LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-PROC-848684

Progress Porting LLNL Monte Carlo Transport Codes to Nvidia GPUs

M. M. Pozulp, R. C. Bleile, P. S. Brantley, S. A. Dawson, M. S. McKinley, M. J. O'Brien, A. P. Robinson, M. Yang

May 3, 2023

The International Conference on Mathematics and Computational Methods Applied to Nuclear Science and Engineering (M&C 2023)
Niagara Falls, Canada
August 13, 2023 through August 17, 2023

Disclaimer

M&C 2023 - The International Conference on Mathematics and Computational Methods Applied to Nuclear Science and Engineering · Niagara Falls, Ontario, Canada · August 13 – 17, 2023

Progress Porting LLNL Monte Carlo Transport Codes to Nvidia GPUs

M. Pozulp, R. Bleile, P. Brantley, S. Dawson, M. McKinley, M. O'Brien, A. Robinson, M. Yang
Lawrence Livermore National Laboratory
{pozulp1, bleile1, brantley1, dawson6, mckinley9, obrien20, robinson124, yang39}@llnl.gov

ABSTRACT

The Lawrence Livermore National Laboratory Monte Carlo Transport Project has made progress porting production code to Nvidia Tesla V100 GPUs on the Sierra supercomputer. The Monte Carlo calculation speedups are now 7.6x for neutronics and 5.8x for thermal photonics in node-to-node comparisons between Sierra and traditional large scale x86 CPU-based compute platforms. We attribute a large portion of the speedups that we achieved to the use of event-based tracking and better code generation.

KEYWORDS: Monte Carlo, LLNL, Nvidia, Sierra, GPU, Imp, Mercury

1. INTRODUCTION

The Lawrence Livermore National Laboratory (LLNL) Monte Carlo Transport Project develops two production codes: Mercury, a Monte Carlo (MC) particle transport code [1], and Imp, an implicit Monte Carlo (IMC) [2] thermal photon transport code [3]. The codes consist of about 240,000 lines of uncommented C++ source code (370,000 with comments), 80% of which is shared, 15% of which is Mercury-specific, and 5% of which is Imp-specific. Mercury and Imp are distributed- and shared-memory parallel using MPI+OpenMP and run in production on x86 servers. In 2018, LLNL sited the Advanced Technology System-2 (ATS-2) machine Sierra, a 4,320 node system with 2 IBM P9 sockets, 4 Nvidia V100 GPUs, 256 GB of dynamic random access memory (DRAM), and 64 GB of high bandwidth memory (HBM) per node [4]. We started porting production Monte Carlo code capabilities to the Sierra GPU architecture in late 2017.

Our goal is to run calculations on ATS-2 Sierra GPUs and assess their performance compared to a Commodity Technology System-1 (CTS-1) system with 2 Intel Xeon E5-2695 v4 Broadwell sockets and 128 GB of DRAM per node. CTS-1 is an example of the aforementioned "x86 servers". At M&C 2019 we reported a 0.81x node-to-node speedup for Mercury and a 1.99x speedup for Imp [5]. Here at M&C 2023 we present a 7.61x speedup for Mercury and a 5.81x speedup for Imp. We describe these results after describing our effort to achieve them. We also present a throughput study demonstrating that Mercury and Imp need about 1000x more particles to saturate an ATS-2 Sierra node than they need to saturate a CTS-1 node.

Porting Monte Carlo transport codes to GPU architectures is an active area of research. Brown and Martin [6] initially developed the "event-based" approach many years ago in porting Monte Carlo transport algorithms to vector computer architectures. Bergmann and Vujic [7] wrote an event-based code for Nvidia GPUs, and Bergmann et al. [8] demonstrated speedups. Bleile et al. [9] ported a research code to Nvidia GPUs and compared event-based to history-based algorithms. Hamilton et al. [10,11] have ported the Shift production Monte Carlo code to Nvidia GPUs. Choi et al. [12] wrote a Monte Carlo code for Nvidia GPUs.

2. DESIGN

The feature set of Mercury and Imp running on the V100 is nearly identical to the x86 production feature set because we retained the original code beneath a new abstraction layer. Mercury and Imp used hardcoded C++ iteration constructs, like `for` loops, which we replaced with C preprocessor macros. When we target x86 the preprocessor substitutes the `for` statement and the original code is recovered, whereas when we target the V100 the preprocessor substitutes a lambda function definition followed by a cuda kernel launch (see Fig. 1).

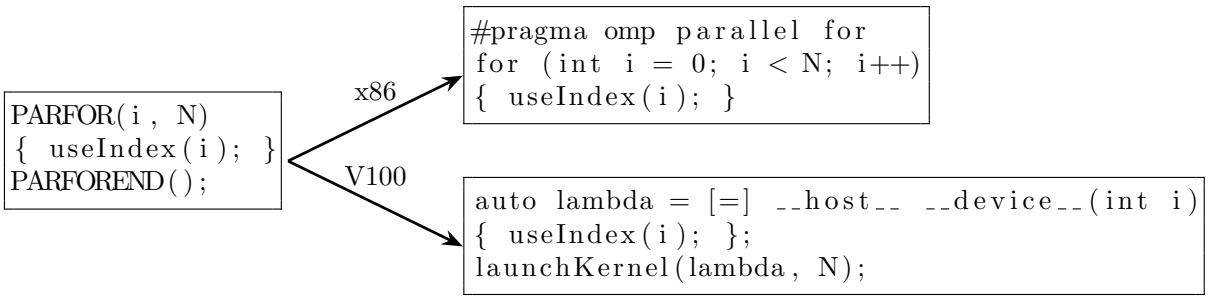


Figure 1: Single source, two architectures. The loop and the kernel execute the same logic.

We use double-precision floating point, Unified Memory ("cudaMallocManaged"), separate compilation, recursive functions, and virtual functions. If we chose to write a new GPU code instead of porting an existing CPU code, we would consider selectively using single-precision, we would use device memory ("cudaMalloc"), and we would avoid separate compilation of device code as well as recursive calls and virtual calls in device code. Our production code has many capabilities and useful features (such as a flexible user-defined tally infrastructure) developed over more than two decades of production use, but this advantage of usefulness may come at a disadvantage in GPU performance compared to a code with fewer features, like a new code. We observed that offering less device code to Nvidia's nvcc compiler reduced register and stack requirements, thereby improving V100 performance. Less code also increases the chances that the code builds and runs. The disadvantage of writing a new production code is verifying, validating, and maintaining it for decades. The first line of Mercury source code was written more than 20 years ago.

We partially ameliorate the performance impact of Unified Memory through pooling using the Umpire memory manager [13]. Pooling amortizes the expense of cudaMallocManaged which we observed to be orders of magnitude slower than glibc malloc. We also reduce the cost of separate compilation using device code link-time optimization, which we describe in Section 4.

3. EVENT-BASED TRACKING IMPLEMENTATION DETAILS

In 2019, our tracking algorithm used a history-based approach in which each GPU thread follows a particle from birth to death. This big kernel covers about 100,000 lines of branching, latency-bound code. The Nvidia nvcc compiler emits the maximum 255 registers per thread for the kernel, which bounds the theoretical occupancy at about 12.5%, and resulted in an achieved occupancy of about 10%. We realized that we needed smaller kernels to address this limitation. This led to adding event-based tracking, where small kernels chain together to handle a single tracked segment. Along

the way, we redesigned our threading model so each thread would handle a particle instead of a chunk of particles. We initially discovered event-based and history-based algorithms performed similar on the GPU with the mini-app, ALPSMC [9]. Hamilton et al. reached similar conclusions for multigroup Monte Carlo [10] but found the need to use event-based algorithms for continuous energy Monte Carlo to obtain improved GPU performance [14].

In our event-based tracking, a group of particles, called a chunk, execute around 9 kernel launches per segment. (A "segment" denotes each movement of a particle to an event, and a sequence of "segments" constitutes a Monte Carlo particle history.) The chunk size is a user settable parameter which greatly impacts performance on the GPU. Around 6 kernels are launched to find the distance to the next event. The remaining kernels sample those events. The number of kernel calls per segment varies by particle type and runtime options. For example, charged particles have more potential events, and time-dependent calculations have an extra census event. The event-based tracking algorithm reduced the number of registers per kernel launch, which did achieve higher occupancy.

3.1. Changing the particleVault

Our original particle container, called particleVault, originally had a list of particles for processing, processed, and extra memory that had not been deallocated yet. Particles were pulled from the top of the processing list, went through many segments, and were placed in the processed list unless removed from the problem. The cycle was finished once the processing list was empty. New particles produced from variance reduction or collisions were placed at the bottom of the processing list. The next cycle started by swapping the list pointers.

We changed our history-based routines to allow for a thread per particle, causing us to add a processingNext list. Instead of new particles being put on the bottom of processing, they would be put on the bottom of processingNext. Once processing was empty, processingNext would swap particles with processing and continue. The cycle was done when processing and processingNext were both empty. The history-based tracking algorithm is shown in Algorithm 1.

The event-based tracking operates on a chunk containing about 2×10^6 Monte Carlo particles. The first series of kernel launches determine the event each particle will undergo. The chunk of particles is moved the shortest distance and a separate vector of indices for each event is updated for each particle scheduled for that event. If an event has non-zero entries in its index vector, a kernel is launched that is the size of the number of particles participating in that event. At the end of each segment, the tracked particles have either died or been copied to a new chunk. Chunks are maintained in linked lists which may be swapped out until all particles have finished for the current cycle. The event-based tracking algorithm is shown in Algorithm 2.

3.2. Attempts to improve event-based tracking

We attempted many strategies to improve the speed of event-based tracking. Early on we thought that history-based tracking may be better when there are few particles left to process in a cycle. We added an option that allows for event-based tracking to switch to history-based tracking when the particle count was low enough, but we saw no improvement in performance with this change. In the case where most particles make it to the next segment (e.g. scatter, facet crossing, energy boundary crossing), we sought to avoid an extra copy by just re-submitting the kernel, which would skip the few finished particles (e.g. captured, census, Russian Roulette). Early testing demonstrated no benefit. We also tried a register ceiling using `--maxregcount` to increase occupancy. This

usually resulted in a break even or a slowdown. We had some success with writing "lite" versions of routines with shallower call stacks. As an example, if no tallies are registered for an event, we may call the lite version which has the tally routines templated out. We added a sidecar concept that would just contain data needed for the duration of a segment such as the cross section sampled or the facet crossed. This reduced the memory per particle which allowed us to run bigger problems, but we saw no noticeable speed increase. We implemented a routine that would sort particles by the cell they were in. This helped history-based tracking but not event-based tracking. We have yet to look at sorting by cell *and* energy. We implemented mesh based variables that store data in a contiguous array for some data over the geometry. This resulted in speedups in several routines such as a cross section pre-computation routine that went from 17% of the runtime down to 0.1%.

We varied the chunk size to see where our performance peaked. For CTS-1, performance plateaus at a chunk size of 10^3 to 10^5. For ATS-2 Sierra, performance improves up to 2×10^6 and remains high until 5×10^6 after which memory problems cause a decline. Bigger chunks amortize the cost of kernel launch latency by providing more work per kernel launch (because a big chunk contains more Monte Carlo particles than a small chunk). We also looked at launching kernels asynchronously. There were two areas where this made sense: computing the distance to events and executing the events. The distance to event had some ordering problems where distances from one calculation may be used to speed up the next calculation. However, the execution of events had no such issue. It resulted in a modest speedup of about 5%. The lack of a better speedup is probably due to our low occupancy. The V100 can make memory-latency-bound codes memory-bandwidth-bound, but only if the code achieves sufficient occupancy, which ours currently does not.

Algorithm 1: History-Based Tracking Algorithm

```
1  Read in nuclear data
2  while cycles remain do
3      cycleInit
4      while particleVault is not empty do
5          Pre-allocate memory
6          Sort particles by cell
7          parallel for particle in currentChunk in processing do
8              while particle is still active do
9                  Compute distance to cell boundary, dCellBoundary
10                 Sample distance to collision, dCollision
11                 Compute distance to census, dCensus
12                 Compute min (dCellBoundary, dCollision, dCensus)
13                 Move particle to event site
14                 case cellBoundary: update particle's cell
15                 case collision: sample rxn, new particles in processingNext or processed
16                 case census: save particle in processed
17         Swap processing and processingNext pointers
18     cycleFinalize
```

Algorithm 2: Event-Based Tracking Algorithm

```
Read in nuclear data
while cycles remain do
    cycleInit
    while particleVault is not empty do
        Pre-allocate memory
        parallel for particle in currentChunk in processing do
            Compute distance to cell boundary, dCellBoundary
        parallel for particle in currentChunk in processing do
            Sample distance to collision, dCollision
        parallel for particle in currentChunk in processing do
            Compute distance to census, dCensus
        parallel for particle in currentChunk in processing do
            Compute min (dCellBoundary, dCollision, dCensus)
        parallel for particle in currentChunk in processing do
            Move particle to event site
        parallel for particle in cellBoundaryChunk do
            update particle's cell and copy to processingNext
        parallel for particle in collisionChunk do
            sample rxn, new particles in processingNext or processed
        parallel for particle in censusChunk do
            save particle in processed
        Swap processing and processingNext pointers
    cycleFinalize
```

4. CODE GENERATION IMPROVEMENTS

Building an executable requires compiling source code into object code and then linking the object files into a single executable file. A toolchain is the software used for application development. A toolchain includes a compiler and a linker and may also include utilities for inspecting binaries and debugging. For example, in the cuda toolchain the compiler is nvcc and the linker is nvlink.

4.1. Compiler

The compiler reads the source file and all the headers it includes and all the headers included in the headers (and so on) to form a translation unit (TU). The TU consists of definitions (e.g. a function implementation), declarations (e.g. a function signature) and references (e.g. a function call). The compiler performs optimizations on an intermediate representation (IR) before generating object code and then optimizing the object code and emitting it in an object file. In order to generate object code for a function call, the compiler needs the function declaration but not the definition, which permits parallel compilation: each TU is independent of all the others and so all TUs may be compiled simultaneously. This is known as separate compilation.

4.2. Linker

The linker links all the object code to form the executable. Unlike the compiler, the linker needs the definition for every reference - that is why the linker can fail with "undefined reference" errors. The main job of the linker is reference resolution. At each function call there is a jump instruction emitted by the compiler with a blank destination that needs to be replaced with the address of the function definition. The linker determines the correct address. (We resolve references at build time, which is called static linking. One can also resolve them at run time, which is called dynamic linking [*].) The linker does not perform optimizations.

4.3. Whole program optimization

The compiler optimizes every TU, but since the compiler operates on a single TU it cannot perform cross-TU optimizations, such as inlining a function which is referenced in one TU but defined in another. However, if one can fit an application into a single TU then this problem goes away. This is known as a unity build, and it creates other problems which we will not enumerate here. Unlike the compiler, the linker sees all the code, so the linker can perform cross-TU optimizations, also known as whole program optimizations. When performed by the linker this is known as link time optimization (LTO).

4.4. LTO in the cuda toolchain

The linker cannot optimize object code but it can optimize IR. Running nvcc with the `-dlto` flag emits NVVM IR into the object file. Invoking nvcc on the objects along with `-dlto` will cause nvlink to perform LTO on the device code. This improves application runtime at the cost of linker runtime. Today it takes 20 minutes for nvlink to link Mercury with LTO, thanks to the ptxas flag `--fast-compile`, without which it takes 8 hours. Other challenges include:

- Stack overflows. Some calculations hang or die with various cuda errors, like "Warp Out-of-range Address", "illegal memory access", and "unspecified launch failure." Our code uses recursion which means that the maximum stack size is not known until runtime. In practice this means increasing `cudaLimitStackSize` and re-running until the calculation runs to completion.
- Build failures. Nvlink failed to link and output the LLVM error "invalid user of intrinsic instruction!" An expert at Nvidia provided a workaround by adding a variable to a class definition in our code so that the class type did not match the signature of an unrelated compiler builtin function in a different NVVM module.
- Miscompiles. Some calculations die with the error "illegal memory access". We discovered that adding the ptxas flag `--disable-optimizer-constants` fixes the problem. For the total tracking time of our regression testing suites, the DOC flag causes the Mercury speedup to drop 11 percentage points from 1.34x to 1.23x and the Imp speedup to drop 3 percentage points from 1.28x to 1.25x.
- Sensitivity. The LTO performance numbers reported in this paper represent the current production release of the code. Seemingly small code changes, such as adding a print statement, can prevent inlining and unexpectedly reduce the LTO speedup. The development branch has lost some LTO performance. We are working to understand and recover this performance.

[*]Some of our references cannot be resolved at build time due to late binding, for example virtual function references.

None of these problems have stopped us from using LTO, and we expect LTO to continue to improve as it has over the three years since Nvidia released it in June 2020. LTO reduces the number of device functions that survived inlining from 938 to 390 in Mercury and 461 to 82 in Imp. Of the 390 that remain in Mercury, 310 come from using a collision library in which we have many virtual functions. A virtual function cannot be inlined.

4.5. ThinLTO in the LLVM toolchain

The maximum cuda LTO speedup of about 27% that we first reported in 2021 on Sierra [15], which is now 36% (see Table 2), compares favorably with 6% observed on CTS-1 with LLVM's ThinLTO build of Mercury [16]. We attribute the difference to the relative cost of function call overhead on the two architectures (higher on V100 than x86).

5. RESULTS

We compare calculations running on CTS-1 and ATS-2 Sierra for which we always run on one node (see Section 1 for descriptions of the CTS-1 and ATS-2 Sierra node architectures). On CTS-1 we compiled with Intel icpc 19.1.0 and on Sierra with Nvidia nvcc 11.7 using clang 14.0.5. We ran version 5.34.0 of Mercury and Imp on two test problems. Godiva in Water [17] is a Godiva critical sphere surrounded by water. We used Mercury to calculate the k-eigenvalue in a 3D Cartesian mesh octant geometry with 27,000 zones, 1 domain, and continuous energy nuclear data. Crooked Pipe [18] is an idealized radiation transport test problem in which a boundary temperature source induces radiative heat flow down a pipe with a bend. We ran Imp for 100 timesteps of variable size and calculated the temperature at five fiducial points in a 2D RZ mesh geometry with 5,000 zones, 1 domain, constant gray opacity, constant specific heat, and a 300 eV black body source.

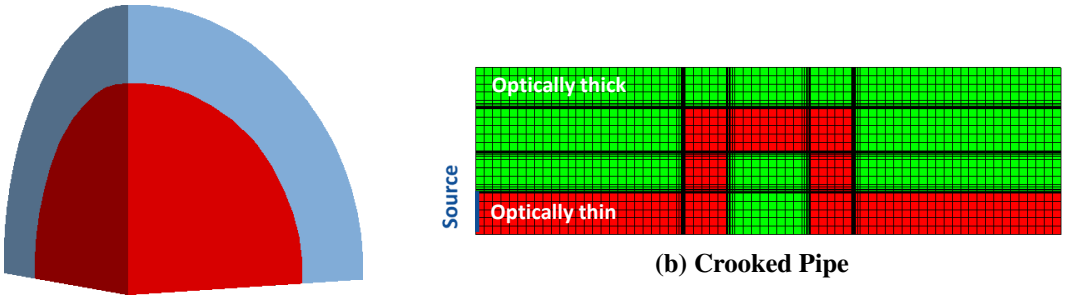
Figure 2 shows the geometry for both problems. Table 1 shows how our GPU speedup has improved over fiscal years (FY) since Sierra came online in 2018. The "speedup" in this table is the ratio of the total runtime on CTS-1 and Sierra. We ran the FY23 calculations with 2^{27} (about 134 million) particles.

We chose 2^{27} particles based on a throughput study which determined 2^{27} to be the saturation point at which the maximum throughput is achieved. We began with 2^{10} particles and increased the number of particles in powers of two. We plotted segments-per-second, which is the number of segments computed divided by the time spent tracking particles (a sequence of "segments" constitutes a Monte Carlo particle history). Figure 3 shows that throughput on a CTS-1 node saturates at only 2^{17} particles whereas an ATS-2 Sierra node saturates at about 2^{27} particles, so about 1000x more work is required to saturate Sierra.

Table 2 and Table 3 show how the new system (Sierra), the new algorithm (Event), and the new build (LTO) all contribute to the overall speedup. For example, LTO improved our Mercury speedup by 7.745/5.707=1.357 or about 36%. The "speedup" in this table is the ratio of segments-per-second. The 7.61x and 5.81x speedups in the FY23 column of Table 1 are slightly less than the 7.75x and 5.92x speedups in Table 2 and Table 3 because the former use total runtime and the latter only tracking time. Table 2 and Table 3 show that tracking time is over 80% of Imp runtime and over 90% of Mercury runtime.

6. CONCLUSIONS AND FUTURE WORK

After six years of intermittent porting, we have achieved a 7.61x speedup for neutronics and 5.81x for thermal photonics in node-to-node comparisons between the ATS-2 Sierra supercomputer and



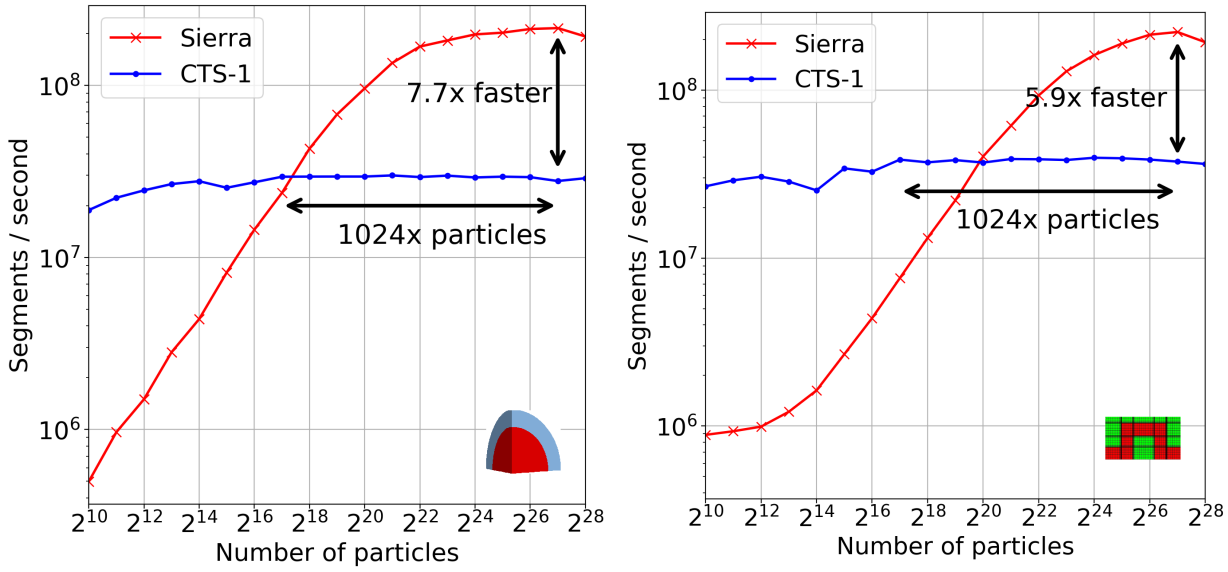
(a) Godiva in Water

(b) Crooked Pipe

Figure 2: Problem geometry for (a) Mercury and (b) Imp calculations.

Table 1: Speedups from FY18 to FY23 (calculations not completely consistent year-to-year)

			Total Runtime Speedup				
	Resources	CPU / GPU	FY18	FY19	FY21	FY22	FY23
Godiva in Water	CTS-1/Sierra	36 CPU cores/4 GPUs	0.47x	0.81x	4.43x	5.15x	7.61x
Crooked Pipe	CTS-1/Sierra	36 CPU cores/4 GPUs	1.45x	1.99x	4.52x	4.90x	5.81x



(a) Godiva in Water

(b) Crooked Pipe

Figure 3: Throughput for (a) Mercury and (b) Imp calculations.

Table 2: Speedups for Mercury Godiva in Water calculation by system, algorithm, and LTO

System	Tracking Algorithm	LTO?	Total Runtime (s)	Tracking Time (s)	Segments (billions)	Seg/s (millions/s)	Speedup
CTS-1	History	No LTO	6170	6006	167.113	27.8	1.000
Sierra	History	No LTO	3744	3710	167.113	45.0	1.619
Sierra	Event	No LTO	1085	1052	167.113	158.8	5.707
Sierra	Event	LTO	811	775	167.113	215.5	7.745

Table 3: Speedups for Imp Crooked Pipe calculation by system, algorithm, and LTO

System	Tracking Algorithm	LTO?	Total Runtime (s)	Tracking Time (s)	Segments (billions)	Seg/s (millions/s)	Speedup
CTS-1	History	No LTO	2711	2334	87.707	37.6	1.000
Sierra	History	No LTO	913	836	87.709	104.9	2.791
Sierra	Event	No LTO	555	477	87.706	183.7	4.888
Sierra	Event	LTO	467	395	87.708	222.3	5.915

traditional large scale x86 systems like CTS-1 for the LLNL production Mercury and Imp codes on the Godiva in Water and Crooked Pipe problems, respectively. Future work includes ongoing GPU performance improvement investigations and porting to El Capitan, a future LLNL system that will have AMD EPYC processors and AMD Radeon Instinct GPUs [19].

ACKNOWLEDGEMENTS

We thank Nvidia employees Jaydeep Marathe for helping us with LTO and Mike Murphy for implementing it. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

REFERENCES

[1] R. Procassini, D. Cullen, G. Greenman, and C. Hagmann. "Verification and Validation of Mercury: A Modern, Monte Carlo Particle Transport Code." In *Proceedings of MC2005*. Chattanooga, TN, USA (2005).

[2] J. A. Fleck and J. D. Cummings. "An implicit Monte Carlo scheme for calculating time and frequency dependent nonlinear radiation transport." *Journal of Computational Physics*, **volume 8**, pp. 313–342 (1971).

[3] P. Brantley, N. Gentile, M. Lambert, M. McKinley, M. O'Brien, and J. Walsh. "A New Implicit Monte Carlo Thermal Photon Transport Capability Developed Using Shared Monte Carlo Infrastructure." In *Proceedings of M&C 2019*, pp. 564–577. Portland, OR, USA (2019).

[4] S. S. Vazhkudai, B. R. de Supinski, A. S. Bland, A. Geist, J. Sexton, J. Kahle, C. J. Zimmer, S. Atchley, S. Oral, D. E. Maxwell, V. G. V. Larrea, and A. Bertsch. "The Design, Deployment, and Evaluation of the CORAL Pre-Exascale Systems." SC '18 (2018).

[5] M. McKinley, R. Bleile, P. Brantley, S. Dawson, M. O'Brien, M. Pozulp, and D. Richards. "Status of LLNL Monte Carlo Transport Codes on Sierra GPUs." In *Proceedings of M&C 2019*, pp. 2160–2165. Portland, OR, USA (2019).

[6] F. B. Brown and W. R. Martin. "Monte Carlo methods for radiation transport analysis on vector computers." *Progress in Nuclear Energy*, **volume 14**(3), pp. 269–299 (1984).

[7] R. M. Bergmann and J. L. Vujić. "Algorithmic choices in WARP – A framework for continuous energy Monte Carlo neutron transport in general 3D geometries on GPUs." *Annals of Nuclear Energy*, **volume 77**, pp. 176–193 (2015).

[8] R. Bergmann, K. Rowland, N. Radnović, R. Slaybaugh, and J. Vujić. "Performance and accuracy of criticality calculations performed using WARP – A framework for continuous energy Monte Carlo neutron transport in general 3D geometries on GPUs." *Annals of Nuclear Energy*, **volume 103**, p. 334–349 (2017).

[9] R. Bleile, P. Brantley, M. O'Brien, and H. Childs. "Algorithmic Improvements for Portable Event-Based Monte Carlo Transport Using the Nvidia Thrust Library." *Transactions American Nuclear Society*, **volume 115**, pp. 535–538 (2016).

[10] S. P. Hamilton, S. R. Slattery, and T. M. Evans. "Multigroup Monte Carlo on GPUs: Comparison of history- and event-based algorithms." *Annals of Nuclear Energy*, **volume 113**, pp. 506–518 (2018).

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

[12] N. Choi, K. M. Kim, and H. Joo. "Optimization of neutron tracking algorithms for GPU-based continuous energy Monte Carlo calculation." *Annals of Nuclear Energy*, **volume 162**, p. 108508 (2021).

[13] D. A. Beckingsale, M. J. McFadden, J. P. S. Dahm, R. Pankajakshan, and R. D. Hornung. "Umpire: Application-focused management and coordination of complex hierarchical memory." *IBM Journal of Research and Development*, **volume 64**(3), pp. 1–10 (2020).

[14] S. P. Hamilton, T. M. Evans, and S. R. Slattery. "Continuous-Energy Monte Carlo Neutron Transport on GPUs in Shift." *Transactions American Nuclear Society*, **volume 118**, pp. 401–403 (2018).

[15] D. F. Richards and B. S. Ryujin. "Enhancements supporting IC usage of PEM libraries on next-gen platforms." (2021).

[16] M. Pozulp, S. Dawson, R. Bleile, P. Brantley, M. S. McKinley, M. O'Brien, and D. Richards. "Transitioning the Scientific Software Toolchain to Clang/LLVM." Paris, France (2020).

[17] D. E. Cullen, C. J. Clouse, R. Procassini, and R. C. Little. "Static and Dynamic Criticality: Are They Different?" Technical report, UCRL-TR-201506 (2003).

[18] F. Graziani and J. LeBlanc. "The Crooked Pipe Test Problem." Technical report, UCRL-MI-143393 (2000).

[19] "LLNL and HPE to partner with AMD on El Capitan, projected as world's fastest supercomputer." (2020). URL https://www.llnl.gov/news/llnl-and-hpe-partner-amd-el-capitan-projected-worlds-fastest-supercomputer.