NAZARBAYEV UNIVERSITY

Efficient computational approaches to GPU-based Monte Carlo radiation transport

by

Tair Askar

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Science Engineering and Technology

Date of Completion
May, 2025

Efficient computational approaches to GPU-based Monte Carlo radiation transport

by

Tair Askar

Submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Science Engineering and Technology

School of Engineering and Digital Sciences

School of Sciences and Humanities

Nazarbayev University

May, 2025

Supervised by

Prof. Ernazar Abdikamalov

Prof. Daniele Malafarina

Prof. Martin Lukac

Prof. Bobomurat Ahmedov

Declaration

I, Tair Askar, declare that the research contained in this thesis, unless otherwise formally indicated within the text, is the author's original work. The thesis has not been previously submitted to this or any other university for a degree and does not incorporate any material already submitted for a degree.

Signature:

Date:

BLANK

Abstract

This thesis focuses on efficient computational approaches for Monte Carlo Radiation Transport (MCRT) simulations using modern Graphics Processing Units (GPUs). Over the last decade, GPUs have become an important part of scientific computing due to their capability to perform large-scale parallel computations, particularly in areas such as radiation transport. Modern MCRT applications are complex simulations that require immense computational power. This work addresses the challenges and opportunities in using GPU architectures for the MCRT simulations and contributes to the understanding of how to optimize these simulations for better performance and energy efficiency.

A detailed performance examination of several parallel pseudorandom number generators (PRNGs) running on various Nvidia GPU cards is presented in the thesis. `MRG32k3a`, `MTGP32`, `PHILOX4_32_10`, `MT19937`, and `XORWOW` are five PRNGs from the `cuRAND` library that are evaluated for their efficiency in producing uniform and non-uniform random numbers using a range of implementation options, including GPU-only, CPU-only, and hybrid CPU/GPU approaches. This assessment advances our knowledge of PRNG performance optimization on GPUs, particularly with regard to the Monte Carlo (MC) simulations.

The thesis also evaluates two popular Python-based GPU programming platforms, CuPy and Numba, benchmarking against CUDA C for the MCRT simulations. This evaluation is based on performance and energy consumption using memory-intensive operations and compute-heavy problems. The analysis was conducted on Nvidia GeForce RTX3080, Tesla V100, and Tesla A100 GPU cards. It offers information about the advantages and disadvantages of these platforms, which is valuable to the scientific community when selecting tools for GPU-based simulations.

Further, the work investigates the performance scaling of MCRT simulations on

multiple GPUs, focusing on strong and weak scaling, optimization strategies such as `fast math` and `block-thread` configuration, and energy consumption. Using an Nvidia DGX-2 server with up to 10 GPUs, the study demonstrates how different scaling strategies and optimization techniques affect both performance and energy efficiency. This research provides practical recommendations for improving the use of multiple GPUs in large-scale MCRT simulations, contributing to the knowledge of multi-GPU programming and optimization.

Overall, this thesis contributes to the understanding of how to efficiently run and optimize MCRT simulations on GPUs. It includes a detailed analysis of PRNGs performance, evaluates popular Python-based computing tools, and explores how well these platforms can scale their applications across multiple GPUs. This work provides useful insights for researchers, students, and professionals who work with GPU computing, particularly in the field of MCRT simulations.

Acknowledgments

I am truly thankful to my Ph.D. supervisor, Dr. Ernazar Abdikamalov, for his invaluable guidance, patience, expertise, and unwavering support throughout my research journey. I also express my profound appreciation to Dr. Bekdaulet Shukirgaliyev for his mentorship in High-Performance Computing and Computational Science, which greatly enriched my practical knowledge in the field. I extend my sincere thanks to my supervisory team, Dr. Martin Lukac, Dr. Bobomurat Ahmedov, and Dr. Daniele Malafarina, for their expert advice and constructive feedback, which have enhanced the quality of my research. Additionally, I am thankful to Dr. Luis R. Rojas-Solórzano, Associate Provost for Graduate Studies, and Dr. Konstantinos Kostas, Ph.D. Program Director, for their support and guidance in navigating the administrative aspects of my Ph.D. Lastly, I am deeply appreciative of the unwavering support from my family throughout this Ph.D. journey.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Overview

This dissertation is composed of five chapters. It begins with an introductory section before moving on to the practical applications in the subsequent chapters. The opening section of the introduction chapter presents the motivation, outlines our contribution, and provides a framework for getting started. The purpose and motivation part 1.1.1 describes the importance of GPU computing in fields such as Monte Carlo radiation transport (MCRT), where GPUs accelerate complex simulations, enabling faster and more efficient solutions to computationally intensive problems. It addresses the question of why the work presented in this document is important. Part 1.1.2 provides key contributions by describing effective computational approaches to address these issues. Additionally, this chapter provides basic information on the GPU architecture, pseudorandom number (PRN) generation, and the Monte Carlo (MC) computational technique. The performance of pseudorandom number generators (PRNG) from the `cuRAND` library in the CUDA programming framework is thoroughly examined in Chapter 2. Chapter 3 assesses the performance of two Python-based platforms, CuPy and Numba, for the one-dimensional (1D) MCRT test case and PRNGs using a single GPU. This assessment is expanded to a multi-GPU setup in Chapter 4. The thesis is finally summarized in Chapter 5, which also discusses potential future research topics and highlights the key contributions.

1.1.1 Purpose and Motivation

GPU computing has rapidly become an important scientific computing tool over the last decade. Compared to past systems, it allows researchers to address complex computational problems faster today. GPUs have enormous potential for large-scale simulations in a variety of fields as their processing power increases. As of today, 9 out of the top 10 supercomputers in the TOP500 ranking [1] use GPUs as accelerators. This shows GPUs' important role in advancing high-performance computing (HPC).

This impact is especially significant in MC simulations, a popular computational technique for solving radiation transport (RT) problems. MCRT relies heavily on PRN generation to model the stochastic behavior of particles as they interact with matter or other particles. The accuracy and efficiency of the PRNG directly influence the overall performance of MCRT applications. Therefore, performance evaluation and optimization of PRNGs on GPU platforms are important in improving speed in such simulations.

Although PRNGs play a crucial role in MCRT simulations, there is a noticeable gap in the research about their performance, particularly for GPU-based MCRT problems. This gap highlights the need for more studies on how PRNGs function on GPUs in these contexts. Understanding how different PRNGs function on various GPU architectures is vital since it can help improve performance in large-scale simulations.

This thesis aims to address this gap by conducting performance testing experiments of PRNGs on GPUs for the MCRT problems. The goal is to provide a thorough examination of different GPU platforms, including their respective programming paradigms — CUDA, Numba, and CuPy. As a result, users will be better able to determine which platform is most effective in enhancing MC simulations due to this. Ultimately, this research aims to contribute to broader efforts focused on optimizing GPU-based simulations to satisfy the current expectations in the field of scientific and engineering studies.

1.1.2 Key Contributions

This thesis delivers the following key contributions emphasized in each chapter, specifically addressing efficient computational approaches to GPU-based MCRT. These contributions focus on techniques that offer valuable insights and optimizations for leveraging GPU architectures effectively for RT simulations:

- In Chapter 2, we present a comprehensive evaluation of the performance of various parallel PRNGs on Nvidia GPUs, specifically the RTX3090, GTX1080Ti, GTX1080, and RTX3080. We analyze five distinct PRNGs from the `cuRAND` library (namely `MRG32k3a`, `PHILOX4_32_10`, `MT19937`, `MTGP32`, and `XORWOW`) and generate uniformly distributed PRNs that are subsequently converted into non-uniform distributions through the acceptance-rejection (AR) technique. Our study includes several implementation approaches: a hybrid CPU/GPU approach where the GPU generates PRNs and transfers them to the CPU; a single CPU core approach where all computations are performed by the CPU; and two GPU-based approaches, one involving data movement between the GPU and CPU, and another excluding this transfer. We analyze performance across different aspects, comparing `cuRAND` library's host API and device API. These evaluations highlight the important parameters affecting the performance of PRNGs on modern GPU cards. They provide useful insights into optimizing their performance for MC simulations;

- Chapter 3 investigates the performance of two GPU programming frameworks, Numba and CuPy, in the scope of MCRT simulations, with a comparison to a CUDA C implementation. The study focuses on two key test cases: a compute-intensive 1D MCRT problem and a memory-intensive task that involves generating pseudo-random numbers and storing them in global memory. The evaluation is conducted on three Nvidia GPUs: GeForce RTX3080, Tesla V100, and Tesla A100. This work aims to provide insights into the advantages and disadvantages of each platform. In the context of GPU-based MC simulations, it highlights the trade-offs between performance, power consumption, and implementation simplicity;

- Chapter 4 evaluates the performance of Numba and CuPy to solve the MCRT

test case using multiple GPUs. The study explores both strong scaling (where the problem size remains fixed while the number of GPUs increases) and weak scaling (where the problem size increases proportionally with the number of GPUs). It also examines the impact of various optimizations, such as `fast math` and `block-thread` configuration adjustments, and assesses energy consumption. The implementations are tested in two modes: single-CPU thread, where one CPU thread manages all GPUs, and multi-CPU thread, where each GPU is controlled by its dedicated CPU thread. The experiments are conducted on an Nvidia DGX-2 server with up to 10 GPUs. This work, therefore, highlights how performance and energy efficiency vary with different scaling and optimization approaches for a better understanding of the use of multiple GPUs for MCRT simulations.

1.2 Overview of the GPU Architecture

CUDA, developed by Nvidia in 2007, revolutionized parallel computing by allowing developers to use Nvidia GPUs not just for graphics rendering but also for scientific computations. The main feature of CUDA is the use of "kernels," which are functions that run on the GPU. These kernels take advantage of the GPU's ability to perform many tasks at once by running across thousands of lightweight threads that work together to perform calculations. This gives GPUs a substantial speed advantage over CPUs, which are built for general tasks and rely on a few powerful cores. GPUs, with their thousands of cores, are highly effective at handling tasks that require a lot of parallel processing, making them more efficient for floating-point calculations and other heavy computational tasks [2].

To take advantage of parallel processing on the GPU, the programmer needs to break the problem into smaller, clear tasks organized in a computational grid (see Figure 1.1). This grid can be one-, two-, or three-dimensional, allowing different tasks to fit the GPU's design. A grid is divided into blocks, and each block contains a number of threads. The programmer decides how many threads are in each block. This layered approach efficiently maps the execution of a problem with the GPU's Streaming

Multiprocessors (SM), therefore, ensuring optimal resource utilization [3].

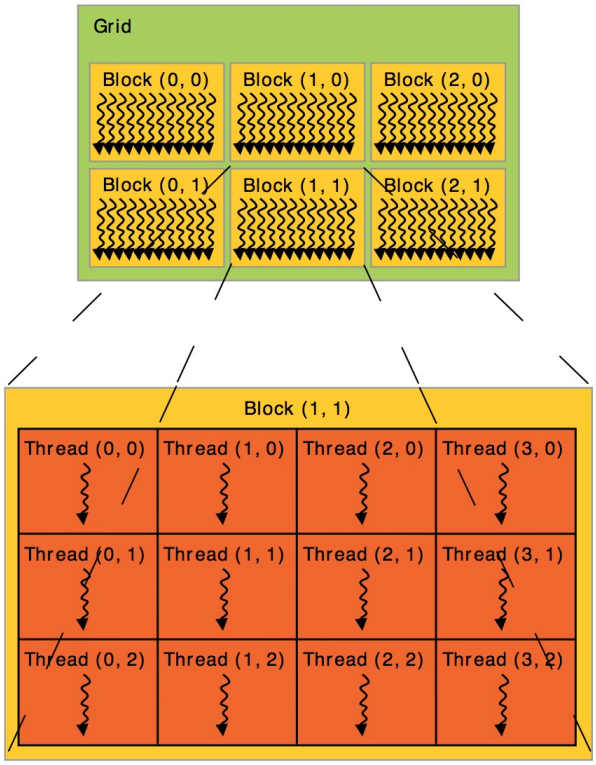


Figure 1.1: The hierarchical organization of a CUDA computational grid. A kernel is a user-defined function executed on a grid, further divided into smaller units called thread blocks (source image: CUDA C programming guide [3]).

As seen in Figure 1.2, each SM is composed of multiple CUDA cores, which are the basic execution units used in parallel computing. These cores are devoted to high-throughput operations, particularly for calculations using floating-point numbers with single and double precision. SMs also represent various levels of memory to enhance data access efficiency. The programmers have to try to achieve coalesced memory access, where a warp (32 threads for Nvidia GPUs), retrieves data from contiguous addresses within global memory space. This memory access mode is much faster than memory accesses that are not contiguous because the GPU has to process such data in one transaction, reducing the number of memory operations needed.

SMs use latency-hiding technique to achieve higher performance. As access to global memory is much slower compared to fast on-chip memory, an SM tries to hide this latency by executing instructions from other warps while one is waiting for its global memory access to complete. This way, memory access latencies are effectively

Figure 1.2: Streaming Multiprocessors (SM) are the main elements of parallel computations, where many CUDA cores are utilized for high-throughput, floating-point performance. Each SM possesses a multilevel memory hierarchy: private registers assigned to every single thread, shared memory used by all threads within one block, and global memory for data storage of a wider scope. A warp scheduler manages the execution of 32-thread groups called warps and ensures that all threads execute the same instructions at every clock cycle in lockstep (source image: Nvidia).

hidden, and the CUDA cores stay active, maximizing the overall processing efficiency and throughput [4].

In addition, every thread block has a shared memory space within the SM, which enables effective data interchange and communication amongst threads in the same block. Each thread also has private registers, which serve as ultra-fast on-chip memory for temporarily storing data.

A thorough understanding of the underlying hardware architecture, particularly the hierarchical organization of threads, is necessary for CUDA programming to achieve optimal performance. Each thread block is split into smaller chunks known as warps in order to maximize the utilization of resources. It is strongly recommended that the number of threads in each block be a multiple of the warp size for best performance.

This ensures the smooth grouping of threads into warps that execute instructions in a lockstep, synchronized way. Overall efficiency can be considerably reduced whenever conditional branching occurs, causing warps to diverge. Programmers should make sure that all threads in a warp follow the same execution path in order to maximize the use of the resources that the SM provides and prevent any thread from sitting idle. This will help to achieve peak performance for parallel computing applications [5].

1.3 Monte Carlo Method for Radiation Transport

The MC method is a common computational technique for simulating physical processes involving randomness or stochastic events. In the case of RT, MC methods are especially useful in modeling the interactions between radiation (photons, neutrons, electrons, etc.) and matter. It employs statistical sampling for tracing individual particle trajectories due to interactions caused by scattering, absorption, and emission. The suitability of this approach for the randomness in physical processes makes it highly suitable for the accurate modeling of radiation behavior in different media [6].

In the MC method of RT simulations, each particle is traced from its emission to its eventual absorption or escape from the system. While traveling, particles undergo probabilistic interactions, as defined by cross-sections or probabilities derived from experimental data or theoretical models. These include scattering and absorption, along with energy and direction changes. The strength of the MC method is its ability to model these events in a manner that is similar to that of the actual processes of RT. By simulating many particle trajectories, a statistical representation of the behavior of the radiation field in different geometries and media is obtained. This flexibility allows this method to be used in a variety of fields, such as astronomy, nuclear engineering, medical physics, and others.

In MC simulations, the source emits particles that travel a certain distance, depending on the mean free path of that material, before interacting with the medium. The type of each interaction is determined by the physical cross-sections of various processes that may occur. In general, after an interaction, the path, energy, and sometimes even the

type of the particle are recalculated before it is allowed to continue its journey. This process is repeated until the particle is either absorbed or escapes the system [7].

The MC method depends on statistical outcomes since particle behavior is random and governed by physical probabilities. As a result, the outcomes are probabilistic, not exact. To achieve accurate results, a large number of particles may need to be simulated, depending on the complexity and required precision of the problem. While this leads to more accurate outcomes, it also demands significant time and computational resources. Finding a balance between accuracy and resource usage is key to optimizing the process.

The MC method is widely applied to solve different kinds of RT problems. In medical physics, for instance, it helps simulate the interaction of radiation with human tissue, thereby helping in treatment planning and the development of diagnostic imaging techniques. In nuclear engineering, MC simulations are important for understanding how neutrons move in reactors, which helps keep them safe and running efficiently. In astrophysics, such techniques have been applied to model RT in stars and other bodies. From these subjects, the flexibility of the MC method makes it a necessary tool for understanding the behavior of radiation in the most complex situations.

However, due to its accuracy and flexibility, the MC method faces major computational challenges. A large number of particles need to be tracked, which demands a lot of computing power, especially when dealing with complex geometries or large systems. HPC systems, especially those with GPUs, have become important for accelerating MC simulations. Given their massively parallel computations, GPUs are fit for the independent particle tracking that characterizes MC methods.

Optimizing MC simulation on GPUs requires effective workload distribution, memory management, and, most importantly, full utilization of computing resources. The important strategies here will be to minimize data transfer between CPU and GPU, optimize the thread and block configuration, and effectively use the fast memory. Large-scale MC simulations can be greatly accelerated with Python-based libraries like CuPy and Numba that support GPU acceleration and direct CUDA programming. These optimizations allow for faster and more scalable computations, making it possible to tackle more complex and resource-intensive RT problems.

The MC method forms the most vital element in RT simulation exercises, allowing the solution of a wide set of applications with great flexibility and accuracy. The capability to model the random interactions of radiation with matter makes the technique invaluable in various fields, from healthcare to nuclear energy. Despite this computational burden, the advancement in GPU computing and multi-GPU systems has opened new avenues for enhancing efficiency and scalability in MC simulations. As technology evolves, the MC method will remain at the frontiers of research and applications in RT while allowing more complex and detailed simulations to be conducted with higher speed and accuracy.

1.4 Pseudo-Random Number Generation

In MCRT simulations, the generation of PRNs is an important component, and the methodology relies on random sampling for modeling the particle's behavior probabilistically in interactions with various media. The PRNG algorithms will create number sequences that behave like random numbers. This enables a simulation that may model a stochastic nature for RT. The quality of the PRNG directly impacts the precision and reliability of the simulation since poor randomness can result in biases or errors within the results. In highly large-scale MCRT simulations, with an enormous number of random samples being required, the efficiency of a PRNG is crucial to ensure that such a simulation runs at an acceptable speed without excessive computational overhead. Therefore, PRNG selection or optimization in parallel computing environments, such as multi-GPU systems, becomes important to realize performance and precision in MCRT simulations.

1.4.1 Inverse Transform Method

In MCRT simulations, after the generation of PRNs, it is quite common to apply the inverse transform method for sampling from particular probability distributions [8]. The inverse transform method uses a uniformly distributed random number to sample any arbitrary probability distribution. It is accomplished by having first the cumulative

distribution function (CDF) of any given distribution and then finding its inverse. Given a uniformly distributed random number U between 0 and 1, the following formula provides a solution:

$$F^{-1}(U) = X, \tag{1.1}$$

where F^{-1} is the inverse of the CDF and X is the sampled value from the desired distribution. This technique is widely employed in MCRT since all the physical processes involved-such as the interaction of particles, their scattering, and absorption-can be modeled using various known probability distributions. The inverse transform method provides a systematic way of mapping uniform random numbers onto these distributions. It is, hence, one of the powerful tools for simulating random behavior essential in RT calculations. When implemented efficiently, it ensures both accuracy and computational efficiency in the large-scale sampling required by MCRT simulations.

1.4.2 Acceptance-Rejection Method

Besides the inverse transform method, another popular technique for sampling from complicated probability distributions in MCRT simulations is the acceptance-rejection (AR) method [9]. This method is especially helpful when it's hard to directly sample from the desired probability distribution. The proposal distribution is a simpler distribution from which the AR method generates sample candidates. The proposal and the desired distribution are then compared, and each sample is either approved or denied.

First random sample X is selected from the proposal distribution, which should to have a distribution that is easy to sample and covers the range of the target distribution. A second random number, U, uniformly distributed between 0 and 1, is then generated. The candidate (random) sample X is accepted on the condition that

$$U \leq \frac{f(X)}{cg(X)}, \tag{1.2}$$

where $f(X)$ is the target distribution, $g(X)$ is the proposal distribution, and c is a constant chosen such that $f(X) \leq cg(X)$ for all X. If the sample is accepted, it will be used in the simulation. Otherwise, repeat the above steps until an appropriate sample is obtained.

AR is a versatile method that applies to all types of distributions. It becomes even more useful for the MCRT problem, whose interactions or events follow an arbitrary complex statistical distribution pattern. Nevertheless, it also depends heavily on the efficiency of a proposed distribution. If the shape of a proposed distribution fits well with the objective distribution, it minimizes the number of rejected samples. Consequently, the methodology remains computationally efficient, even in the case of large-scale RT simulations.

BLANK

Chapter 2

Analyzing Performance of GPU-based Pseudo-Random Number Generators on Nvidia GPU Cards

Chapter 2 of this PhD dissertation is based on the following paper:

Askar, Tair, Bekdaulet Shukirgaliyev, Martin Lukac, and Ernazar Abdikamalov. 2021. "Evaluation of Pseudo-Random Number Generation on GPU Cards." Computation 9, no. 12: 142.

2.1 Literature review

Numerous studies have explored parallel pseudo-random number (PRN) generation on multi-core CPUs [10, 11, 12, 13, 14] and GPUs [15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 2, 39], focusing on how to assign distinct pseudo-random number generator (PRNG) instances to different threads, each generating unique random number sequences. This can be attained through two main methods: splitting and parameterization [10, 31, 39]. By using the splitting technique, the full sequence of PRNs produced by a serial operation is split up into smaller subsequences that are distributed over several threads or processors. The parameterization method runs the same or similar random number generators on

different threads, with each thread using different algorithm settings to make them unique. It should be compact in size to fit in local memory (the memory on the chip in this case) for the best performance. For example, Langdon [40] implemented the Park-Miller PRNG on a GPU using Compute Unified Device Architecture (CUDA), achieving at least a 44× speedup compared to its CPU counterpart. Gong et al.'s [41] implementation of a PRNG for Monte Carlo (MC) particle transport on a GPU showed a speedup of up to 8.1× over a CPU with 4 to 6 cores. Further, Nandapalan et al. [42] compared the xorgenGP PRNG to the MTGP32 generator and found comparable speeds. Kargaran et al. [43] used Fortran on CUDA to present a PRNG developed on GPU, utilizing shared and global memory to store pre-generated seed tables. The shared and global memory implementations achieved 470× and 150× speedups over a single CPU core, respectively, although the shared memory PRNG was 13% slower than the `cuRAND` `XORWOW` generator. Riesinger et al. [44] found that an optimized `SHR3` generator for solving differential equations performed 79% better than the `cuRAND`'s `XORWOW` PRNG for uniform distributions and 38% better for normal distributions. Lastly, Jun et al. [45] showed that a comparison between PRNGs from the `VecRNG` package and the `cuRAND` library, generating 10^7 double-precision PRNs, revealed that the `PHILOX4_32_10` PRNG from `cuRAND` library was 5× faster than `VecRNG`'s version, whereas both libraries' `MRG32k3a` PRNGs performed similarly.

2.2 Motivation

The main objective of this research is to assess the generation of uniform and non-uniform PRN samples, which is a fundamental aspect of MC simulations. This component frequently contributes substantially to the overall execution time. Building on previous research, we expand the analysis to include a broader set of performance metrics, such as the overhead from API calls (e.g., device synchronization, allocation of memory), data movement times between the GPU and the CPU, state setup time, block and thread configuration, and other performance-affecting factors [46]. In this study we use several contemporary Nvidia GPUs to evaluate 5 distinct `cuRAND` library

generators: MRG32k3a, MTGP32, PHILOX4_32_10, MT19937, and XORWOW, which generate uniformly distributed PRNs. Then we transform these uniform sequences into three non-uniform distributions — Rayleigh, Beta, and Gamma [47, 48, 49] — using the acceptance-rejection (AR) method. These computational experiments are designed as simplified test cases to assess the most basic aspects of MC simulations.

We explore the effects of various implementation options on performance. Our study examines both CPU and GPU methods along with a hybrid approach in which uniform PRNs are produced on the GPU and then converted into a non-uniform samples on the CPU via the AR technique. We evaluate two approaches for the GPU implementations, utilizing either the host or device APIs. We also examine the effects of the number of blocks allocated to each streaming multiprocessor (SM) and the threads per block configuration on performance. Lastly, the performance across various GPUs is compared.

2.3 Experimental Setting

Table 2.1 provides the main characteristics of the 5 distinct PRNGs that we examined in this study. The splitting technique is used by the MRG32k3a, MT19937, and XORWOW PRNGs, whereas the parameterization technique is used by the MTGP32 PRNG. PHILOX4_32_10 uses a parameterization approach in its host API version, but it uses a splitting method in the device API version. [50]. The period of a PRNG determines the maximum number of random numbers it can generate before the sequence starts to repeat. The subsequence length, in turn, indicates the amount of random numbers each thread or stream can generate without colliding with other streams or threads. Additional information on the above-mentioned PRNGs is available in [50].

A single AMD Ryzen Threadripper 3990X CPU core and multiple Nvidia GPU cards, such as the RTX3090, RTX3080, GTX1080Ti, and GTX1080, are used for our computations (see Table 2.2 for GPU specifications). To make a fair comparison, we apply the same application and parameter settings (such as ordering and seed) for all tests while maintaining a similar combination of blocks and threads. Additionally, we

Table 2.1: The main parameters of the 5 `cuRAND` library PRNGs analyzed in this study.

	MT19937	**XORWOW**	**MRG32k3a**	**MTGP32**	**PHILOX4_32_10**
Algorithm	Twisted generalized feedback shift register generator [51]	Linear feedback shift registers [52]	Combined Multiple Recursive [53]	Twisted generalized feedback shift register generator [54]	Counter-Based Random Number Generation [55]
Period	2^{19937}-1	2^{192}-1	2^{191}	2^{11214}	2^{128}
Subsequence length	2^{1000}	2^{67}	2^{67}	–	2^{64}
Parallelization method	Sequence splitting	Sequence splitting	Sequence splitting	Parameterization	Sequence splitting, parameterization

explore various thread-per-block combinations on the different GPU cards to identify the range that provides the best performance. Our code is developed in C/C++ using the CUDA 11.4 version with the Nvidia 470.57.02 driver version [3]. We use profiling software such as `nvprof` and `Nsight Systems`, along with tools like `cudaEvent_t` and `clock_t` to assess computation times. The data type used for all computations is single-precision floating point.

Table 2.2: The main technical details of the Nvidia GPUs used in the study.

	RTX3080	**GTX1080Ti**	**GTX1080**	**RTX3090**
CUDA cores	4352	3584	2560	10496
SMs	68	28	20	82
Global memory	10 GB	11 GB	8 GB	24 GB
Max clock rate	1.8 GHz	1.58 GHz	1.73 GHz	1.7 GHz
Bandwidth	760.3 GB/s	484.4 GB/s	320.3 GB/s	936.2 GB/s
Theoretical performance	29.77 TFLOPS (FP32)	11.34 TFLOPS (FP32)	8.873 TFLOPS (FP32)	35.58 TFLOPS (FP32)

In assessing PRNG performance on GPUs, we take into account 4 distinct approaches: (1) In the GPU-only approach (`G_imp`), all computations are performed on the GPU, and uniform PRNs are produced on the GPU before being converted into non-uniform distributions (such Gamma, Beta, and Rayleigh) using the AR technique. (2) In the

(`C_imp`), all computations, including PRN generation and transformation, are performed on a single CPU core. (3) The CPU/GPU hybrid approach (`H_imp`) uses the GPU to produce uniform PRNs, which are subsequently moved to the CPU RAM, where a single CPU core uses the AR technique to transform the uniform PRNs into non-uniform distributions. (4) In the GPU implementation with memory copy (`G_imp_mcpy`), the time required to transfer data from the GPU to the CPU after the calculation is complete is taken into account.

We will examine both a device API and a host API for calling the PRNG functions in the `G_imp` approach. In the latter case, the CPU calls the PRNG library functions, but the GPU performs the actual computing. The generated random numbers can be kept in the GPU's global memory space or transferred back to the host side for further processing. In the former case, the device API makes it possible to use the PRNG functions directly from within a GPU kernel, allowing the device to set up parameters such as state, seed, and sequence. Without being stored in global memory, the generated PRNs are directly usable by the GPU kernel. We use a single kernel to implement all PRNG operations, including seed setup and state update, inside the device API in order to minimize data movement within GPU memory. These two API strategies are compared in detail. The AR method makes a series of PRNs with probability distribution $f(x)$ by generating 2 equal-length sets of uniformly distributed PRNs, x and y. If the values of x meet the requirement that $y < f(x)$, then they are accepted. The acceptance rates of the Rayleigh, Gamma, and Beta distributions utilized in this study are 0.41, 0.36, and 0.67, respectively. These distribution functions are plotted for $0 \leq x \leq 1$ in Figure 2.1, normalized so that $f(x) \leq 1$.

We use computation execution time as the main performance metric, with an average of more than 100 iterations for all reported findings. Table 2.3 describes our standard configuration, which makes use of an RTX3090 GPU, Beta distribution, and the `MRG32k3a` PRNG in device API mode. We examine each parameter's implications in the following sections. It's important to note that this default configuration does not represent the optimal parameters but serves as a baseline against which other parameter configurations are compared for performance.

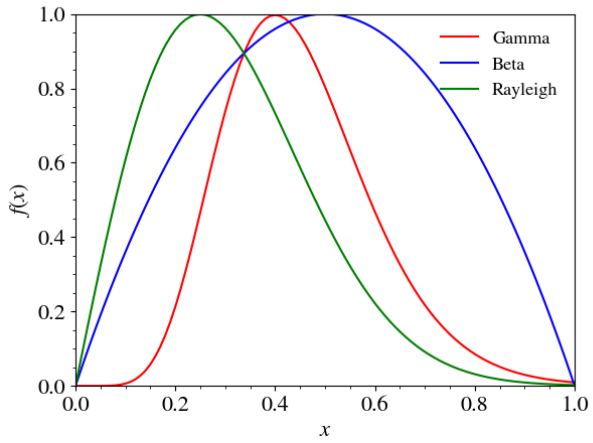


Figure 2.1: The Rayleigh, Gamma, and Beta distributions' probability density functions are normalized to $f(x) \leq 1$. For these distributions, the corresponding areas under the curves are 0.41, 0.36, and 0.67.

Table 2.3: The experimental setup's standard settings are used as a reference point. We vary the values of each parameter to examine how performance is affected by these changes.

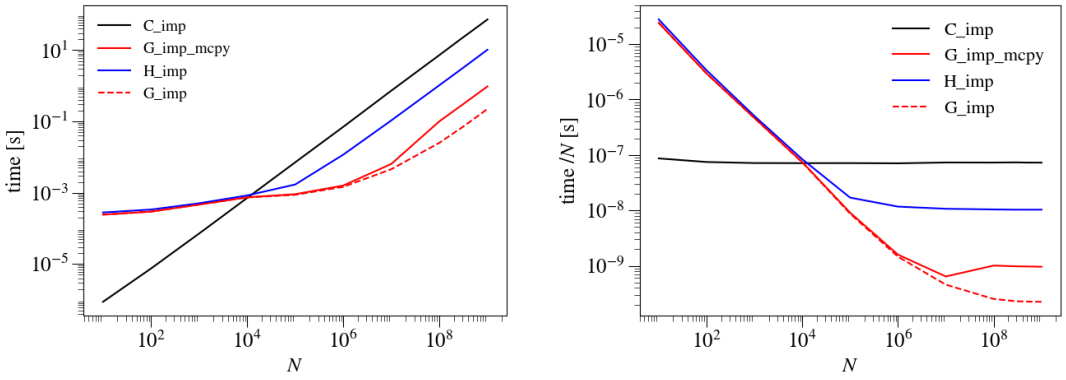
Parameter	**Value**
Distribution	Beta distribution
GPU	RTX 3090
Implementation	`G_imp`
API	device API
PRNG	MRG32k3a

2.4 Results

2.4.1 Benchmarking of CPU and GPU Methods

Figure 2.2a shows the computation time for various methods relative to the amount of generated random candidate samples (N). The CPU method `C_imp` is up to 2 orders of magnitude faster than the hybrid `H_imp` and GPU `G_imp` implementations for smaller values of N (e.g. $N \lesssim 10^4$). A low GPU occupancy around $\sim$3% is the cause of this performance disparity (occupancy is the ratio of active warps to the maximum number of warps that can execute on the GPU). Unlike CPU cores, GPU cores are underutilized due to the small N. A partially utilized GPU executes more slowly than a fully utilized CPU because a single CPU core with its complicated architecture is a way faster than a single GPU core. In hybrid implementation `H_imp`, when GPU generates the PRNs but

the CPU performs the *AR-selection* , performance is similar to `G_imp` due to the same low GPU occupancy bottleneck.



(a) (b)

Figure 2.2: Overall computation time (**a**) and computation time per candidate sample (**b**) for the Beta distribution across various methods as a function of N. `C_imp` is represented by the solid black line; `G_imp` is represented by the dashed red line; `G_imp_mcpy` is represented by the solid red line; `H_imp` is represented by the solid blue line.

GPU utilization rises with N, surpassing CPU performance at $N \gtrsim 10^4$. At $N \gtrsim 10^8$, the `G_imp` is $\gtrsim 10^2$ times faster than the `C_imp`. The `H_imp` is approximately $7\times$ faster than `C_imp`, yet it is $\simeq 46$ times slower than the `G_imp`.

For $N \lesssim 10^6$, the computation times for `G_imp` and `G_imp_mcpy` are about the same, but they begin to differ at bigger N. This divergence occurs due to the `G_imp_mcpy` encounters a bottleneck in data transfer from GPU to CPU for $N \gtrsim 10^6$. In contrast, at smaller N, the time required for data transport is negligible in comparison to other computational tasks such as API calls, PRNG state updates, and seed setup.

The computation time per candidate sample as a function of N, displayed in Figure 2.2b, reflects these outcomes. In the `C_imp` implementation, the execution time per candidate point remains constant, regardless of N, indicating that the CPU core is 100% used for all values of N (as observed using the `htop` program). In the `G_imp` method, the lowest execution time is achieved at $\sim 10^8$ due to the large number of cores available. For `G_imp_mcpy` method, the minimum computation time per candidate sample is reached at $N \sim 10^7$. Further, with increasing N, the execution time increases because of the overhead from device to host data movement. This time reduces in the

`H_imp` implementation until $N \sim 10^5$, after this point it stabilizes. The fact that the execution time levels off indicates that the CPU becomes the computational bottleneck for $N \gtrsim 10^5$ in the `H_imp` due to the AR algorithm which runs on the CPU. For `G_imp` the saturation is reached at a much higher N.

2.4.2 Per-Operation Load Evaluation

API functions, PRNG state update, seed initialization, and the AR algorithm are the components that contribute to the total cost for generating non-uniform PRNs. Each of these components' execution times in the `G_imp` implementation are shown in Figure 2.3a, and Figure 2.3b shows each component's percentage contribution to the overall execution time.

For $N \lesssim 10^7$, seed initialization accounts for over $\sim 30\%$ of the overall computation time, peaking at $\sim 80\%$ for $10^4 \lesssim N \lesssim 10^5$. The PRNG state update remains under $\sim 10\%$ for $N \lesssim 10^7$, but its share increases to $\sim 40\%$ by $N \sim 10^9$. For $N \lesssim 10^5$, the AR technique consumes less than $\sim 20\%$ of the total time, but its contribution steadily increases with N, reaching $\sim 50\%$ at $N \sim 10^9$. API function calls make up $\sim 40\%$ of the time for $N \lesssim 10^2$, but this share drops to below $\sim 20\%$ for $N \gtrsim 10^3$. However, as discussed later, these results are influenced by the PRNG parameters, such as state size and seed initialization. The `G_imp_mcpy` method, illustrated in Figures 2.3c and 2.3d, displays a similar cost breakdown for $N \lesssim 10^6$. Beyond $N > 10^6$, however, data movement to the GPU becomes the prevailing factor, comprising $\sim 80\%$ of the time at $N \sim 10^8$. As a result, the relative contribution of other tasks, including the AR technique, is reduced compared to the `G_imp` method.

2.4.3 Comparing Device and Host API Modes of PRNG

In the `G_imp`, as previously noted, we explore 2 distinct methods for invoking the PRNG processes: utilizing the device API and the host API. Figure 2.4a illustrates the computation time for the Beta distribution using the `MRG32k3a` PRNG in both device and host API modes. The device API outperforms the host API by roughly

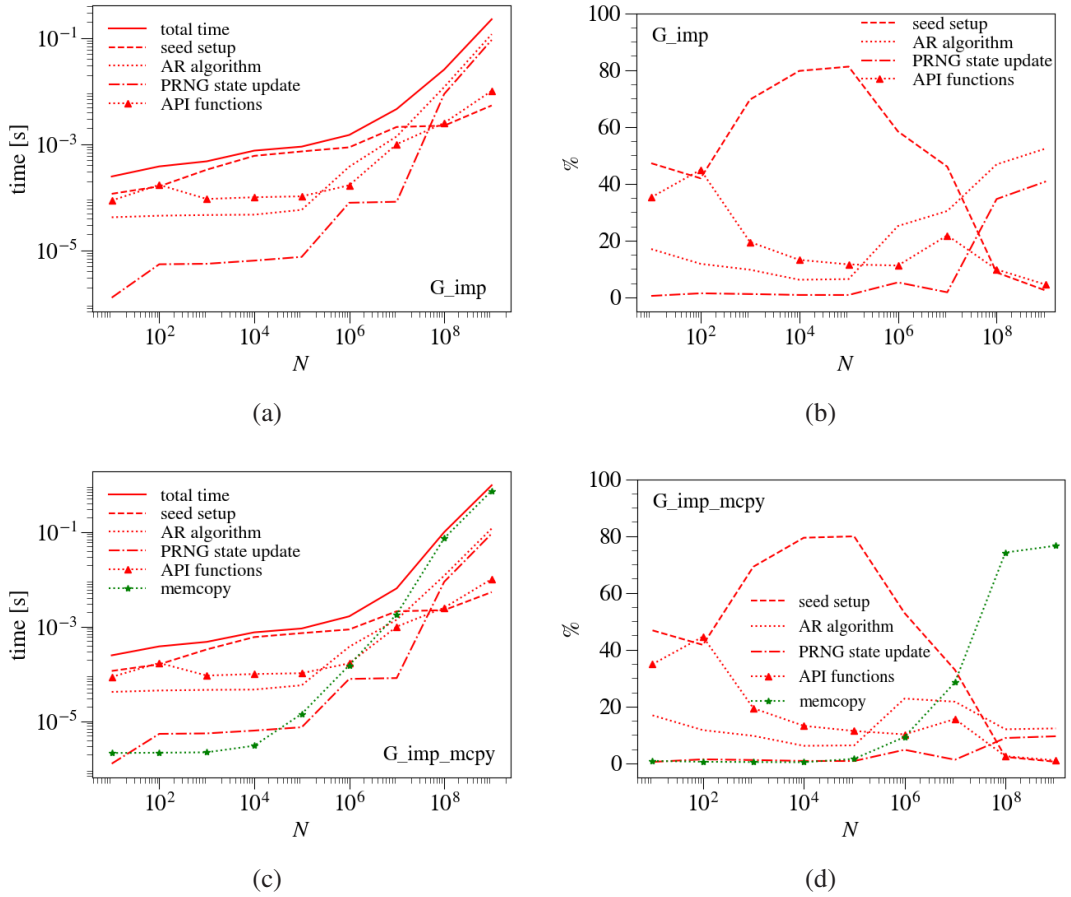


Figure 2.3: Computation time and percentage breakdown of various computing components for the `G_imp` (**a** and **b**) and `G_imp_mcpy` (**c** and **d**) methods as a function of N. The PRNG seed setup time is represented by the dashed line, while the overall execution time is shown by the solid line. The PRNG state update is displayed by the dashed-dotted line, while the dotted line represents the AR technique. The memory copy time from device to host is shown by the green dotted line with star markers, and API function calls are represented by the dotted line with triangle markers.

an order of magnitude for $N \lesssim 10^6$. The dashed blue line indicates the PRNG seed setup time, which is a bottleneck for the host API mode under these conditions and is the main source of this performance difference. The default implementation of the `MRG32k3a` generator via the host API initializes 32768 threads regardless of the quantity of produced PRNs. The host API has a consistent seed initialization time of $\sim$6.1 ms for all N because every thread maintains its state. However, the PRNG seed setup time can be decreased through altering the ordering parameter. The order of PRNs in GPU memory is determined by the ordering option in PRNGs [3]. For example, the `CURAND_ORDERING_PSEUDO_LEGACY` setting for `MRG32k3a` will decrease the seed initialization time by about six times in comparison to the standard configuration, since

it initializes fewer threads (4096). In contrast, the device API allows us to choose the number of initialized threads based on N. Consequently, as seen by the red dashed line in Figure 2.4a, the PRNG seed initialization time for the device API grows with N, going from 0.116 ms at N=10 to 5.36 ms at N=10^9. It has also been described in [56, 57] how to speed up seed initialization time by adjusting the ordering choice.

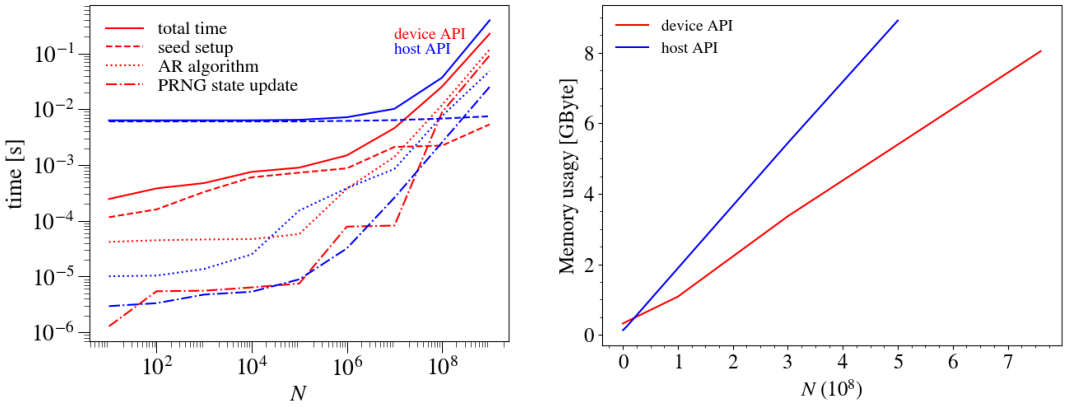


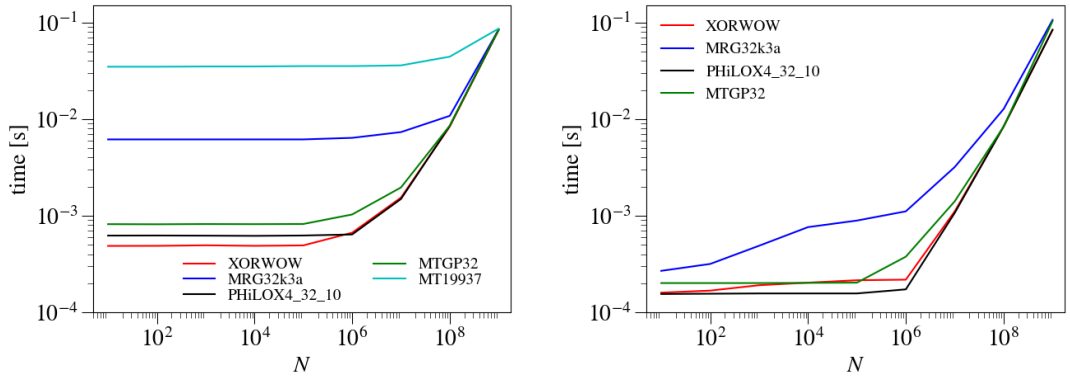
Figure 2.4: The device and host API methods' execution times (**a**) and GPU global memory utilization (**b**) as a function of N. In (**a**), dashed lines show the seed initialization time, dashed-dotted lines show the PRNG state update time, and solid lines show the whole computation time. The dotted lines show the time used applying the AR technique.

The computation times for the AR technique and the PRNG state update (represented by the dotted and dash-dotted lines) grow as N rises for both modes. Consequently, the seed setup's related importance decreases as N increases. This results in a smaller computation time difference between the host API and device API modes, which eventually reduces to a factor of $\sim$1.5 at $N\sim10^8$.

The global memory utilization for both device and host API implementations is shown in Figure 2.4b. The host API mode consumes more memory, as it needs to store the produced random numbers in global memory during computation. In contrast, PRNs can be used instantly with the device API mode, which eliminates the need for global memory storage. Therefore, if the random numbers have to be reused multiple times or transferred to the host, the host API mode is more suitable. If not, in order to save memory and prevent further loading and storing activities in global memory, the device API is better.

2.4.4 Performance Evaluation of Various PRNGs

The computation time of different PRNGs as a function of N for both device and host API modes is shown in Figure 2.5. In order to eliminate the influence of AR computations, we only consider uniform distribution sequences in this subsection. The `MT19937` PRNG is excluded from Figure 2.5b because it does not support the device API. As previously discussed for the `MRG32k3a` generator (see Subsection 2.4.3), the host API's longer seed setup time results in extended execution times, particularly at smaller N. A similar performance gap between the host and device modes is observed for the other PRNGs analyzed here.



(a) (b)

Figure 2.5: Computation time for various PRNGs as a function of N for the host (**a**) and device API (**b**) implementations. The `MT19937` PRNG is excluded from (**b**) because it does not support the device API [3].

The fastest PRNGs in both modes are `PHILOX4_32_10` and `XORWOW`, with `MTGP32` coming next. While `PHILOX4_32_10`, a non-state PRNG that uses thread IDs as its state, achieves higher speeds by generating PRNs based on a counter [55], `XORWOW` relies on XOR and shift bitwise operations, which execute in a single clock cycle [58]. The device API mode's slowest PRNG is `MRG32k3a`, while the host API mode's slowest is `MT19937`, which is, as previously said, not supported in the device API. `MT19937`'s large state size of 2.5 kB reduces SM core occupancy due to limited on-chip memory per SM. Additionally, the `MRG32k3a` generator's use of the modulus operator, which has a latency of 22-29 clock cycles [59, 58], contributes to its longer computation times.

The performance gap between various PRNGs is especially noticeable for $N \lesssim 10^7$

but becomes minimal at $N \gtrsim 10^8$. This is due to API function calls dominate the execution time for the three fast PRNGs (`MTGP32`, `PHILOX4_32_10`, and `XORWOW`) (see Figure 2.6), making their speeds more comparable to the 2 slower PRNGs (`MRG32k3a` and `MT19937`). In the case of `MT19937`, seed setup is the main time-consuming part of the total execution time for $N \lesssim 10^7$. For `PHILOX4_32_10`, nearly all execution time in the device API implementation is spent on API function calls.

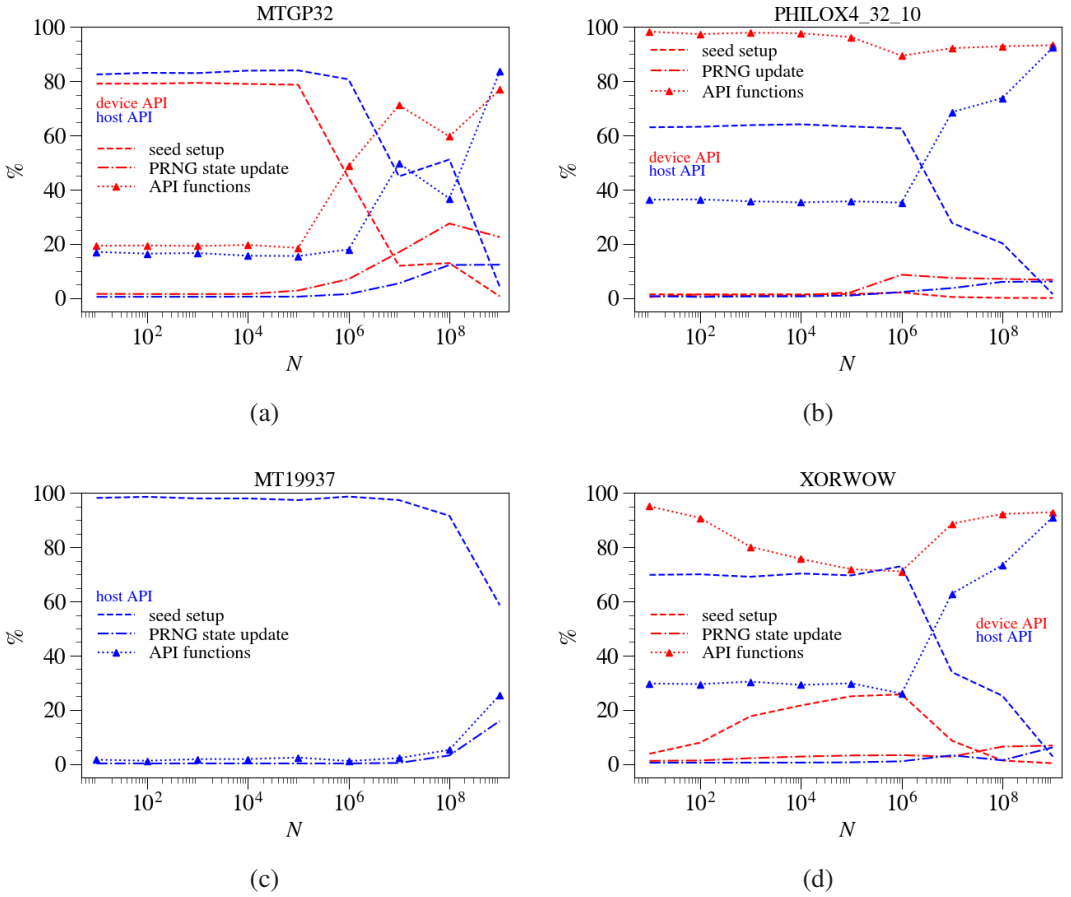


Figure 2.6: Percentage breakdown of various computing components for a range of PRNGs as a function of N. (**a**) `MTGP32`, (**b**) `PHILOX4_32_10`, (**c**) `MT19937` (unsupported on device API), (**d**) `XORWOW`. The time spent on API function calls is represented by the dotted lines with triangle markers, the dashed lines show the PRNG seed initialization time, and the dashed-dotted lines show the PRNG state update time.

Figure 2.7 illustrates the normalized computation time for four PRNGs' GPU occupancy, excluding the `MT19937` due to not supporting device API. The times are normalized against the lowest possible execution time achievable by each PRNG on the GPU for certain values of N. The `PHiLOX4_32_10` PRNG performs best when the GPU reaches full occupancy (100%). This is primarily due to its small

state size (see Table 1), allowing more threads to be accommodated in GPU memory. Additionally, its short seed setup time (refer to Figure 2.6) means that increasing the number of threads—which extends the seed setup duration—does not negatively impact performance. On the contrary, more threads result in faster PRNG state updates and AR computations because of increased parallel calculations. For $N = 10^7$ and $N = 10^5$, the `XORWOW` generator performs optimally at 5% occupancy, whereas for $N = 10^9$, performance remains largely unaffected by occupancy levels.

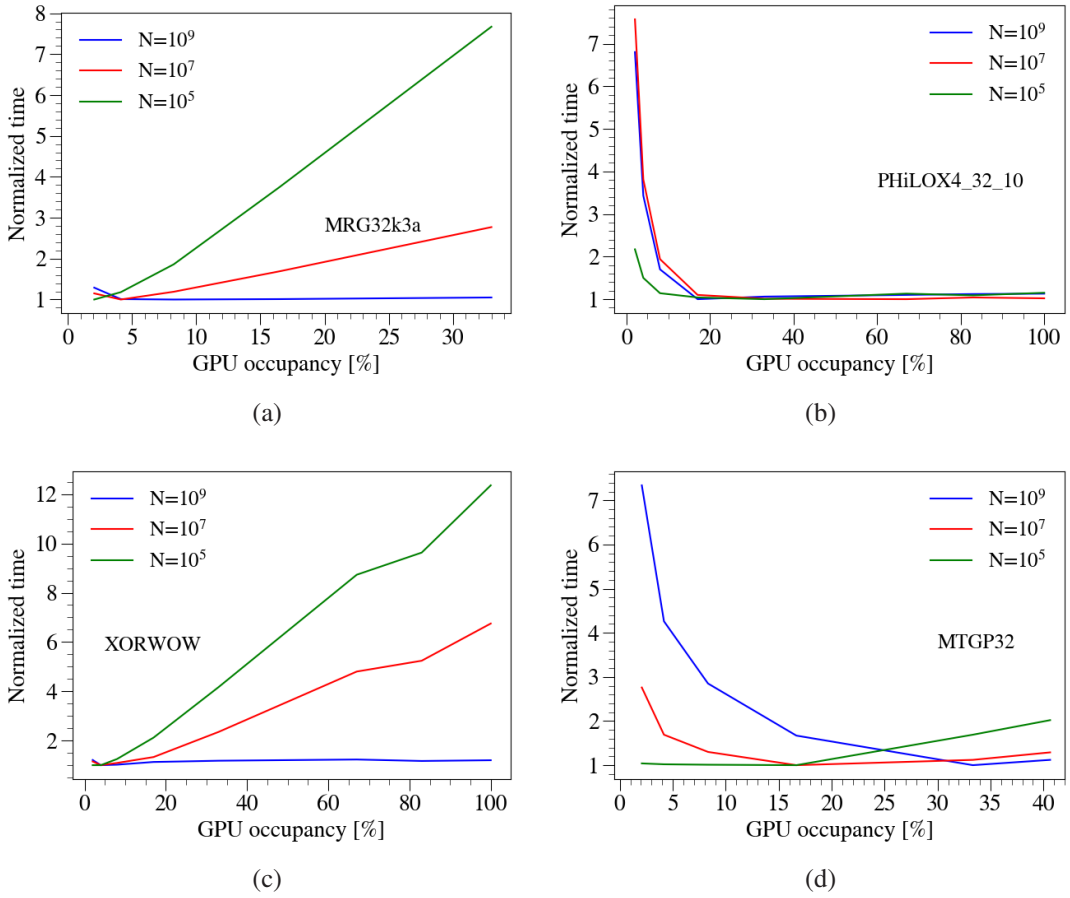


Figure 2.7: Normalized time of execution utilizing the device API approach plotted versus GPU occupancy across different PRNGs. The subfigures show results for (**a**) `MRG32k3a` PRNG, (**b**) `PHILOX4_32_10` PRNG, (**c**) `XORWOW` PRNG, and (**d**) `MTGP32` PRNG. The times are normalized based on the minimum execution time achievable for a given N on the RTX3090 GPU using the device API. Different lines represent varying numbers of generated PRNs, uniformly distributed. GPU occupancy refers to the ratio of active warps running on an SM compared to the maximum number of warps which can run concurrently.

For the other two PRNGs, the scenarios are significantly different. Due to the constrained amount of blocks and threads permitted to produce PRNs depending on its

GPU realization, the `MTGP32` PRNG, for example, never exceeds 41% occupancy [3]. Similarly, the `MRG32k3a` achieves less than 35% occupancy, primarily due to the memory-intensive modulus operation required for seed setup and state updates [59, 53]. As shown in Figure 2.8, increasing N beyond $N \simeq 10^9$ does not lead to higher occupancy for these generators.

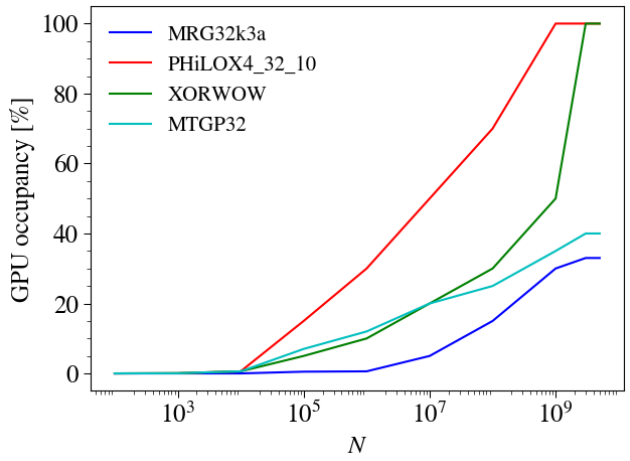


Figure 2.8: GPU occupancy for various PRNGs generating different quantities N of PRNs using the device API implementation.

2.4.5 Assessing the Performance of Various GPUs

Figure 2.9 illustrates the computation time as a function of N for different GPU cards using the default fiducial parameters. The GPUs are underutilized for small values $N \lesssim 10^4$, resulting in similar performance across all cards. However, as N increases and the occupancy of the GPU rises, the execution times begin to vary across the cards. The fastest card is the RTX3090. The RTX3080 is next, then the GTX1080Ti, and then the GTX1080. That fully aligns with the amount of computing cores in each of those cards. In addition, RTX series GPU cards are built upon the new Ampere architecture, whereas GTX cards are based on the older Pascal architecture, further contributing to such performance differences.

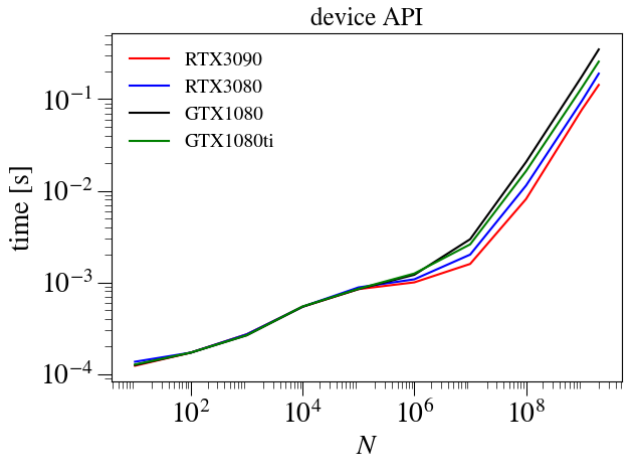


Figure 2.9: Computation time of PRN generation as a function of N across several GPUs utilizing the `MRG32k3a` PRNG and the Beta distribution function.

2.4.6 Impact of Distributions

Figure 2.10 illustrates the computation time for both uniform and non-uniform distributions in the `G_imp` method, based on the amount of "accepted" PRNs (N_a). The uniform distribution bypasses the AR technique and is the quickest to calculate. For instance, at N_a=10^6, it outperforms the Beta, Rayleigh, and Gamma distributions by factors of 1.6, 1.9, and 2.8, respectively. Due to the varying acceptance rates required to produce these distributions, non-uniform distributions have different execution times (see Section 2.3). As N increases, the time differences between distributions become more pronounced. At smaller N_a, the overall execution time is dominated by state setup and API function calls, but as N_a increases, the PRN state update and the AR technique play a larger role, amplifying these gaps. For instance, the uniform distribution outperforms the Beta, Rayleigh, and Gamma distributions by factors of 2.5, 4.3, and 10.7, respectively, for N_a=10^8.

2.5 Conclusion

In this study, we assessed the performance of various parallel pseudo-random number generators (PRNGs) on several Nvidia GPUs, including the GTX1080, GTX1080Ti, RTX3080, and RTX3090. Five PRNGs — `MRG32k3a`, `XORWOW`, `MTGP32`, `PHILOX4_32_10`, and `MT19937` — from the `cuRAND` library were evaluated. The

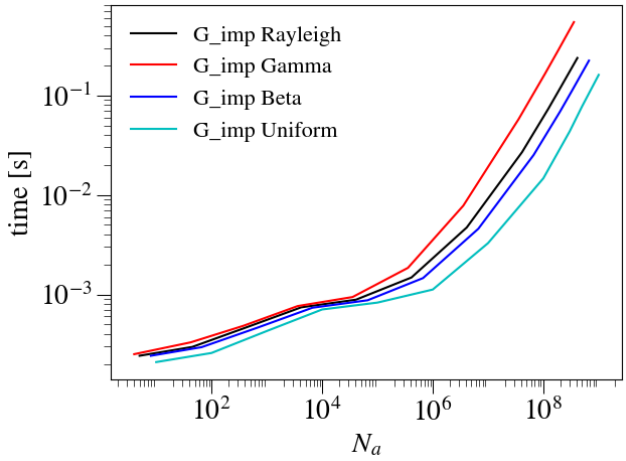


Figure 2.10: Computation time for generating random number samples with various distributions utilizing the `MRG32k3a` generator and the device API implementation.

generated random numbers, initially uniform, were transformed into non-uniform distributions through the acceptance-rejection (AR) technique. We implemented various settings to utilize the PRNGs, including a single-core CPU version (on an AMD Ryzen Threadripper 3990X), a GPU approach with and without data movement to the host memory space, and a CPU/GPU hybrid approach when random numbers are generated on the device side and transferred to the CPU for applying the AR technique. Both host and device application programming interface (API) approaches were explored for the GPU implementation. Performance was analyzed under various configurations compared to a default setup (refer to Table 2.3).

We observe that a single CPU core performs better than the GPU when the pseudo-random number (PRN) sequence length is smaller than $\sim 10^4$. GPU cores are not completely utilized in this range, whereas CPU cores operate at full capacity since individual CPU cores are inherently faster than GPU cores. A partially loaded GPU results in slower performance than a fully utilized CPU. However, as the number of PRNs exceeds $\sim 10^4$, GPU cores become more evenly loaded, improving performance. For sequences $\gtrsim 10^6$ PRNs, the GPU achieves a performance advantage of two orders of magnitude over the CPU.

To best utilize the GPU, minimizing the time spent setting up the PRNG seeds and updating the states will be important. The setup time will increase with more parallel threads while the update time decreases. Optimally, these competing factors must be

tuned against one another. The optimal number of parallel threads will depend on how many PRNs will be generated. When the number of PRNs significantly exceeds the number of GPU cores (e.g., $N \gtrsim 10^7$), maximizing the number of threads that fit into the GPU minimizes the seed setup's impact on total execution time. The optimal number of threads for smaller PRN sequences is a trade-off between seed setup and state update times, varying by PRNG. These insights will help choose performance settings for Monte Carlo radiation transport and other upcoming scientific computer applications.

BLANK

Chapter 3

Performance Analysis of Python-based CuPy and Numba Platforms in GPU-Based Monte Carlo Radiation Simulations

Chapter 3 of this PhD dissertation is based on the following paper:

Askar, Tair, Argyn Yergaliyev, Bekdaulet Shukirgaliyev, and Ernazar Abdikamalov. 2024. "Exploring Numba and CuPy for GPU-Accelerated Monte Carlo Radiation Transport." Computation 12, no. 3: 61.

3.1 Background

GPU accelerators have become an important instrument for advancing the development of numerous scientific and technical fields. Their usage goes beyond conventional high-performance computing (HPC) tasks [60, 61], in fact, they have become essential for the development and training of artificial intelligence models [62, 63]. Many of the world's leading supercomputers get most of their computational power from these accelerators, as highlighted in a recent review by Matsuoka et al. [64]. As of December 2024, nine of the top 10 supercomputers on the Top500 list utilize GPUs as accelerators to enhance their computational power [1].

Several programming platforms have emerged to harness the processing power of

GPUs from low-level and high-level abstractions. Low-level platforms can optimize code to specific accelerator architectures but are usually at the cost of reduced portability. Notable examples of such platforms include Nvidia Compute Unified Device Architecture (CUDA) [59] and AMD ROCm HIP [65]. In contrast, high-level frameworks prioritize implementation simplicity and portability across different hardware but offer fewer opportunities for hardware-specific optimizations. Examples of these higher-level paradigms include OneAPI [66], RAJA [67], Kokkos [68], Legate Numpy [69], CuPy [70, 71], Numba [72, 73], Alpaka [74], SYCL [75], OpenCL [76], OpenACC [77], and OpenMP[78]. Depending on the application's specific requirements and hardware environment, these platforms offer varying trade-offs between performance optimization and ease of use.

Numerous studies have examined the performance capabilities of GPUs across a range of applications [79, 80, 81, 82], comparing various programming platforms to assess their efficiency. These evaluations focus on both the strengths and limitations of widely used platforms, including CUDA C [83, 84, 85], CUDA Fortran [86, 87, 88], OpenCL [89, 90, 91], OpenACC [92, 93], OpenMP [94, 95], and Python-based tools such as Numba, CuPy, and Python CUDA [96, 97, 98, 99, 100, 101]. Even with the progress made, a big challenge in scientific computing remains. The goal is to find a programming approach that is easy to use, works well on different systems, and still delivers fast, efficient performance [102, 103, 104, 105].

Radiation transport (RT) is crucial in various scientific and technical disciplines, from nuclear physics to astronomy (e.g., [106, 107, 108]). It is quite complicated, seven-dimensional problem in its most comprehensive form, taking into consideration time, two-directional dimensions, one energy dimension, and three space dimensions [109]. Due to this, solving the problem has become computationally expensive. The most common approach to this challenge is through a Monte Carlo (MC) method that simulates the radiation behavior by utilizing random numbers. It models a collection of radiation particles as individual MC particles. Monte Carlo radiation transport (MCRT) methods are widely favored for their simplicity, especially in cases with complex geometries and physics. However, the main disadvantage of MCRT is its high

computing demand [106].

A variety of contemporary software packages have been developed to solve real-life radiation transport problems across domains such as medical physics, nuclear engineering, and astrophysics (e.g., [110, 111, 112]. While many legacy codes remain CPU-bound, a growing subset is adopting GPU acceleration to enhance computational efficiency. `OpenMC` [113], for instance, now includes GPU support (AMD, Nvidia, Intel), while codes like `Shift` from Oak Ridge National Laboratory have been adapted to efficiently run on GPUs. This adaptation encompasses the full continuous-energy MC transport routines, supporting both eigenvalue and fixed-source simulations, along with a significant portion of the tally capabilities present in the CPU version [114]. Other platforms, such as `Geant4` [115] and `MCNP` [116], have seen experimental GPU extensions or third-party efforts. `Geant4`, for example, has inspired projects like `AdePT` [117] and `Celeritas` [118], which accelerate electromagnetic processes using hybrid CPU-GPU models and have reported considerable speedups, while the `Opticks` [119] framework enables GPU-accelerated optical photon tracking. For `MCNP`, although its core remains CPU-based, research efforts have focused on offloading specific components, such as particle tracking and tallying, to GPUs using CUDA or OpenCL, and institutions like Los Alamos National Laboratory have explored prototyping GPU-capable transport kernels, though full GPU integration into production versions remains limited. Astrophysical codes like `RADMC-3D` [120] have also begun exploring GPU acceleration; while the main code is still CPU-bound, prototype efforts using OpenCL and custom CUDA kernels for radiative transfer tasks such as ray tracing and dust emission have shown promising, though not yet widely adopted, results.

The use of GPUs to accelerate MCRT simulations is well-established due to the parallel nature of the calculations [121, 122, 123, 124, 125, 26]. Since MCRT evolves a lot of independent MC particles, it is suitable for GPUs' many-core design, which enables parallel processing and substantial speed improvements over conventional serial computations [126, 127, 128]. Numerous studies have explored various aspects of GPU-accelerated MCRT [129, 130, 131, 132]. For instance, these works [34, 133, 39] examined random number generation on GPUs, while Bossler and Valdez [134]

compared the performance of MCRT kernels implemented in Kokkos and CUDA. Hamilton et al. [135] also evaluated history-based versus event-based MCRT algorithms. Hamilton et al. [136] and Choi et al. [137] explored parallelization techniques like domain decomposition, and Bleile et al. [138] introduced the "thin-threads" approach to reduce memory usage and enhance performance in history-based MCRT. Humphrey et al. [139] used the Titan supercomputer to scale a reverse MC ray tracing technique to 16384 GPUs. Utilizing CUDA, Silvestri and Pecnik [140] ported a reciprocal MC method for radiative heat transfer in turbulent flows, resulting in notable speedups compared to CPU-based versions. Heymann and Siebenmorgen [141] used GPU-based MCRT for modeling dust radiation transfer in active galactic nuclei. At the same time, Ramon et al. [142] used a GPU-accelerated MC code, written in CUDA with a Python interface, to model RT in ocean-atmosphere systems. Lee et al. [143] created gCMCRT, a GPU-based MCRT code written in CUDA Fortran, for 3D simulation of exoplanet atmospheres. Numerous research groups have also studied integrating machine learning techniques to reduce noise in MCRT simulations [144, 145, 146, 147].

3.2 Motivation and Related Works

In this work, we assess the performance of two popular GPU computing platforms in the Python ecosystem, Numba [73] and CuPy [71], particularly with regard to MCRT simulations. Both CuPy and Numba are high-level tools designed for ease of use (e.g., [101, 148]). Numba is a just-in-time (JIT) compiler that speeds up Python code by converting it into optimized machine code, delivering performance close to that of manually optimized C code [73]. CuPy, in contrast, offers GPU acceleration with a NumPy-like interface, making it an easy choice for Python users looking to harness GPU power [71]. It can be a preferred platform for scientific computing due to its wide support for mathematical operations and compatibility with Python modules (e.g., [149, 150]).

Numerous research has examined how well Numba and CuPy work in different applications. Di Domenico et al. [99] found that Numba achieved performance on

par with CUDA-based C++ when employing NAS parallel benchmark kernels. In contrast, Oden's comparison [101] between Numba and CUDA C highlighted Numba's slower execution by 50–85% for computational heavy tasks. Peng Xu et al. [148] examined data transfer speeds between Numba and CuPy, showing Numba's superiority in handling large data transfers, with single-precision operations being approximately 20% faster than double-precision ones. Azizi [151] used a variety of Python-based tools, including CuPy and Numba, to optimize expectation-maximization algorithms with positive outcomes. Marowka [97] evaluated Numba's performance in matrix multiplication, while Dogaru R. and Dogaru I. [152] applied it to reaction-diffusion cellular nonlinear networks. These earlier studies serve as a basis for our thorough examination of CuPy and Numba in the scope of MCRT.

This study's primary goal is to evaluate the CuPy and Numba platforms' capability for the MCRT simulations. To do this, we ran simple tests, such as generating random numbers and simulating a basic one-dimensional (1D) MCRT test case in a purely absorbing medium (see Appendix A for derivation). A comprehensive evaluation of execution times across three different GPU models, considering both single- and double-precision calculations. For benchmarking, the performance was compared against CUDA C implementations. As far as we know, no previous study has explicitly looked into using CuPy and Numba for MCRT applications. This work's novelty lies in the useful insights it provides into the performance and applicability of Python-based GPU frameworks for MCRT applications.

3.3 Experimental settings

We have compared CuPy and Numba's performance in the MCRT problem's solution by considering two idealized test cases. The first involved generating PRNs and storing them in global memory to simulate a memory-intensive case. The second test case dealt with a 1D MC radiation transport problem in a purely absorbing medium with plane-parallel geometry, emphasizing computational load with numerous arithmetic operations like logarithms, divisions, and multiplications. In this case, global memory

storage was avoided by using the generated PRNs directly within the kernel. We benchmarked our findings against a test realization in CUDA C. Remarkably, the 1D MCRT test case in CuPy only needed 15 lines of code, 26 in Numba, and 37 in CUDA C, indicating that CuPy and Numba are easier to implement than CUDA C [99].

In order to make a fair comparison, we ensured that the calculations were as similar across all three platforms as possible. The same seed and `Xorshift`-type PRNG were used across the tests: CuPy and CUDA C employed the `XORWOW` generator from the `cuRAND` library, while Numba employed its `Xoroshiro128p` generator, which is part of the `Xorshift` random number generators family. Although Numba doesn't support `cuRAND`, using similar algorithms allowed for consistent performance comparisons. Performance testing between these generators showed no significant differences, confirming that both PRNGs aligned well for a balanced assessment of random number generation efficiency across CUDA C, Numba, and CuPy. To ensure that any performance variance is only attributable to platforms, we employed the identical constants and coefficients in every test instance. To maximize the GPU's potential in Numba and CUDA C, we employed an optimal grid, block, and thread configuration. Conversely, CuPy depends on its default setup, which represents platform-specific parallel computing resource management.

Since Numba and CuPy utilize JIT compilation, functions are compiled during runtime when called. As a result, the first time a function is invoked, it includes the compilation process, which can substantially increase execution time. We excluded the timing for the initial iteration in order to prevent the distortion of performance results. We did not count the time to set up the PRNG state for CuPy and CUDA C in the GPU performance tests because Numba sets up its PRNG state on the CPU. We explicitly set the data type for single-precision computations because CuPy and Numba use double-precision for all floating-point variables and constants by default [101, 148].

We used `nvprof` and `Nsight Systems` profiling tools to evaluate performance. The specific versions of Python, CuPy, CUDA C, and Numba utilized were 3.10.6, 12.2.0, 11.8.0, and 0.58.1, respectively. To ensure accuracy, we ran every test 100 times and averaged the outcomes. Each test lasted 20 minutes and used the `nvidia-smi` tool

to track power usage. We collected the power and GPU utilization data every 5 seconds, and the average of these measurements was taken to obtain reliable power consumption insights.

We evaluated three different Nvidia GPUs: RTX3080, V100 (from a DGX-2 server), and the A100 (from a DGX-A100 server). The A100 was selected since our organization had access to this relatively new Nvidia card, which is intended for HPC and artificial intelligence. The V100, which came before the A100, was also included for comparison. The RTX3080 represents a high-performance consumer-grade GPU (see Table 3.1 for specifications). These cards provide a good representation of GPUs actively used in scientific computing. While we did not have access to the newest Nvidia H100 card or GPUs from AMD or Intel at the time of this study, our comparison of the RTX3080, V100, and A100 suggests that the differences we observed across software platforms should remain consistent, regardless of the specific GPU cards used.

Table 3.1: Key Technical Specifications of the Nvidia GPUs Utilized in This Study.

GPU Card	**Base Clock [MHz]**	**Bandwidth [GB/s]**	**FP32 (Float) [TFLOPS]**	**FP64 (Double) [TFLOPS]**	**CUDA Cores**
RTX3080	1440	760	29.77	0.465	8704
V100	1230	897	14.13	7.066	5120
A100	765	1,555	19.49	9.746	6912

Unless stated otherwise, all the results shown in Section 3.4 were collected with the A100 GPU. For comparison purposes, results from the RTX3080 and V100 GPUs are also provided in Section 3.4.3. In Section 3.4.1, single-precision and double-precision calculations are compared, although all other computations were performed in single-precision.

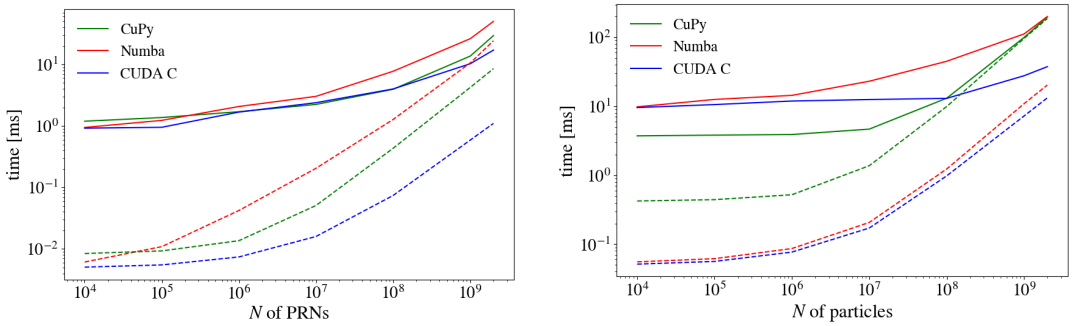
3.4 Results

A comparison of the random number generation performance of CUDA C, Numba, and CuPy is illustrated in Figure 3.1a. The number of PRNs generated is shown on the x-axis. Solid lines indicate the overall computation time, encompassing host and GPU

computations. The GPU kernel computation time, on the other hand, is represented by dashed lines and only accounts for time spent on the GPU. For $N \leq 10^6$, all three platforms displayed similar total execution times. However, for larger N, Numba's performance lagged behind CuPy and CUDA C, with execution times 1.87× and 3.22× slower at $N = 2{\times}10^9$, respectively. CuPy performed comparably to CUDA C up to $N = 10^8$ but was outperformed by CUDA C by 1.72× at $N = 2{\times}10^9$. Regarding GPU kernel performance, CUDA C surpassed Numba and CuPy even at $N = 10^4$, being 1.21× and 1.67× faster. This performance gap increased with growing N, reaching about 22× and 7.8× compared to CuPy and Numba at $N = 2{\times}10^9$. These results are consistent with other research [101, 99].

A comparison of CUDA C, Numba, and CuPy's performance in solving the 1D MCRT problem is shown in Figure 3.1b. In terms of total computation time, CuPy performed better than both CUDA C and Numba for smaller workloads ($N < 10^8$) (solid lines). At $N = 10^6$, CuPy became 3.06 times and 4.72 times faster than CUDA C and Numba, respectively. However, as the problem grew beyond $N > 10^8$, CUDA C became the better option. For $N = 2{\times}10^9$, CUDA C surpassed Numba and CuPy by 5.78× and 5.24×, respectively. When focusing on the GPU kernel performance alone (dashed lines), CuPy lagged behind both CUDA C and Numba. For $N = 10^4$, CuPy was already 8.5× slower compared to CUDA C, with this gap increasing to 14.2× at $N = 2 \times 10^9$. In contrast, Numba showed competitive performance with CUDA C, with only a 1.53× slowdown at $N = 2 \times 10^9$. This discrepancy was primarily attributed to 2 factors: (1) the difference in random number generators utilized (see Section 3.3), and (2) the use of double-precision for the logarithm function (see Section 3.4.3). These differences reflect both algorithmic structure, as the choice of PRNG affects computational and memory behavior, and software/API limitations, as Numba's handling of transcendental functions offers less control. Notably, when the same PRNG was used across platforms, the performance of Numba closely matched that of CUDA C for this test case.

The similar performance of CUDA C and Numba in the 1D MCRT application, contrasted with their significant differences in the random number generation task, highlights the underlying cause of this behavior. In the PRN generation problem, where



(a) (b)

Figure 3.1: Computation time as a function of the number of PRNs/particles for CUDA, Numba, and CuPy on the Nvidia A100 GPU. The left figure displays results for the random number generation benchmark, whereas the right figure focuses on the 1D MCRT problem. The GPU kernel times are shown by dashed lines, while the overall computation time is shown by solid lines.

Numba performed poorly, random number samples were stored in GPU global memory space, causing substantial performance overhead due to data movement. However, in the 1D MCRT test case, random numbers are generated and utilized locally without much data transfer, allowing Numba to perform more efficiently. This implies that Numba is slower for large-scale data transfers and faster for small-scale data movement [101]. This indicates that the observed performance differences stem from memory access and API limitations.

The faster execution times observed for smaller workloads ($N \leq 10^8$) in the 1D MCRT problem using CuPy can be referred to its efficient memory management. CuPy reduces the cost of repetitive memory allocation and CPU/GPU synchronization by "caching" previously allocated memory blocks [70]. This is particularly beneficial in the 1D MCRT test case, where frequent allocations of memory blocks occur, allowing CuPy to outperform both Numba and CUDA C in terms of overall performance for smaller problem sizes $N \leq 10^8$. However, in terms of GPU kernel execution, CuPy was slower because intermediate computed values were temporarily stored in global memory after each calculation, which added some latency. Overall, CuPy's performance is influenced by a combination of algorithm structure, hardware architecture, and limitations in APIs and compilers.

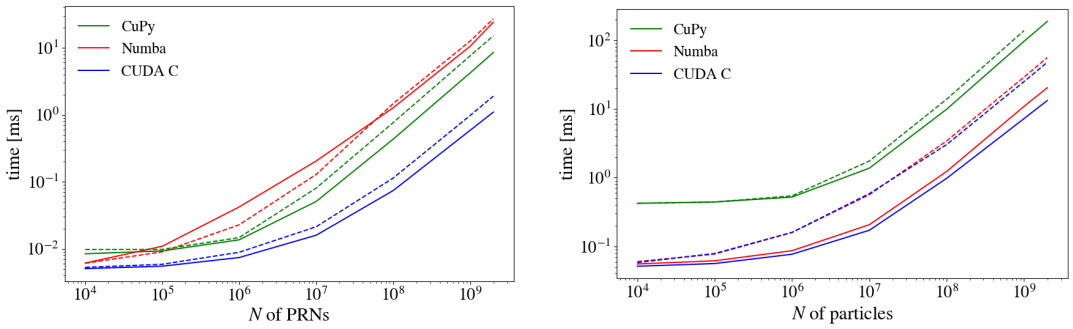
3.4.1 Effects of Precision on Performance

This section performs calculations in both single-precision and double-precision formats to investigate the impact of precision on performance. See [148] for a comparable analysis used with the finite-difference method for Burgers' equation. Additionally, comprehensive performance evaluations of CUDA C and Numba for tasks like 3D stencil operations, parallel reduction, and matrix-matrix multiplication in both precision levels can be found in [101].

Figure 3.2a illustrates the computation time of GPU kernels for random number generation using CUDA C, Numba, and CuPy, comparing single- and double-precision computations as a function of N. At $N = 10^6$, the single-precision execution of CUDA C outperformed its double-precision version by 1.17×, as expected. As N increased, this differenece grew to 1.77× at $N = 2 \times 10^9$. These outcomes align with prior research on the influence of precision in CUDA C [153, 154]. CuPy showed a similar performance trend, which is likely expected since it's PRN generation depends on the CUDA-provided cuRAND library [70, 50].

For Numba, single-precision computations were slower for $N \leq 6 \times 10^7$ but became $\sim$12.5% faster for larger N. The cause of this behavior is two competing factors. First, as confirmed by PTX assembly code inspection and profiling using the `Nsight Systems` tool, Numba generates random numbers in double-precision by default. Converting these values to single-precision incurs additional overhead. However, single-precision data is quicker to store in global memory than double-precision, allowing single-precision calculations to outperform double-precision for larger N.

The GPU kernel execution time for the single- and double-precision 1D MCRT application as a function of N is shown in Figure 3.2b. For CuPy, similar performance was observed for both precisions up to $N = 10^6$. Beyond this, single-precision performance becomes faster, reaching a maximum value of 1.43× at $N = 10^9$. This difference can be described by the significant data transfers to and from global memory space for temporary storage following every computation.



(a) (b)

Figure 3.2: GPU kernel execution times as a function of the number of random numbers (particles) for CUDA C, Numba, and CuPy on the Nvidia A100 GPU. The left figure illustrates the results of the random number generation application, whereas the right figure shows the 1D MCRT application. Solid lines represent single-precision computations, and dashed lines indicate double-precision outcomes.

For Numba, double-precision calculations were always slower than those using single precision for all the tested values of N. Numba internally converts double-precision PRNs to single-precision but does so more efficiently than CuPy by avoiding the need to temporarily store data in global memory between calculations. This approach enhances performance. For instance, at $N = 10^4$, the speed difference between the two precisions was a factor of 1.08, whereas, at $N = 2\times\ 10^9$, the gap increased to 2.74×.

3.4.2 Energy Usage Analysis

An overview of the power usage and GPU utilization characteristics, as well as the average energy consumption to generate a single random number (or track a single particle in the 1D MCRT test case), is given in Table 3.2. We record values for $N = 10^9$ using the A100 GPU. According to the results, Numba uses less energy than CuPy. But, CUDA C outperforms both of them. Specifically, CUDA C requires 2.1 and 2.4 times less energy per random number than CuPy and Numba, respectively. For the 1D MCRT simulation, CUDA C consumes 5.1 times less energy than CuPy and 3.7 times less than Numba. CUDA C's energy efficiency is attributed to its fine-tuned optimizations for the GPU architecture, maximizing parallel computing efficiency and minimizing idle time, resulting in higher computational output per unit of energy. In contrast, CuPy and Numba, being Python-based JIT compilation frameworks, introduce overheads and

inefficiencies that impact their energy performance relative to CUDA C.

Table 3.2: Power usage, GPU utilization, and energy consumption. These parameters were measured by utilizing the A100 GPU to track 10^9 particles for the 1D MCRT problem and to generate 10^9 PRNs for the random number generation problem. Details on the measurement process can be found in Section 3.3.

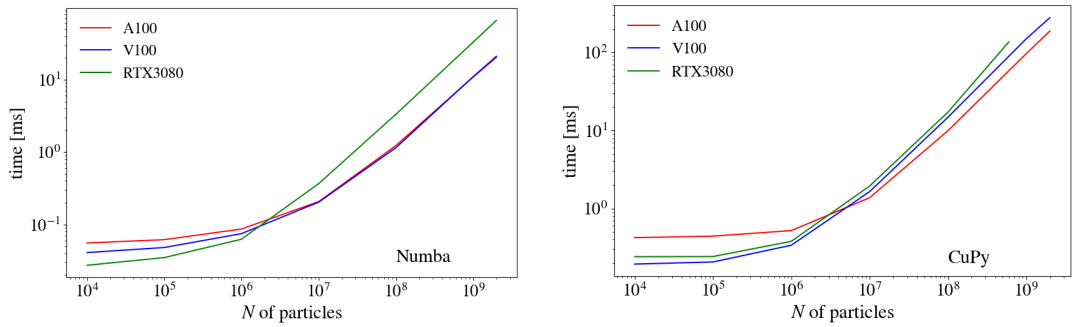
	PRN Generation			**1D MCRT**		
	Energy per PRN [nanojoule]	**Power [Watt]**	**Util [%]**	**Energy per Particle [nanojoule]**	**Power [Watt]**	**Util [%]**
CUDA C	1.17	76	100	4.34	125	46
Numba	2.46	81	100	15.9	68	10
CuPy	2.82	206	100	22.1	221	100

Energy consumption in the PRN generation test shows that CUDA C uses 7% less power than Numba and 171% less than CuPy. However, for the 1D MCRT test case (as shown in Table 3.2), Numba consumes 225% and 84% less power than CuPy and CUDA C, respectively. This is because Numba leverages the GPU less intensively compared to CUDA C. In Numba's case, the random number generator's state setup occurs on the host side, which is then moved to the GPU, introducing delays that reduce GPU utilization. The excessive data transfer brought on by CuPy's default memory management [70] is the reason for its high power consumption [155]. These differences in power consumption between CUDA C, Numba, and CuPy involve some important considerations for practical applications. Focusing on GPU parallelism and optimization, CUDA C generally provides better computation and energy efficiency performance. It would be ideal in applications where time-sensitive outcomes and power constraints are required [156, 157]. While Numba and CuPy simplify GPU implementation, they represent overheads that affect energy efficiency, scalability, and performance.

3.4.3 Benchmarking Selected Nvidia GPUs

This subsection evaluates the performance of the 3 GPUs by applying them to a 1D MCRT application. For a comparable analysis of PRN generation in CUDA C, refer to Chapter 2 and the study by Askar et al. [158] on GPU performance evaluation.

In Figure 3.3a, the computation time for Numba is represented. The Numba-generated PTX assembly code shows that the `log` function is run in double precision even though all variables and constants are explicitly defined as single-precision. This leads to performance penalties, especially on the RTX3080 GPU card, which processes double-precision tasks up to 64× slower than single-precision operations. On the other hand, the V100 and A100 GPUs have a much smaller performance gap, with double-precision operations being only twice as slow as single-precision (refer to Table 3.1 for performance metrics of GPU cards for each precision type). As a result, Numba performs significantly slower on the RTX3080, with a performance difference of up to 6.7× when processing $N = 2 \times 10^9$ particles compared to the A100 and V100.



(a) (b)

Figure 3.3: GPU kernel computation time as a function of number of particles for the 1D MCRT test problem, comparing CuPy and Numba across 3 GPU cards. The left figure illustrates the performance of Numba, while the right figure shows the results for CuPy.

The Figure 3.3b presents CuPy's performance results. For $N < 3{\times}10^6$, CuPy ran faster on the V100 and RTX3080 compared to the A100. Though, for $N \geq 3{\times}10^6$, the A100 outperforms both the RTX3080 and V100, achieving speed-ups of 1.72× and 1.49× at $N = 10^8$, respectively. The reason for the RTX3080's slower performance for $N \geq 3{\times}10^6$ is due to its memory bandwidth is 18% and 105% less than that of the V100 and A100, respectively. The A100's faster performance compared to the V100 for $N \geq 3{\times}10^6$ is also due to its 73% higher memory bandwidth, as highlighted in Table 3.1. In this case, a GPU card with higher memory bandwidth is an important factor for CuPy's overall efficiency.

Table 3.3 compares the power consumption of 3 GPUs during random number

generation and 1D MCRT problem cases. The A100 GPU consumes the least power, followed by the V100, with the RTX3080 using the most. For example, the A100 GPU card uses 26.8% less power than the V100 and 64.7% less power than the RTX3080 GPU cards while using CUDA C. Numba is 105.2% more efficient than the RTX3080 and 52% more efficient than the V100 while operating on the A100. Similarly, CuPy, on the A100 consumes 3% less power than on the V100 and 38.3% less than on the RTX3080.

Table 3.3: Power usage for tracking 10^9 particles in the 1D MCRT task and generating 10^9 samples in the PRN generation task across RTX3080, V100, and A100 GPU cards.

	PRN Generation [Watt]			**1D MCRT [Watt]**		
	A100	**V100**	**RTX3080**	**A100**	**V100**	**RTX3080**
CUDA C	76	85	129	141	200	225
Numba	81	145	174	68	85	133
CuPy	206	211	293	221	229	297

3.5 Discussion

The performance of GPU programming tools — more especially, Python-based frameworks such as CuPy and Numba — is assessed in this study using MCRT simulations. We illustrate the trade-offs between computational efficiency and implementation simplicity for each platform by examining two different problem sets and comparing the outcomes with a CUDA C solution.

Our results show that performance varies depending on the workload type, including compute- and memory-intensive workloads. We also do computations in single and double precision to evaluate the effect of numerical precision (for more information, see Subsection 3.4.1). We benchmark these frameworks across several NVIDIA GPU architectures, analyzing their effectiveness under various setups to obtain a deeper knowledge of hardware-dependent performance (see Section 3.4.3).

Although our study offers valuable insights, it should be noted that it has a number of limitations. Factors such as memory access patterns, optimization constraints, and

framework-specific overheads influence performance. Future research will explore additional optimizations, alternative GPU architectures (e.g. AMD, Intel), and their impact on large-scale simulations involving higher-dimensional and more complex geometries.

In addition to performance analysis, our findings provide useful recommendations for choosing GPU programming frameworks based on particular computing needs. This paper helps researchers in maximizing performance for MCRT simulations and other related computational tasks by outlining the advantages and disadvantages of CuPy and Numba.

3.6 Conclusion

In this study, we evaluated the performance of 2 Python-based GPU programming frameworks, Numba and CuPy, in the scope of Monte Carlo radiative transfer (MCRT) simulations. Two problems were used: one focused on generating and storing random numbers in global memory (a memory-intensive task), and the other involved a one-dimensional (1D) MCRT simulation in a fully absorbing medium. For the purpose of comparison, we benchmarked all tests against CUDA C on three different GPUs: the GeForce RTX 3080, the NVIDIA Tesla A100, and the Tesla V100.

The outcomes present that CUDA C provided the better energy efficiency and fastest execution time. Numba's performance was comparable to that of CUDA C when there was minimal data movement, but it drastically decreased when large data transfers were required (such as storing random numbers in global memory). CuPy showed better memory management than Numba, making it faster for random number generation. However, CuPy's performance was slower in compute-intensive tasks, with significantly lower performance in the 1D MCRT test problem and higher energy consumption, resulting in lower efficiency. CuPy's code implementation process was simpler, which can reduce the complexity of development despite its slower performance.

These findings outline the performance characteristics of each framework, highlighting the strengths and weaknesses of each platform. It provides useful insights

for selecting the most appropriate programming platform for specific computational requirements.

Chapter 4

Performance Evaluation of Numba and CuPy in Multi-GPU Environments

This chapter of the PhD thesis is based on the following paper submitted to the Journal of Cluster Computing and is currently under peer review:

Tair Askar, Martin Lukac, Bekdaulet Shukirgaliyev, and Ernazar Abdikamalov. 2024. "Evaluating Multi-GPU Computing Capabilities of Numba and CuPy." Cluster Computing, Springer (accepted).

4.1 Introduction

GPU computing has been an important part of scientific computing in recent years, driving advancements in numerous areas (e.g., [159, 160, 161, 162, 163, 164, 165]). Single-GPU systems are frequently falling short in terms of performance as computational needs are rising. Multi-GPU systems present a more powerful alternative by distributing workloads across several GPUs. However, the performance improvements in such systems largely depend on the efficient utilization of GPU resources [166, 167, 168]. Modern applications require scalability, especially those dealing with big dataset processing or running complex simulations. Nvidia developed the DGX servers [169] in direct response to these demands by using multiple GPUs on a single node, explicitly optimized for improved scalability and computational performance.

The Python programming language has gained widespread popularity [170, 171] because of its user-friendliness and the abundance of scientific libraries it offers, such as NumPy [172] and SciPy [173], which simplify complex calculations without the need for in-depth knowledge of low-level programming. Due to Python's high-level abstraction, this often comes at the cost of performance. Libraries like Numba [72] and CuPy [70] have been developed to address this issue. Numba is a just-in-time (JIT) compiler that converts Python code into optimized machine code, which enables near-native performance on the GPU. CuPy, on the other hand, is a GPU-accelerated library similar to NumPy, allowing Python users to harness GPU power for array-based computations. While CUDA C offers more granular control and enhanced performance by directly interfacing with the GPU, it requires more complex programming, which makes libraries like Numba and CuPy more appealing for many researchers seeking a balance between ease of use and performance [174, 175].

4.2 Literature Review

Numerous studies have investigated using Numba and CuPy across various applications, utilizing single- and multi-GPU setups. Di Domenico et al. [176] demonstrate that employing Numba for GPU-based applications in Python delivers performance on par with C++ using CUDA and outperforms C++ with OpenACC. Additionally, Numba requires fewer GPU management tasks than CUDA, though slightly more than OpenACC. Rao et al. [177] compared Numba and C for analyzing historical Phasor Measurement Unit data. They found that Numba substantially improves performance compared to standard Python and makes coding easier. However, even with these improvements, Numba still performs slower than C on the NVIDIA Jetson Xavier GPU. Villalobos and Meneses [178] conducted benchmarking of Numba and CuPy, along with other GPU-accelerated libraries, on an NVIDIA V100 GPU. Their findings showed that Numba outperformed CuPy in Monte Carlo (MC) simulations, particle interactions, and stencil computations, though CuPy provided a simpler coding experience. Xu et al. [148] report that Numba surpasses CuPy in performance when the grid size exceeds

10^7, and using single-precision in Numba improves computation time by 20% compared to double-precision. Oden's [101] benchmarking of Numba against CUDA C reveals that Numba performs 50–85% slower in computationally intensive tasks. Kriebel et al. [179] demonstrate that utilizing CuPy for tensor contraction tasks on a single V100 GPU achieves a speedup of 10 to 16 times compared to performing the same computations with NumPy on 36 CPU cores. Guerrero-Hurtado et al. [180] demonstrate that employing Numba with CUDA for fluid-structure interaction simulations on a single GPU achieves a speedup of 34 to 54 times compared to a CPU-based solver utilizing 96 to 128 cores. Almgren-Bell et al. [150] enhance the performance of particle-in-cell codes through the use of CuPy and Numba, achieving a processing speed of 1.4 nanoseconds per particle per time step on a single GPU, with scalability reaching up to 16 GPUs. Pata et al. [181] demonstrate that leveraging Numba to offload computations to multiple GPUs can accelerate high-energy physics data analysis, achieving speeds up to 10 times faster than multi-threaded CPUs.

4.3 Motivation

Although Numba and CuPy are becoming more popular for multi-GPU computing, there are few detailed studies on how well they perform and scale in multi-GPU setups. It's important to fill this gap to better understand how these tools handle large workloads in such environments. This work extensively analyzes multi-GPU in Numba and CuPy compared to CUDA C, focusing on performance, scalability, and energy efficiency for various scenarios. The research looks at both weak and strong scaling scenarios and explores optimizations like `fast math` and `block-thread` adjustments. The outputs provide useful insights into the strengths and weaknesses of these Python-based frameworks in real-world applications.

4.4 Study Design and Methods

We assess the performance of Numba and CuPy in tackling the Monte Carlo radiative transfer (MCRT) application in a multi-GPU environment, comparing these results with a CUDA C implementation. As discussed in Chapter 3, the same two idealized test cases are used for this evaluation. In the first case, a scenario with high memory utilization is simulated by generating and storing pseudo-random numbers (PRNs) in global memory. The second case involves solving a one-dimensional (1D) MC radiation transfer test case within a fully absorbing medium (see Appendix A for a derivation), emphasizing a computationally intensive workload that includes numerous arithmetic operations (e.g., division, logarithms). Our tests cover both strong scaling (constant problem size with increasing GPU units) and weak scaling (problem size grows with the number of GPUs) while also analyzing the effects of optimizations such as `fast math` and `block-thread` configuration. Additionally, energy consumption is evaluated. We implement two approaches: a multi-CPU thread model, in which an individual CPU thread controls one GPU, and a single-CPU thread model, in which a single thread controls all GPUs.

To ensure a fair comparison, we standardized the computational approach across CuPy, Numba, and CUDA C computing platforms. We used the same pseudo-random number generator (PRNG) type with an identical seed. Specifically, we employed `cuRAND`'s `XORWOW` PRNG [50] for both CuPy and CUDA C, while Numba used `Xoroshiro128p` from the same `Xorshift` family [52, 182]. No significant performance differences were observed between these PRNG algorithms [183]. Furthermore, we kept all problem settings the same for each test case (e.g., absorption coefficient, position) and used the same block and thread configurations. For the multi-CPU thread implementation, we used `Pthreads` with CUDA C and Python's `threading` module for CuPy and Numba, as they are comparable in managing CPU threads on GPUs. This ensures that performance differences are due to the platform implementations, not the PRNG or setup variations.

Since both Numba and CuPy rely on JIT compilation, we omit the timing for the

first function execution to exclude the overhead of the initial compilation from our performance evaluations. Furthermore, to maintain consistency, we ensure that all floating-point variables and constants are explicitly defined as single-precision across all platforms.

We conducted our experiments on the Nvidia DGX-2 system, which has 16 NVIDIA V100 GPUs (refer to Table 4.1 for key specifications). Even with continuous improvements in GPU technology, the DGX-2 continues to be widely utilized for high-performance computing (HPC) applications. We used profiling tools such as `nvprof` and `Nsight Systems` [184] for performance evaluation. The software versions include CuPy 12.3.0, Numba 0.58.1, CUDA C 11.4.0, and Python 3.8.10. We ran each method 100 times and recorded the average value to make sure the outcomes were reliable. To monitor power consumption, we utilized `nvidia-smi` instrument [185]. We conduct every test for 20 minutes and record consumption every second. Then we calculated the average value.

Table 4.1: NVIDIA DGX-2 key specifications.

CPUs	Dual Intel Xeon Platinum 8168, 2.7 GHz, 24-cores
GPUs	16X NVIDIA Tesla V100 32GB HBM2
System Memory	1.5TB DDR4
GPU Memory	512GB total HBM2 (16 $\times$ 32 GB)
Storage	OS: 2X 960GB SSDs; Internal Storage: 30TB (8X 3.84TB) SSDs
Network	8X 100Gb/sec Infiniband/100GigE Dual 10/25Gb/sec Ethernet
Maximum Power Usage	10 kW
Performance	2 petaFLOPS
NVIDIA CUDA Cores	81920

4.5 Topology of Multi-GPU systems

Performance optimization of multi-GPU systems requires an understanding of their topology. The efficiency of data transfers and computational performance are substantially affected by the way in which GPUs are connected to the CPUs and to one another, typically via PCIe lanes or high-speed interconnects such as

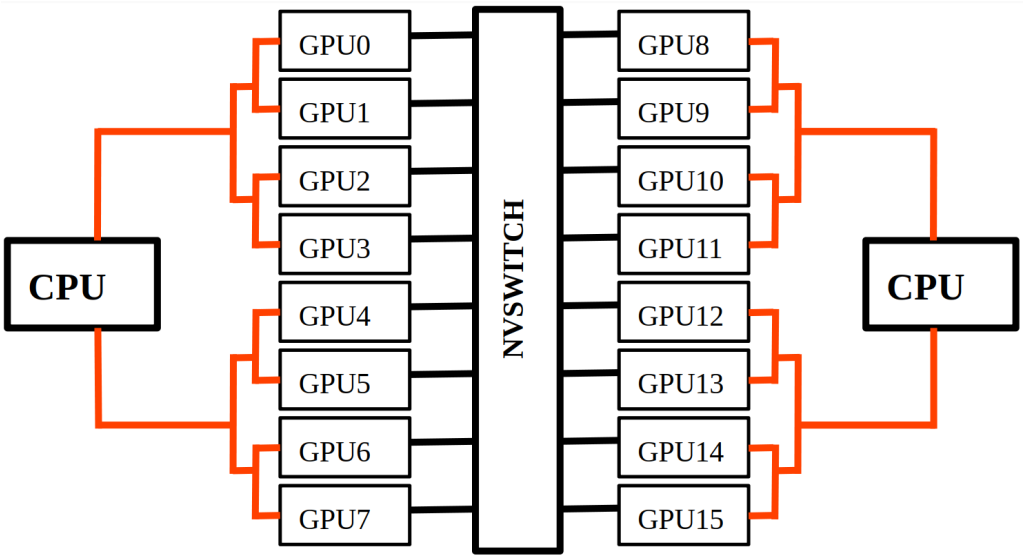


Figure 4.1: Simplified topology of Nvidia DGX-2 server. Red lines correspond to the PCIe lanes, while black lines to the NVlink.

NVLink. Understanding this design allows users to allocate resources for their specific applications in an efficient manner.

The connection architecture is complex in systems such as the Nvidia DGX-2, which has 16 GPUs. While NVLink facilitates fast GPU-to-GPU communication, CPU-to-GPU communication primarily occurs over PCIe lanes. Figure 4.1shows the general topology of the DGX-2 server where these lanes are grouped, with several GPUs sharing the same PCIe connection to the CPU. If a user is unaware of how these GPUs are grouped, they may accidentally allocate multiple GPUs from the same group, forcing those GPUs to share a single PCIe lane. This can lead to bandwidth sharing, causing performance bottlenecks for applications where data movement between CPU and GPU is heavy.

For instance, if a user needs four GPUs for their application and doesn't indicate which GPUs to utilize, the system may automatically choose GPUs 0–3. Since these GPUs share a PCIe lane in a DGX-2 system, the available bandwidth must be divided between them. This shared bandwidth could limit performance, especially for applications that require frequent data transfers between the CPU and the GPUs.

However, if the user explicitly selects GPUs from different PCIe groups—such as GPUs 0, 4, 8, and 12 (see Figure 4.1 for reference)—each GPU would have its own

dedicated PCIe lane for communication with the CPU. This configuration maximizes the available PCIe bandwidth, allowing each GPU to utilize its full potential. In this case, the performance could be up to four times better than using GPUs from the same PCIe group, as each GPU would benefit from independent, non-competing data transfers.

Understanding and leveraging the topology of a multi-GPU system can lead to significant performance improvements, particularly in bandwidth-intensive applications. For optimal performance, especially in CPU-to-GPU or GPU-to-CPU intensive tasks, selecting GPUs from different PCIe domains is advisable to ensure efficient use of the available bandwidth.

4.6 Results

4.6.1 Strong Scalability Assessment

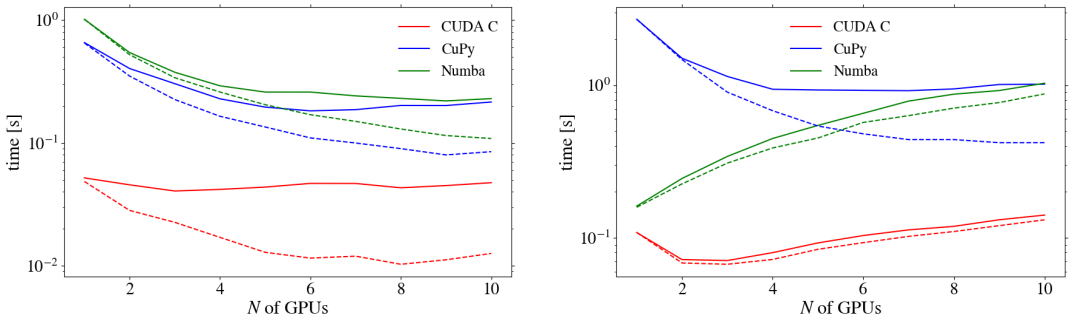


Figure 4.2: Strong scaling performance measured as execution time versus the number of GPUs for CUDA C, CuPy, and Numba (lower values indicate better performance). Solid lines represent single-CPU control results, while dashed lines show results for multi-CPU control implementations. The left panel displays results for the random number generation, and the right panel shows the 1D MCRT results.

The experiments involving CUDA C, CuPy, and Numba for testing PRN generation will be labeled as $\mathcal{R}_{\mathrm{CUDA}}$, $\mathcal{R}_{\mathrm{CuPy}}$, and $\mathcal{R}_{\mathrm{Numba}}$, respectively. Similarly, for the 1D MCRT application, the experiments will be named $\mathcal{C}_{\mathrm{CUDA}}$, $\mathcal{C}_{\mathrm{CuPy}}$, and $\mathcal{C}_{\mathrm{Numba}}$. Each experiment will further be annotated with a superscript $^{\mathrm{S}}$ or $^{\mathrm{M}}$ to indicate whether a single-CPU or multi-CPU thread version is used. To simplify, we will refer to

all experiments as $\mathcal{R}^{\mathrm{S}}$, $\mathcal{R}^{\mathrm{M}}$ (for PRN generation) and $\mathcal{C}^{\mathrm{S}}$, $\mathcal{C}^{\mathrm{M}}$ (for the 1D MCRT application), depending on the CPU configuration. For example, experiments using single- and multi-CPU threads in CUDA C for the 1D MCRT application will be denoted as $\mathcal{C}_{\mathrm{CUDA}}$.

Figure 4.2a illustrates the execution time required to generate 8×10^9 PRNs using CUDA C, CuPy, and Numba as a function of the number of GPUs. The solid lines represent the single-threaded $\mathcal{R}^{\mathrm{S}}$ implementation, while the dashed lines shows the multi-threaded $\mathcal{R}^{\mathrm{M}}$ implementation. For the $\mathcal{R}^{\mathrm{S}}_{\mathrm{CUDA}}$ implementation, the execution time remains constant regardless of the number of GPUs utilized. This happens because of the combined effect of GPU computation time and application programming interface (API) calls. The calculation on one GPU overlaps with the API calls for another due to the CPU thread issues API commands sequentially, which causes a performance plateau across multiple GPUs.

The execution times for both $\mathcal{R}^{\mathrm{S}}_{\mathrm{CuPy}}$ and $\mathcal{R}^{\mathrm{S}}_{\mathrm{Numba}}$ improve with the addition of up to six GPUs. Profiling with `Nsight Systems` reveals that in $\mathcal{R}^{\mathrm{S}}_{\mathrm{CuPy}}$, around 80% of API calls time, is not overlapped with GPU computation. Still, this non-overlapping time decreases as more GPUs are utilized, leading to better performance. However, once the number of GPUs exceeds six, the benefit of adding more GPUs is offset by the overhead from kernel launches, causing execution time to stabilize. For $\mathcal{R}^{\mathrm{S}}_{\mathrm{Numba}}$, GPU computation time surpasses API calls time by 110%. As the number of GPUs increases, the execution time drops due to efficient overlapping of computations across multiple GPUs. Nevertheless, similar to $\mathcal{R}^{\mathrm{S}}_{\mathrm{CuPy}}$, performance gains level off beyond six GPUs because kernel launch overheads counterbalance further improvements.

The $\mathcal{R}^{\mathrm{S}}_{\mathrm{CUDA}}$ implementation demonstrates better performance over $\mathcal{R}^{\mathrm{S}}_{\mathrm{CuPy}}$ and $\mathcal{R}^{\mathrm{S}}_{\mathrm{Numba}}$ across all GPU configurations. At 10 GPUs, $\mathcal{R}^{\mathrm{S}}_{\mathrm{CUDA}}$ surpasses $\mathcal{R}^{\mathrm{S}}_{\mathrm{CuPy}}$ and $\mathcal{R}^{\mathrm{S}}_{\mathrm{Numba}}$ by factors of 4.54 and 4.84, respectively. Although both $\mathcal{R}^{\mathrm{S}}_{\mathrm{CUDA}}$ and $\mathcal{R}^{\mathrm{S}}_{\mathrm{CuPy}}$ use the `cuRAND` library, $\mathcal{R}^{\mathrm{S}}_{\mathrm{CuPy}}$ suffers from inefficiency in generating PRNs [183], primarily due to the GPU remains idle during memory allocation after initializing the PRNG state. $\mathcal{R}^{\mathrm{S}}_{\mathrm{Numba}}$ has two main limitations: its PRNG state is initialized on the CPU and then transferred to the GPU, and it generates PRNs about three times slower than

$\mathcal{R}^{\mathrm{S}}_{\mathrm{CUDA}}$, largely because it initially generates them in double precision by default and then converts them to single precision [183].

In an $\mathcal{R}^{\mathrm{M}}$ setup, execution time decreases across all tested computing platforms as the number of GPUs increases. This happens because each CPU thread independently manages its assigned GPU, enabling parallel computations across multiple GPUs. In these experiments, $\mathcal{R}^{\mathrm{M}}_{\mathrm{CUDA}}$ demonstrates a performance advantage over both $\mathcal{R}^{\mathrm{M}}_{\mathrm{CuPy}}$ and $\mathcal{R}^{\mathrm{M}}_{\mathrm{Numba}}$, outperforming them by factors of 6.74 and 8.62, respectively, when using a configuration of ten GPUs.

Figure 4.2b illustrates the execution times for the $\mathcal{C}_{\mathrm{CUDA}}$, $\mathcal{C}_{\mathrm{CuPy}}$, and $\mathcal{C}_{\mathrm{Numba}}$ experiments using different numbers of GPU cards. As anticipated, $\mathcal{C}_{\mathrm{CUDA}}$ consistently outperforms both $\mathcal{C}_{\mathrm{CuPy}}$ and $\mathcal{C}_{\mathrm{Numba}}$, regardless of the number of GPUs used, in both the $\mathcal{C}^{\mathrm{S}}$ and $\mathcal{C}^{\mathrm{M}}$ implementations. In the $\mathcal{C}^{\mathrm{S}}$ setup, $\mathcal{C}^{\mathrm{S}}_{\mathrm{CUDA}}$ is 7.2 times faster than $\mathcal{C}^{\mathrm{S}}_{\mathrm{CuPy}}$ and 7.31 times faster than $\mathcal{C}^{\mathrm{S}}_{\mathrm{Numba}}$ when utilizing 10 GPUs. In the $\mathcal{C}^{\mathrm{M}}$ configuration, $\mathcal{C}^{\mathrm{M}}_{\mathrm{CUDA}}$ is 3.2 times faster than $\mathcal{C}^{\mathrm{M}}_{\mathrm{CuPy}}$ and 6.68 times faster than $\mathcal{C}^{\mathrm{M}}_{\mathrm{Numba}}$. The performance of $\mathcal{C}_{\mathrm{CUDA}}$ improves up to the three GPUs, but starting from the fourth GPU, there is a 20% drop in performance, with the decline continuing to around 10% at 10 GPUs. The MCRT problem involves several time steps and frequent API invocations, which results in increasing API call overhead that grows with the number of GPUs. This is the main cause of this delay. In contrast, $\mathcal{C}_{\mathrm{CuPy}}$ observes performance improvements up to four GPUs in the $\mathcal{C}^{\mathrm{S}}$ mode and up to eight GPUs in the $\mathcal{C}^{\mathrm{M}}$ mode, after which the execution time stabilizes. This stabilization can be explained by two factors: 1) caching of memory blocks reduces memory allocation time and synchronization overhead, and 2) higher memory bandwidth reduces data transfer time [183, 70]. $\mathcal{C}_{\mathrm{Numba}}$, unlike the other implementations, experiences a drop in performance as the number of GPUs increases. This is largely due to the CPU-based initialization of the PRNG state, which causes the GPU to idle during this process. Since the PRNG state is initialized on the host, the use of multiple CPU threads is constrained by the `Global Interpreter Lock (GIL)`. Given that the MCRT test problem involves many time steps, repeated PRNG state initialization results in further performance degradation, which becomes more pronounced as additional GPUs are used.

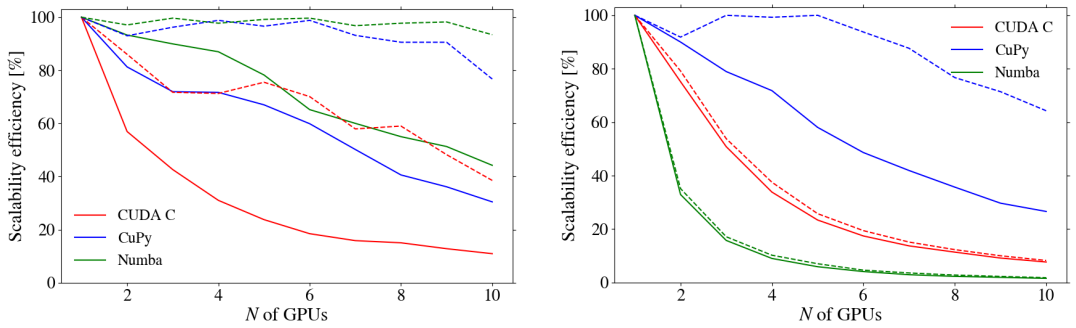


Figure 4.3: Strong scalability efficiency comparison for CUDA C, CuPy, and Numba implementations (higher values indicate better scalability). The left panel displays the results of the random number generation problem, whereas the right panel focuses on the 1D MCRT application. A perfect scalability score is 100%, with values between 99% and 51% indicating acceptable scalability, and values below 51% considered poor scalability.

Scalability efficiency is expressed as a percentage and serves as a metric to assess performance improvement when using multiple GPUs. It quantifies how well the addition of more GPUs boosts computational efficiency. The formula for scalability efficiency is:

$$\text{Scalability Efficiency (\%)} = \left(\frac{\text{Speedup}}{\text{Number of GPUs}}\right) \times 100\% \tag{1}$$

where the **speedup** is defined as:

$$\text{Speedup} = \frac{\text{Execution Time with 1 GPU}}{\text{Execution Time with N GPUs}} \tag{2}$$

A scalability efficiency of 100% represents ideal scalability, where performance increases linearly with the addition of GPUs. Efficiency values between 99% and 51% indicate reasonable scalability, meaning performance is improving, though at a reduced rate compared to perfect scaling. On the other hand, values below 51% indicate poor scalability, meaning that adding more GPUs has little impact on performance because of issues like communication delays or more frequent API calls.

Figure 4.3 illustrates the scalability efficiency of each platform when using multiple GPUs for PRN generation (Figure 4.3a) and for the 1D MCRT test case (Figure 4.3b) under strong scaling conditions. In Figure 4.3a, $\mathcal{R}_{\text{Numba}}$ demonstrates the highest efficiency across all platforms. With 10 GPUs, $\mathcal{R}_{\text{Numba}}$ achieves 93.4% efficiency in

$\mathcal{R}^{\mathrm{M}}$ mode and 44.23% efficiency in $\mathcal{R}^{\mathrm{S}}$ mode. Compared to $\mathcal{R}_{\mathrm{CUDA}}$, $\mathcal{R}_{\mathrm{Numba}}$ shows a $\sim 2.42\times$ higher efficiency in $\mathcal{R}^{\mathrm{M}}$ mode and a $\sim 4.04\times$ improvement in $\mathcal{R}^{\mathrm{S}}$ mode. Likewise, when compared to $\mathcal{R}_{\mathrm{CuPy}}$, $\mathcal{R}_{\mathrm{Numba}}$ outperforms by $\sim 1.22\times$ in $\mathcal{R}^{\mathrm{M}}$ mode and $\sim 1.45\times$ in $\mathcal{R}^{\mathrm{S}}$ mode.

In Figure 4.3b, the 1D MCRT test demonstrates that $\mathcal{C}_{\mathrm{CuPy}}$ exhibits higher scalability efficiency in both implementation modes. In the $\mathcal{C}^{\mathrm{M}}$mode, it achieves 64.3% efficiency across 10 GPUs, which is 7.78 times greater than $\mathcal{C}^{\mathrm{M}}_{\mathrm{CUDA}}$ and 35.5 times higher than $\mathcal{C}^{\mathrm{M}}_{\mathrm{Numba}}$. Similarly, in $\mathcal{C}^{\mathrm{S}}$ mode, $\mathcal{C}^{\mathrm{S}}_{\mathrm{CuPy}}$ reaches 26.6% efficiency with 10 GPUs, surpassing $\mathcal{C}^{\mathrm{S}}_{\mathrm{CUDA}}$ by 3.46 times and $\mathcal{C}^{\mathrm{S}}_{\mathrm{Numba}}$ by 16.97 times.

4.6.2 Weak Scalability Assessment

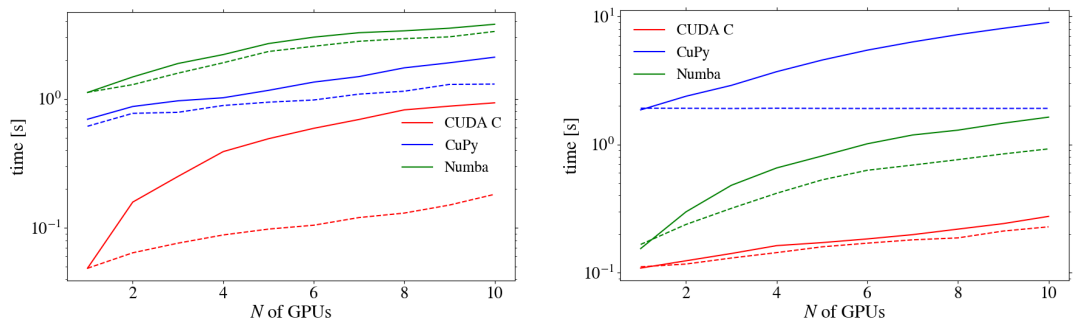


Figure 4.4: Weak scaling performance measured as execution time versus the number of GPUs for CUDA C, CuPy, and Numba (lower values indicate better performance). Solid lines represent single-CPU control results, while dashed lines show results for multi-CPU control implementations. The left panel displays the results for the random number generation, and the right panel shows the 1D MCRT results.

Figure 4.4a illustrates the execution times of $\mathcal{R}_{\mathrm{CUDA}}$, $\mathcal{R}_{\mathrm{CuPy}}$, and $\mathcal{R}_{\mathrm{Numba}}$ across multiple GPUs in a weak scaling scenario. As the number of GPUs increases, execution times rise for all three implementations. $\mathcal{R}_{\mathrm{CUDA}}$ consistently demonstrates better performance over both $\mathcal{R}_{\mathrm{CuPy}}$ and $\mathcal{R}_{\mathrm{Numba}}$. In single-thread mode ($\mathcal{R}^{\mathrm{S}}$), $\mathcal{R}^{\mathrm{S}}_{\mathrm{CUDA}}$ is 2.26 times faster than $\mathcal{R}^{\mathrm{S}}_{\mathrm{CuPy}}$ and 4.06 times faster than $\mathcal{R}^{\mathrm{S}}_{\mathrm{Numba}}$ when utilizing 10 GPUs. In multi-thread mode ($\mathcal{R}^{\mathrm{M}}$), $\mathcal{R}^{\mathrm{M}}_{\mathrm{CUDA}}$ surpasses $\mathcal{R}^{\mathrm{M}}_{\mathrm{CuPy}}$ by a factor of 7.15 and $\mathcal{R}^{\mathrm{M}}_{\mathrm{Numba}}$ by 18.28 times.

In Figure 4.4b, the execution times for $\mathcal{C}_{\mathrm{CUDA}}$, $\mathcal{C}_{\mathrm{CuPy}}$, and $\mathcal{C}_{\mathrm{Numba}}$ are compared under weak scaling across different numbers of GPUs. $\mathcal{C}_{\mathrm{CUDA}}$ consistently outperforms both $\mathcal{C}_{\mathrm{CuPy}}$ and $\mathcal{C}_{\mathrm{Numba}}$ across all configurations. While all implementations observe a performance decline in the $\mathcal{C}^{\mathrm{S}}$ mode, $\mathcal{C}^{\mathrm{M}}_{\mathrm{CuPy}}$ shows stable execution times in the $\mathcal{C}^{\mathrm{M}}$ mode. This stability arises from $\mathcal{C}_{\mathrm{CuPy}}$'s heavy reliance on temporary storage for intermediate calculations. As the number of GPUs grows, the increased bandwidth reduces the data movement bottleneck, leading to consistent performance. However, despite this steadiness, $\mathcal{C}^{\mathrm{M}}_{\mathrm{CuPy}}$ is 8.4 times slower than $\mathcal{C}^{\mathrm{M}}_{\mathrm{CUDA}}$ and 2.07 times slower than $\mathcal{C}^{\mathrm{M}}_{\mathrm{Numba}}$ with 10 GPUs. In $\mathcal{C}^{\mathrm{S}}$ mode, $\mathcal{C}^{\mathrm{S}}_{\mathrm{CUDA}}$ surpasses $\mathcal{C}^{\mathrm{S}}_{\mathrm{CuPy}}$ by 32.66 times and $\mathcal{C}^{\mathrm{S}}_{\mathrm{Numba}}$ by 5.95 times with 10 GPUs.

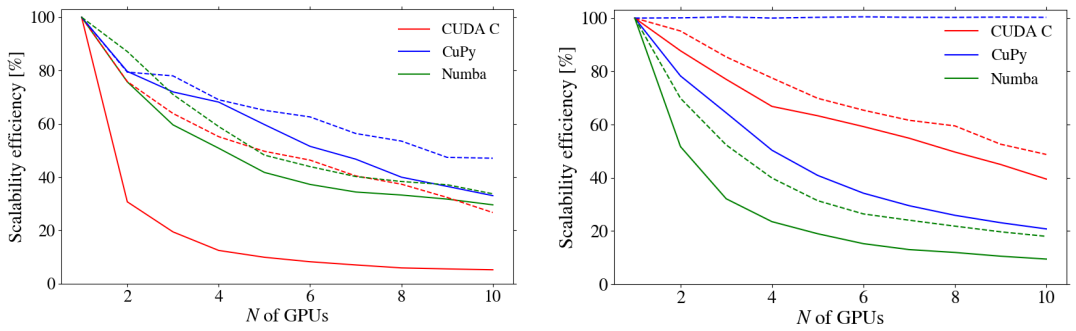


Figure 4.5: Weak scalability efficiency comparison for CUDA C, CuPy, and Numba implementations (higher values indicate better scalability). The left panel displays the results of the random number generation problem, whereas the right panel focuses on the 1D MCRT application. A perfect scalability score is 100%, with values between 99% and 51% indicating acceptable scalability, and values below 51% considered poor scalability.

Figure 4.5a illustrates the scalability efficiency of the PRN generation test under weak scaling conditions. Among the three platforms, $\mathcal{R}_{\mathrm{CuPy}}$ shows better scalability in both modes. Specifically, when using 10 GPUs, $\mathcal{R}_{\mathrm{CuPy}}$ achieves 47.1% efficiency in $\mathcal{R}^{\mathrm{M}}$ mode and 33.04% in $\mathcal{R}^{\mathrm{S}}$ mode. Compared to $\mathcal{R}_{\mathrm{CUDA}}$, $\mathcal{R}_{\mathrm{CuPy}}$ delivers approximately 1.76 times higher efficiency in $\mathcal{R}^{\mathrm{M}}$ mode and 6.32 times in $\mathcal{R}^{\mathrm{S}}$ mode. Similarly, when compared to $\mathcal{R}_{\mathrm{Numba}}$, $\mathcal{R}_{\mathrm{CuPy}}$ demonstrates around 1.4 times better efficiency in $\mathcal{R}^{\mathrm{M}}$ mode and 1.12 times better in $\mathcal{R}^{\mathrm{S}}$ mode.

Figure 4.5b demonstrates the scalability efficiency of the 1D MCRT test problem

under weak scaling. The $\mathcal{C}_{\mathrm{CuPy}}$ implementation maintains 100% efficiency in $\mathcal{C}^{\mathrm{M}}$ mode across all GPU counts. In contrast, the efficiency of $\mathcal{C}^{\mathrm{M}}_{\mathrm{CUDA}}$ and $\mathcal{C}^{\mathrm{M}}_{\mathrm{Numba}}$ decreases as the number of GPUs increases, falling to 48.74% and 17.96%, respectively, when using 10 GPUs. In $\mathcal{C}^{\mathrm{S}}$ mode, $\mathcal{C}^{\mathrm{S}}_{\mathrm{CUDA}}$ outperforms both $\mathcal{C}^{\mathrm{S}}_{\mathrm{CuPy}}$ and $\mathcal{C}^{\mathrm{S}}_{\mathrm{Numba}}$ in terms of efficiency. With 10 GPUs, $\mathcal{C}^{\mathrm{S}}_{\mathrm{CUDA}}$ achieves an efficiency of 39.49%, which is 1.9 times higher than $\mathcal{C}^{\mathrm{S}}_{\mathrm{CuPy}}$ and 4.19 times greater than $\mathcal{C}^{\mathrm{S}}_{\mathrm{Numba}}$.

4.6.3 Performance Tuning

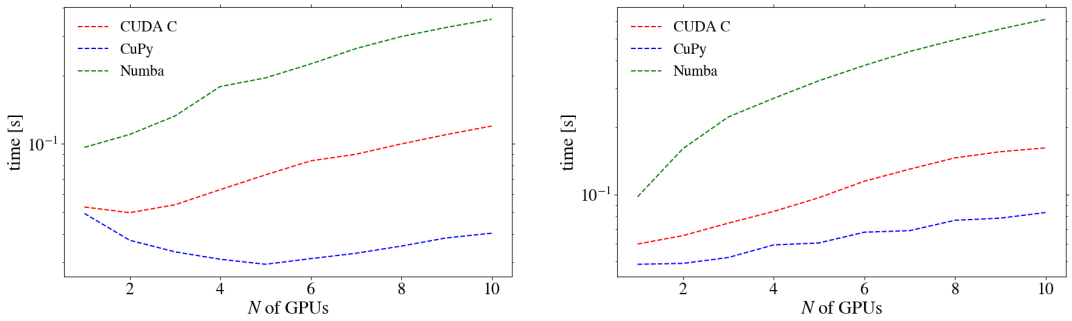


Figure 4.6: Computation time as a function of the number of GPUs for optimized implementations of CUDA C, CuPy, and Numba (lower is better). The left figure presents the outcomes for strong scaling, whereas the right figure displays the results for weak scaling. Only the multi-CPU implementations for the 1D MCRT test problem are included.

This section focuses on applying optimizations to the single-precision version of our 1D MCRT test case. The two optimization strategies explored are:

1) **Optimal Block and Thread Configurations**: Identifying the most effective combination of blocks and threads is essential for maximizing GPU resource utilization. For each platform (CUDA C, CuPy, and Numba), we performed experiments to identify the optimal grid configuration by varying the number of blocks and threads per block. This approach is supported by previous studies that emphasize the impact of grid configuration on GPU performance [186, 187, 188]. CUDA C and Numba allow direct control over these parameters, whereas CuPy requires the use of `Raw Kernels` to achieve similar control.

2) **`Fast Math` Optimization**: This technique enables faster but less precise mathematical operations, including fused multiply-add instructions and approximations of transcendental functions (e.g., logarithm and exponential), to enhance performance. Although this leads to a reduction in precision, it remains suitable for applications where exact numerical accuracy is not essential [189, 190, 191, 192, 193]. In this study, we evaluate the performance gains achieved through both optimization strategies and document the corresponding source code modifications.

This section focuses on optimizing our 1D MCRT test problem using the $\mathcal{C}^{\mathrm{M}}$ implementation. Two optimization strategies are considered: 1) determining the optimal block and thread configuration for each platform [186, 187, 194], and 2) enabling `fast math` [167] operations for computations. The latter can be utilized in cases where lower or mixed precision is acceptable, even though it is not appropriate for tasks requiring high precision (e.g., [190, 191, 192, 193]). We evaluate the performance gains achieved through these optimizations and explain the changes needed in the source code.

Figure 4.6a illustrates the execution times for various platforms after optimizations were applied for strong scaling. The $\mathcal{C}^{\mathrm{M}}_{\mathrm{CuPy}}$ implementation outperforms both $\mathcal{C}^{\mathrm{M}}_{\mathrm{CUDA}}$ and $\mathcal{C}^{\mathrm{M}}_{\mathrm{Numba}}$ across all GPU configurations. With 10 GPUs, $\mathcal{C}^{\mathrm{M}}_{\mathrm{CuPy}}$ is 2.98 times faster than $\mathcal{C}^{\mathrm{M}}_{\mathrm{CUDA}}$ and 8.83 times faster than $\mathcal{C}^{\mathrm{M}}_{\mathrm{Numba}}$. Figure 4.6b shows the execution times for weak scaling, where $\mathcal{C}^{\mathrm{M}}_{\mathrm{CuPy}}$ again surpasses $\mathcal{C}^{\mathrm{M}}_{\mathrm{CUDA}}$ and $\mathcal{C}^{\mathrm{M}}_{\mathrm{Numba}}$ in performance at any number of GPUs used. At 10 GPUs, $\mathcal{C}^{\mathrm{M}}_{\mathrm{CuPy}}$ is 1.95 times faster than $\mathcal{C}^{\mathrm{M}}_{\mathrm{CUDA}}$ and 7.36 times faster than $\mathcal{C}^{\mathrm{M}}_{\mathrm{Numba}}$.

$\mathcal{C}^{\mathrm{M}}_{\mathrm{CuPy}}$ outperforms $\mathcal{C}^{\mathrm{M}}_{\mathrm{CUDA}}$ primarily due to its default memory "caching" feature, which minimizes the overhead from memory allocation and CPU/GPU synchronization [70]. In simple situations, such as our 1D MCRT test problem, where memory usage is stable and predictable, this caching approach works highly effectively. However, in more complex cases, such as 3D MCRT simulations with time-varying source terms or dynamic geometry changes [195, 196], the default caching behavior of $\mathcal{C}^{\mathrm{M}}_{\mathrm{CuPy}}$ can lead to inefficiencies. This includes problems like memory fragmentation and inefficient memory use, which lower performance because cached memory blocks might not be properly freed or reused. On the other hand, $\mathcal{C}^{\mathrm{M}}_{\mathrm{CUDA}}$ does not use memory

caching by default, offering more flexibility in memory management. This approach lets programmers manage and adjust memory use to fit specific applications. Although implementing memory caching in $\mathcal{C}^{\mathrm{M}}_{\mathrm{CUDA}}$ is possible, but it requires extra code changes customized for the specific task [59, 167].

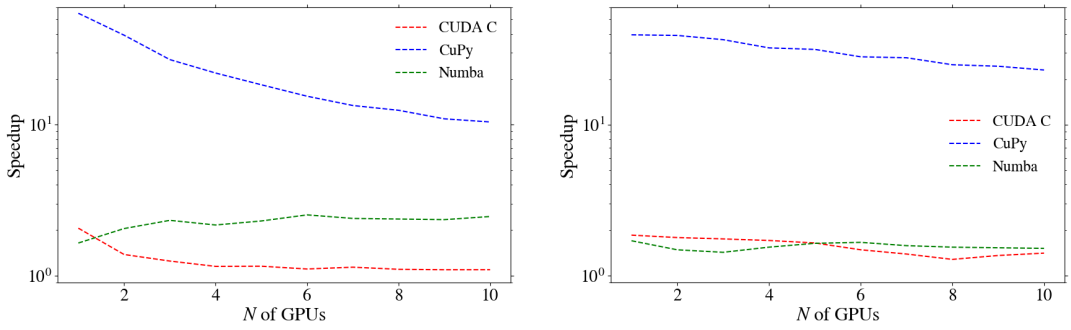


Figure 4.7: Speedup of optimized code implementation over the non-optimized implementation of CUDA C, CuPy, and Numba for multi-CPU thread control versions of the 1D MCRT application. The left figure illustrates strong scaling results, whereas the right figure depicts weak scaling results.

Figure 4.7 illustrates the performance boost of optimized versus non-optimized code for both strong and weak scaling scenarios. In the case of strong scaling (Figure 4.7a), the $\mathcal{C}^{\mathrm{M}}_{\mathrm{CuPy}}$ implementation achieves a speedup ranging from 55× down to 11× as more GPUs are utilized. Meanwhile, $\mathcal{C}^{\mathrm{M}}_{\mathrm{Numba}}$ shows a more modest speedup, between 1.7× and 2.5×, and $\mathcal{C}^{\mathrm{M}}_{\mathrm{CUDA}}$ experiences speedups from 2.1× to 1.1×. For weak scaling (Figure 4.7b), $\mathcal{C}^{\mathrm{M}}_{\mathrm{CuPy}}$ maintains high speedups, ranging from 40× to 24×, while $\mathcal{C}^{\mathrm{M}}_{\mathrm{Numba}}$ consistently delivers a 1.7× increase, and $\mathcal{C}^{\mathrm{M}}_{\mathrm{CUDA}}$ fluctuates between 1.9× and 1.5×. Overall, while $\mathcal{C}^{\mathrm{M}}_{\mathrm{Numba}}$ and $\mathcal{C}^{\mathrm{M}}_{\mathrm{CUDA}}$ maintain relatively stable speedup performance with increasing GPU count, $\mathcal{C}^{\mathrm{M}}_{\mathrm{CuPy}}$ experiences a decline in speedup as more GPUs are added but retain a high overall speedup advantage.

Figure 4.8 illustrates the impact of optimization parameters on performance improvements. In strong scaling (Figure 4.8a), both $\mathcal{C}^{\mathrm{M}}_{\mathrm{CUDA}}$ and $\mathcal{C}^{\mathrm{M}}_{\mathrm{CuPy}}$ benefit similarly from `block-thread tuning` (48.91% and 46.54%, respectively) and `fast math` usage (51.09% and 53.46%). However, $\mathcal{C}^{\mathrm{M}}_{\mathrm{Numba}}$ benefits more from `block-thread tuning` (83%) compared to `fast math` (17%). In weak scaling (Figure 4.8b), $\mathcal{C}^{\mathrm{M}}_{\mathrm{CUDA}}$ and $\mathcal{C}^{\mathrm{M}}_{\mathrm{CuPy}}$ achieve 36.9% and 38.44% improvements from `block-thread tuning`

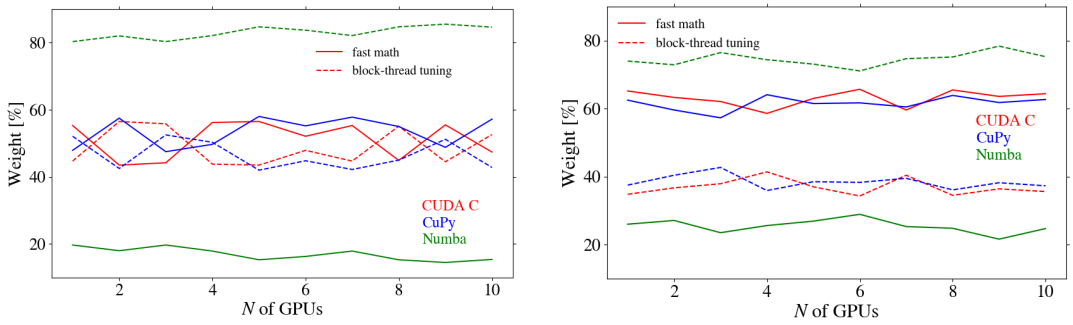


Figure 4.8: The average impact of each optimization factor on the performance improvements in the multi-CPU implementation of the 1D MCRT test problem across CUDA C, CuPy, and Numba. The solid line represents the effect of `fast math` optimizations, while the dashed line represents the contribution from `block-thread` configuration tuning. The left panel presents results for strong scaling, and the right panel illustrates the outcomes for weak scaling.

and 63.1% and 61.56% from `fast math`, respectively. Like in strong scaling, $\mathcal{C}^{\mathrm{M}}_{\mathrm{Numba}}$ derives more benefit from `block-thread tuning` (74.56%) than from `fast math` (25.44%).

We apply the optimizations as follows: for both $\mathcal{C}^{\mathrm{M}}_{\mathrm{CUDA}}$ and $\mathcal{C}^{\mathrm{M}}_{\mathrm{Numba}}$, we introduce minimal modifications by setting the optimal block and thread configurations directly within the code. Additionally, we enable the `-use_fast_math` flag for $\mathcal{C}^{\mathrm{M}}_{\mathrm{CUDA}}$ at the compilation stage and activate the `fastmath=True` option in the `jit` decorator for $\mathcal{C}^{\mathrm{M}}_{\mathrm{Numba}}$ [72]. In contrast, optimizing $\mathcal{C}^{\mathrm{M}}_{\mathrm{CuPy}}$ is more involved due to the use of `Raw Kernel`, which requires a complete modification of the code structure [70].

In Figure 4.9, the number of instructions executed for the non-optimized (Figure 4.9a) and optimized (Figure 4.9b) GPU kernels across $\mathcal{C}^{\mathrm{M}}_{\mathrm{CUDA}}$, $\mathcal{C}^{\mathrm{M}}_{\mathrm{CuPy}}$, and $\mathcal{C}^{\mathrm{M}}_{\mathrm{Numba}}$ are shown. The terms `fp32` (`fp64`) refer to instructions using 32-bit (64-bit) floating-point operations, while `int` represents instructions for integer data, and `control` instructions manage program execution flow. In Figure 4.9a, $\mathcal{C}^{\mathrm{M}}_{\mathrm{CuPy}}$ has the highest count of instructions for all categories except `fp64`, which aligns with expectations since $\mathcal{C}^{\mathrm{M}}_{\mathrm{CuPy}}$'s non-optimized code, due to its abstraction layers, executes substantially more instructions compared to $\mathcal{C}^{\mathrm{M}}_{\mathrm{CUDA}}$ and $\mathcal{C}^{\mathrm{M}}_{\mathrm{Numba}}$. Specifically, $\mathcal{C}^{\mathrm{M}}_{\mathrm{CuPy}}$ performs 1.15 and 1.21 times more `fp32` operations, 6 and 24 times more `int` instructions, and

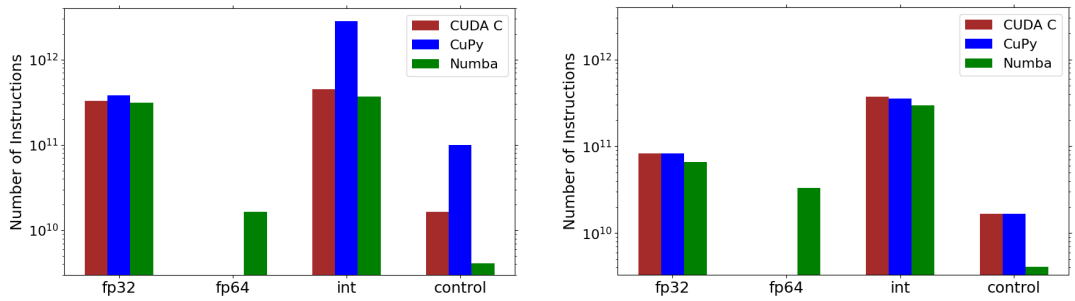


Figure 4.9: Instruction counts for GPU kernel execution using CUDA C, CuPy, and Numba. The left panel presents results for non-optimized code, while the right panel displays those for optimized code. The data includes the multi-CPU implementation of the 1D MCRT test case. Instruction types are categorized as: `fp32` for 32-bit floating-point operations, `fp64` for 64-bit floating-point operations, `int` for integer operations, and `control` for control flow instructions.

6.3 and 7.7 times more `control` instructions than $\mathcal{C}^{\mathrm{M}}_{\mathrm{CUDA}}$ and $\mathcal{C}^{\mathrm{M}}_{\mathrm{Numba}}$, respectively. Unlike the others, $\mathcal{C}^{\mathrm{M}}_{\mathrm{Numba}}$ includes `fp64` operations because its default PRN generation uses double-precision [183]. In Figure 4.9b, `fp32` instructions decrease across all implementations due to `fast math` optimizations, which apply techniques like `flush-to-zero` to speed up computation by approximating small values [59, 72]. After optimization, $\mathcal{C}^{\mathrm{M}}_{\mathrm{CuPy}}$ and $\mathcal{C}^{\mathrm{M}}_{\mathrm{CUDA}}$ have identical instruction counts since $\mathcal{C}^{\mathrm{M}}_{\mathrm{CuPy}}$'s `Raw Kernel` optimization matches that of $\mathcal{C}^{\mathrm{M}}_{\mathrm{CUDA}}$ [70]. However, $\mathcal{C}^{\mathrm{M}}_{\mathrm{Numba}}$'s `fp64` instruction count doubles due to the `DSETP` instruction, a double-precision comparison identified by analyzing the SASS Assembly code. This comparison adds an extra double-precision operation. Its comparison result affects subsequent calculations [59].

4.6.4 Evaluation of Power Usage

Figure 4.10a illustrates the energy consumption per particle for $\mathcal{C}^{\mathrm{M}}_{\mathrm{CUDA}}$, $\mathcal{C}^{\mathrm{M}}_{\mathrm{CuPy}}$, and $\mathcal{C}^{\mathrm{M}}_{\mathrm{Numba}}$ under strong scaling conditions. In the non-optimized configurations (solid lines), $\mathcal{C}^{\mathrm{M}}_{\mathrm{CUDA}}$ demonstrates substantially lower energy consumption compared to both $\mathcal{C}^{\mathrm{M}}_{\mathrm{CuPy}}$ and $\mathcal{C}^{\mathrm{M}}_{\mathrm{Numba}}$, across any number of GPUs. For instance, with 10 GPUs, $\mathcal{C}^{\mathrm{M}}_{\mathrm{CUDA}}$ uses 7.27 times less energy than $\mathcal{C}^{\mathrm{M}}_{\mathrm{CuPy}}$ and 6.7 times less than $\mathcal{C}^{\mathrm{M}}_{\mathrm{Numba}}$. Conversely, in the optimized versions (dashed lines), $\mathcal{C}^{\mathrm{M}}_{\mathrm{CuPy}}$ becomes the most energy-efficient, consuming 2.45 times less energy than $\mathcal{C}^{\mathrm{M}}_{\mathrm{CUDA}}$ and 7.48 times less than $\mathcal{C}^{\mathrm{M}}_{\mathrm{Numba}}$ with 10

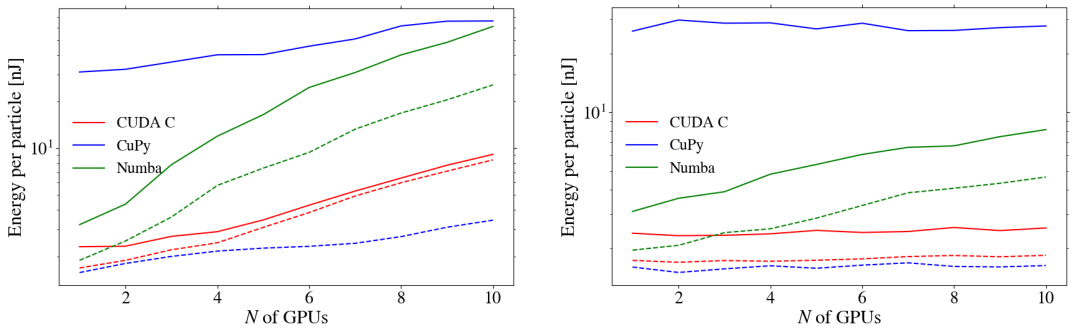


Figure 4.10: Energy usage per particle as a function of the number of GPUs for CUDA C, Namba, and CuPy implementations. Solid lines correspond to non-optimized versions, while dashed lines represent optimized multi-CPU implementations for the 1D MCRT application. The left figure displays outcomes for strong scaling, and the right figure illustrates weak scaling outcomes.

GPUs.

Figure 4.10b presents the weak scaling performance results. In the non-optimized implementation (solid lines), when using 10 GPUs, $\mathcal{C}^{\mathrm{M}}_{\mathrm{CUDA}}$ is considerably more efficient than $\mathcal{C}^{\mathrm{M}}_{\mathrm{CuPy}}$ and $\mathcal{C}^{\mathrm{M}}_{\mathrm{Numba}}$, outperforming them by factors of 10.88 and 3.2, respectively. Conversely, in the optimized version (dashed lines), $\mathcal{C}^{\mathrm{M}}_{\mathrm{CuPy}}$ shows improved performance, surpassing both $\mathcal{C}^{\mathrm{M}}_{\mathrm{CUDA}}$ and $\mathcal{C}^{\mathrm{M}}_{\mathrm{Numba}}$ by factors of 1.13 and 2.85, respectively, also with 10 GPUs.

In non-optimized code, the energy efficiency of $\mathcal{C}^{\mathrm{M}}_{\mathrm{CUDA}}$ surpasses that of $\mathcal{C}^{\mathrm{M}}_{\mathrm{CuPy}}$ and $\mathcal{C}^{\mathrm{M}}_{\mathrm{Numba}}$ due to its efficient resource utilization. $\mathcal{C}^{\mathrm{M}}_{\mathrm{Numba}}$ not only consumes energy during computation but also uses additional power while idling, as the GPU waits for the PRNG state to be initialized on the host [72]. This idle time increases the overall energy consumption. $\mathcal{C}^{\mathrm{M}}_{\mathrm{CuPy}}$ demonstrates lower energy efficiency because of frequent memory transfers of intermediate data throughout the computation process. Since data transfers are among the most energy-intensive operations in GPU computing [155, 197, 198], this results in excessive power consumption.

In the optimized versions of the code, $\mathcal{C}^{\mathrm{M}}_{\mathrm{CuPy}}$ exhibits better energy efficiency compared to both $\mathcal{C}^{\mathrm{M}}_{\mathrm{CUDA}}$ and $\mathcal{C}^{\mathrm{M}}_{\mathrm{Numba}}$, regardless of the number of GPUs used. This advantage is attributed to the use of `Raw Kernel` implementation, which wraps $\mathcal{C}^{\mathrm{M}}_{\mathrm{CUDA}}$ kernels, and applies the memory caching technique in CuPy by default [70]. On the

other hand, $\mathcal{C}^{\mathrm{M}}_{\mathrm{CUDA}}$ does not enable memory block caching by default to offer more flexibility in programming. This is the main reason why $\mathcal{C}^{\mathrm{M}}_{\mathrm{CuPy}}$ and $\mathcal{C}^{\mathrm{M}}_{\mathrm{CUDA}}$ exhibits different levels of energy efficiency.

4.7 Discussion

Our experimental results provide certain important details on how effectively Numba and CuPy perform in multi-GPU configurations, especially when compared to CUDA C, in a variety of test scenarios.

In the non-optimized version, CUDA C consistently performs better than CuPy and Numba in all scaling tests, including strong and weak scaling, across both single- and multi-CPU thread configurations. Additionally, CUDA C demonstrates higher energy efficiency in every test scenario. This highlights CUDA C's advanced development, maturity, and reliability as a programming model.

After implementing optimizations such as `fast math` and `block-thread tuning` configurations, CuPy outperforms both CUDA C and Numba in terms of execution time and energy efficiency. To achieve these improvements, though, major code modifications are needed, such as wrapping the CUDA C kernel inside the `Raw Kernel` feature [70], which implies expertise in CUDA C programming.

CuPy demonstrates better speedups when transitioning from non-optimized to optimized code (ranging from 11× to 55×), surpassing Numba and CUDA C in strong and weak scaling tests. Although CuPy's speedup decreases as the number of GPUs increases, it still consistently outperforms Numba and CUDA, which maintain more stable but lower speedups (between 1.5× and 2.5×). The optimized CuPy version is much faster than the non-optimized version. This is due to the non-optimized version using simpler, high-level abstractions that make coding easier but reduce performance. In the meantime, the optimized version uses advanced techniques that demand a higher level of coding knowledge, which improves speed [183]. In contrast, Numba and CUDA C, which are more finely tuned for performance, show less dramatic improvements after optimization, indicating they are already closer to optimal efficiency.

The performance evaluation of different optimization factors revealed that the improvements from `block-thread tuning` and `fast math` in strong scaling tests are nearly the same for CUDA and CuPy. But in weak scaling scenarios, `fast math` provides a larger advantage (62%), mostly because the kernels are compute-intensive. On the other hand, Numba demonstrates a higher dependency on `block-thread tuning` compared to `fast math` in both strong scaling (83%) and weak scaling (75%). This is largely due to the expensive computational implementation of the PRNG state initialization, which involves non-GPU operations [158].

According to our findings, CuPy can be less practical due to the complexity of code modification, even though it provides better performance with specific optimizations. Numba, on the other hand, improves performance slightly, mostly because of the expensive PRNG state initialization, but it does so without requiring major changes to the code. Future research will focus on evaluating the performance of Numba and CuPy in more sophisticated MC simulations and exploring the effects of different GPU communication strategies.

4.8 Conclusion

We examine the performance of Numba and CuPy when solving the Monte Carlo radiative transfer (MCRT) test case using multiple GPUs. Both strong scaling (increasing GPU count while keeping the problem size fixed) and weak scaling (growing the problem size proportionally with the number of GPUs) are investigated. Additionally, we explore the effects of optimizations such as `fast math` and `block-thread` configuration adjustments alongside with an assessment of energy consumption. We tested our implementations in two modes: single-CPU thread, where one CPU thread controls all GPUs, and multi-CPU thread, where each GPU is controlled by its own CPU thread. With these configurations, we can assess energy efficiency and performance differences across different scaling scenarios and optimization techniques.

We used up to ten GPUs in our studies on an Nvidia DGX-2 server. Our results show that CUDA C consistently outperforms the Python-based alternatives and delivers

the best energy efficiency in its default, non-optimized configuration. However, we also found that CuPy can achieve substantial performance gains through optimizations, surpassing CUDA C and Numba. These improvements, achieved through techniques like `fast math` and `block-thread tuning`, considerably boost CuPy's efficiency, but they require extensive changes to the code, making it more complex. On the other hand, Numba, like CUDA C, shows only small improvements with optimization, which suggests that both are already well-tuned for performance and don't need considerable changes.

BLANK

Chapter 5

Conclusion

5.1 General Conclusions

This chapter highlights the main findings of the study that has been done thus far and suggests future research directions. As discussed in the previous chapters, the thesis has focused on efficient computational approaches for GPU-based Monte Carlo radiation transport (MCRT) simulations. The findings have advanced our understanding of optimizing the important aspects of these simulations for improved performance:

- Within Chapter 2, we deliver an exhaustive investigation concerning the efficiency of various parallel pseudo-random number generators (PRNGs), specifically on Nvidia GPU models, namely the RTX3090, GTX1080, RTX3080, and GTX1080Ti. Our analysis examines a selection of five distinct PRNGs available in the `cuRAND` library, including `MRG32k3a`, `XORWOW`, `MTGP32`, `PHILOX4_32_10`, and `MT19937`. These PRNGs are utilized to produce sequences of uniformly distributed random numbers. The acceptance-rejection (AR) technique is then used to transform these sequences into non-uniform distributions. The work involves several implementation methods: a single CPU core approach utilizing the AMD Ryzen Threadripper 3990X, a GPU-based method with and without data movement between the device and host, and a CPU/GPU hybrid approach where random numbers are generated on the device side and then transferred to the CPU for further processing. Our performance evaluation examines various configurations, comparing GPU device and host Application Programming Interfaces (APIs). We analyzed in detail key PRNG parameters affecting execution speed, memory usage, and efficiency. Additionally, we determined the optimal GPU occupancy value for each PRNG across a wide range of generated random numbers to

achieve maximum performance. We also measured memory consumption for PRNG implementations in both host and device APIs, providing insights into their resource usage and efficiency. Overall, this study identifies the main factors affecting PRNG performance on modern GPU architectures and offers useful guidance for optimizing performance, especially in Monte Carlo (MC) applications;

- Chapter 3 provides a comprehensive comparison of the performance of two widely used Python-based programming frameworks, Numba and CuPy, in the scope of MCRT simulations, alongside a reference implementation in CUDA C. The study focuses on two distinct test cases: a memory-intensive task of generating and storing random numbers, and a computationally demanding one-dimensional (1D) MCRT problem. These test scenarios were chosen to reflect typical MCRT simulation problems, providing a thorough assessment of the frameworks' advantages and disadvantages. Three distinct Nvidia GPUs—the Tesla A100, Tesla V100, and GeForce RTX3080—are used in this investigation. Each GPU has its own performance and architecture features, offering different conditions for testing platforms on various hardware. The performance of CuPy and Numba in these test cases was systematically assessed in this study, with an emphasis on important metrics such as computational efficiency and execution time. Additionally, we investigated the impact of single and double precisions on the performance of each platform. This research is particularly relevant to scientific computing, where accuracy and simulation performance can be considerably affected by precision. In addition to performance, one of the most important aspects of our evaluation was energy usage. In order to determine each framework's energy efficiency, we monitored power usage across the test scenarios. This is an important consideration in large-scale simulations, where optimizing energy consumption can lead to cost savings and environmental benefits. Furthermore, we compared the various Nvidia GPUs to investigate how their memory bandwidth, computing capability, and architecture affect the simulation outcomes. This gave us information about the best hardware options for GPU-accelerated MCRT simulations and allowed us determine how specific features of each GPU affect performance differences. The primary goal of this research is to provide a clear understanding of the relative strengths and limitations

of CUDA C, Numba, and CuPy in the context of MCRT simulations. This study provides valuable insights for researchers to optimize GPU-accelerated MC simulations by examining the trade-offs in performance, energy efficiency, and development simplicity, helping them choose the best platform based on the simulation's needs, available hardware, and desired balance between speed and energy efficiency;

- Chapter 4 provides a detailed evaluation of the performance of CUDA C, Numba, and CuPy in solving the MCRT test case across multiple GPUs, using up to 10 GPUs on an Nvidia DGX-2 system. Two implementation modes are assessed: single-CPU thread, where one CPU controls all GPUs, and multi-CPU thread, where each GPU is managed by its own CPU thread. The evaluation covers strong scaling (fixed problem size with increasing GPUs) and weak scaling (problem size grows with GPU count) to understand each platform's efficiency in handling larger workloads. The chapter also examines optimization methods such as `block-thread` adjustments and `fast math` operations, comparing the power consumption and performance of optimized and non-optimized implementations. A detailed analysis of instruction counts is included to gain insights into each platform's execution behavior and computational efficiency. This comprehensive analysis highlights the impact of optimizations on performance and energy efficiency, providing valuable guidance for researchers looking to optimize GPU-based MCRT simulations, while identifying the trade-offs between scalability, efficiency, and energy consumption.

In conclusion, these works contribute to efficient computational approaches for GPU-accelerated MCRT simulations and parallel PRNG on modern Nvidia GPUs. By evaluating multiple PRNGs and programming frameworks such as CuPy, Numba, and CUDA C, the research identifies key factors influencing computational speed, memory usage, and efficiency. It also explores the effects of various optimization techniques, precision levels, and hardware architectures on performance and energy consumption. A review of contemporary software implementations for real-life radiation transport problems shows that although GPU support is increasingly being adopted, many existing tools remain partially accelerated or are still reliant on CPU-based execution. This study can directly impact such efforts by offering tested methodologies

and performance insights that help optimize GPU usage, inform architectural choices, and guide future development. The findings offer valuable guidance for researchers to make informed decisions when selecting platforms, hardware, and optimization strategies, supporting the development of more efficient and scalable simulations with a focus on balancing performance, energy efficiency, and ease of implementation. The findings offer valuable guidance for researchers to make informed decisions when selecting platforms, hardware, and optimization strategies, supporting the development of more efficient and scalable simulations with a focus on balancing performance, energy efficiency, and ease of implementation.

5.2 Limitations, Future Work, and Perspectives

One of the main limitations in this research has been sharing computing resources, such as the Nvidia DGX-2, DGX-A100, and other servers, with many other researchers within the university. Due to their high demand, these resources frequently become overloaded, resulting in a long wait time. Since most of our research time activities ($\sim$ 70-80%) involve running simulations, performance evaluations, debugging, and profiling various application codes, the limited access to these resources has slowed down our progress.

As a future work, we plan to extend our research by analyzing more complex MCRT codes in 2D and 3D, incorporating complicated geometries to better simulate real-world applications. We will also evaluate additional programming platforms such as OpenCL, HIP, SYCL, Vulkan, OneAPI, and others to widen our research and find other effective solutions for GPU-based computations. Furthermore, we aim to integrate Artificial Intelligence and Machine Learning techniques into MCRT problems to enhance performance optimization and predictive modeling. Also, we intend to provide a software tool in near future that will assist researchers in choosing the best block and thread configurations for particular GPU cards and PRNG types, providing optimal performance for their applications.

As active members of the CuPy and Numba communities, we have contributed to

improving these platforms by addressing performance issues, especially with PRNGs. Some of our suggestions have been successfully implemented, leading to better performance. We are committed to continuing our contributions to these communities in order to increase their effectiveness and relevance for a wider variety of applications. Additionally, we are planning to support the NU community by establishing a small computational facility to assist researchers in using GPU computing for their work. Many universities have dedicated the whole department for this purpose, and we, from our side, hope to create a similar resource to advance research and GPU-based computing at our institution.

BLANK

Bibliography

[1] Top500, "Top500 supercomputer sites." https://www.top500.org/, 2024. Accessed: 2024-12-02.

[2] D. B. Kirk and W. H. Wen-Mei, *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann, 2016.

[3] NVIDIA Corporation, *NVIDIA CUDA Toolkit Documentation*. NVIDIA Corporation, Santa Clara, CA, USA, 2024. Version 12.6.

[4] M. Pharr and R. Fernando, *GPU Gems 2: Programming techniques for high-performance graphics and general-purpose computation (gpu gems)*. Addison-Wesley Professional, 2005.

[5] N. Wilt, *The cuda handbook: A comprehensive guide to gpu programming*. Pearson Education, 2013.

[6] A. Haghighat, *Monte Carlo methods for particle transport*. Crc Press, 2020.

[7] M. H. Kalos and P. A. Whitlock, *Monte carlo methods*. John Wiley & Sons, 2009.

[8] R. C. Cheng, "Random variate generation," *Handbook of Simulation*, pp. 139–172, 1998.

[9] L. Martino, D. Luengo, J. Míguez, L. Martino, D. Luengo, and J. Míguez, "Accept–reject methods," *Independent Random Sampling Methods*, pp. 65–113, 2018.

[10] P. L'Ecuyer, "Random number generation with multiple streams for sequential and parallel computing," in *2015 Winter Simulation Conference (WSC)*, pp. 31–44, IEEE, 2015.

[11] A. De Matteis and S. Pagnutti, "Parallelization of random number generators and long-range correlations," *Numerische Mathematik*, vol. 53, pp. 595–608, 1988.

[12] K. Entacher, "On the cray-system random number generator," *Simulation*, vol. 72, no. 3, pp. 163–169, 1999.

[13] K. Entacher, A. Uhl, and S. Wegenkittl, "Parallel random number generation: long-range correlations among multiple processors," in *International Conference of the Austrian Center for Parallel Computation*, pp. 107–116, Springer, 1999.

[14] A. T. Karl, R. Eubank, J. Milovanovic, M. Reiser, and D. Young, "Using rngstreams for parallel random number generation in c++ and r," *Computational Statistics*, vol. 29, pp. 1301–1320, 2014.

[15] B. Yang, Q. Hu, J. Liu, and C. Gong, "Gpu optimized pseudo random number generator for mcnp," in *IEEE Conference Anthology*, pp. 1–6, IEEE, 2013.

[16] W.-M. Pang, T.-T. Wong, and P.-A. Heng, "Generating massive high-quality random numbers using gpu," in *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*, pp. 841–847, IEEE, 2008.

[17] S. K. Monfared, O. Hajihassani, M. S. Kiarostami, S. M. Zanjani, D. Rahmati, and S. Gorgin, "Bsrng: a high throughput parallel bitsliced approach for random number generators," in *Workshop Proceedings of the 49th International Conference on Parallel Processing*, pp. 1–10, 2020.

[18] S. Gao and G. D. Peterson, "Gasprng: Gpu accelerated scalable parallel random number generator library," *Computer Physics Communications*, vol. 184, no. 4, pp. 1241–1249, 2013.

[19] G. Beliakov, M. Johnstone, D. Creighton, and T. Wilkin, “An efficient implementation of bailey and borwein’s algorithm for parallel random number generation on graphics processing units,” *Computing*, vol. 95, no. 4, pp. 309–326, 2013.

[20] A. Fog, “Pseudo-random number generators for vector processors and multicore processors,” *Journal of modern applied statistical methods*, vol. 14, no. 1, p. 23, 2015.

[21] J. Passerat-Palmbach, C. Mazel, and D. R. Hill, “Pseudo-random number generation on gp-gpu,” in *2011 IEEE Workshop on Principles of Advanced and Distributed Simulation*, pp. 1–8, IEEE, 2011.

[22] S. Ayubian, S. Alawneh, and J. Thijssen, “Gpu-based monte-carlo simulation for a sea ice load application,” in *Proceedings of the Summer Computer Simulation Conference*, pp. 1–8, 2016.

[23] J. Spiechowicz, M. Kostur, and L. Machura, “Gpu accelerated monte carlo simulation of brownian motors dynamics with cuda,” *Computer Physics Communications*, vol. 191, pp. 140–149, 2015.

[24] S. Okada, K. Murakami, S. Incerti, K. Amako, and T. Sasaki, “Mpexs-dna, a new gpu-based monte carlo simulator for track structures and radiation chemistry at subcellular scale,” *Medical Physics*, vol. 46, no. 3, pp. 1483–1500, 2019.

[25] J. Bert, H. Perez-Ponce, Z. El Bitar, S. Jan, Y. Boursier, D. Vintache, A. Bonissent, C. Morel, D. Brasse, and D. Visvikis, “Geant4-based monte carlo simulations on gpu for medical applications,” *Physics in Medicine & Biology*, vol. 58, no. 16, p. 5593, 2013.

[26] E. Alerstam, T. Svensson, and S. Andersson-Engels, “Parallel computing with graphics processing units for high-speed monte carlo simulation of photon migration,” *Journal of biomedical optics*, vol. 13, no. 6, pp. 060504–060504, 2008.

[27] X. Jia, X. Gu, J. Sempau, D. Choi, A. Majumdar, and S. B. Jiang, "Development of a gpu-based monte carlo dose calculation code for coupled electron–photon transport," *Physics in Medicine & Biology*, vol. 55, no. 11, p. 3077, 2010.

[28] D. B. Thomas, L. Howes, and W. Luk, "A comparison of cpus, gpus, fpgas, and massively parallel processor arrays for random number generation," in *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, pp. 63–72, 2009.

[29] T. Preis, P. Virnau, W. Paul, and J. J. Schneider, "Gpu accelerated monte carlo simulation of the 2d and 3d ising model," *Journal of Computational Physics*, vol. 228, no. 12, pp. 4468–4477, 2009.

[30] L. Howes and D. Thomas, "Efficient random number generation and application using cuda," *GPU gems*, vol. 3, pp. 805–830, 2007.

[31] M. Sussman, W. Crutchfield, and M. Papakipos, "Pseudorandom number generation on the gpu," in *Graphics Hardware*, pp. 87–94, 2006.

[32] T. Bradley, J. du Toit, R. Tong, M. Giles, and P. Woodhams, "Parallelization techniques for random number generators," in *GPU Computing Gems Emerald Edition*, pp. 231–246, Elsevier, 2011.

[33] L. Y. Barash and L. N. Shchur, "Prand: Gpu accelerated parallel random number generation library: Using most reliable algorithms and applying parallelism of modern gpus and cpus," *Computer Physics Communications*, vol. 185, no. 4, pp. 1343–1353, 2014.

[34] Y. Kim and G. Hwang, "Efficient parallel cuda random number generator on nvidia gpus," *Journal of KIISE*, vol. 42, no. 12, pp. 1467–1473, 2015.

[35] V. Demchik, "Pseudorandom numbers generation for monte carlo simulations on gpus: Opencl approach," *Numerical Computations with GPUs*, pp. 245–271, 2014.

[36] T. Ciglarič, R. Češnovar, and E. Štrumbelj, "An opencl library for parallel random number generators," *The Journal of Supercomputing*, vol. 75, pp. 3866–3881, 2019.

[37] J. Wadden, N. Brunelle, K. Wang, M. El-Hadedy, G. Robins, M. Stan, and K. Skadron, "Generating efficient and high-quality pseudo-random behavior on automata processors," in *2016 IEEE 34th International Conference on Computer Design (ICCD)*, pp. 622–629, IEEE, 2016.

[38] P. L'Ecuyer, B. Oreshkin, and R. Simard, "Random numbers for parallel computers: requirements and methods," *Manuscript submitted for publication*, 2014.

[39] M. Manssen, M. Weigel, and A. K. Hartmann, "Random number generators for massively parallel simulations on gpu," *The European Physical Journal Special Topics*, vol. 210, no. 1, pp. 53–71, 2012.

[40] W. B. Langdon, "Prng random numbers on gpu," 2007.

[41] C. Gong, J. Liu, L. Chi, Q. Hu, L. Deng, and Z. Gong, "Accelerating pseudo-random number generator for mcnp on gpu," in *AIP Conference Proceedings*, vol. 1281, pp. 1335–1337, American Institute of Physics, 2010.

[42] N. Nandapalan, R. P. Brent, L. M. Murray, and A. P. Rendell, "High-performance pseudo-random number generation on graphics processing units," in *Parallel Processing and Applied Mathematics: 9th International Conference, PPAM 2011, Torun, Poland, September 11-14, 2011. Revised Selected Papers, Part I 9*, pp. 609–618, Springer, 2012.

[43] H. Kargaran, A. Minuchehr, and A. Zolfaghari, "The development of gpu-based parallel prng for monte carlo applications in cuda fortran," *AIP Advances*, vol. 6, no. 4, 2016.

[44] C. Riesinger, T. Neckel, F. Rupp, A. P. Hinojosa, and H.-J. Bungartz, "Gpu optimization of pseudo random number generators for random ordinary differential equations," *Procedia Computer Science*, vol. 29, pp. 172–183, 2014.

[45] S. Jun, P. Canal, J. Apostolakis, A. Gheata, and L. Moneta, "Vectorization of random number generation and reproducibility of concurrent particle transport simulation," in *Journal of Physics: Conference Series*, vol. 1525, p. 012054, IOP Publishing, 2020.

[46] C. Gregg and K. Hazelwood, "Where is the data? why you cannot debate cpu vs. gpu performance without the answer," in *(IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 134–144, IEEE, 2011.

[47] A. Papoulis, *Random variables and stochastic processes*. McGraw Hill, 1965.

[48] S. Theodoridis, "Probability and stochastic processes," in *Machine learning: a bayesian and optimization perspective*, vol. 2015, pp. 9–51, Academic Press London, UK, 2015.

[49] D. Hoffman and O. J. Karst, "The theory of the rayleigh distribution and some of its applications," *Journal of Ship Research*, vol. 19, no. 03, pp. 172–191, 1975.

[50] N. Corporation, *cuRAND Library*. NVIDIA Corporation, 2024. Available at: https://docs.nvidia.com/cuda/curand/.

[51] M. Matsumoto and T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 8, no. 1, pp. 3–30, 1998.

[52] G. Marsaglia, "Xorshift rngs," *Journal of Statistical software*, vol. 8, pp. 1–6, 2003.

[53] P. L'ecuyer, "Good parameters and implementations for combined multiple recursive random number generators," *Operations Research*, vol. 47, no. 1, pp. 159–164, 1999.

[54] M. Saito and M. Matsumoto, "Variants of mersenne twister suitable for graphic processors," *ACM Transactions on Mathematical Software (TOMS)*, vol. 39, no. 2, pp. 1–20, 2013.

[55] J. K. Salmon, M. A. Moraes, R. O. Dror, and D. E. Shaw, "Parallel random numbers: as easy as 1, 2, 3," in *Proceedings of 2011 international conference for high performance computing, networking, storage and analysis*, pp. 1–12, 2011.

[56] M. Fatica and G. Ruetsch, "Cuda fortran for scientists and engineers," *Best Practices for Efficient CUDA Fortran Programming; Elsevier Inc./Morgan Kaufmann: Waltham, MA, USA*, 2014.

[57] M. Fatica and E. Phillips, "Pricing american options with least squares monte carlo on gpus," in *Proceedings of the 6th Workshop on High Performance Computational Finance*, pp. 1–6, 2013.

[58] A. Fog, "Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs," tech. rep., Copenhagen University College of Engineering, 2021.

[59] NVIDIA Corporation, *CUDA C Programming Guide*. NVIDIA Corporation, 2024. Last accessed on November 5, 2024.

[60] W. Pazner, T. Kolev, and J.-S. Camier, "End-to-end gpu acceleration of low-order-refined preconditioning for high-order finite element discretizations," *The International Journal of High Performance Computing Applications*, vol. 37, no. 5, pp. 578–599, 2023.

[61] A. Abdelfattah, V. Barra, N. Beams, R. Bleile, J. Brown, J.-S. Camier, R. Carson, N. Chalmers, V. Dobrev, Y. Dudouit, *et al.*, "Gpu algorithms for efficient exascale discretizations," *Parallel Computing*, vol. 108, p. 102841, 2021.

[62] M. Pandey, M. Fernandez, F. Gentile, O. Isayev, A. Tropsha, A. C. Stern, and A. Cherkasov, "The transformational role of gpu computing and deep learning in drug discovery," *Nature Machine Intelligence*, vol. 4, no. 3, pp. 211–221, 2022.

[63] Y. Hu, Y. Liu, and Z. Liu, "A survey on convolutional neural network accelerators: Gpu, fpga and asic," in *2022 14th International Conference on Computer Research and Development (ICCRD)*, pp. 100–107, IEEE, 2022.

[64] S. Matsuoka, J. Domke, M. Wahib, A. Drozd, and T. Hoefler, "Myths and legends in high-performance computing," *The International Journal of High Performance Computing Applications*, vol. 37, no. 3-4, pp. 245–259, 2023.

[65] AMD Inc., "HIP: Heterogeneous-Compute Interface for Portability." https://github.com/ROCm-Developer-Tools/HIP, 2024. Accessed: 2024-12-02.

[66] A. Fortenberry and S. Tomov, "Extending magma portability with oneapi," in *2022 Workshop on Accelerator Programming Using Directives (WACCPD)*, pp. 22–31, IEEE, 2022.

[67] Lawrence Livermore National Laboratory, "RAJA: A Performance Portability Layer for Parallel Applications." https://raja.readthedocs.io/, 2023. Accessed: 2024-12-02.

[68] C. R. Trott, D. Lebrun-Grandié, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D. S. Hollman, D. Ibanez, *et al.*, "Kokkos 3: Programming model extensions for the exascale era," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 805–817, 2021.

[69] M. Bauer and M. Garland, "Legate numpy: Accelerated and distributed array computing," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–23, 2019.

[70] CuPy Developers, *CuPy Documentation*, 2024.

[71] R. Nishino and S. H. C. Loomis, "Cupy: A numpy-compatible library for nvidia gpu calculations," *31st confernce on neural information processing systems*, vol. 151, no. 7, 2017.

[72] I. Anaconda, *Numba Documentation*, 2024. Accessed: 2024-11-14.

[73] S. K. Lam, A. Pitrou, and S. Seibert, "Numba: A llvm-based python jit compiler," in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, pp. 1–6, 2015.

[74] E. Zenker, B. Worpitz, R. Widera, A. Huebl, G. Juckeland, A. Knüpfer, W. E. Nagel, and M. Bussmann, "Alpaka–an abstraction library for parallel kernel acceleration," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 631–640, IEEE, 2016.

[75] R. Reyes and V. Lomüller, "Sycl: Single-source c++ accelerator programming," in *Parallel Computing: On the Road to Exascale*, pp. 673–682, IOS Press, 2016.

[76] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, vol. 12, no. 3, p. 66, 2010.

[77] S. Wienke, P. Springer, C. Terboven, and D. an Mey, "Openacc—first experiences with real-world applications," in *Euro-Par 2012 Parallel Processing: 18th International Conference, Euro-Par 2012, Rhodes Island, Greece, August 27-31, 2012. Proceedings 18*, pp. 859–870, Springer, 2012.

[78] M. Martineau, S. McIntosh-Smith, and W. Gaudin, "Evaluating openmp 4.0's effectiveness as a heterogeneous parallel programming model," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 338–347, IEEE, 2016.

[79] T. Kalaiselvi, P. Sriramakrishnan, and K. Somasundaram, "Survey of using gpu cuda programming model in medical image analysis," *Informatics in Medicine Unlocked*, vol. 9, pp. 133–144, 2017.

[80] U. Aydonat, S. O'Connell, D. Capalija, A. C. Ling, and G. R. Chiu, "An opencl™ deep learning accelerator on arria 10," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 55–64, 2017.

[81] J. C. Knight and T. Nowotny, "Larger gpu-accelerated brain simulations with procedural connectivity," *Nature Computational Science*, vol. 1, no. 2, pp. 136–142, 2021.

[82] S. Lim and P. Kang, "Implementing scientific simulations on gpu-accelerated edge devices," in *2020 International Conference on Information Networking (ICOIN)*, pp. 756–760, IEEE, 2020.

[83] S. Memeti, L. Li, S. Pllana, J. Kołodziej, and C. Kessler, "Benchmarking opencl, openacc, openmp, and cuda: programming productivity, performance, and energy consumption," in *Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing*, pp. 1–6, 2017.

[84] S. Christgau, J. Spazier, B. Schnor, M. Hammitzsch, A. Babeyko, and J. Waechter, "A comparison of cuda and openacc: accelerating the tsunami simulation easywave," in *ARCS 2014; 2014 workshop proceedings on architecture of computing systems*, pp. 1–5, VDE, 2014.

[85] L. Kuan, J. Neves, F. Pratas, P. Tomás, and L. Sousa, "Accelerating phylogenetic inference on gpus: an openacc and cuda comparison.," in *IWBBIO*, pp. 589–600, 2014.

[86] S.-i. Satake, H. Yoshimori, and T. Suzuki, "Optimizations of a gpu accelerated heat conduction equation by a programming of cuda fortran from an analysis of a ptx file," *Computer Physics Communications*, vol. 183, no. 11, pp. 2376–2385, 2012.

[87] J. Herdman, W. Gaudin, S. McIntosh-Smith, M. Boulton, D. A. Beckingsale, A. C. Mallinson, and S. A. Jarvis, "Accelerating hydrocodes with openacc, opencl and cuda," in *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pp. 465–471, IEEE, 2012.

[88] B. Cloutier, B. K. Muite, and P. Rigge, "Performance of fortran and c gpu extensions for a benchmark suite of fourier pseudospectral algorithms," in

2012 Symposium on Application Accelerators in High Performance Computing, pp. 145–148, IEEE, 2012.

[89] J. Fang, A. L. Varbanescu, and H. Sips, "A comprehensive performance comparison of cuda and opencl," in *2011 International Conference on Parallel Processing*, pp. 216–225, IEEE, 2011.

[90] K. Karimi, N. G. Dickson, and F. Hamze, "A performance comparison of cuda and opencl," *arXiv preprint arXiv:1005.2581*, 2010.

[91] M. Malik, T. Li, U. Sharif, R. Shahid, T. El-Ghazawi, and G. Newby, "Productivity of gpus under different programming paradigms," *Concurrency and computation: practice and experience*, vol. 24, no. 2, pp. 179–191, 2012.

[92] T. Hoshino, N. Maruyama, S. Matsuoka, and R. Takaki, "Cuda vs openacc: Performance case studies with kernel benchmarks and a memory-bound cfd application," in *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, pp. 136–143, IEEE, 2013.

[93] X. Li and P.-C. Shih, "An early performance comparison of cuda and openacc," in *MATEC Web of Conferences*, vol. 208, p. 05002, EDP Sciences, 2018.

[94] X. Guo, J. Wu, Z. Wu, and B. Huang, "Parallel computation of aerial target reflection of background infrared radiation: Performance comparison of openmp, openacc, and cuda implementations," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 9, no. 4, pp. 1653–1662, 2016.

[95] T. L. Gimenes, F. Pisani, and E. Borin, "Evaluating the performance and cost of accelerating seismic processing with cuda, opencl, openacc, and openmp," in *2018 IEEE international parallel and distributed processing symposium (IPDPS)*, pp. 399–408, IEEE, 2018.

[96] A. Boytsov, I. Kadochnikov, M. Zuev, A. Bulychev, Y. Zolotuhin, and I. Getmanov, "Comparison of python 3 single-gpu parallelization technologies on

the example of a charged particles dynamics simulation problem," in *Proceedings of the CEUR Workshop Proceedings, Dubna, Russia*, pp. 10–14, 2018.

[97] A. Marowka, "Python accelerators for high-performance computing," *The Journal of Supercomputing*, vol. 74, no. 4, pp. 1449–1460, 2018.

[98] H. H. Holm, A. R. Brodtkorb, and M. L. Sætra, "Gpu computing with python: Performance, energy efficiency and usability," *Computation*, vol. 8, no. 1, p. 4, 2020.

[99] D. Di Domenico, J. V. Lima, and G. G. Cavalheiro, "Nas parallel benchmarks with python: a performance and programming effort analysis focusing on gpus," *The Journal of Supercomputing*, vol. 79, no. 8, pp. 8890–8911, 2023.

[100] W. F. Godoy, P. Valero-Lara, T. E. Dettling, C. Trefftz, I. Jorquera, T. Sheehy, R. G. Miller, M. Gonzalez-Tallada, J. S. Vetter, and V. Churavy, "Evaluating performance and portability of high-level programming models: Julia, python/numba, and kokkos on exascale nodes," in *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 373–382, IEEE, 2023.

[101] L. Oden, "Lessons learned from comparing c-cuda and python-numba for gpu-computing," in *2020 28th Euromicro international conference on parallel, distributed and network-based processing (PDP)*, pp. 216–223, IEEE, 2020.

[102] T. Deakin, J. Cownie, W.-C. Lin, and S. McIntosh-Smith, "Heterogeneous programming for the homogeneous majority," in *2022 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pp. 1–13, IEEE, 2022.

[103] M. Thavappiragasam, W. Elwasif, and A. Sedova, "Portability for gpu-accelerated molecular docking applications for cloud and hpc: can portable compiler directives provide performance across all platforms?," in *2022 22nd*

IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid), pp. 975–984, IEEE, 2022.

[104] Z.-X. Ma, Y.-Y. Jin, S.-Z. Tang, H.-J. Wang, W.-C. Xue, J.-D. Zhai, and W.-M. Zheng, "Unified programming models for heterogeneous high-performance computers," *Journal of Computer Science and Technology*, vol. 38, no. 1, pp. 211–218, 2023.

[105] M. Bhattacharya, P. Calafiura, T. Childers, M. Dewing, Z. Dong, O. Gutsche, S. Habib, X. Ju, M. Kirby, K. Knoepfel, *et al.*, "Portability: a necessary approach for future scientific software," *arXiv preprint arXiv:2203.09945*, 2022.

[106] U. M. Noebauer and S. A. Sim, "Monte carlo radiative transfer," *Living Reviews in Computational Astrophysics*, vol. 5, pp. 1–103, 2019.

[107] M. S. Patterson, B. C. Wilson, and D. R. Wyman, "The propagation of optical radiation in tissue i. models of radiation transport and their application," *Lasers in Medical Science*, vol. 6, pp. 155–168, 1991.

[108] J. C. Wagner, D. E. Peplow, S. W. Mosher, T. M. Evans, *et al.*, "Review of hybrid (deterministic/monte carlo) radiation transport methods, codes, and applications at oak ridge national laboratory," *Progress in nuclear science and technology*, vol. 2, no. 104, pp. 808–814, 2011.

[109] J. I. Castor, "Radiation hydrodynamics," tech. rep., Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2003.

[110] D. Sarrut, T. Baudier, D. Borys, A. Etxebeste, H. Fuchs, J. Gajewski, L. Grevillot, S. Jan, G. C. Kagadis, H. G. Kang, *et al.*, "The opengate ecosystem for monte carlo simulation in medical physics," *Physics in Medicine & Biology*, vol. 67, no. 18, p. 184001, 2022.

[111] M. Woo and S. G. Hong, "Straum-matxst: A code system for multi-group neutron-gamma coupled transport calculation with unstructured tetrahedral meshes," *Nuclear Engineering and Technology*, vol. 54, no. 11, pp. 4280–4295, 2022.

[112] J. Pelle, O. Reula, F. Carrasco, and C. Bederian, "Skylight: a new code for general-relativistic ray-tracing and radiative transfer in arbitrary space–times," *Monthly Notices of the Royal Astronomical Society*, vol. 515, no. 1, pp. 1316–1327, 2022.

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

[114] S. P. Hamilton and T. M. Evans, "Continuous-energy monte carlo neutron transport on gpus in the shift code," *Annals of Nuclear Energy*, vol. 128, pp. 236–247, 2019.

[115] S. Agostinelli, J. Allison, K. a. Amako, J. Apostolakis, H. Araujo, P. Arce, M. Asai, D. Axen, S. Banerjee, G. Barrand, *et al.*, "Geant4—a simulation toolkit," *Nuclear instruments and methods in physics research section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 506, no. 3, pp. 250–303, 2003.

[116] Los Alamos National Laboratory, "Mcnp – a general monte carlo n-particle transport code." https://mcnp.lanl.gov/, 2025. Accessed: 2025-05-08.

[117] AdePT Development Team, "AdePT: A performant Monte Carlo transport library for GPUs." https://github.com/apt-sim/AdePT, 2021. Accessed: 2025-04-29.

[118] S. R. Johnson, J. Esseiva, E. Biondo, P. Canal, M. Demarteau, T. Evans, S. Y. Jun, G. Lima, A. Lund, P. Romano, *et al.*, "Celeritas: Accelerating geant4 with gpus," in *EPJ Web of Conferences*, vol. 295, p. 11005, EDP Sciences, 2024.

[119] S. Blyth, "Opticks: Gpu optical photon simulation for particle physics using nvidia® optixtm," in *EPJ Web of Conferences*, vol. 214, p. 02027, EDP Sciences, 2019.

[120] C. Dullemond, A. Juhasz, A. Pohl, F. Sereshti, R. Shetty, T. Peters, B. Commercon, and M. Flock, "Radmc-3d: A multi-purpose radiative transfer tool," *Astrophysics Source Code Library*, pp. ascl–1202, 2012.

[121] C. J. Zoller, A. Hohmann, F. Foschum, S. Geiger, M. Geiger, T. P. Ertl, and A. Kienle, "Parallelized monte carlo software to efficiently simulate the light propagation in arbitrarily shaped objects and aligned scattering media," *Journal of Biomedical Optics*, vol. 23, no. 6, pp. 065004–065004, 2018.

[122] R. M. Bergmann and J. L. Vujić, "Algorithmic choices in warp–a framework for continuous energy monte carlo neutron transport in general 3d geometries on gpus," *Annals of Nuclear Energy*, vol. 77, pp. 176–193, 2015.

[123] J. Lippuner and I. A. Elbakri, "A gpu implementation of egsnrc's monte carlo photon transport for imaging applications," *Physics in Medicine & Biology*, vol. 56, no. 22, p. 7145, 2011.

[124] B. Huang, J. Mielikainen, H. Oh, and H.-L. A. Huang, "Development of a gpu-based high-performance radiative transfer model for the infrared atmospheric sounding interferometer (iasi)," *Journal of computational Physics*, vol. 230, no. 6, pp. 2207–2221, 2011.

[125] A. Badal and A. Badano, "Monte carlo simulation of x-ray imaging using a graphics processing unit," in *2009 IEEE Nuclear Science Symposium Conference Record (NSS/MIC)*, pp. 4081–4084, IEEE, 2009.

[126] J. Shao, K. Zhu, and Y. Huang, "A fast gpu monte carlo implementation for radiative heat transfer in graded-index media," *Journal of Quantitative Spectroscopy and Radiative Transfer*, vol. 269, p. 107680, 2021.

[127] S. Hissoiny, B. Ozell, H. Bouchard, and P. Després, "Gpumcd: a new gpu-oriented monte carlo dose calculation platform," *Medical physics*, vol. 38, no. 2, pp. 754–764, 2011.

[128] X. Jia, X. Gu, Y. J. Graves, M. Folkerts, and S. B. Jiang, "Gpu-based fast monte carlo simulation for radiotherapy dose calculation," *Physics in Medicine & Biology*, vol. 56, no. 22, p. 7017, 2011.

[129] M. Shi, M. Myronakis, M. Jacobson, D. Ferguson, C. Williams, M. Lehmann, P. Baturin, P. Huber, R. Fueglistaller, I. V. Lozano, *et al.*, "Gpu-accelerated monte carlo simulation of mv-cbct," *Physics in Medicine & Biology*, vol. 65, no. 23, p. 235042, 2020.

[130] D. Ma, B. Yang, Q. Zhang, J. Liu, and T. Li, "Evaluation of single-node performance of parallel algorithms for multigroup monte carlo particle transport methods," *Frontiers in Energy Research*, vol. 9, p. 705823, 2021.

[131] B. Ma, M. Gaens, L. Caldeira, J. Bert, P. Lohmann, L. Tellmann, C. Lerche, J. Scheins, E. R. Kops, H. Xu, *et al.*, "Scatter correction based on gpu-accelerated full monte carlo simulation for brain pet/mri," *IEEE Transactions on Medical Imaging*, vol. 39, no. 1, pp. 140–151, 2019.

[132] T. Young-Schultz, S. Brown, L. Lilge, and V. Betz, "Fullmontecuda: a fast, flexible, and accurate gpu-accelerated monte carlo simulator for light propagation in turbid media," *Biomedical optics express*, vol. 10, no. 9, pp. 4711–4726, 2019.

[133] P. L'Ecuyer, D. Munger, B. Oreshkin, and R. Simard, "Random numbers for parallel computers: Requirements and methods, with emphasis on gpus," *Mathematics and Computers in Simulation*, vol. 135, pp. 3–17, 2017.

[134] K. Bossler and G. D. Valdez, "Comparison of kokkos and cuda programming models for key kernels in the monte carlo transport algorithm.," tech. rep., Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2018.

[135] S. P. Hamilton, S. R. Slattery, and T. M. Evans, "Multigroup monte carlo on gpus: Comparison of history-and event-based algorithms," *Annals of Nuclear Energy*, vol. 113, pp. 506–518, 2018.

[136] S. P. Hamilton, T. M. Evans, K. E. Royston, and E. D. Biondo, "Domain decomposition in the gpu-accelerated shift monte carlo code," *Annals of Nuclear Energy*, vol. 166, p. 108687, 2022.

[137] N. Choi and H. G. Joo, "Domain decomposition for gpu-based continuous energy monte carlo power reactor calculation," *Nuclear Engineering and Technology*, vol. 52, no. 11, pp. 2667–2677, 2020.

[138] R. Bleile, P. Brantley, D. Richards, S. Dawson, M. S. McKinley, M. O'Brien, and H. Childs, "Thin-threads: An approach for history-based monte carlo on gpus," in *2019 International Conference on High Performance Computing & Simulation (HPCS)*, pp. 273–280, IEEE, 2019.

[139] A. Humphrey, D. Sunderland, T. Harman, and M. Berzins, "Radiative heat transfer calculation on 16384 gpus using a reverse monte carlo ray tracing approach with adaptive mesh refinement," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 1222–1231, IEEE, 2016.

[140] S. Silvestri and R. Pecnik, "A fast gpu monte carlo radiative heat transfer implementation for coupling with direct numerical simulation," *Journal of Computational Physics: X*, vol. 3, p. 100032, 2019.

[141] F. Heymann and R. Siebenmorgen, "Gpu-based monte carlo dust radiative transfer scheme applied to active galactic nuclei," *The Astrophysical Journal*, vol. 751, no. 1, p. 27, 2012.

[142] D. Ramon, F. Steinmetz, D. Jolivet, M. Compiègne, and R. Frouin, "Modeling polarized radiative transfer in the ocean-atmosphere system with the gpu-accelerated smart-g monte carlo code," *Journal of Quantitative Spectroscopy and Radiative Transfer*, vol. 222, pp. 89–107, 2019.

[143] E. K. Lee, J. P. Wardenier, B. Prinoth, V. Parmentier, S. L. Grimm, R. Baeyens, L. Carone, D. Christie, R. Deitrick, D. Kitzmann, *et al.*, "3d radiative transfer for

exoplanet atmospheres. gcmcrt: a gpu-accelerated mcrt code," *The Astrophysical Journal*, vol. 929, no. 2, p. 180, 2022.

[144] D. Sarrut, A. Etxebeste, E. Muñoz, N. Krah, and J. M. Létang, "Artificial intelligence for monte carlo simulation in medical physics," *Frontiers in Physics*, vol. 9, p. 738112, 2021.

[145] R. H. van Dijk, N. Staut, C. J. Wolfs, and F. Verhaegen, "A novel multichannel deep learning model for fast denoising of monte carlo dose calculations: preclinical applications," *Physics in Medicine & Biology*, vol. 67, no. 16, p. 164001, 2022.

[146] M. Raayai Ardakani, L. Yu, D. R. Kaeli, and Q. Fang, "Framework for denoising monte carlo photon transport simulations using deep learning," *Journal of Biomedical Optics*, vol. 27, no. 8, pp. 083019–083019, 2022.

[147] Z. Peng, H. Shan, T. Liu, X. Pei, G. Wang, and X. G. Xu, "Mcdnet–a denoising convolutional neural network to accelerate monte carlo radiation transport simulations: A proof of principle with patient dose from x-ray ct imaging," *Ieee Access*, vol. 7, pp. 76680–76689, 2019.

[148] P. Xu, M.-Y. Sun, Y.-J. Gao, T.-J. Du, J.-M. Hu, and J.-J. Zhang, "Influence of data amount, data type and implementation packages in gpu coding," *Array*, vol. 16, p. 100261, 2022.

[149] M. M. Radmanović, "A comparison of computing spectral transforms of logic functions using python frameworks on gpu," in *2022 57th International Scientific Conference on Information, Communication and Energy Systems and Technologies (ICEST)*, pp. 1–4, IEEE, 2022.

[150] J. Almgren-Bell, N. Al Awar, D. S. Geethakrishnan, M. Gligoric, and G. Biros, "A multi-gpu python solver for low-temperature non-equilibrium plasmas," in *2022 IEEE 34th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pp. 140–149, IEEE, 2022.

[151] I. Azizi, "Parallelization in python-an expectation-maximization application," 2023.

[152] R. Dogaru and I. Dogaru, "A python framework for fast modelling and simulation of cellular nonlinear networks and other finite-difference time-domain systems," in *2021 23rd International Conference on Control Systems and Computer Science (CSCS)*, pp. 221–226, IEEE, 2021.

[153] J. M. Cohen and M. J. Molemaker, "A fast double precision cfd code using cuda," *Journal of the Physical Society of Japan*, vol. 1, no. 2, pp. 237–341, 2009.

[154] H.-V. Dang and B. Schmidt, "Cuda-enabled sparse matrix–vector multiplication on gpus using atomic operations," *Parallel Computing*, vol. 39, no. 11, pp. 737–750, 2013.

[155] S. Collange, D. Defour, and A. Tisserand, "Power consumption of gpus from a software perspective," in *Computational Science–ICCS 2009: 9th International Conference Baton Rouge, LA, USA, May 25-27, 2009 Proceedings, Part I 9*, pp. 914–923, Springer, 2009.

[156] A. Eklund, P. Dufort, D. Forsberg, and S. M. LaConte, "Medical image processing on the gpu–past, present and future," *Medical image analysis*, vol. 17, no. 8, pp. 1073–1094, 2013.

[157] P. Després and X. Jia, "A review of gpu-based medical image reconstruction," *Physica Medica*, vol. 42, pp. 76–92, 2017.

[158] T. Askar, B. Shukirgaliyev, M. Lukac, and E. Abdikamalov, "Evaluation of pseudo-random number generation on gpu cards," *Computation*, vol. 9, no. 12, p. 142, 2021.

[159] T. Le, B. Vo, H. Fujita, N.-T. Nguyen, and S. W. Baik, "A fast and accurate approach for bankruptcy forecasting using squared logistics loss with gpu-based extreme gradient boosting," *Information Sciences*, vol. 494, pp. 294–310, 2019.

[160] A. Kirimtat and O. Krejcar, "Gpu-based parallel processing techniques for enhanced brain magnetic resonance imaging analysis: A review of recent advances," *Sensors*, vol. 24, no. 5, p. 1591, 2024.

[161] H. Li, A. Li, Y. Liu, Y. Lin, and Y. Shi, "Ai face recognition and processing technology based on gpu computing," *Journal of Theory and Practice of Engineering Science*, vol. 4, no. 05, pp. 9–16, 2024.

[162] D. F. Fernandes, M. C. Santos, A. C. Silva, and A. M. Lima, "Comparative study of cuda-based parallel programming in c and python for gpu acceleration of the 4th order runge-kutta method," *Nuclear Engineering and Design*, vol. 421, p. 113050, 2024.

[163] F. Santoro, I. Petrelli, G. Massaro, G. Filios, F. V. Pepe, L. Amoruso, M. Ieronimaki, S. Burri, E. Charbon, P. Mos, *et al.*, "Gpu-based data processing for speeding-up correlation plenoptic imaging," *arXiv preprint arXiv:2407.20692*, 2024.

[164] M. Moniruzzaman, A. H. Okilly, S. Choi, J. Baek, T. I. Mannan, and Z. Islam, "A comprehensive study of machine learning algorithms for gpu based real-time monitoring and lifetime prediction of igbts," in *2024 IEEE Applied Power Electronics Conference and Exposition (APEC)*, pp. 2678–2684, IEEE, 2024.

[165] Q. Xiong, S. Huang, Z. Yuan, B. Sharma, L. Kuang, K. Jiang, and L. Yu, "Gpic: A set of high-efficiency cuda fortran code using gpu for particle-in-cell simulation in space physics," *Computer Physics Communications*, vol. 295, p. 108994, 2024.

[166] J. Cao, Y. Guan, K. Qian, J. Gao, W. Xiao, J. Dong, B. Fu, D. Cai, and E. Zhai, "Crux: Gpu-efficient communication scheduling for deep learning training," in *Proceedings of the ACM SIGCOMM 2024 Conference*, pp. 1–15, 2024.

[167] NVIDIA, "Cuda c++ best practices guide," 2024. Accessed: 2024-11-14.

[168] A. Gharaibeh, S. Al-Kiswany, and M. Ripeanu, "Crystalgpu: Transparent and efficient utilization of gpu power," *arXiv preprint arXiv:1005.1695*, 2010.

[169] NVIDIA, "Nvidia dgx systems." https://www.nvidia.com/en-us/data-center/dgx-systems/, 2024. Accessed: 2024-09-21.

[170] L. A. Barba, "The python/jupyter ecosystem: Today's problem-solving environment for computational science," *Computing in Science & Engineering*, vol. 23, no. 3, pp. 5–9, 2021.

[171] A. N. Ziogas, T. Schneider, T. Ben-Nun, A. Calotoiu, T. De Matteis, J. de Fine Licht, L. Lavarini, and T. Hoefler, "Productivity, portability, performance: Data-centric python," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–13, 2021.

[172] C. R. Harris, K. J. Millman, S. J. Van Der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, *et al.*, "Array programming with numpy," *Nature*, vol. 585, no. 7825, pp. 357–362, 2020.

[173] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, *et al.*, "Scipy 1.0: fundamental algorithms for scientific computing in python," *Nature methods*, vol. 17, no. 3, pp. 261–272, 2020.

[174] A. Graas, W. J. Palenstijn, B. van Werkhoven, and F. Lucka, "Astra kernelkit: Gpu-accelerated projectors for computed tomography using cupy," *Applied Mathematics for Modern Challenges*, vol. 2, no. 1, pp. 70–92, 2024.

[175] A. A. de Moura Meneses, L. M. Araujo, and R. Schirru, "A gpu-accelerated linear system solution for the galerkin finite element method applied to neutron diffusion equation," *Nuclear Engineering and Design*, vol. 421, p. 113103, 2024.

[176] D. Di Domenico, G. G. Cavalheiro, and J. V. Lima, "Nas parallel benchmark kernels with python: A performance and programming effort analysis focusing on gpus," in *2022 30th Euromicro international conference on parallel, distributed and network-based processing (PDP)*, pp. 26–33, IEEE, 2022.

[177] N. Rao, N. Liebers, A. S. Leger, and S. J. Matthews, "Comparing the performance of numba and cuda for historical analysis of synchrophasor data," in *2024 IEEE Power & Energy Society Innovative Smart Grid Technologies Conference (ISGT)*, pp. 1–5, IEEE, 2024.

[178] J. Villalobos and E. Meneses, "Evaluation of alternatives to accelerate scientific numerical calculations on graphics processing units using python," in *Latin American High Performance Computing Conference*, pp. 3–20, Springer, 2023.

[179] M. H. Kriebel, P. Tecmer, M. Gałyńska, A. Leszczyk, and K. Boguslawski, "Accelerating pythonic coupled-cluster implementations: A comparison between cpus and gpus," *Journal of Chemical Theory and Computation*, vol. 20, no. 3, pp. 1130–1142, 2024.

[180] M. Guerrero-Hurtado, J. M. Catalan, M. Moriche, A. Gonzalo, and O. Flores, "A python-based flow solver for numerical simulations using an immersed boundary method on single gpus," *arXiv preprint arXiv:2406.19920*, 2024.

[181] J. Pata, I. Dutta, N. Lu, J.-r. Vlimant, A. Newman, M. Spiropulu, C. Reissel, and D. Ruini, "Data analysis with gpu-accelerated kernels," in *40th International Conference on High Energy Physics*, vol. 390, p. 908, SISSA, 2021.

[182] D. Lemire and M. E. O'Neill, "Xorshift1024*, xorshift1024+, xorshift128+ and xoroshiro128+ fail statistical tests for linearity," *Journal of Computational and Applied Mathematics*, vol. 350, pp. 139–142, 2019.

[183] T. Askar, A. Yergaliyev, B. Shukirgaliyev, and E. Abdikamalov, "Exploring numba and cupy for gpu-accelerated monte carlo radiation transport," *Computation*, vol. 12, no. 3, p. 61, 2024.

[184] NVIDIA Corporation, *NVIDIA Nsight Systems*, 2025. Accessed: 2025-02-07.

[185] N. Corporation, "nvidia-smi: Nvidia system management interface," 2023. Accessed: 2025-01-29.

[186] C. Woolley, "Gpu optimization fundamentals," *Technical report, Tech. Rep*, 2013.

[187] B. Schussler, P. Rigon, A. F. Lorenzon, and P. O. Navaux, "Bto, block and thread optimization of gpu kernels on geophysical exploration," in *2024 32nd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pp. 9–16, IEEE, 2024.

[188] N.-P. Tran and M. Lee, "Parameter tuning model for optimizing application performance on gpu," in *2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, pp. 78–83, 2016.

[189] NVIDIA Corporation, *CUDA C++ Best Practices Guide*, 2025. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html.

[190] J. Yun, B. Kang, F. Rameau, and Z. Fu, "In defense of pure 16-bit floating-point neural networks," *arXiv preprint arXiv:2305.10947*, 2023.

[191] X. Jia, "Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes," *arXiv preprint arXiv:1807.11205*, 2018.

[192] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, *et al.*, "Mixed precision training," *arXiv preprint arXiv:1710.03740*, 2017.

[193] J. Hayford, J. Goldman-Wetzler, E. Wang, and L. Lu, "Speeding up and reducing memory usage for scientific machine learning via mixed precision," *Computer Methods in Applied Mechanics and Engineering*, vol. 428, p. 117093, 2024.

[194] N.-P. Tran and M. Lee, "Parameter tuning model for optimizing application performance on gpu," in *2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS* W)*, pp. 78–83, IEEE, 2016.

[195] E. Abdikamalov, A. Burrows, C. D. Ott, F. Löffler, E. O'Connor, J. C. Dolence, and E. Schnetter, "A new monte carlo method for time-dependent neutrino radiation transport," *The Astrophysical Journal*, vol. 755, no. 2, p. 111, 2012.

[196] J. Seco and F. Verhaegen, *Monte Carlo techniques in radiation therapy*. CRC press Boca Raton, 2013.

[197] H. Shen, "Enhancing gpu performance and energy efficiency: Innovative strategies for sustainable computing," *Applied and Computational Engineering*, vol. 49, pp. 242–246, 03 2024.

[198] T. M. Aamodt, W. W. L. Fung, T. G. Rogers, and M. Martonosi, *General-purpose graphics processor architectures*. Springer, 2018.

Appendices

BLANK

Appendix A

1D Monte Carlo Radiation Transport in Purely Absorbing Medium

A.1 Introduction

Our research mainly focuses on neutron transport, especially its computational modeling and analysis. This appendix provides detailed derivations of important equations and explains how they are applied to the Monte Carlo method, a key tool in our studies.

The full analytical balance equation for neutrons, known as the neutron transport equation, describes how neutrons behave within a nuclear system. This equation considers various processes that affect neutrons, such as their production, absorption, scattering, and leakage from the system. It provides a comprehensive framework for understanding and predicting the movement and interactions of neutrons in different environments. The equation can be expressed as:

$$\begin{aligned}\frac{1}{v(\mathrm{r},\Omega,E)}\frac{\partial\psi(\mathrm{r},\Omega,E,t)}{\partial t} &+ \Omega\cdot\nabla\psi(\mathrm{r},\Omega,E,t) + \Sigma_t(\mathrm{r},E)\psi(\mathrm{r},\Omega,E,t)\\ &= \int_0^\infty\int_{4\pi}\Sigma_s(\mathrm{r},E'\to E,\Omega'\to\Omega)\psi(\mathrm{r},\Omega',E',t)\,dE'\,d\Omega'\\ &+ \frac{\chi(E)}{4\pi}\int_0^\infty\nu\Sigma_f(\mathrm{r},E')\int_{4\pi}\psi(\mathrm{r},\Omega',E',t)\,d\Omega'\,dE'\\ &+ S(\mathrm{r},\Omega,E,t)\end{aligned} \tag{A.1}$$

where:

- $\psi(\mathrm{r}, \Omega, E, t)$: Neutron angular flux at position r, direction Ω, energy E, and time t;
- $v(\mathrm{r}, \Omega, E)$: Neutron speed at position r, direction Ω, and energy E;
- $\Sigma_t(\mathrm{r}, E)$: Total macroscopic cross-section at position r and energy E;
- $\Sigma_s(\mathrm{r}, E' \to E, \Omega' \to \Omega)$: Differential scattering cross-section;
- $\Sigma_f(\mathrm{r}, E)$: Fission macroscopic cross-section;
- ν: Average number of neutrons produced per fission;
- $\chi(E)$: Neutron spectrum from fission;
- $S(\mathrm{r}, \Omega, E, t)$: External source term.

Solving this neutron transport equation analytically is impossible due to seven independent variables: three for position, two for direction, one for time, and one for energy. These make the equation complex. Since an exact solution is impossible, scientists use numerical methods and make certain approximations to simplify the equation.

A.2 Derivation

Based on equation A.1, we need to derive a one-dimensional balance equation for neutrons going through a purely absorbing medium. Also, we assume that our problem is in a steady state and moves only in one direction, i.e., perpendicular to the medium side, as shown in Figure A.1.

We make several assumptions to simplify our transport equation in a purely absorbing medium. Particularly, we assume:

- **One-dimensional geometry**: The spatial dependence is only in one direction, so $r = x$;

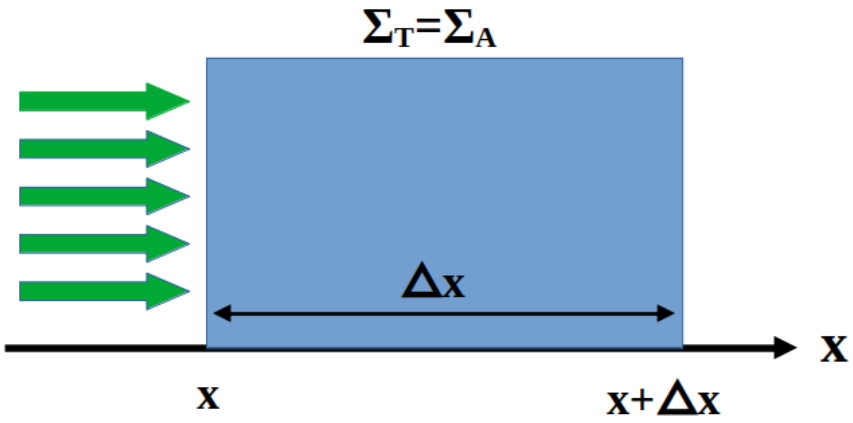


Figure A.1: Neutrons traveling through a purely absorbing medium entering it in a perpendicular direction.

- **Pure absorption**: There is no scattering and fission, so the scattering cross section $\Sigma_s = 0$ and the fission term are absent;
- **Single energy state**: We only consider neutrons with a single energy, so the energy dependence can be dropped;
- **Single direction**: All neutrons travel in the same direction, so angular dependence is simplified, and we assume a single direction of travel $\Omega = 1$;
- **Time-independent**: No time dependency in our transport equation.

Applying these assumptions our initial neutron transport equation A.1 simplifies to the following:

$$\Omega \cdot \nabla\psi(\mathrm{x}) + \Sigma_t\psi(\mathrm{x}) = 0 \tag{A.2}$$

In 1-D geometry and with the assumption that $\Omega = 1$ (i.e., neutrons are moving in the positive x-direction), this becomes:

$$\frac{d\psi(\mathrm{x})}{d\mathrm{x}} + \Sigma_t\psi(\mathrm{x}) = 0 \tag{A.3}$$

However, we need to deal with scalar flux rather than angular flux for Monte Carlo simulations. We must integrate angular flux over all its directions to convert it into

scalar flux. Since we are using only one direction in our experiments therefore, the equation for the scalar flux $\phi(\mathrm{x})$ becomes the same as for the angular flux:

$$\frac{d\phi(\mathrm{x})}{d\mathrm{x}} + \Sigma_t \phi(\mathrm{x}) = 0 \tag{A.4}$$

This is the first-order differential equation, and the solution for it is the following:

Firstly, let's rearrange our equation:

$$\frac{d\phi(\mathrm{x})}{\phi(\mathrm{x})} = -\Sigma_t d\mathrm{x} \tag{A.5}$$

Then, integrate both sides:

$$\ln \phi(\mathrm{x}) = -\Sigma_t \mathrm{x} + C \tag{A.6}$$

Further, by exponentiating both sides, we get:

$$\phi(\mathrm{x}) = e^{-\Sigma_t \mathrm{x} + C} = e^C e^{-\Sigma_t \mathrm{x}} \tag{A.7}$$

Let $\phi_0 = e^C$ be the scalar flux at $x = 0$. Hence, the scalar neutron flux $\phi(x)$ is equal to:

$$\phi(\mathrm{x}) = \phi_0 e^{-\Sigma_t \mathrm{x}} \tag{A.8}$$

We have simplified the neutron transport equation to a form that can be solved analytically. However, to apply the MC method, we need to convert this differential equation into a counting algorithm. By sampling random numbers, we can then solve this test problem. The MC method is particularly well-suited for neutron transport simulations, as it allows us to track the path of each particle and count the number of particles that undergo specific events.

From Figure A.1, we can see that it's possible to count how many particles enter, leave, or get absorbed inside the medium. However, this doesn't inform us how the flux (particle distribution) behaves inside this medium. To understand what happens within the medium, we must break it into smaller 'cells' as depicted in Figure A.2. This allows us to gather detailed information about the flux within each cell.

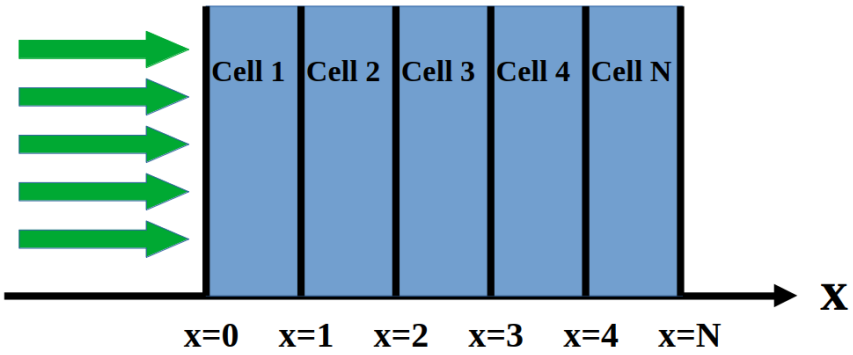


Figure A.2: Neutrons traveling through a purely absorbing medium divided into small cells.

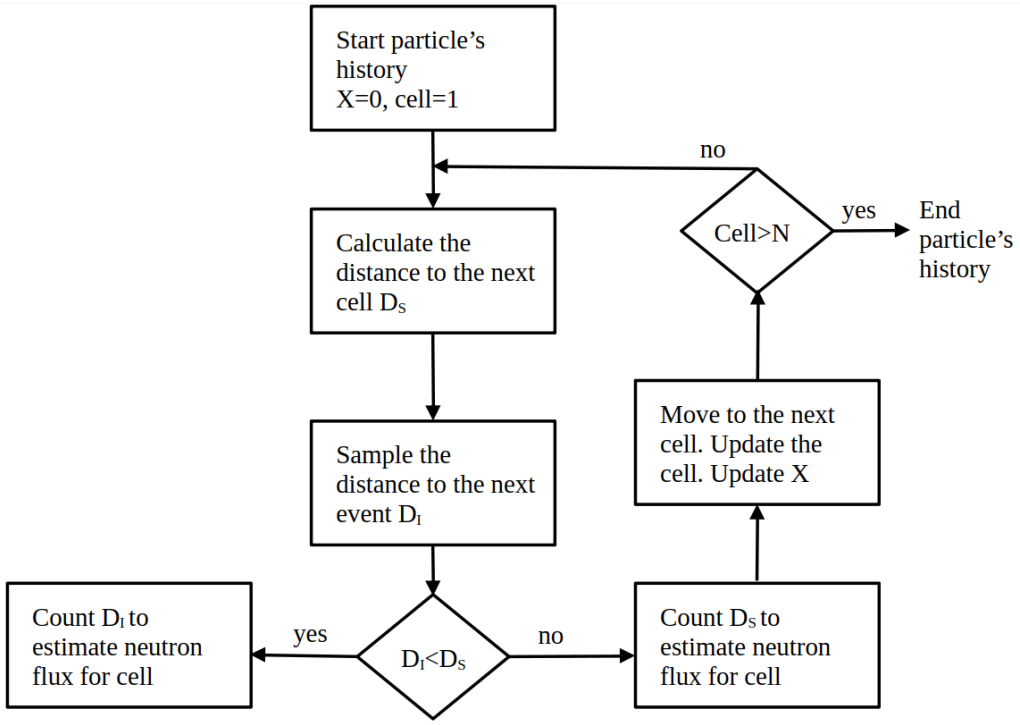


Figure A.3: Flowchart of the MC algorithm for neutron transport in a multi-cell medium.

Now, we can count the number of reactions and estimate flux in each cell. This makes a histogram showing how the flux is distributed inside the medium. It's important to note that dividing the medium into more cells will give a more accurate flux estimation. With this background information, let us define a step-by-step process for using the MC method to simulate particle transport in the multi-cell medium (see Figure A.3 for flowchart of the MC algorithm).

Appendix B

Source Code

B.1 Code Implementations

The source codes developed for this thesis work are available in the following GitHub repository: https://github.com/tairaskar/](https://github.com/tairaskar/. This repository contains all the key implementations and scripts we used in this research work. It is a resource for anyone interested in reviewing, replicating, or extending the work presented in this thesis.

BLANK

Appendix C

Implementation Guidelines for GPU-Based Radiation Transport Simulations

To support users in improving the efficacy of GPU-based radiation transport simulations, we have summarized our key findings in the form of a comparative Table C.1 in this appendix. The table provides practical implementation guidelines across CUDA C, Numba, and CuPy, covering aspects such as performance, scalability, memory management, PRNG handling, optimization gains, and energy efficiency. These insights are drawn from systematic testing across multiple scenarios and platforms.

The optimizations evaluated in this work—namely, `block-thread` configuration tuning and `fast math` approximations—are particularly valuable for extending performance gains to full 3D physically motivated radiation transfer problems. In such simulations, computational and memory demands grow significantly due to increased spatial complexity and particle interactions. `Block-thread tuning` helps maximize GPU occupancy and efficiency, especially in large-scale 3D domains, while `fast math` reduces the overhead of complex mathematical functions commonly used in radiation transport, where applicable, without substantially affecting accuracy. These optimizations collectively contribute to faster execution, better scalability, and lower energy consumption. These results give developers and researchers practical strategies to build more efficient, scalable, and portable 3D GPU-based MCRT codes.

Table C.1: Implementation guidelines based on comparative GPU evaluation of CUDA C, Numba, and CuPy.

Aspect	**CUDA C**	**Numba**	**CuPy**	**Guideline / Insight**
Baseline Performance	High; efficient in both PRNG and MCRT simulations	Comparable to CUDA C in MCRT with minimal global memory access	Lower, due to intermediate results stored in global memory	Use CUDA C for best default performance; Numba is effective when GPU computations involve minimal global memory access. CuPy is slow, however, it offers ease of code development.
Multi-GPU Scalability	Scales well across multiple GPUs	Declines due to CPU-side PRNG initialization overhead	Moderate; limited by excessive data movement	Prefer CUDA C for multi-GPU scaling; optimize Numba to reduce CPU-side initialization overhead.
Ease of Use	Requires low-level programming expertise	Pythonic and JIT-compiled; easy to develop code	High-level, user-friendly API	Use Numba/CuPy for rapid development; CUDA C for fast and efficient high-performance applications.
Memory Management	Manual but efficient control	Automatic, less tunable	Includes default memory pooling	CuPy benefits from memory pooling; optimize memory reuse across frameworks.
PRNG Handling	GPU-native `cuRAND` library	CPU-side PRNG initialization introduces overhead	`cuRAND` library with abstraction overhead	Redesign PRNG in Numba for better performance (hard to implement); CUDA C is most efficient.
Optimization Gains	Small; already well-tuned for performance	Modest; limited by CPU-side PRNG initialization	Substantial with `Raw kernels` and memory pooling	Focus optimization effort on CuPy; others benefit less from tuning.
Fast Math Impact	Effective for compute-heavy kernels	Less effective due to CPU-side PRNG bottlenecks	Useful when combined with `Raw kernels`	Apply selectively in compute-intensive tasks to maintain accuracy; less effective in Numba.
Block-Thread Tuning	Effective and flexible	Highly impactful due to Global Interpreter Lock (GIL)	Needed but requires `Raw kernel` customization	Critical for Numba; CuPy needs custom kernels.
Energy Efficiency	Best overall in default and optimized versions	Improves with tuning, remains higher	Substantially improves after optimization	CuPy can match CUDA C efficiency with optimization.
Code Modification Effort	Low (small changes required)	Low (small changes required)	High with applying of `Raw kernels`	Consider effort vs. performance when choosing framework.
Recommended Use Case	Large-scale, high-performance-critical applications	Moderate problems with minimal memory transfer	Small to moderate workloads prioritizing ease of use	Choose based on balance between scalability, performance, and usability.