# Toward Portable GPU Acceleration of the OpenMC Monte Carlo Particle Transport Code

**Conference Paper** · May 2022

DOI: 10.13182/PHYSOR22-37847



**8 authors**, including:

John R. Tramm
Argonne National Laboratory
**25** PUBLICATIONS **300** CITATIONS
SEE PROFILE

Paul Romano
Argonne National Laboratory
**55** PUBLICATIONS **2,166** CITATIONS
SEE PROFILE

Patrick Shriwise
Argonne National Laboratory
**31** PUBLICATIONS **82** CITATIONS
SEE PROFILE

Andrew R. Siegel
Argonne National Laboratory
**93** PUBLICATIONS **2,473** CITATIONS
SEE PROFILE

# Toward Portable GPU Acceleration of the OpenMC Monte Carlo Particle Transport Code

**John R. Tramm[1*], Paul K. Romano[1], Johannes Doerfert[1], Amanda L. Lund[1], Patrick C. Shriwise[1], Andrew R. Siegel[1], Gavin Ridley[2], Andrew Pastrello[3]**

[1]Argonne National Laboratory
9700 S Cass Ave., Lemont IL, 60439, USA

[2]Massachusetts Institute of Technology
77 Massachusetts Ave, Cambridge MA, 02139, USA

[3]University of New South Wales
Sydney, NSW 2502, Australia

[*]jtramm@anl.gov

## ABSTRACT

OpenMC is an open source Monte Carlo particle transport code originally designed for CPU-based systems. In this work, we use the OpenMP target offload programming model to port OpenMC to a variety of GPU architectures from NVIDIA and AMD. We detail our porting strategy and provide preliminary performance comparisons. The GPU version of OpenMC run on an NVIDIA A100 GPU achieves performance equivalent to approximately 200 Xeon CPU cores for a depleted fuel reactor benchmark problem.

KEYWORDS: Monte Carlo, Particle Transport, GPU, OpenMP Offloading, High Performance Computing

## 1. INTRODUCTION

In the field of computational nuclear engineering, the Monte Carlo (MC) method is commonly used for general purpose, high-fidelity particle transport simulations. While MC is often considered the "gold standard" in accuracy, it typically has a very high computational cost. The high computational costs are further exacerbated by MC's stochastic branching nature and random memory access patterns, inhibiting the efficient use of modern computer hardware. As computers become more powerful, there has been growing interest in using high-fidelity MC methods for routine design work that has historically been performed by deterministic applications. It is therefore important that MC codes continue to evolve to leverage the increased computing power of modern computing architectures.

OpenMC [1] is a continuous-energy MC particle transport application that has demonstrated excellent performance and scalability on CPU-based systems. In addition to being an open source application, OpenMC also provides a host of advanced modeling and simulation capabilities including depletion, advanced geometry representations, on-the-fly Doppler broadening, and multigroup cross section generation. OpenMC can also utilize current CPU-based supercomputers using both distributed-memory parallelism (via the Message Passing Interface (MPI)) and shared-memory parallelism (via OpenMP threading). Together, the combination of advanced capabilities in OpenMC has led to rapid community adoption and research across a variety of fields involving particle transport. In recent years the OpenMC community has grown both in

terms of open source developers active in the project that contribute new modules as part of their research, as well as an ever growing international user base.

One major challenge facing OpenMC is its inability to utilize GPU hardware. OpenMC runs on nearly all CPU-based systems thanks to widespread support for MPI and OpenMP. GPUs, however, do not support either of these models. Without GPU support, OpenMC cannot utilize many current-generation supercomputers, including seven of the top ten most powerful supercomputers in the world [2], where over 90% of the total performance is provided by the GPUs. Moreover, many small and mid-sized HPC systems in the coming years are likely to benefit from the cost/performance advantages of GPUs, which in recent generations have substantially outpaced CPUs in performance gains. Therefore, to make use of current and next generation GPU-based supercomputers, OpenMC must be ported to a new parallel programming model. We emphasize that this effort is in no sense a rote line-for-line porting — due to the profound differences between CPU and GPU architectures, significant and fundamental algorithmic restructuring is inevitable.

There have been a number of previous efforts to run portions of OpenMC on GPUs. Salmon et al. [3] utilized NVIDIA's proprietary Optix ray tracing library for performing geometry operations on NVIDIA GPUs. However, only the modeling of bare cubes, spheres, and other standard graphics rendering geometries (all featuring a single material containing a single nuclide) were demonstrated. The overall effort was limited in scope and served mainly as a proof of concept for scientific geometry modeling on GPUs.

Another recent CUDA-based porting effort [4] targets a much larger set of OpenMC's capabilities. However, the use of the CUDA programming model has some key limitations. As NVIDIA's proprietary language, CUDA is not natively supported on Intel or AMD GPUs. Furthermore, CUDA code cannot execute on CPUs, meaning that either 1) multiple code bases must be maintained to allow CPU/GPU portability and/or 2) `ifdef` compiler macros must be used in some parts of the code to maintain multiple versions of some portions of the code. Thus, there is an incentive for investigation into portable programming models that would allow running on a wider variety of computer architectures.

Beyond OpenMC, there is a growing body of research porting MC applications to GPUs. For instance, Shift [5] and PRAGMA [6] both utilize CUDA to leverage NVIDIA GPUs. The present work differs from these efforts in a number of ways: 1) OpenMC supports a wider range of computational geometries rather than being restricted to PWR-like lattices, 2) OpenMC is open source, and 3) the choice of the OpenMP target offload programming model for OpenMC offers greater portability than is possible with CUDA.

## 2. PORTING STRATEGY

### 2.1. Selection of Programming Model

To date, most GPU-based scientific computing has been done on NVIDIA GPUs using the NVIDIA proprietary CUDA programming model. The GPU market, however, is becoming more competitive. For instance, the Aurora [7] exascale supercomputer at Argonne National Laboratory will feature Intel GPUs while the Frontier [8] exascale supercomputer at Oak Ridge National Laboratory is based on AMD GPUs. Therefore, due to the wider variety of vendors that are now competing in the market, portability has become a more important aspect for GPU HPC applications than it has been historically.

Recently, new programming models have emerged that offer GPU support with greater portability. OpenMP target offloading, HIP, SYCL/DPC++, and higher level performance portability frameworks like Kokkos, RAJA, and OCCA are all viable options that provide portability across multiple GPU vendors. Among these, OpenMP has first-party support by all three major GPU vendors. NVIDIA currently supports OpenMP offloading with its NVIDIA HPC SDK, Intel supports OpenMP offloading with its OneAPI SDK, and AMD supports offloading with its AOMP compiler. Additionally, the LLVM Clang compiler (which several of the GPU vendors' compilers are based on) is also capable of compiling OpenMP target offloading

code for a variety of GPUs. Furthermore, OpenMP offloading permits code to run natively on the CPU by simply "offloading" to the host CPU as if it were a device, which is not possible with all GPU programming models. This ability allows for a single unified code base to run on nearly any CPU or GPU without the need to silo off CPU vs. GPU versions.

### 2.2. Strategy and Methods

Our GPU porting effort involved both large-scale algorithmic changes as well as a number of more targeted porting challenges. This section details all significant structural changes. The goal is to create a unified single-source version of OpenMC for both CPU and GPU architectures. However, as a first step we wanted to gain first-hand experience running OpenMC on GPUs before committing to transitioning the main community branch of the code. To that end, the effort detailed in this study was implemented in a separate (closed source) branch of OpenMC specifically tailored to GPUs. In the future, once we have optimized GPU performance and full support of the OpenMP 5.0 standard is implemented in most compilers, we will evaluate transitioning the main branch of OpenMC to use the new OpenMP offloading version.

#### 2.2.1. Event-based transport

The first area where large algorithmic changes were needed was the overall parallelization strategy. OpenMC, like most MC codes, relies on *history-based* transport, where each thread sequentially processes a single particle from birth to death. For each thread, at most one particle is kept in memory at any point in time, meaning that the total number of particles in-flight (and resident in memory) is equal to the number of active threads. Parallelization is achieved by assigning a group of particles to each thread. Thus, at a given point in time, in general each thread will be performing a different task. Thus, the history-based transport method is inherently MIMD and precludes effective use of data-level parallelism. On GPU architectures, which rely heavily on data-level parallelism, history-based transport is likely to lead to suboptimal performance. Instead, *event-based* [9] transport can be used wherein many particles are stored in a large buffer in memory and the simulation is split up into discrete events (e.g., calculation of cross sections, surface crossing, and collision). Events are executed in turn, and parallelization is achieved by parallelizing over all particles that require that event, which reduces branching and improves the overall vector efficiency of the application. This also leaves open the possibility for different threads to operate on different particles at each new event stage, meaning that some structural changes to the code are often required when using event-based parallelism. The introduction of an event-based mode to OpenMC led to a number of important implementation questions that will be addressed in this section.

**Granularity** To achieve perfect vectorization and no thread divergence, it would be ideal to have many fine-grained events that are small enough to have no conditional branching. However, restructuring OpenMC into such small kernels is both impractical and may potentially degrade performance by limiting available parallelism. Previous work [5,6] has shown that in general coarser grained events achieve better performance while simplifying the implementation. We followed this model for OpenMC, defining eight discrete events for use in the event-based transport implementation.

The cross section (XS) lookup event was split into fuel and nonfuel events because of the large disparity in nuclde counts for reactor applications (depleted fuel will contain hundreds of nuclides while the nonfuel materials many contain only a few). This can create a significant work imbalance if different threads in a GPU thread team (or block) are each processing a different material type. While this is not likely to be an important difference for CPUs, on GPUs this coarse-grained material sorting strategy has been shown to reduce thread divergence and improve vector efficiency, resulting in better overall performance [5].

**Queuing** There are several viable methods for managing how events are executed. Our implementation adopts the strategy described in [5], wherein a single buffer of particles is stored in device memory. Then, a queue is stored for each event type. As a particle finishes an event, it atomically adds an entry to the queue for the subsequent event. The queue entry contains the index in the particle buffer that the particle resides

at. This atomic operations cause inherent overheads when enqueuing but there is the benefit of not having to perform a sort on the particles or perform any physical copies of particle objects.

**Reproducibility** Switching to event-based parallelism naturally changes the order in which different operations occur on a device. In practice, this can result in differences in the order that a random number generator is used and the order in which particle events occur, causing different results to be generated between history-based and event-based modes. While there is nothing wrong with changing the order from a statistical standpoint, it can create practical problems. To overcome these we adopted the reproducibility strategy described in [10]. The basic approach is to ensure that all particles have their own unique random number streams, and then to add several fields that allow for a consistent (and fast) sorting operation to occur at the end of the iteration. Use of the algorithm in [10] enables reproducibile calculations when using either event-based or history-based transport modes in OpenMC.

### 2.2.2. Data structures

Beyond supporting event-based transport, another significant porting challenge is mapping the complex, deeply nested data structures and polymorphism to the device. While some GPUs do support the use of virtual functions on device, many do not because of the lack of device-side virtual function lookup tables for objects initialized on the host. Even for architectures and programming models that might support virtual functions in some capacity, there may be serious performance implications when using them, as device compilers typically perform best when they are able to inline all function calls. Thus, we decided to eliminate the use of polymorphism and adopt alternative approaches.

For surface classes, virtual functions were eliminated by creating a single `Surface` class that stores an extra data member indicating the type of the surface along with the union of all attributes needed by the various surface types. Any method call that would have resulted in a virtual table lookup was replaced by a switch statement that checks the surface type and dispatches to an appropriate (non-virtual) method for the given surface type. The same approach was used for lattices and boundary conditions as well. While storing the union of all surface fields is a suboptimal solution in terms of memory usage, surface data is typically not a major contributor to overall simulation memory usage, at least for typical light water reactor simulations. In practice, we found the increase in memory usage for storing surfaces, lattices, and boundary conditions was negligible.

Removing polymorphism in classes within the nuclear data hierarchy (probability and angle-energy distributions) was not as simple. A different approach was adopted whereby all data for a single nuclear reaction is serialized into an opaque buffer. Each class within that hierarchy is required to have a `serialize()` method that writes data into a buffer that is passed and increments an index to indicate the offset past which no data has been written yet. This effectively "flattens" the hierarchy of classes.

### 2.2.3. Management of device memory

As OpenMC simulations are based on highly stochastic memory access patterns, we chose to manually migrate data initialized on the host to the device data environment rather than relying on OpenMP unified shared memory. Use of explicit manual data migration avoids potentially expensive paging delays associated with unified shared memory. Therefore, we wrote a number of memory migration routines that migrate simulation data to (and sometimes from) the device by way of simulation-persistent OpenMP `target enter/exit data` mapping directives and accompanying `update` directives when required. Additionally, in OpenMP (and all other current GPU programming models), the C++ standard template library is not supported for regions of the code being run on device. Because OpenMC makes substantial use of C++ containers such as `std::vector`, some work was required to remove this usage in favor of simpler pointers and length fields.

### 2.3. OpenMC Features Not Yet Supported on GPU

Our GPU port of OpenMC supports nearly all the same capabilities as the original CPU-oriented version. Even advanced features were included in the port (e.g., multipole cross section representations and photon transport), with only a few exceptions. To start the performance analysis and optimization phase of the project as soon as possible, user-defined tallies and multigroup transport are not yet supported on device. As such, we only report performance for inactive batches in this study as all problems contain no tallies. We expect these remaining capabilities to be added soon.

## 3. RUNTIME SPEED COMPARISONS

### 3.1. Benchmark Problem

The full core benchmark problem used in this analysis is the Hoogenboom-Martin (HM) "large" variant [11] representing a depleted fuel reactor with 272 nuclides in total. Our testing confirmed that, thanks to the efforts towards reproducibility detailed in Section 2.2.1, identical results can be produced for simulations run on both GPU and CPU and using both the history-based and event-based transport modes. Some small differences in results between computer architectures were noticed only when very large particle counts were run, likely as a result of minute differences in architecture-specific library math functions. Because the history-based CPU version of OpenMC is already well validated, further validation or alternative testing for the GPU version was deemed unnecessary.

### 3.2. Optimization

Our primary test system is the NVIDIA A100 GPU using a developmental build of the LLVM Clang compiler (git version 8871d82). The baseline version of our port did not initially run to completion. However, after working closely with the LLVM compiler team we identified a series of optimizations that enabled both correctness as well as significant improvements in performance. The impact of each optimization is summarized in Table I.

**Table I: Performance improvements from different optimizations on OpenMC for the HM-large benchmark problem. All optimizations are added to the ones above them in the table.**

| | Optimization Description | Inactive Batch Performance [particles/sec] | Additional Speedup |
|---|---|---|---|
| A | CMake unity build | 602 | - |
| B | -fopenmp-cuda-mode flag usage | 7,714 | 12.8 |
| C | LLVM update() clause optimization | 58,529 | 7.6 |
| D | OpenMC particle object size reduction | 89,732 | 1.5 |
| E | XS lookup kernel optimizations: inlining + reference removal | 117,067 | 1.3 |
| F | Continuous particle refill | 129,345 | 1.1 |
| G | XS lookup queue sort by energy | 164,114 | 1.3 |
| H | Removal of microscopic XS cache | 336,636 | 2.1 |
| I | Increasing number of particles in-flight to 8 million | 349,237 | 1.1 |

**Optimization A** The first optimization was to instruct CMake (OpenMC's build system generator) to perform a "unity build", wherein all source files are copied into a single translation unit before being passed to the compiler. This has an effect similar to link-time optimization where the compiler can better

optimize across functions located in different source files. Use of the unity build allowed for correct results to be obtained, though still with poor performance. While the LLVM compiler is introducing device-side link-time optimization that would negate the need for use of a unity build, the work is still ongoing.

**Optimization B** To temporarily remedy globalization issues in the LLVM compiler while a fix is integrated into the main branch, we used the undocumented `-fopenmp-cuda-mode` flag of Clang which enables CUDA semantics for local variables, indicating to the compiler that local variables will not be accessed by other threads. Notably, both the unity build and `-fopenmp-cuda-mode` optimizations are not likely to be required once device link-time optimization and other improvements are formally added to the LLVM compiler in the near future.

**Optimization C** Another optimization opportunity was discovered when we realized a significant discrepancy between the kernel times reported by the kernel timing utility `nsys nvprof` and OpenMC's own function timers.* We found that a surprisingly large amount of time was spent in `#pragma omp target update` directives that were transferring single scalar variables (usually particle queue size information) after each event kernel launch. In working with the LLVM compiler team a runtime optimization was implemented and used in our developmental build of LLVM, greatly improving the performance of these operations. This optimization is currently undergoing review to be included in the main branch of LLVM.

**Optimization D** Our original GPU port had several statically sized arrays in the `Particle` object that were sized so as to pass all of OpenMC's regression tests. We found these arrays could be greatly reduced in size when running more typical light water reactor problems. The original unoptimized particle size was 92 kB, which was reduced to 47 kB for the HM-large problem, which allowed us to greatly increase the number of particles in-flight. In the future, we will simply allocate fields within the `Particle` object dynamically at program initialization rather than using static arrays.

**Optimization E** The cross section lookup kernel is generally the most expensive kernel in OpenMC. In this kernel, we found that there were redundant references to global memory that could sometimes be replaced with a single load into a thread-local stack variable. For instance, rather than accumulating microscopic cross section (XS) data directly into the particle's macroscopic XS field after each nuclide, local microscopic variables could be used with only one write into the particle's field after all XS lookups were performed. Two key functions within the XS lookup kernel (S($\alpha, \beta$) and unresolved resonance region (URR) probability tables) were also manually inlined to reduce the number of global data references.

**Optimization F** In the original event-based implementation, if the number of particles run per batch was greater than what could be fit in GPU memory, multiple sub-batches were run in serial to complete the number of particle histories required by each outer simulation batch. One optimization is to rebirth particles on-the-fly so that the event-queues are refilled. This has the effect of amortizing the cost of low-efficiency GPU operations as particles die off and less work is being performed in each event at the end of a batch.

**Optimization G** There is benefit in sorting the fuel XS lookup queue by particle energy as has been demonstrated in the PRAGMA application [6]. This optimization has the effect of greatly increasing the memory locality of adjacent threads inside each GPU block as they are likely to access the same cross section data. While a performant on-device sorting library will eventually be available to link to for all major GPU architectures, they are not yet universally available, so for portability we implemented the sort by transferring data back to the CPU host, sorting in parallel using OpenMP threading, and then transferring the sorted queue data back to the device. This is highly suboptimal due to the high cost of migrating data between host and device, but even with these costs included, the improved memory locality of the XS lookup kernel outweighed the added costs from the sort. In subsection 3.5, we will quantify the impacts of linking to a vendor-specific on-device sorting library.

*The profiling available with LLVM/OpenMP, as described at https://openmp.llvm.org, also indicated this problem.

**Optimizations H and I** A final optimization was found by *removing* a legacy CPU-oriented optimization from OpenMC. The original CPU version of OpenMC contained a caching data structure that would store results from microscopic cross section lookups for each nuclide and each particle in-flight. Due to our port's use of an event-based algorithm and an energy sort, it turns out that the loading and storing of cached microscopic XS results is actually slower than simply performing the full XS lookup whenever needed. This is due to energy locality resulting in shared access to the same XS data between adjacent GPU threads, whereas cached data is always unique to each thread and never shared. A beneficial side effect of removing the microscopic XS cache is that the overall size of each `Particle` object is dramatically reduced, from 47 kB to 4 kB. This greatly increases the number of particles that can be in-flight at once on the GPU and further improves performance.

### 3.3. CPU vs. GPU Results

In Table II, we compare performance on several GPUs and CPUs. Performance data is presented when using pointwise cross sections and multipole Doppler broadened cross sections. In all cases, the developmental build of the LLVM Clang compiler (git version 8871d82) was used. Results for the A100 and MI100 GPUs were obtained with OpenMC's event-based mode, while results for the CPU architecture was obtained using OpenMC's history-based mode, as these were the optimal configurations for each architecture. All GPU results were obtained with the microscopic XS cache disabled (as described in Optimization H of subsection 3.2), while results for the CPU node were obtained with the cache enabled (as disabling the cache on CPU caused performance to degrade by 25%). The results show that OpenMC run on a single A100 GPU is comparable in performance to roughly 200 CPU cores, and that OpenMC run on a single MI100 GPU is comparable in performance to 55–75 CPU cores depending on the choice of compiler. We note that LLVM is slightly slower than AMD's AOMP compiler on the MI100, likely because LLVM support for AMD GPUs is less mature than for NVIDIA GPUs. Overall, these are promising early results, especially given the portable nature of the programming model. Notably, the same executable could be run on both the A100 and CPU without requiring recompilation.

### 3.4. Particle Count Performance Scaling

Our performance analysis revealed that increasing the number of particles in-flight at once on GPU can significantly improve performance. To quantify this effect, we ran a series of tests on both CPU and GPU (and for both history-based and event-based modes) varying the number of particles per batch. In Figure 1 we can see that the A100 continues to see performance increases until the device memory has been exhausted (at the 8 million particle mark), suggesting that future GPU architectures featuring larger on-device memory could achieve additional performance gains. The history-based version on the A100 GPU also showed some benefit from having more particles in-flight, but the impact was small compared to improvements achieved when running in event-based mode on GPU. Additionally, the history-based GPU mode was far slower than the event-based mode assuming an adequate number of particles in flight.

**Table II: OpenMC GPU and CPU performance comparisons on the depleted HM-large reactor benchmark with 272 nuclides.**

| XS Lookup Method | Inactive Batches Calculation Rate [Particles/Sec] | | | |
|---|---|---|---|---|
| | 2x Xeon 8180M CPU (56c/112t) (LLVM CLang) | A100 GPU (LLVM Clang) | MI100 GPU (AMD AOMP) | MI100 GPU (LLVM Clang) |
| Pointwise | 97,159 | 349,237 | 131,904 | 93,918 |
| Multipole | 87,963 | 306,767 | 122,188 | 83,829 |

Comparatively, the dual socket CPU node achieved much better performance in history-based mode, likely as a result of the higher costs of accessing particle data.

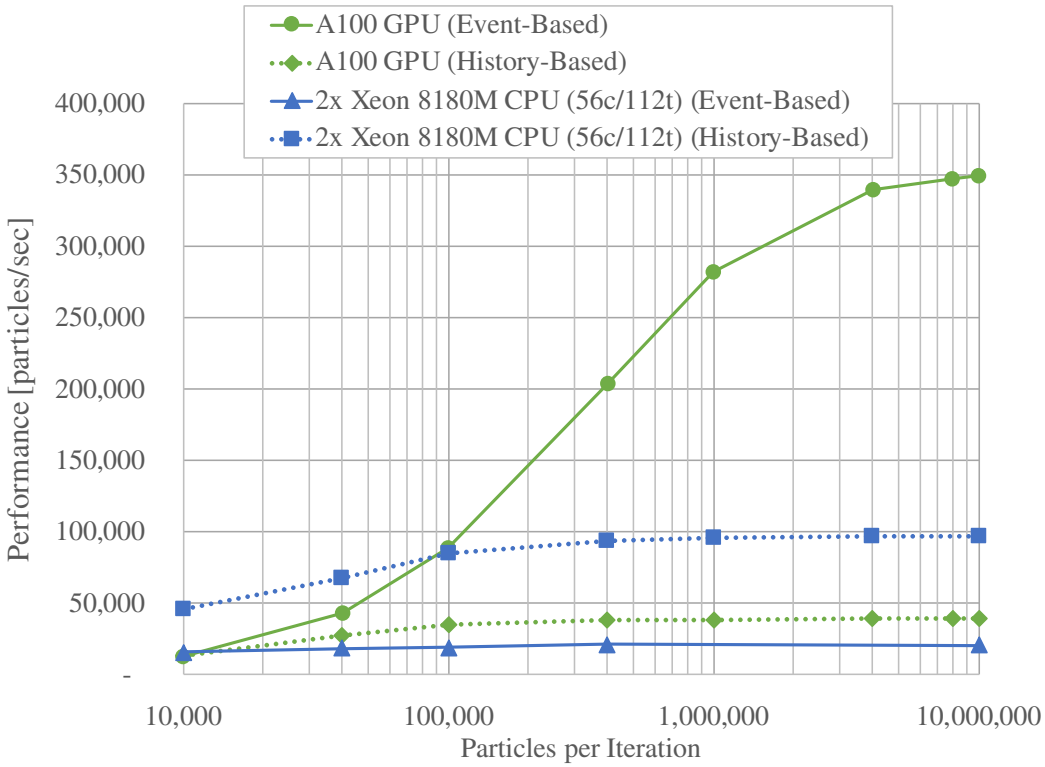


**Figure 1: OpenMC performance for the HM-large depleted reactor benchmark problem on CPU and GPU as a function of the number of particles per inactive iteration. Event-based and history-based algorithms are also compared.**

### 3.5. Performance Impacts of Different Physics Capabilities

To investigate the performance costs of various physics capabilities in OpenMC we ran the code in a variety of configurations. Table III shows the effect of enabling several physics capabilities as well as the result of removing some physics modules that are usually enabled. By default, OpenMC uses the free gas, constant cross section scattering model, which has historically been used by MC codes to sample target velocities, to treat the target motion of all nuclides. Two higher-fidelity methods also exist in OpenMC: the DBRC and RVS models [12]. As shown in Table III, use of the DBRC model reduces performance by nearly 20%, but the RVS model shows less than a 1% drop in performance, meaning that high-fidelity resonant scattering is possible without sacrificing performance on GPU. We also experimented with a non-portable optimization by sorting the cross section lookup queue on-device using CUDA Thrust, yielding a 5% performance gain. Finally, we disabled (by commenting these capabilities out from the code and re-compiling) several advanced physics features in OpenMC. While it is desirable to have all of these capabilities present in the code to maximize fidelity, their removal enables performance comparison of OpenMC to other MC applications that may lack these capabilities. When using non-portable device sorting and removing the thermal and unresolved resonance range treatments from the cross section lookup kernel, OpenMC is able to achieve an overall 46% performance improvement to over 510,000 particles per second while still offering a robust arbitrary constructive solid geometry treatment and using 64-bit floating point arithmetic.

**Table III: Performance impacts of different physics modules, capabilities, and portability constraints on an A100 GPU for the HM-large depleted fuel benchmark problem.**

| | Change Description | Inactive Batch Performance [particles/sec] |
|---|---|---|
| | Baseline | 349,237 |
| Capability Additions | Using DBRC resonant scattering method | 297,086 |
| | Using RVS resonant scattering method | 346,765 |
| Capability or Portability Subtractions | XS lookup queue energy sort on-device by linking with CUDA Thrust | 367,037 |
| | URR probability tables disabled | 420,516 |
| | S($\alpha, \beta$) disabled | 352,495 |
| | URR and S($\alpha, \beta$) disabled | 425,960 |
| | URR, S($\alpha, \beta$), and multipole disabled | 479,657 |
| | URR, S($\alpha, \beta$), and multipole disabled + on-device XS lookup queue sort | 511,451 |

## 4. CONCLUSIONS

We have successfully completed an initial port of OpenMC into the OpenMP target offloading model. We were able run OpenMC on several different GPU architectures from a variety of vendors, including the NVIDIA A100 and AMD MI100. Focused optimization efforts and close collaboration with the LLVM compiler team yielded strong preliminary results on the A100 GPU, with performance for that GPU in the range of 200 Xeon CPU cores on a depleted fuel reactor benchmark problem. We were also able to achieve acceptable performance on an MI100 GPU, with performance for that GPU in the range of 55 to 75 Xeon CPU cores. We also found that the event-based mode is superior to the history-based mode on GPU, whereas the history-based mode is optimal for CPU, confirming findings from previous work in the field. This work provides an excellent baseline for future optimization efforts.

A notable secondary benefit of this work is the co-maturation of new GPU technology in the context of Monte Carlo style applications. Specifically, our collaboration with the LLVM compiler team resulted in the identification of a number of subtle issues affecting performance in LLVM Clang. Due to this collaboration, these issues have now largely been remedied in the main branch of LLVM to the benefit of future OpenMP offloading application development efforts.

## ACKNOWLEDGEMENTS

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. The material was based upon work supported by the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357.

We gratefully acknowledge the computing resources provided and operated by the Joint Laboratory for System Evaluation (JLSE) at Argonne National Laboratory.

## REFERENCES

[1] P. K. Romano, N. E. Horelik, B. R. Herman, A. G. Nelson, and B. Forget. "OpenMC: A state-of-the-art Monte Carlo code for research and development." *Ann Nucl Energy*, **volume 82**, pp. 90–97

(2015). https://doi.org/10.1016/j.anucene.2014.07.048.

[2] E. Strohmaier, J. Dongarra, H. Simon, and M. Meuer. "TOP 500 November 2021 List." https://www.top500.org/lists/top500/2021/11 (2021).

[3] J. Salmon and S. McIntosh-Smith. "Exploiting Hardware-Accelerated Ray Tracing for Monte Carlo Particle Transport with OpenMC." In *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pp. 19–29 (2019). URL https://doi.org/10.1109/PMBS49563.2019.00008.

[4] G. Ridley and B. Forget. "Design and Optimization of GPU Capabilities in OpenMC." *Trans Am Nucl Soc*, **volume 125**(1), pp. 456–459 (2021).

[5] S. Hamilton and T. Evans. "Continuous-energy Monte Carlo neutron transport on GPUs in the Shift code." *Ann Nucl Energy*, **volume 128**, pp. 236–247 (2019).

[6] N. Choi, K. M. Kim, and H. G. Joo. "Optimization of neutron tracking algorithms for GPU-based continuous energy Monte Carlo calculation." *Annals of Nuclear Energy*, **volume 162** (2021). URL https://www.sciencedirect.com/science/article/pii/S0306454921003844.

[7] Argonne Leadership Computing Facility. "Aurora." https://www.alcf.anl.gov/aurora (2020).

[8] Oak Ridge Leadership Computing Facility. "Frontier." https://www.ornl.gov/news/us-department-energy-and-cray-deliver-record-setting-frontier-supercomputer-ornl (2019).

[9] F. Brown and W. Martin. "Monte Carlo Methods for Radiation Transport Analysis on Vector Computers." *Progress in Nuclear Energy*, **volume 14**(3), pp. 269–299 (1984).

[10] F. B. Brown and T. M. Sutton. "Reproducibility and Monte Carlo Eigenvalue Calculations." *Trans Am Nucl Soc*, **volume 65**, pp. 235–236 (1992).

[11] J. E. Hoogenboom, W. R. Martin, and B. Petrovic. "The Monte Carlo performance benchmark test—Aims, Specifications, and First Results." In *Int. Conf. on Mathematics and Computational Methods Applied to Nuclear Science and Engineering*. Rio de Janeiro, Brazil (2011).

[12] P. K. Romano and J. A. Walsh. "An improved target velocity sampling algorithm for free gas elastic scattering." *Ann Nucl Energy*, **volume 114**, pp. 318–324 (2018).