computation

MDPI

Article

Exploring Numba and CuPy for GPU-Accelerated Monte Carlo Radiation Transport

Tair Askar [1,2,*], Argyn Yergaliyev [3], Bekdaulet Shukirgaliyev [2,4,5,6] and Ernazar Abdikamalov [2,3]

1 School of Engineering and Digital Sciences, Nazarbayev University, Astana 010000, Kazakhstan
2 Energetic Cosmos Laboratory, Nazarbayev University, Astana 010000, Kazakhstan; b.shukirgaliyev@hw.ac.uk (B.S.); ernazar.abdikamalov@nu.edu.kz (E.A.)
3 Department of Physics, Nazarbayev University, Astana 010000, Kazakhstan; argyn.yergaliyev@nu.edu.kz
4 Heriot-Watt International Faculty, Zhubanov University, Aktobe 030000, Kazakhstan
5 Fesenkov Astrophysical Institute, Almaty 050020, Kazakhstan
6 Department of Computation and Data Science, Astana IT University, Astana 010000, Kazakhstan
* Correspondence: tair.askar@nu.edu.kz

Abstract: This paper examines the performance of two popular GPU programming platforms, Numba and CuPy, for Monte Carlo radiation transport calculations. We conducted tests involving random number generation and one-dimensional Monte Carlo radiation transport in plane-parallel geometry on three GPU cards: NVIDIA Tesla A100, Tesla V100, and GeForce RTX3080. We compared Numba and CuPy to each other and our CUDA C implementation. The results show that CUDA C, as expected, has the fastest performance and highest energy efficiency, while Numba offers comparable performance when data movement is minimal. While CuPy offers ease of implementation, it performs slower for compute-heavy tasks.

Keywords: GPU; CUDA; Numba; CuPy; performance

check for updates

Citation: Askar, T.; Yergaliyev, A.; Shukirgaliyev, B.; Abdikamalov, E. Exploring Numba and CuPy for GPU-Accelerated Monte Carlo Radiation Transport. *Computation* **2024**, *12*, 61. https://doi.org/10.3390/computation12030061

Academic Editor: Demos T. Tsahalis

Received: 26 December 2023
Revised: 25 January 2024
Accepted: 31 January 2024
Published: 20 March 2024

Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

GPU accelerators have emerged as a powerful tool in many areas of science and engineering. They are used for accelerating not only traditional high-performance computing (HPC) applications (e.g., [1,2]) but also for the training and application of AI models (e.g., [3,4]). Many supercomputers in the top 500 list (https://www.top500.org (accessed on 15 November 2023)) obtain most of their computing power from these accelerators (e.g., see [5] for a recent review). Examples include the Aurora, Frontier, and Leonardo supercomputers that use Intel Max, AMD Instinct MI250X, and Nvidia A100 GPUs, respectively.

There are several programming platforms developed for harnessing the power of GPUs, ranging from low- to high-level paradigms. The low-level solutions offer the capability to optimize code for a given family of accelerators at the expense of portability. Popular examples are Nvidia CUDA [6] and the AMD ROCm HIP [7]. The high-level platforms offer ease of implementation and portability at the expense of limited options for optimization for specific hardware. Examples include OpenMP [8], OpenACC [9], OpenCL [10], SYCL [11], Alpaka [12], Numba [13,14], CuPy [15–17], Legate Numpy [18], Kokkos [19], RAJA [20], and OneAPI [21].

Many studies have explored how well GPUs perform in various applications [22–25]. These investigations compare the performance and highlight the strength and weaknesses of popular programming platforms such as CUDA C [26–28], CUDA Fortran [29–31], OpenCL [32–34], OpenACC [35,36], OpenMP [37,38] and Python-based compilers and libraries like Numba, CuPy, and Python CUDA [39–44]. Despite these efforts, a quest for a paradigm that offers simplicity of implementation and portability in combination with high performance remains one of the main goals of scientific computing (e.g., [45–48]).

Radiation transport is used in many areas of science and engineering, ranging from astronomy to nuclear physics (e.g., [49]). In the most general case, it represents a seven-dimensional problem (three dimensions for space, two dimensions for direction, one for energy, and one for time) [50]. This makes it particularly expensive to solve. One popular method for solving the equation is the Monte Carlo method (MC). This method directly simulates the behavior of radiation using random numbers. A group of radiation particles is modeled as an MC particle. MC radiative transfer (MCRT) methods are relatively easy to implement, especially when complicated physics and geometries are involved. The main drawback of the method is its immense computational cost [49].

It is possible to accelerate MCRT calculations using GPUs [51–56]. Since the MCRT utilizes a large number of MC particles that are evolved independently from each other (at least, with a timestep), the many-core architecture can evolve particles in parallel, leading to a significant speed-up compared to serial calculations [57–59]. There have been many studies and applications of GPU-accelerated MCRT [60–63]. The generation of random numbers on GPUs was investigated by, e.g., [64–66]. Bossler and Valdez [67] compared Kokkos and CUDA implementations of the main MCRT kernels. Hamilton et al. [68] compared history- and event-based MCRT algorithms. The domain decomposition parallelization techniques were explored by, e.g., [69,70]. Bleile et al. [71] explored the so-called "thin-threads" approach to history-based MCRT that achieves lighter memory usage, leading to faster performance. Humphrey et al. [72] implemented a reverse MC ray tracing approach and scaled it to 16,384 GPUs on the Titan supercomputer. Silvestri and Pecnik [73] ported the reciprocal MC algorithm to model radiative heat transfer in turbulent flows using the CUDA programming language, finding a significant speed-up compared to the CPU implementation. Heymann and Siebenmorgen [74] applied the GPU-based MC dust radiative method to active galactic nuclei. Ramon et al. [75] modeled radiation transport in ocean–atmosphere systems with the GPU-accelerated MC code. Their core code is written in CUDA, while the user interface is written in Python CUDA. Lee et al. [76] developed the GPU-accelerated code gCMCRT using CUDA Fortran to perform 3D modeling of exoplanet atmospheres. Several groups explored noise reduction in MCRT using machine learning techniques [77–80].

In this work, in the context of MC radiation transport, we assess the performance of two popular platforms that allow GPU computing from the Python programming language: Numba [14] and CuPy [16,17]. Both Numba and CuPy are examples of high-level popular platforms (e.g., [39,81]). Their main advantage is their simplicity. Numba is a just-in-time compiler for Python and enables faster performance by translating Python functions into optimized machine code, which is often competitive with manually optimized C code [14]. CuPy provides a simple NumPy-like interface with GPU acceleration [16,17]. Its compatibility with existing Python libraries and support for a wide range of mathematical operations have made it a popular choice for scientific computing tasks (e.g., [82,83]).

There are a number of works that study these platforms in various contexts. Di Domenico et al. [41] observed Numba achieving performance levels comparable to CUDA-based C++ using NAS parallel benchmark kernels. Oden [39] compared Numba with CUDA C, revealing the slower performance of Numba by 50–85% for compute-intensive benchmarks. Peng Xu et al. [84] compared Numba and CuPy in terms of data transfer speed and analyzed the influence of data type on the performance. They found that Numba outperforms CuPy for heavy data transfers, and single precision exhibits a 20% faster performance than double precision. Marowka [43] assessed the performance of Numba in matrix multiplication application. Dogaru R. and Dogaru I. [85] evaluated Numba in reaction–diffusion cellular non-linear networks. Azizi [86] utilized Numba and CuPy, among other Python-based platforms, to optimize expectation-maximization algorithms, yielding promising results. These related works set the foundations for our focused exploration of Numba and CuPy in the context of MCRT.

Our main aim was to study the potential of Numba and CuPy platforms for MCRT calculations. We performed a series of idealized test problems involving random number

generation and the one-dimensional MCRT problem in a purely absorbing medium. We performed a detailed analysis of the execution times for three different cards and single- and double-precision calculations. As a benchmark, we compared the results to computations implemented in CUDA C. To the best of our knowledge, there is no existing evaluation of the potential of Numba and CuPy specifically for MCRT problems. The novelty of our work lies in contributing valuable insights into the performance and suitability of these platforms for MCRT simulations.

This paper is organized as follows. In Section 2, we describe our methodology. In Section 3, we present our results. We discuss our findings in Section 4. Finally, in Section 5, we summarize the results and provide conclusions.

2. Methodology

We evaluated the performance of CuPy and Numba for the MC radiation transport problem. We considered two different idealized test problems. In the first problem, we generated pseudo-random numbers (PRNs). We stored the generated PRNs in the global memory, mimicking a scenario that involved heavy memory usage. In the second test problem, we considered a one-dimensional MC transport problem in a purely absorbing medium in a plane parallel geometry. This represented a compute-intensive test as it performed many arithmetic operations (e.g., logarithm, division, multiplication, etc.). The generated PRNs are immediately consumed within the kernel without storing them in the global memory. We compared our results to our implementation in CUDA C. Note that the 1D MCRT test problem was implemented with 15, 26, and 37 lines of code in CuPy, Numba, and CUDA C, respectively, reflecting the ease of implementation in CuPy and Numba relative to CUDA C [41].

We made the computations as similar as possible across the three platforms to ensure a fair comparison. We used the same seed and the same PRNG type `Xorshift`. More specifically, we used the `XORWOW` generator from the `cuRAND` library for the CuPy and CUDA C tests. Since Numba does not support the `cuRAND` library, we used its `Xoroshiro128p` PRNG, which belongs to the same `Xorshift` family. This ensured that all computing platforms were using similar algorithms, facilitating a fair performance comparison. Additionally, we tested the performance of these two generator algorithms and we found no significant difference between them. Therefore, the choice of PRNGs from the Xorshift family aligned to ensure fairness and comparability in the evaluation of PRN generation performance between CuPy, Numba, and CUDA C. Additionally, we ensured that our experiments utilized consistent values for constants and coefficients across all test problems. This guaranteed that any observed differences in performance could be attributed to the platforms themselves rather than variations in PRN generation. For Numba and CUDA C, we took the further step of applying a consistent configuration of grid size, specifying the number of blocks and threads, to efficiently map computations onto the GPU. However, it is important to note that CuPy used its default configuration, reflecting the inherent differences in how each platform manages its parallel computing resources.

Since Numba and CuPy used a just-in-time compilation process, functions were compiled at runtime when they were invoked. Consequently, the first invocation of a function included the compilation step, which can be considerably slower. To avoid this potential bottleneck, we excluded the timing of the first iteration. Additionally, we excluded the PRNG state setup time of CUDA C and CuPy from the GPU kernel performance time, since Numba's PRNG state setup took place on the host side. Also, unless specified explicitly, the Numba and CuPy platforms assigned double-precision types by default for all floating-point variables and constants. In this regard, we explicitly specified a type for single-precision calculations [39,81].

To assess the performance, we employed the profiling tools `nvprof` and `Nsight Systems`. We utilized versions 12.2.0, 0.58.1, 11.8.0, and 3.10.6 of CuPy, Numba, CUDA C, and Python, respectively. We performed 100 measurements for each method and recorded the average value to obtain reliable results. To measure power consumption, we used the

tool `nvidia-smi`. We ran each of our tests for 20 min and recorded the power consumption and GPU utilization parameters every 5 s. Then, we took the average value from these measurements.

We considered three distinct GPU cards: the A100 (in a DGX-A100 server), V100 (in a DGX-2 server), and RTX3080 card. We selected the A100 GPU because it is a relatively recent-generation Nvidia card for AI and HPC and our group had access to it. The V100 GPU is a predecessor to A100. Lastly, the RTX3080 GPU is an example of a high-end card in consumer space (see Table 1 for key specifications). These cards represent examples of devices that are currently in active use by the scientific community (At the time of preparing this work, we had no access to the latest-generation Nvidia H100 card. Similarly, we have no access to AMD or Intel cards. However, our comparison between A100, V100, and RTX3080 that we present below suggests that the qualitative differences between the three software platforms that we found should remain valid irrespective of the cards used).

Table 1. Technical specifications of GPU cards used in this work.

GPU Card	CUDA Cores	Base Clock [MHz]	FP32 (Float) [TFLOPS]	FP64 (Double) [TFLOPS]	Bandwidth [GB/s]
A100	6912	765	19.49	9.746	1555
V100	5120	1230	14.13	7.066	897
RTX3080	8704	1440	29.77	0.465	760

Unless otherwise noted, all results presented in Section 3 were obtained using the A100 GPU card. However, for the purpose of comparison, we included the results for the V100 and RTX3080 GPUs in Section 3.3. All computations were performed in single precision, except in Section 3.1, where we compared single- and double-precision calculations.

3. Results

Figure 1 (left panel) shows a performance comparison of CuPy, Numba, and CUDA C for the PRN generation problem. The x-axis shows the number of generated PRNs. The solid lines represent the total execution time from the beginning until the end of the computation, including computations on the host and GPU side. The dashed lines show the GPU kernel execution time, which only measures the time on the GPU. In terms of the total execution time, all platforms performed at the same level for $N \leq 10^6$. For larger N, Numba demonstrated slower performance than CuPy and CUDA C, e.g., by 1.87x and 3.22x at $N = 2 \times 10^9$, respectively. CuPy performed similarly compared to CUDA C up to $N = 10^8$. However, for a larger N, CUDA C outperformed CuPy by 1.72x at $N = 2 \times 10^9$. When comparing the performance of GPU kernels, CUDA C was faster than Numba and CuPy by 1.21x and 1.67x even at $N = 10^4$. The performance gap widened with increasing N and reached about 22x and 7.8x compared to Numba and CuPy for $N = 2 \times 10^9$. These findings are in line with those of previous studies (e.g., [39,41]).

Figure 1 (right panel) shows a performance comparison of CuPy, Numba, and CUDA C for the 1D MCRT problem. In terms of the total execution time (solid lines), for loads $N < 10^8$, CuPy outperformed Numba and CUDA C. At $N = 10^6$, the difference in performance reached ∼4.72x and ∼3.06x, respectively. As the workload increased ($N > 10^8$), CUDA C became faster. For $N = 2 \times 10^9$, CUDA C was faster by 5.78x and 5.24x as compared to Numba and CuPy. When comparing GPU kernels (dashed lines), CuPy was slower than both Numba and CUDA C. Specifically, it was already 8.5 times slower than CUDA C for $N = 10^4$, and the difference reached 14.2x for $N = 2 \times 10^9$. However, Numba demonstrated competitive performance with CUDA C. The slowdown was only 1.53x at $N = 2 \times 10^9$. We found that this was mainly caused by two factors: (1) the difference in PRN algorithms used in these two platforms (cf. Section 2); (2) the `log` function is executed in double precision (cf. Section 3.3). If we used the same PRNG, the difference between Numba and CUDA C became negligible for this test.

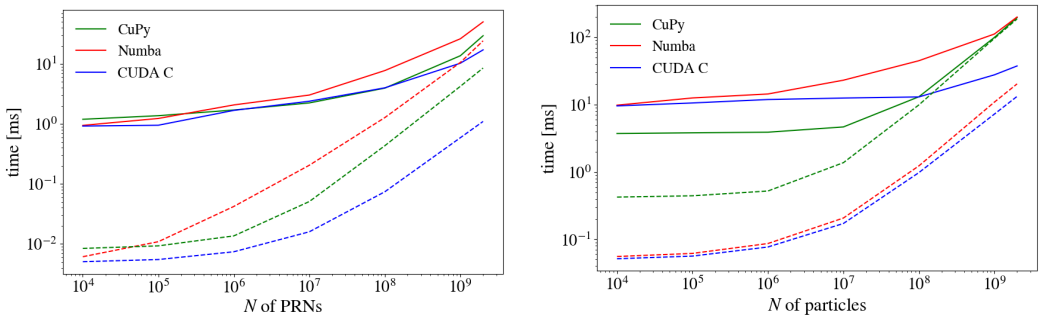

Figure 1. Execution time as a function of the number of PRNs/particles for CuPy, Numba, and CUDA C using the A100 GPU card. The left panel represents the PRN generation test problem, while the right panel is for the 1D MCRT test. The solid lines show the total execution time and the dashed lines correspond to the GPU kernel times.

The similar performances of Numba and CUDA C for the 1D MCRT problem and their drastic differences in the PRN generation problem point to the reason behind this behavior. In the former problem, where Numba performed relatively slowly, the PRN values are stored in the GPU global memory. In the 1D MCRT problem, PRNs are generated and then used locally. Thus, Numba's performance was fast (slow) when small (large) data movement was involved [39].

The faster execution time for lower loads (up to $N \leq 10^8$) exhibited by CuPy in the 1D MCRT test can be attributed to the utilization of memory spaces by reserving memory blocks [15]. This reduces the overheads associated with memory allocation and CPU/GPU synchronization due to the "caching" of allocated memory blocks. Since the 1D MCRT problem involves numerous memory allocations, CuPy achieved better overall performance compared to Numba and CUDA C for $N \leq 10^8$. However, in GPU kernel time, CuPy's performance was slower due to the temporary storing of the calculated values in the global memory after each calculation.

3.1. Impact of Precision

In this section, we study the impact of precision by performing computations in single and double precision. For a similar investigation in the context of the finite-difference approach to Burgers' equation, see [81]. Extensive tests of Numba and CUDA C for matrix–matrix multiplication, parallel reduction, and 3D stencil application in single and double precisions can be found in [39].

Figure 2 (left panel) shows the GPU kernel execution time for the PRN generation problem using CuPy, Numba, and CUDA C for single and double precisions, as a function of N. As expected, the CUDA C single-precision calculations were faster than the double-precision calculations. The performance difference reached 1.17x at $N = 10^6$. As N increased, the gap widened, reaching 1.77x at $N = 2 \times 10^9$. These results are in line with those of previous studies on the impact of precision for CUDA C [87,88]. The same performance pattern was observed in CuPy. This was expected behavior since its PRN generation is based on CUDA's cuRAND library [15,89].

For Numba, the single precision was slower for $N \leq 6 \times 10^7$ and faster for larger N by about ∼12.5%. The reason for this behavior is a combination of two competing factors. On the one hand, Numba generates PRNs in double precision by default (this can be established by analyzing the `Nsight Systems` profiling tool and examining the PTX assembly code). Conversion to single precision is then performed, introducing an additional overhead. On the other hand, it is faster to write single-precision values to the global memory than double-precision variables. For this reason, single-precision calculations become competitive with (and faster than) double-precision calculations for large N.

Figure 2 (right panel) shows the execution time of the GPU kernel for the single- and double-precision 1D MCRT test problem as a function of N. For CuPy, we observed a

similar performance in both precisions up to $N = 10^6$. For larger N, the performance difference only reached 1.43x at $N = 10^9$. This small difference is explained by excessive data movement to/from the global memory for temporary storage after each calculation.

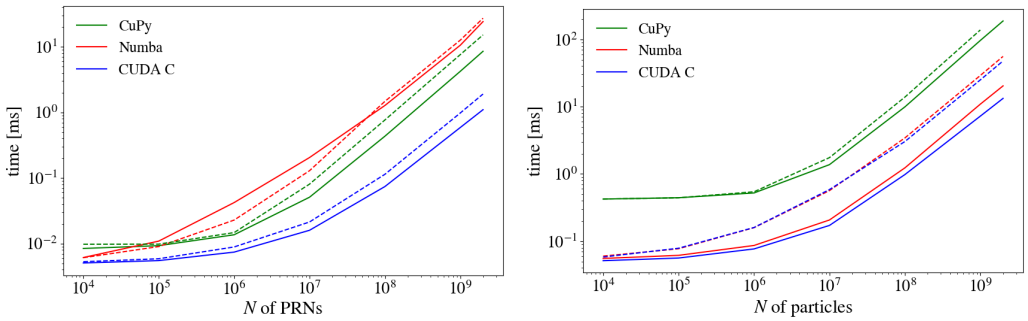

Figure 2. Execution times of GPU kernels as a function of the number of PRNs/particles for CuPy, Numba, and CUDA C using the A100 GPU card. The left panel represents the PRN generation test problem, while the right panel is for the 1D MCRT test problem. The solid lines correspond to single-precision calculations and the dashed lines show double-precision results.

In the case of Numba, the double precision was slower than the single precision for all values of N. Numba converts the double-precision PRN to a single-precision PRN, but unlike CuPy, it does this operation without temporarily storing the data in the global memory after each calculation, which is more efficient. For example, for $N = 10^4$, the difference between the two precisions was 1.08x, while for $N = 2 \times 10^9$, the difference rose to 2.74x.

3.2. Energy Consumption

Table 2 shows the average energy spent to generate a single PRN (or to track a single particle in the 1D MCRT problem), the power consumption, and GPU utilization. These results were obtained for $N = 10^9$ for the A100 card. We can see that Numba was more energy efficient than CuPy. However, CUDA C was more efficient than both Numba and CuPy. For the PRN generation test, CUDA C spent less energy per PRN than Numba and CuPy by a factor of 2.1 and 2.4, respectively. In the case of 1D MCRT, CUDA C spent less energy by approximately 3.7x than Numba and 5.1x than CuPy. CUDA C is energy efficient compared to Numba and CuPy due to its optimization for the GPU architecture and the efficient utilization of parallel computing capabilities. The optimized utilization of GPU resources minimizes idle times, allowing CUDA C to accomplish more computations per unit of energy. Unlike CUDA C, Numba and CuPy, being just-in-time frameworks for Python, introduce additional overheads and limitations that impact energy efficiency.

Table 2. Average energy per particle, power usage, and GPU utilization. The values were obtained by generating 10^9 PRNs for the PRN generation test and by tracking 10^9 particles for the 1D MCRT test using the A100 GPU card. See Section 2 for details of the measurement method.

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

In terms of power consumption, CUDA C used 7% less power than Numba and 171% less power than CuPy in the PRN generation test. However, for the 1D MCRT problem (see Table 2), Numba used less power by 84% and 225% than CUDA C and CuPy, respectively.

This is attributed to the fact that Numba utilizes the GPU to a lesser extent than CUDA C. For Numba, the PRNG state setup takes place on the host CPU, which is then transferred to the GPU. The wait times associated with this transfer limit the GPU utilization. The high power consumption of CuPy can be explained by excessive data movement [90] due to automatic memory management [15]. The power consumption variations among CuPy, Numba, and CUDA C have implications for practical applications. CUDA C's emphasis on GPU parallelization and optimization generally leads to higher computational performance and energy efficiency, making it well-suited for applications with time-sensitive results and power constraints [91,92]. Numba and CuPy, while providing relatively easy implementation of GPU applications, introduce overheads that impact performance, and their scalability and energy efficiency are comparatively lower.

3.3. Comparison of GPU Cards

In this section, we compare the performance of three GPU cards using the 1D MCRT test problem. For a similar comparison in the context of PRN generation in CUDA C, see, e.g., [93].

Figure 3 (left panel) shows the execution time for Numba. The PTX assembly code of Numba reveals that, despite explicit specification of all variables and constants as single-precision type, the `log` function is executed in double precision. This introduces overheads for the RTX3080 GPU card, which can perform double-precision operations 64 times slower than in single precision. For the A100 and V100, the ratio is 1:2 (see Table 1). Consequently, Numba was far slower on RTX3080 than on A100 and V100. The performance gap reached $\sim$6.7x for $N = 2 \times 10^9$ particles.

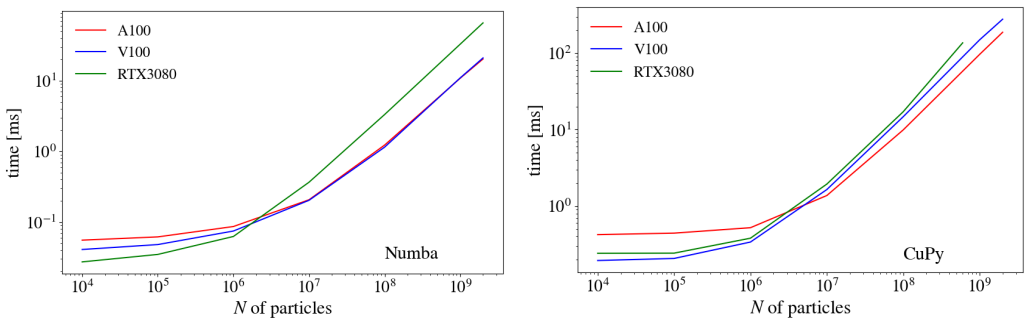

Figure 3. Execution time of GPU kernel as a function of the number of particles for Numba and CuPy using three GPU cards for the 1D MCRT test problem. The left panel shows the results for Numba, while the right panel is for CuPy.

Figure 3 (right panel) shows the results for CuPy. For $N < 3 \times 10^6$, CuPy on RTX3080 and V100 performed faster than on A100. However, at $N \geq 3 \times 10^6$, A100 was faster than V100 and RTX3080, reaching speed-ups of 1.49x and 1.72x for $N = 10^8$, respectively. The slower performance of CuPy on RTX3080 for $N \geq 3 \times 10^6$ is caused by the memory bandwidth, which is less than on A100 and V100 by 105% and 18%, respectively. The better performance of the A100 compared to the V100 at $N \geq 3 \times 10^6$ can also be attributed to the higher memory bandwidth of A100, i.e., by 73% (see Table 1). In the context of CuPy, this factor significantly contributes to the overall performance.

Table 3 shows the power consumption of three GPU cards for the PRN generation and 1D MCRT test problems. The A100 GPU card used less power than V100, which, in turn, used less power than RTX3080. For example, in CUDA C, A100 consumed less power than V100 and RTX3080 by an average of 26.8% and 64.7%, respectively. In the context of Numba, A100 was more efficient by an average of 52% than V100 and 105.2% than RTX3080. For CuPy, A100 used less power by an average of 3% and 38.3% than V100 and RTX3080, respectively.

Table 3. Power consumption while generating 10^9 PRNs in the PRN generation problem and tracking 10^9 particles in the 1D MCRT problem for the A100, V100, and RTX3080 GPU cards.

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

4. Discussion

Our research work evaluates the performance of GPU programming platforms, specifically CuPy and Numba, within the context of MCRT. By addressing two distinct test problems and comparing outcomes with the CUDA C implementation, we highlight trade-offs between performance and ease of implementation across these platforms. We explored variations in performance for a comprehensive set of test problems, ranging from memory intensive to compute intensive tasks. We also looked at the impact of the precision by considering calculations in single and double precision (see Section 3.1 for more details). Further evaluations involved comparing the performance of various NVIDIA GPU cards. Conducting these comparisons helped us to gain a better understanding of how efficiently each platform performs across different hardware configurations (see Section 3.3). The limitations that we identify suggest avenues for future research. In addition, our findings can serve as a guide for choosing various computational methods for more complicated simulations covering multiple spatial dimensions and complex geometries. Our study contributes valuable insights concerning the strengths and limitations of the GPU programming platforms CuPy and Numba, helping researchers make decisions for optimal performance in MCRT simulations.

5. Conclusions

We investigated the performance of two prominent GPU programming platforms, CuPy and Numba, in the context of the Monte Carlo (MC) radiation transport problem. We considered two idealized test problems. In the first problem, we performed random number generation with subsequent storage of the numbers in the global memory. This problem represented the scenario with heavy memory usage. In the second problem, we considered a one-dimensional MC radiation transport test in a purely absorbing medium. We compared the results to our implementation in CUDA C. We considered three different cards: NVIDIA Telsa A100, Tesla V100, and GeForce RTX3080.

The results show that, overall, CUDA C had the fastest performance and highest energy efficiency. However, Numba demonstrated competitive performance to CUDA C when the data movement was small. In contrast, when data movement was heavy (e.g., storing PRNs in the global memory), Numba was significantly slower than CUDA C. CuPy showed better performance than Numba in the PRN generation test problem due to its efficient memory allocation. However, for compute-heavy tasks (in the case of the 1D MCRT test problem), it lagged behind these platforms substantially. Also, it showed less energy efficiency. The slower performance of CuPy was somewhat compensated by the relative ease of implementation within this platform. These outcomes highlighted the trade-offs between performance and ease of implementation across the programming platforms considered in this work.

The main limitation of our work was that we used an idealized 1D MC radiation transport test in a purely absorbing and static medium. Realistic calculations often take place in multiple spatial dimensions. In general, matter can be non-static. In addition to absorption, matter can emit and scatter radiation. Such scenarios will be considered in future work. Moreover, other platforms that enable GPU computing within the Python

language will also be considered. Multiple GPU configurations and GPUs from other vendors will also be explored.

Author Contributions: T.A. wrote the main manuscript, programming codes, prepared figures, and tables. A.Y. verified the Numba code and reviewed and edited the manuscript. B.S. reviewed and edited the manuscript, provided technical support regarding GPU hardware and software. E.A. reviewed and edited the manuscript, supervised the research work. All authors have read and agreed to the published version of the manuscript.

Funding: This research is supported by the Science Committee of the Ministry of Science and Higher Education of the Republic of Kazakhstan, under Grants No. AP19677351 and AP13067834. Additionally, funding is provided through the Nazarbayev University Faculty Development Competitive Research Grant Program, with Grant No. 11022021FD2912. Furthermore, this work is supported by the Aerospace Committee of the Ministry of Digital Development, Innovations and Aerospace Industry of the Republic of Kazakhstan, through Grant No. BR21881880.

Data Availability Statement: The source codes can be found at the following GitHub repository for reproducing the results: https://github.com/tairaskar/CuPy-Numba-CUDA-C (accessed on 26 November 2023).

Acknowledgments: The authors acknowledge the Institute of Smart Systems and Artificial Intelligence at Nazarbayev University for granting access to the DGX servers (issai.nu.edu.kz).

Conflicts of Interest: The authors declare that there are no competing interests associated with the research presented in this paper.

References

1. Abdelfattah, A.; Barra, V.; Beams, N.; Bleile, R.; Brown, J.; Camier, J.S.; Carson, R.; Chalmers, N.; Dobrev, V.; Dudouit, Y.; et al. GPU algorithms for efficient exascale discretizations. *Parallel Comput.* **2021**, *108*, 102841. [CrossRef]
2. Pazner, W.; Kolev, T.; Camier, J.S. End-to-end GPU acceleration of low-order-refined preconditioning for high-order finite element discretizations. *Int. J. High Perform. Comput. Appl.* **2023**, *37* , 10943420231175462. [CrossRef]
3. Hu, Y.; Liu, Y.; Liu, Z. A Survey on Convolutional Neural Network Accelerators: GPU, FPGA and ASIC. In Proceedings of the 2022 14th International Conference on Computer Research and Development (ICCRD), Shenzhen, China, 7–9 January 2022; pp. 100–107. [CrossRef]
4. Pandey, M.; Fernandez, M.; Gentile, F.; Isayev, O.; Tropsha, A.; Stern, A.C.; Cherkasov, A. The transformational role of GPU computing and deep learning in drug discovery. *Nat. Mach. Intell.* **2022**, *4*, 211–221. [CrossRef]
5. Matsuoka, S.; Domke, J.; Wahib, M.; Drozd, A.; Hoefler, T. Myths and legends in high-performance computing. *Int. J. High Perform. Comput. Appl.* **2023**, *37*, 245–259. [CrossRef]
6. CUDA Best Practices. Available online: https://developer.nvidia.com/cuda-best-practices (accessed on 8 August 2023).
7. AMD ROCm HIP Documentation. Available online: https://rocm.docs.amd.com/projects/HIP/en/latest/ (accessed on 10 December 2023).
8. Martineau, M.; McIntosh-Smith, S.; Gaudin, W. Evaluating OpenMP 4.0's Effectiveness as a Heterogeneous Parallel Programming Model. In Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Chicago, IL, USA, 23–27 May 2016; pp. 338–347. [CrossRef]
9. Wienke, S.; Springer, P.; Terboven, C.; an Mey, D. OpenACC—First Experiences with Real-World Applications. In Proceedings of the Euro-Par 2012 Parallel Processing, Rhodos, Greece, 27–31 August 2012; Kaklamanis, C., Papatheodorou, T., Spirakis, P.G., Eds.; Springer : Berlin/Heidelberg, Germany, 2012; pp. 859–870.
10. Stone, J.E.; Gohara, D.; Shi, G. OpenCL: A parallel programming standard for heterogeneous computing systems. *Comput. Sci. Eng.* **2010**, *12*, 66. [CrossRef] [PubMed]
11. Reyes, R.; Lomüller, V. SYCL: Single-source C++ accelerator programming. In Proceedings of the International Conference on Parallel Computing, Edinburgh, UK, 1–4 September 2015.
12. Zenker, E.; Worpitz, B.; Widera, R.; Huebl, A.; Juckeland, G.; Knupfer, A.; Nagel, W.E.; Bussmann, M. Alpaka–An Abstraction Library for Parallel Kernel Acceleration. In Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Chicago, IL, USA, 23–27 May 2016; pp. 631–640. [CrossRef]
13. Numba Developers . *Numba Documentation*. Numba. 2023. Available online: https://numba.pydata.org/ (accessed on 8 December 2023).
14. Lam, S.K.; Pitrou, A.; Seibert, S. Numba: A LLVM-Based Python JIT Compiler. In Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC, Austin, TX, USA, 15 November 2015; LLVM'15. [CrossRef]
15. CuPy Developers. CuPy Documentation. 2023. Available online: https://docs.cupy.dev/en/stable/ (accessed on 30 January 2024).

16. Okuta, R.; Unno, Y.; Nishino, D.; Hido, S.; Loomis, C. Cupy: A numpy-compatible library for nvidia gpu calculations. In Proceedings of the Workshop on Machine Learning Systems (LearningSys) in the Thirty-First Annual Conference on Neural Information Processing Systems (NIPS), Long Beach, CA, USA, 4–9 December 2017; Volume 6.
17. Nishino, R.; Loomis, S.H.C. Cupy: A numpy-compatible library for nvidia gpu calculations. In Proceedings of the 31st Confernce on Neural Information Processing Systems, Long Beach, CA, USA, 4–9 December 2017; Volume 151.
18. Bauer, M.; Garland, M. Legate NumPy: Accelerated and Distributed Array Computing. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, Denver, CO, USA, 17–19 November 2019; SC'19. [CrossRef]
19. Trott, C.R.; Lebrun-Grandié, D.; Arndt, D.; Ciesko, J.; Dang, V.; Ellingwood, N.; Gayatri, R.; Harvey, E.; Hollman, D.S.; Ibanez, D.; et al. Kokkos 3: Programming Model Extensions for the Exascale Era. *IEEE Trans. Parallel Distrib. Syst.* **2022**, *33*, 805–817. [CrossRef]
20. Hornung, R.D.; Keasler, J.A. *The RAJA Portability Layer: Overview and Status*; Lawrence Livermore National Lab.(LLNL): Livermore, CA, USA, 2014.
21. Fortenberry, A.; Tomov, S. Extending MAGMA Portability with OneAPI. In Proceedings of the 2022 Workshop on Accelerator Programming Using Directives (WACCPD), Dallas, TX, USA, 13–18 November 2022; IEEE: Piscataway, NJ, USA , 2022; pp. 22–31.
22. Lim, S.; Kang, P. Implementing scientific simulations on GPU-accelerated edge devices. In Proceedings of the 2020 International Conference on Information Networking (ICOIN), Barcelona, Spain, 7–10 January 2020; IEEE: Piscataway, NJ, USA, 2020; pp. 756–760.
23. Knight, J.C.; Nowotny, T. Larger GPU-accelerated brain simulations with procedural connectivity. *Nat. Comput. Sci.* **2021**, *1*, 136–142. [CrossRef] [PubMed]
24. Aydonat, U.; O'Connell, S.; Capalija, D.; Ling, A.C.; Chiu, G.R. An OpenCL™ Deep Learning Accelerator on Arria 10. In Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, 22–24 February 2017; FPGA'17; pp. 55–64. [CrossRef]
25. Kalaiselvi, T.; Sriramakrishnan, P.; Somasundaram, K. Survey of using GPU CUDA programming model in medical image analysis. *Informatics Med. Unlocked* **2017**, *9*, 133–144. [CrossRef]
26. Kuan, L.; Neves, J.; Pratas, F.; Tomás, P.; Sousa, L. Accelerating Phylogenetic Inference on GPUs: An OpenACC and CUDA comparison. In Proceedings of the IWBBIO, Granada, Spain, 7–9 April 2014; pp. 589–600.
27. Christgau, S.; Spazier, J.; Schnor, B.; Hammitzsch, M.; Babeyko, A.; Waechter, J. A comparison of CUDA and OpenACC: Accelerating the tsunami simulation easywave. In Proceedings of the ARCS 2014; 2014 Workshop Proceedings on Architecture of Computing Systems, Luebeck, Germany, 25–28 February 2014; VDE: Frankfurt am Main, Germany; Springer: Cham, Switzerland, 2014; pp. 1–5.
28. Memeti, S.; Li, L.; Pllana, S.; Kołodziej, J.; Kessler, C. 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, Washington, DC, USA, 28 July 2017; pp. 1–6.
29. Cloutier, B.; Muite, B.K.; Rigge, P. Performance of FORTRAN and C GPU extensions for a benchmark suite of Fourier pseudospectral algorithms. In Proceedings of the 2012 Symposium on Application Accelerators in High Performance Computing, Argonne, IL, USA, 10–11 July 2012; IEEE: Piscataway, NJ, USA, 2012; pp. 145–148.
30. Herdman, J.; Gaudin, W.; McIntosh-Smith, S.; Boulton, M.; Beckingsale, D.A.; Mallinson, A.; Jarvis, S.A. Accelerating hydrocodes with OpenACC, OpenCL and CUDA. In Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis, Salt Lake City, UT, USA, 10–16 November 2012; IEEE: Piscataway, NJ, USA, 2012; pp. 465–471.
31. Satake, S.I.; Yoshimori, H.; Suzuki, T. Optimizations of a GPU accelerated heat conduction equation by a programming of CUDA Fortran from an analysis of a PTX file. *Comput. Phys. Commun.* **2012**, *183*, 2376–2385. [CrossRef]
32. Malik, M.; Li, T.; Sharif, U.; Shahid, R.; El-Ghazawi, T.; Newby, G. Productivity of GPUs under different programming paradigms. *Concurr. Comput. Pract. Exp.* **2012**, *24*, 179–191. [CrossRef]
33. Karimi, K.; Dickson, N.G.; Hamze, F. A performance comparison of CUDA and OpenCL. *arXiv* **2010**, arXiv:1005.2581.
34. Fang, J.; Varbanescu, A.L.; Sips, H. A comprehensive performance comparison of CUDA and OpenCL. In Proceedings of the 2011 International Conference on Parallel Processing, Taipei, Taiwan, 13–16 September 2011; IEEE: Piscataway, NJ, USA, 2011; pp. 216–225.
35. Li, X.; Shih, P.C. An early performance comparison of CUDA and OpenACC. *Matec Web Conf.* **2018**, *208*, 05002. [CrossRef]
36. Hoshino, T.; Maruyama, N.; Matsuoka, S.; Takaki, R. CUDA vs OpenACC: Performance case studies with kernel benchmarks and a memory-bound CFD application. In Proceedings of the 2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, Delft, The Netherlands, 13–16 May 2013; IEEE: Piscataway, NJ, USA, 2013; pp. 136–143.
37. Gimenes, T.L.; Pisani, F.; Borin, E. Evaluating the performance and cost of accelerating seismic processing with cuda, opencl, openacc, and openmp. In Proceedings of the 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS), Vancouver, BC, Canada, 21–25 May 2018; IEEE: Piscataway, NJ, USA, 2018; pp. 399–408.
38. Guo, X.; Wu, J.; Wu, Z.; Huang, B. Parallel computation of aerial target reflection of background infrared radiation: Performance comparison of OpenMP, OpenACC, and CUDA implementations. *IEEE J. Sel. Top. Appl. Earth Obs. Remote Sens.* **2016**, *9*, 1653–1662. [CrossRef]

39. Oden, L. Lessons learned from comparing C-CUDA and Python-Numba for GPU-Computing. In Proceedings of the 2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), Västerås, Sweden, 11–13 March 2020; IEEE: Piscataway, NJ, USA, 2020; pp. 216–223.
40. Godoy, W.F.; Valero-Lara, P.; Dettling, T.E.; Trefftz, C.; Jorquera, I.; Sheehy, T.; Miller, R.G.; Gonzalez-Tallada, M.; Vetter, J.S.; Churavy, V. Evaluating performance and portability of high-level programming models: Julia, Python/Numba, and Kokkos on exascale nodes. *arXiv* **2023**, arXiv:2303.06195.
41. Di Domenico, D.; Lima, J.V.; Cavalheiro, G.G. NAS Parallel Benchmarks with Python: A performance and programming effort analysis focusing on GPUs. *J. Supercomput.* **2023**, *79*, 8890–8911. [CrossRef]
42. Holm, H.H.; Brodtkorb, A.R.; Sætra, M.L. GPU computing with Python: Performance, energy efficiency and usability. *Computation* **2020**, *8*, 4. [CrossRef]
43. Marowka, A. Python accelerators for high-performance computing. *J. Supercomput.* **2018**, *74*, 1449–1460. [CrossRef]
44. Boytsov, A.; Kadochnikov, I.; Zuev, M.; Bulychev, A.; Zolotuhin, Y.; Getmanov, I. 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, 10–14 September 2018; pp. 518–522.
45. Bhattacharya, M.; Calafiura, P.; Childers, T.; Dewing, M.; Dong, Z.; Gutsche, O.; Habib, S.; Ju, X.; Ju, X.; Kirby, M.; et al. Portability: A Necessary Approach for Future Scientific Software. In Proceedings of the Snowmass 2021, Seattle, WA, USA, 17–26 July 2022. [CrossRef]
46. Ma, Z.X.; Jin, Y.Y.; Tang, S.Z.; Wang, H.J.; Xue, W.C.; Zhai, J.D.; Zheng, W.M. Unified Programming Models for Heterogeneous High-Performance Computers. *J. Comput. Sci. Technol.* **2023**, *38*, 211–218. [CrossRef]
47. Thavappiragasam, M.; Elwasif, W.; Sedova, A. Portability for GPU-accelerated molecular docking applications for cloud and HPC: Can portable compiler directives provide performance across all platforms? In Proceedings of the 2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid), Taormina, Italy, 16–19 May 2022; pp. 975–984. [CrossRef]
48. Deakin, T.; Cownie, J.; Lin, W.C.; McIntosh-Smith, S. Heterogeneous Programming for the Homogeneous Majority. In Proceedings of the 2022 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), Dallas, TX, USA, 13–18 November 2022; pp. 1–13. [CrossRef]
49. Noebauer, U.M.; Sim, S.A. Monte Carlo radiative transfer. *Living Rev. Comput. Astrophys.* **2019**, *5*, 1. [CrossRef]
50. Castor, J.I. *Radiation Hydrodynamics*; Cambridge University Press: Cambridge, UK, 2004.
51. Alerstam, E.; Svensson, T.; Andersson-Engels, S. Parallel computing with graphics processing units for high-speed Monte Carlo simulation of photon migration. *J. Biomed. Opt.* **2008**, *13*, 060504. [CrossRef]
52. Badal, A.; Badano, A. Monte Carlo simulation of X-ray imaging using a graphics processing unit. In Proceedings of the 2009 IEEE Nuclear Science Symposium Conference Record (NSS/MIC), Orlando, FL, USA, 24 October–1 November 2009; pp. 4081–4084. [CrossRef]
53. Huang, B.; Mielikainen, J.; Oh, H.; Allen Huang, H.L. Development of a GPU-based high-performance radiative transfer model for the Infrared Atmospheric Sounding Interferometer (IASI). *J. Comput. Phys.* **2011**, *230*, 2207–2221. [CrossRef]
54. Lippuner, J.; Elbakri, I.A. A GPU implementation of EGSnrc's Monte Carlo photon transport for imaging applications. *Phys. Med. Biol.* **2011**, *56*, 7145. [CrossRef]
55. Bergmann, R.M.; Vujić, J.L. Algorithmic choices in WARP—A framework for continuous energy Monte Carlo neutron transport in general 3D geometries on GPUs. *Ann. Nucl. Energy* **2015**, *77*, 176–193. [CrossRef]
56. Zoller, C.; Hohmann, A.; Foschum, F.; Geiger, S.; Geiger, M.; Ertl, T.P.; Kienle, A. Parallelized Monte Carlo Software to Efficiently Simulate the Light Propagation in Arbitrarily Shaped Objects and Aligned Scattering Media. *J. Biomed. Opt.* **2018**, *23*, 065004. [CrossRef] [PubMed]
57. Jia, X.; Gu, X.; Graves, Y.J.; Folkerts, M.; Jiang, S.B. GPU-based fast Monte Carlo simulation for radiotherapy dose calculation. *Phys. Med. Biol.* **2011**, *56*, 7017. [CrossRef] [PubMed]
58. Hissoiny, S.; Ozell, B.; Bouchard, H.; Després, P. GPUMCD: A new GPU-oriented Monte Carlo dose calculation platform. *Med. Phys.* **2011**, *38*, 754–764. [CrossRef]
59. Shao, J.; Zhu, K.; Huang, Y. A fast GPU Monte Carlo implementation for radiative heat transfer in graded-index media. *J. Quant. Spectrosc. Radiat. Transf.* **2021**, *269*, 107680. [CrossRef]
60. Young-Schultz, T.; Brown, S.; Lilge, L.; Betz, V. FullMonteCUDA: A fast, flexible, and accurate GPU-accelerated Monte Carlo simulator for light propagation in turbid media. *Biomed. Opt. Express* **2019**, *10*, 4711–4726. [CrossRef]
61. Ma, B.; Gaens, M.; Caldeira, L.; Bert, J.; Lohmann, P.; Tellmann, L.; Lerche, C.; Scheins, J.; Rota Kops, E.; Xu, H.; et al. Scatter Correction Based on GPU-Accelerated Full Monte Carlo Simulation for Brain PET/MRI. *IEEE Trans. Med. Imaging* **2020**, *39*, 140–151. [CrossRef]
62. Ma, D.; Yang, B.; Zhang, Q.; Liu, J.; Li, T. Evaluation of Single-Node Performance of Parallel Algorithms for Multigroup Monte Carlo Particle Transport Methods. *Front. Energy Res.* **2021**, *9*, 705823. [CrossRef]
63. Shi, M.; Myronakis, M.; Jacobson, M.; Ferguson, D.; Williams, C.; Lehmann, M.; Baturin, P.; Huber, P.; Fueglistaller, R.; Lozano, I.V.; et al. GPU-accelerated Monte Carlo simulation of MV-CBCT. *Phys. Med. Biol.* **2020**, *65*, 235042. [CrossRef] [PubMed]
64. Manssen, M.; Weigel, M.; Hartmann, A.K. Random number generators for massively parallel simulations on GPU. *Eur. Phys. J. Spec. Top.* **2012**, *210*, 53–71. [CrossRef]

65. L'Écuyer, P.; Munger, D.; Oreshkin, B.; Simard, R. *Random Numbers for Parallel Computers: Requirements and Methods, with Emphasis on Gpus*; GERAD, HEC Montréal: Montréal, QC, Canada, 2015.
66. Kim, Y.; Hwang, G. Efficient Parallel CUDA Random Number Generator on NVIDIA GPUs. *J. Kiise* **2015**, *42*, 1467–1473. [CrossRef]
67. Bossler, K.; Valdez, G.D. Comparison of Kokkos and CUDA Programming Models for Key Kernels in the Monte Carlo Transport Algorithm. In Proceedings of the Nuclear Explosives Code Development Conference (NECDC) 2018, Los Alamos, NM, USA, 15–19 October 2018; Technical Report; Sandia National Lab.(SNL-NM): Albuquerque, NM, USA, 2018.
68. Hamilton, S.P.; Slattery, S.R.; Evans, T.M. Multigroup Monte Carlo on GPUs: Comparison of history- and event-based algorithms. *Ann. Nucl. Energy* **2018**, *113*, 506–518. [CrossRef]
69. Choi, N.; Joo, H.G. Domain decomposition for GPU-Based continuous energy Monte Carlo power reactor calculation. *Nucl. Eng. Technol.* **2020**, *52*, 2667–2677. [CrossRef]
70. Hamilton, S.P.; Evans, T.M.; Royston, K.E.; Biondo, E.D. Domain decomposition in the GPU-accelerated Shift Monte Carlo code. *Ann. Nucl. Energy* **2022**, *166*, 108687. [CrossRef]
71. Bleile, R.; Brantley, P.; Richards, D.; Dawson, S.; McKinley, M.S.; O'Brien, M.; Childs, H. Thin-Threads: An Approach for History-Based Monte Carlo on GPUs. In Proceedings of the 2019 International Conference on High Performance Computing & Simulation (HPCS), Dublin, Ireland, 15–19 July 2019; pp. 273–280. [CrossRef]
72. Humphrey, A.; Sunderland, D.; Harman, T.; Berzins, M. Radiative Heat Transfer Calculation on 16384 GPUs Using a Reverse Monte Carlo Ray Tracing Approach with Adaptive Mesh Refinement. In Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Chicago, IL, USA, 23–27 May 2016; pp. 1222–1231. [CrossRef]
73. Silvestri, S.; Pecnik, R. A fast GPU Monte Carlo radiative heat transfer implementation for coupling with direct numerical simulation. *J. Comput. Phys. X* **2019**, *3*, 100032. [CrossRef]
74. Heymann, F.; Siebenmorgen, R. GPU-based Monte Carlo Dust Radiative Transfer Scheme Applied to Active Galactic Nuclei. *Astrophys. J.* **2012**, *751*, 27. [CrossRef]
75. Ramon, D.; Steinmetz, F.; Jolivet, D.; Compiègne, M.; Frouin, R. Modeling polarized radiative transfer in the ocean-atmosphere system with the GPU-accelerated SMART-G Monte Carlo code. *J. Quant. Spectrosc. Radiat. Transf.* **2019**, *222-223*, 89–107. [CrossRef]
76. Lee, E.K.H.; Wardenier, J.P.; Prinoth, B.; Parmentier, V.; Grimm, S.L.; Baeyens, R.; Carone, L.; Christie, D.; Deitrick, R.; Kitzmann, D.; et al. 3D Radiative Transfer for Exoplanet Atmospheres. gCMCRT: A GPU-accelerated MCRT Code. *Astrophys. J.* **2022**, *929*, 180. [CrossRef]
77. Peng, Z.; Shan, H.; Liu, T.; Pei, X.; Wang, G.; Xu, X.G. 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* **2019**, *7*, 76680–76689. [CrossRef]
78. Ardakani, M.R.; Yu, L.; Kaeli, D.; Fang, Q. Framework for denoising Monte Carlo photon transport simulations using deep learning. *J. Biomed. Opt.* **2022**, *27*, 083019. [CrossRef]
79. van Dijk, R.H.W.; Staut, N.; Wolfs, C.J.A.; Verhaegen, F. A novel multichannel deep learning model for fast denoising of Monte Carlo dose calculations: Preclinical applications. *Phys. Med. Biol.* **2022**, *67*, 164001. [CrossRef]
80. Sarrut, D.; Etxebeste, A.; Muñoz, E.; Krah, N.; Létang, J.M. Artificial Intelligence for Monte Carlo Simulation in Medical Physics. *Front. Phys.* **2021**, *9*, 738112. [CrossRef]
81. Xu, P.; Sun, M.Y.; Gao, Y.J.; Du, T.J.; Hu, J.M.; Zhang, J.J. Influence of data amount, data type and implementation packages in GPU coding. *Array* **2022**, *16*, 100261. [CrossRef]
82. Almgren-Bell, J.; Awar, N.A.; Geethakrishnan, D.S.; Gligoric, M.; Biros, G. A Multi-GPU Python Solver for Low-Temperature Non-Equilibrium Plasmas. In Proceedings of the 2022 IEEE 34th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), Bordeaux, France, 2–5 November 2022; pp. 140–149. [CrossRef]
83. Radmanović, M.M. A Comparison of Computing Spectral Transforms of Logic Functions using Python Frameworks on GPU. In Proceedings of the 2022 57th International Scientific Conference on Information, Communication and Energy Systems and Technologies (ICEST), Ohrid, North Macedonia, 16–18 June 2022; pp. 1–4. [CrossRef]
84. Xu, A.; Li, B.T. Multi-GPU thermal lattice Boltzmann simulations using OpenACC and MPI. *Int. J. Heat Mass Transf.* **2023**, *201*, 123649. [CrossRef]
85. Dogaru, R.; Dogaru, I. A Python Framework for Fast Modelling and Simulation of Cellular Nonlinear Networks and other Finite-difference Time-domain Systems. In Proceedings of the 2021 23rd International Conference on Control Systems and Computer Science (CSCS), Bucharest, Romania, 26–28 May 2021; IEEE: Piscataway, NJ, USA, 2021; pp. 221–226.
86. Azizi, I. Parallelization in Python-An Expectation-Maximization Application. 2023. Available online: https://iliaazizi.com/projects/em_parallelized/ (accessed on 18 December 2023).
87. Cohen, J.; Molemaker, M.J. A fast double precision CFD code using CUDA. *J. Phys. Soc. Japan* **1997**, *66*, 2237–2341.
88. Dang, H.V.; Schmidt, B. CUDA-enabled Sparse Matrix–Vector Multiplication on GPUs using atomic operations. *Parallel Comput.* **2013**, *39*, 737–750. [CrossRef]
89. cuRAND-NVIDIA's CUDA Random Number Generation Library. Available online: https://developer.nvidia.com/curand (accessed on 24 July 2023).

90. Collange, S.; Defour, D.; Tisserand, A. Power consumption of GPUs from a software perspective. In Proceedings of the Computational Science–ICCS 2009: 9th International Conference, Baton Rouge, LA, USA, 25–27 May 2009; Proceedings, Part I 9; Springer: Berlin/Heidelberg, Germany, 2009; pp. 914–923.
91. Eklund, A.; Dufort, P.; Forsberg, D.; LaConte, S.M. Medical image processing on the GPU–Past, present and future. *Med. Image Anal.* **2013**, *17*, 1073–1094. [CrossRef]
92. Després, P.; Jia, X. A review of GPU-based medical image reconstruction. *Phys. Medica* **2017**, *42*, 76–92. [CrossRef]
93. Askar, T.; Shukirgaliyev, B.; Lukac, M.; Abdikamalov, E. Evaluation of pseudo-random number generation on GPU cards. *Computation* **2021**, *9*, 142. [CrossRef]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.