Performance Portable Monte Carlo Neutron Transport in MCDC via Numba

Joanna Piper Morgan[1,2*] Ilham Variansyah[1,3 †]
Braxton Cuneo[1,4] Todd S. Palmer[1,3] Kyle E. Niemeyer[1,2]

[1]Center for Exascale Monte Carlo Neutron Transport (CEMeNT)[‡]
[2]School of Mechanical Industrial and Manufacturing Engineering, Oregon State University
[3]School of Nuclear Science and Engineering, Oregon State University
[4]Department of Computer Science, Seattle University

Abstract

Finding a software engineering approach that allows for portability, rapid development, open collaboration, and performance for high performance computing on GPUs and CPUs is a challenge. We implement a portability scheme using the Numba compiler for Python in Monte Carlo / Dynamic Code (MC/DC), a new neutron transport application for rapid Monte Carlo methods development. Using this scheme, we have built MC/DC as a single source, single language, single compiler application that can run as a pure Python, compiled CPU, or compiled GPU solver. In GPU mode, we use Numba paired with an asynchronous GPU scheduler called Harmonize to increase GPU performance. We present performance results for a time-dependent problem on both the CPU and GPU and compare them to a production code.

Developing software to simulate physical problems that demand high-performance computing (HPC) is difficult. Modern HPCs commonly use both CPUs and GPUs from various vendors. Years can be spent porting a code from CPUs to run on GPUs, then again when moving from one GPU vendor to the next [1]. Portability issues compound when designing software for rapidly developing numerical methods where algorithms need to be both implemented and tested at scale. Finding a software engineering approach that balances the need for portability, rapid development, open collaboration, and performance can be challenging especially when numerical schemes do not rely on commonly library implemented operations (i.e., linear algebra as in LAPACK or

*morgajoa@oregonstate.edu
†variansi@oregonstate.edu
‡https://cement-psaap.github.io/

Intel MKL). Common HPC software engineering requirements are often met using a Python-as-glue-based approach, where peripheral functionality (e.g., MPI calls, I/O) is implemented using Python packages but compiled functions are called through Python's C-interface where performance is needed. Python-as-glue does not necessarily assist in the production of the compiled compute kernels themselves—what the Python is gluing together—but can go a long way in simplifying the overhead of peripheral requirements of HPC software. With this technique, environment management and packaging uses `pip`, `conda`, or `spack`, input/output with `h5py`, MPI calls with `mpi4py`, and automated testing with `pytest`, which can all ease initial development and continued support for these imperative operations.

Many tools have been developed to extend the Python-as-glue scheme to allow producing single-source compute kernels for both CPUs and GPUs. One tactic is to use a domain-specific language to avoid needing a low-level language (e.g., FORTRAN, C). A domain-specific language is designed to alleviate development difficulties for a group of subject-area experts and can abstract hardware targets if defined with that goal. PyFR, for example, is an open-source computational fluid dynamics solver that implements a domain-specific language plus Python structure to run on CPUs and Nvidia, Intel, and AMD GPUs [2]. Witherden et al. [2] discussed how this scheme allows PyFR developers to rapidly deploy numerical methods at deployment HPC scales and have demonstrated performance at the petascale. Other projects have addressed the need to write user-defined compute kernels entirely in Python script. Numba is a compiler that lowers a small subset of Python code with NumPy arrays and functions into LLVM, then just in time (JIT) compiles to a specific hardware target [3]. Numba also compiles global and device functions for Nvidia GPUs from compute kernels defined in Python. API calls are made through Numba on both the Python side (e.g., allocate and move data to and from the GPU) and within compiled device functions (e.g., to execute atomic operations). When compiling to GPUs, Numba supports an even smaller subset of Python, losing most of the operability with NumPy functions. If functions are defined using only that smallest subset, Numba can compile the same functions to CPUs or GPUs, or execute those functions purely in Python. Numba data allocations on the GPU can be consumed and ingested by functions from CuPy if linear-algebra operations are required in conjunction with user-defined compute kernels.

We used a Numba+Python development scheme to abate portability issues and allow for rapidly developing novel numerical methods at the HPC scale on CPUs and GPUs when we developed a new Monte Carlo neutron transport application called Monte Carlo / Dynamic Code (MC/DC) [4, 5]. In this paper we first provide an overview of neutron transport and the Monte Carlo solution method. We next describe MC/DC's novel (for the field) software engineering approach in greater detail. We discuss how novel numerical methods are prototyped and developed in MC/DC. Then, we analyze the compute performance of MC/DC on both CPUs and GPUs and, where possible, compare it against modern production Monte Carlo neutron transport solvers. Finally, we provide concluding remarks and outline future work.

1 Monte Carlo / Dynamic Code

Predicting how neutrons move through space and time is important when modeling inertial confinement fusion systems, pulsed neutron sources, and nuclear criticality safety experiments, among other systems. Simulating these problems is computationally difficult using any numerical method, as the neutron distribution is a function of seven independent variables: three in space, three in velocity, and time. Modern HPC systems now enable high-fidelity simulation of neutron transport for problem types that have seldom been modeled before due to limitations of previous computers. Specifically, large scale, highly dynamic transport problems require thousands of compute nodes using modern hardware accelerators (i.e., GPUs) [6, 7].

The behavior of neutrons can be modeled with a Monte Carlo simulation, where particles with statistical importance are created and transported to produce a particle history [8]. A particle's path and the specific set of events that occur within its history are governed by pseudo-random numbers, known probabilities (e.g., from material data), and known geometries. Data about how particles move and/or interact with the system are tallied to solve for parameters of interest with an associated statistical error from the Monte Carlo process. The analog Monte Carlo method is slow to converge (with a convergence rate of $\mathcal{O}(1/\sqrt{n})$ where n is the number of simulated particles). New Monte Carlo schemes could converge the solution faster in wall-clock time with fewer simulated particles and may be needed to effectively simulate some systems. Monte Carlo/Dynamic Code (MC/DC) was written to enable the rapidly developing these novel numerical methods.

MC/DC has a similar feature set to other Monte Carlo neutron transport applications (e.g., OpenMC [7], Shift [6]) with support for k-eigenvalue and fully time-dependent simulation modes in full 3D constructive solid geometry. It can model the neutron distribution in energy by either using continuous energy or multigroup nuclear data. It also supports domain decomposition. All these features are supported on CPU (x86, ARM, POWERPC) and GPU targets (Nvidia and, soon, AMD GPUs).

The number of novel schemes and simulation techniques implemented in MC/DC in a short time illustrates the success in its software engineering structure MC/DC supports the use of the iterative quasi-Monte Carlo (iQMC) method where deterministic and Monte Carlo transport operations are used in tandem to converge solutions faster than they would in a pure Monte Carlo method. MC/DC supports hash-based random number generation to have a fully-replicated solution for testing, even when running a domain decomposed problem. It supports population control techniques for use in the time-dependent mode where particle populations can grow or decay exponentially. MC/DC has the ability to do uncertainty and global sensitivity analysis to isolate the variance introduced by the Monte Carlo process from the variance due to uncertainty in input parameters (i.e., uncertainty associated with material data). Continuous movement of surfaces (other codes move geometry as discrete steps between time steps) is implemented to better model transient events, such as fuel rod insertion into a

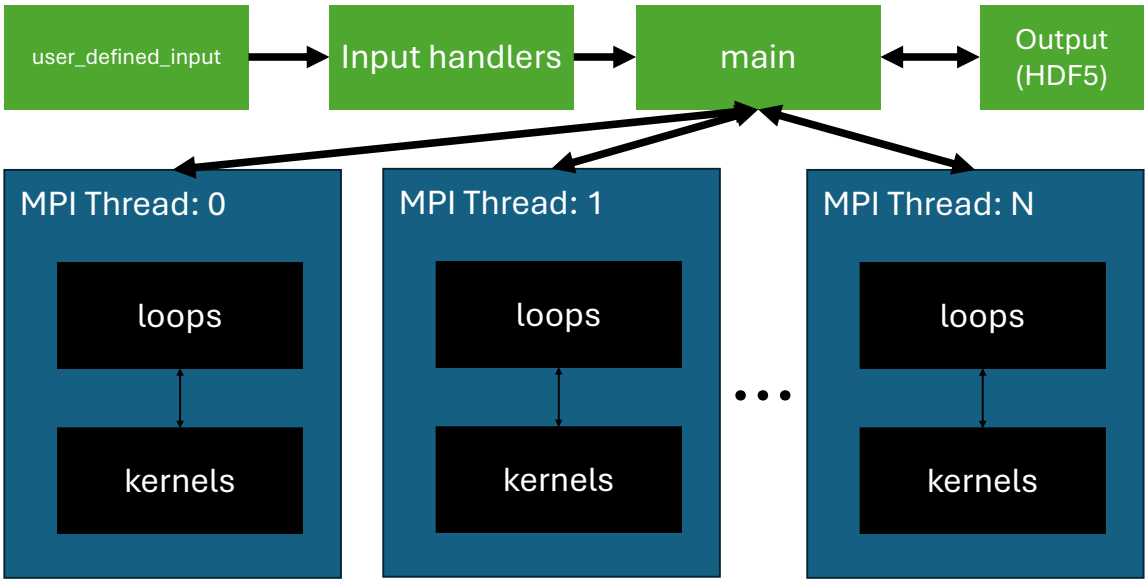

Figure 1: MC/DC's overall structure and how functions get called and interact. Green functions are entirely Python, and black functions are compiled compute kernels if they are running in Numba CPU or GPU modes.

reactor or criticality safety experiments. Several ongoing developments include Quasi Monte Carlo, residual Monte Carlo, and machine learning techniques for dynamic node scheduling.

Figure 1 shows MC/DC's functional layout when running in both MPI and Numba mode. First, a user writes a Python script and imports `mcdc` as a package. Then, the user interfaces with functions described in the input handler within MC/DC describing the physical models, material data, and simulation parameters. This input layout is similar to how other Monte Carlo neutron transport applications input problems. This input script calls a run function, which starts initialization functions within MC/DC. In initialization, a global variable is allocated and constructed containing the user-defined inputs, meshes, particle banks, event tallies, and current global states. This global variable is constructed from a statically typed Numpy `ndarray`, which acts like a Python dictionary, where keywords are used to get numerical arrays out. After the global variable is built, initialization functions dispatch MPI processes if running in MPI mode, and begin the Monte Carlo neutron transport simulation.

Within each MPI process, loops containing the various transport algorithms and modes that MC/DC supports are called. Each transport function is decorated with a Numba JIT (`@jit`) compilation flag declaring that each function must be compiled before being executed if running in Numba mode. These transport logic loops are the highest level at which Python functions will be compiled in MC/DC. For example, a fixed source problem will *loop* over all

```
import numpy

part = numpy.dtype([
    ('x', float64), ('y', float64),
    ('z', float64), ('ux', float64),
    ('uy', float64), ('uz', float64),
    ('v', float64) ])

@jit
def move_particle(P: part, distance):
    P['x'] += P['ux'] * distance
    P['y'] += P['uy'] * distance
    P['z'] += P['uz'] * distance
    P['t'] += distance / P['v']
```

Figure 2: Example of a decorated function and MC/DC's data structures based on `numpy.ndarray`.

the particles and transport them until the particle's history is terminated from a physical event (e.g., capture, fission, time or space boundary), a simulation event (e.g., time census), or a variance-reduction event (e.g., population control, implicit capture). The specific functions within each algorithm that conducts the actual transport operations (e.g., moving particles, tallying events, generating daughter particles from fission) are contained in the kernel set where all functions are `@jit` decorated.

Figure 2 shows an example compute kernel that updates the position and time of a particle as it moves. It also shows the declaration of an `ndarray` data structure used in MC/DC.

After all transport is completed and the simulation is finished, the program returns to the Python interpreter and calls finalization functions. Here requested tally information along with statistical error provided from the Monte Carlo process is saved in an HDF5 file. Data can be extracted from this HDF5 file and used in Python scripts to do post-processing analysis and/or data visualization with tools like Matplotlib, or post-processing can be done in other applications like Visit or Paraview.

Moving to compile and run Numba `jit`ed functions to the GPU requires making a few alterations to the kernels themselves. An even smaller subset of Python functions work in GPU-compiled code, with operations supported on Numba-CPU like `numpy.linalg.solve()` losing support. Atomic operations are required to preserve the side-effects of individual threads in global memory (e.g., adding to a tally). To allow for a unified kernel base in MC/DC for both CPUs and GPUs we track alternate function implementations which are registered through decorators. Figure 3 shows how we implement alternate tally accumulation functions using `@for_cpu` and `@for_gpu` decorator. Here

```
from numba import cuda

@for_cpu
def add(array, value, idx):
    array[idx] += value

@for_gpu
def add(array, value, idx):
    cuda.atomic_add(array, value, idx)
```

Figure 3: Example of GPU and CPU specific API calls as defined in MC/DC.

the `@for_cpu` is adding one to an array, as this is within a single MPI rank we can assume this is a thread safe operation; however on the GPU this may result in a memory race condition thus requiring an `numba.cuda.atomic_add` API call. While this does increase complexity for a programmer implementing numerical methods, it is nowhere near the complexity that might be required to accomplish a similar implementation in a compiled language.

Just because a function compiles for a GPU target does not mean it will perform efficiently on the GPU. MC/DC does implement some specific algorithms designed to increase GPU performance (e.g. event based algorithms, branchless collisions), but more is needed to enable a Monte Carlo neutron transport application to be performant on GPUs. So, when compiling and running MC/DC for GPUs, an asynchronous event scheduling library called Harmonize [9] is used to re-organize the execution of business logic and storage of data to better fit the strengths of GPUs. Harmonize examines the operations that are due to be executed, segregating them into like-operations so that like-work may be executed together in batches. Monte Carlo transport functions lend themselves to asynchronous programming schemes as is intuitive to provide a function for each particle operation, like in the example of a `move_particle` function in Figure 2. Executing in this mode, MC/DC becomes a library of device functions that Harmonize ingests. This has been shown to increase GPU performance by reducing divergence [9].

When compiling functions to GPU targets with Harmonize, intermediate compiler representations (e.g., LLVM-IR or PTX) are used to inject the Monte Carlo neutron transport kernels within MC/DC into the GPU-optimized control flow defined in Harmonize. While Harmonize was built with MC/DC in mind, theoretically it could be used for any generic GPU workflow that exhibits significant thread divergence [9]. Currently, MC/DC only supports execution on the GPU with Harmonize. When running MC/DC in GPU mode, device memory for the global array is allocated then moved from the host (CPU) to the device (GPU) during initialization. After that, MC/DC's transport kernels are executed in Harmonize on the GPU until transport for a given collection of work is complete. Communication of the global variable between the device

and the host may be required during transport for some simulation modes like when conducting a census event for time dependent transport. When transport is completed the global variable is moved back to the host for a final time, and the simulation is finalized.

When initially exploring a novel transport method, a developer can work in a pure Python environment where functions are entirely executed in Python. In this mode, any package can be brought into any function, typing can be done dynamically, and any Python data structure can be used. MC/DC can be executed in MPI mode in Python or compiled CPU mode. While a full Python development environment is great for initial proofs of concept, it proves to be too slow for problems of interest.

When more performance is required, Numba mode can be used to compile functions to approach C-like speeds. However, Numba only supports a small subset of the Python ecosystem. Some Python data structures like dictionaries and lists can no longer be used and must instead come from NumPy implementations. Thus, when using Numba, the small subset of functions supported effectively becomes a domain-specific language. When kernels are written to support Numba mode, they can be compiled to any supported CPU targets (x86, ARM64, and PPC64) automatically. If even more performance is required, the subset of supported Python shrinks further and specific GPU API calls must be appropriately abstracted with function calls defined with `@for_cpu` and `@for_gpu`. At this point, an optimized kernel can be executed on the GPU with Harmonize.

2 Performance

To compare MC/DC to another Monte Carlo neutronics software package, we use a time-dependent version of the Kobayashi dog-leg void-duct problem [10, 5]. Figure 4 at left shows the void duct and the location of the neutron source at the opening of the duct. The initial condition is zero flux everywhere. Radiation moves down the void quickly and then penetrates into the walls of the problem. Figure 4 at right shows the duct clearly with the scalar flux solution in a time-averaging window of 10 s at somepoint time from MC/DC and OpenMC [7] (an open-source code written in `C++`) .

Figure 5 at left shows the runtime results of MC/DC compared with OpenMC on a single node of the Quartz supercomputer (two socket 18-core Intel Xeon E5-2695 v4) at Lawrence Livermore National Laboratory (LLNL). Caching in MC/DC was enabled for these results, meaning there is no JIT compilation overhead. We examine the impact of a variance reduction technique called implicit capture [8], which is supported in both codes. OpenMC is about 20 times faster than MC/DC for this problem with implicit capture. Without implicit capture that performance gap falls to 12 times. This performance gap is due to OpenMC's tracking algorithm, which does not stop particles at mesh crossing events as MC/DC currently does. OpenMC is currently limited to the use of the collision estimator to estimate the scalar flux spatial distribution in

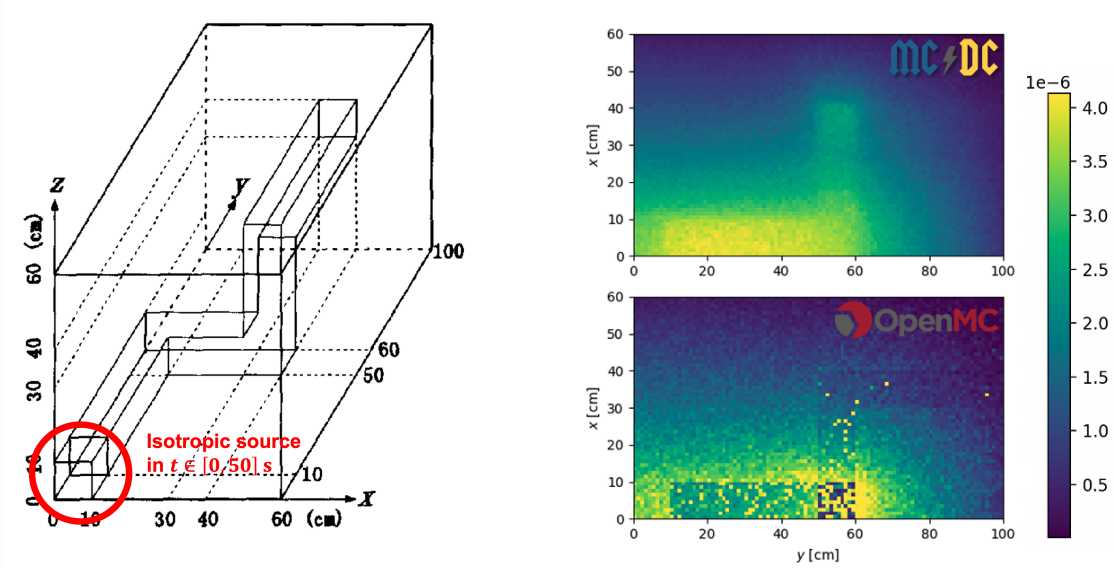

Figure 4: Left: Kobayashi problem schematic. Right: Time and space averaged scalar flux solution to the Kobayashi problem run with 1×10^9 particle histories from MC/DC (top) and OpenMC (bottom).

time-dependent simulation, whereas MC/DC uses the track length estimator [8], which is computationally more expensive but yields better statistics. This is shown in figure 4 at bottom right where OpenMC's result has tally cells with a poorly converged solution (displaying as yellow spots) and MC/DC's (top right) doesn't. Work is ongoing to implement this particle stop command. When this is enabled, we suspect the performance gaps for this problem will close substantially between MC/DC and OpenMC.

MC/DC is the only open-source Monte Carlo neutron transport code that can perform fully transient analysis for systems like the Kobyashi problem on GPUs. Figure 5 at right shows the wall clock runtime results of the same Kobyashi problem on a single Tesla V100 GPU on the Lassen supercomputer at LLNL as compared to MC/DC on 36 CPU cores (this time with caching disabled) on LLNL's Quartz supercomputer across particle counts and with-/without implicit capture. The JIT compilation overhead incurs a constant runtime before enough work saturates the GPU architecture. For the Kobyashi problem with implicit capture, the GPU versions experience a 21 times speed up with saturation occurring around 10^6 particles. For this problem without implicit capture, the GPU remains unsaturated up to 10^9 but incurs a maximum observed speedup of 24 times over on CPU. This speedup of a whole CPU node to a single GPU compares well to the performance of other production Monte Carlo neutron transport applications [11].

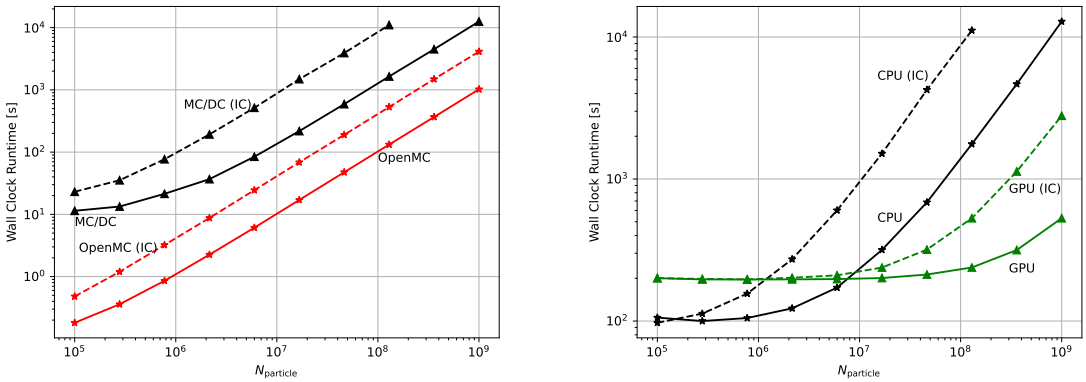

Figure 5: Left: Wall clock runtime of the Kobyashi problem over particle counts with and without implicit capture (IC) between cached MC/DC (black) and OpenMC (red). Right: Wall clock runtime of the Kobyashi problem over particle counts in uncached MC/DC CPU (black) and GPU (green) modes.

3 Discussions, Conclusions, and Future Work

Monte Carlo/Dynamic Code is a single-source, single-language, single-compiler Monte Carlo neutron transport code that targets modern HPC architectures with CPUs and GPUs. Our performance results demonstrate that MC/DC's structure (using a Python + Numba JIT + MPI scheme) can produce similar performance to other Monte Carlo neutron transport solvers. While MC/DC currently performs somewhat worse than other production code, we believe this is primarily due to algorithmic differences rather than architectural ones. MC/DC currently supports many novel transient transport methods and variance reduction techniques that are not supported in any other Monte Carlo neutron transport application.

Numba has been found to have performance deficiencies as compared to other high-level languages and portability frameworks in some workloads [12]. We initially explored other high-level language portability frameworks including PyCUDA and PyKokkos and found that for a Monte Carlo neutron test-bed Numba's performance was in-line with the others considered [13]. Numba has been described as being as difficult to develop in as a traditional low-level language for complex algorithms [14]. We agree with those findings that it can be as difficult—or even harder when debugging—as developing compiled code, but since we are primarily using Numba as a Python-based portability framework, any ease-of-development advantages are only added benefits. MC/DC's software engineering approach has the added benefit of unifying our glue language and kernel production language.

Work in MC/DC is ongoing. We are continually exploring novel variance

reduction techniques and adding new functionality. We will implement a no-stop mesh-boundary tracking algorithm, optimize our data structures, and explore methods of kernel profiling with the goal of identifying code hotspots sections of MC/DC for targeted optimizations. We also plan on adding better tracking schemes to OpenMC to improve it's solution for transient problems as well. On GPUs specifically, work is ongoing to support compilation to AMD GPUs via a patch to Numba that allows for AMD compilation and execution[1]. Intel also offers a patch to compile to their GPUs via Numba[2] and future work include building support for their accelerators as well. On all GPU types we also plan to extend support to multi-GPU simulations via existing MPI frameworks. We will continue to improve MC/DC, making it a portable application for rapid methods development enabled by Python and Numba.

4 Acknowledgments

This work was supported by the Center for Exascale Monte-Carlo Neutron Transport (CEMeNT) a PSAAP-III project funded by the Department of Energy, grant number: DE-NA003967.

References

[1] M. Pozulp, R. Bleile, P. Brantley, S. Dawson, M. McKinley, and A. R. M. Y. M. O'Brien, "Progress Porting LLNL Monte Carlo Transport Codes to Nvidia GPUs," in *International Conference on Mathematics and Computational Methods Applied to Nuclear Science and Engineering*, Niagara Falls, Ontario, Canada, 2023.

[2] F. D. Witherden, "Python at petascale with PyFR or: How I learned to stop worrying and love the snake," *Computing in Science & Engineering*, vol. 23, no. 4, pp. 29–37, 2021.

[3] 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*. Austin Texas: ACM, Nov. 2015, pp. 1–6.

[4] J. P. Morgan, I. Variansyah, S. Pasmann, K. B. Clements, B. Cuneo, A. Mote, C. Goodman, C. Shaw, J. Northrop, R. Pankaj, E. Lame, B. Whewell, R. McClarren, T. Palmer, L. Chen, D. Anistratov, C. T. Kelley, C. Palmer, and K. E. Niemeyer, "Monte Carlo / Dynamic Code (MC/DC): An accelerated Python package for fully transient neutron transport and rapid methods development," *Journal of Open Source Software*, vol. 95, p. 6415, 2024.

[1] https://github.com/ROCm/numba-hip
[2] https://github.com/IntelPython/numba-dpex

[5] I. Variansyah, J. P. Morgan, J. Northrop, K. E. Niemeyer, and R. G. McClarren, "Development of MC/DC: a performant, scalable, and portable Python-based Monte Carlo neutron transport code," in *International Conference on Mathematics and Computational Methods Applied to Nuclear Science and Engineering*, Niagara Falls, Ontario, Canada, 2023.

[6] 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, Jun. 2019.

[7] 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.

[8] E. E. Lewis and W. F. Miller, *Computational Methods of Neutron Transport*. New York, NY, USA: John Wiley and Sons, Inc., 1984.

[9] B. Cuneo and M. Bailey, "Divergence reduction in Monte Carlo neutron transport with on-GPU asynchronous scheduling," *ACM Trans. Model. Comput. Simul.*, oct 2023, just Accepted.

[10] K. Kobayashi, N. Sugimura, and Y. Nagaya, "3D radiation transport benchmark problems and results for simple geometries with void region," *Progress in Nuclear Energy*, vol. 39, pp. 119–144, 2001.

[11] J. P. Morgan, A. Mote, S. Pasmann, G. Ridley, T. S. Palmer, K. E. Niemeyer, and R. G. McClarren, "The Monte Carlo computational summit – October 25 & 26, 2023 – Notre Dame, Indiana, USA," *Journal of Computational and Theoretical Transport*, 2024.

[12] 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)*. IEEE, May 2023.

[13] J. P. Morgan, T. S. Palmer, and K. E. Niemeyer, "Explorations of Python-based automatic hardware code generation for neutron transport applications," in *Transactions of the American Nuclear Society*, vol. 126, Anaheim, CA, USA, 2022, pp. 318–320.

[14] S. Kailasa, T. Wang, L. A. Barba, T. Betcke, K. Hinsen, and A. Dubey, "PyExaFMM: An exercise in designing high-performance software with Python and Numba," *Computing in Science & Engineering*, vol. 24, no. 5, pp. 77–84, 2022.