Status of LLNL Monte Carlo Transport Codes on Sierra GPUs

M.S. McKinley, R. Bleile, P.S. Brantley, S. Dawson, M. O'Brien, M. Pozulp, D. Richards
{mckinley9, bleile1, brantley1, dawson6, obrien20, pozulp1, richards12}@llnl.gov

ABSTRACT

The LLNL Monte Carlo Transport Project has been working to port Mercury, a Monte Carlo particle transport code, along with Imp, an IMC thermal photon transport code, for the new Sierra architecture. Sierra is an IBM Power9 CPU and Nvidia Volta V100 GPU supercomputer. We present our approaches to porting code to this next generation machine including code sharing, memory management, threading models, mini-apps, load balancing, data replication, and loop speedups outside of direct particle tracking. We share our latest performance speedups compared to a commodity technology system (CTS-1) node with 36 Intel Broadwell CPU cores. Finally, we outline our future plans to further improve performance.

KEYWORDS: Monte Carlo, Sierra GPU, Imp, Mercury

1. INTRODUCTION

Mercury [1] is a Monte Carlo particle transport code developed at the Lawrence Livermore National Laboratory. Mercury can transport neutrons, gamma rays and light charged particles through meshes and/or 3-D combinatorial geometry. The collision physics is handled by the GIDI library [2] which supports multigroup and continuous energy cross sections with all collisions being continuous energy. Imp is a new implicit Monte Carlo x-ray thermal photon transport code [3] being developed using Monte Carlo infrastructure shared with Mercury. A major motivation for developing a new IMC code using Monte Carlo shared infrastructure was to enable the porting of both capabilities to the Sierra GPU architecture.

Parallelism in Mercury and Imp is handled through domain replication and domain decomposition on top of MPI, OpenMP, and Cuda. The codes are about 300k lines of C++ with a Python interface. Monte Carlo transport is not naturally well suited for GPUs due to substantial branch divergence, lots of random memory access, and few floating point operations.

Lawrence Livermore National Laboratory (LLNL) has accepted the Sierra supercomputer, which is currently ranked 2nd in the world [4]. The Sierra architecture is based on 3.1 GHz IBM Power9 CPUs and Nvidia Tesla V100 GPUs. A Sierra node has 4 GPUs and 2 CPUs with 20 cores each. Over 95% of the FLOPS are on the GPUs. For comparisons to a CPU-only architecture, we use a CTS-1 machine with dual-socket 2.1 GHz Intel Xeon Broadwell nodes providing 36 CPU cores per node.

In this paper, we discuss our multiple approaches to modifying Mercury and Imp to adapt to new computing architectures. We also present selected performance results. Finally, we end with planned future work.

2. APPROACHES TO PORTING TO GPU

Porting the Monte Carlo codes to the Sierra GPU architecture involved several years of effort and planning. Our first realization was that we did not want to duplicate work for particle and implicit thermal photon Monte Carlo, so we worked to share infrastructure between Mercury and Imp. Along the way, we developed mini-apps to scope out algorithmic and memory issues. The nuclear data library, GIDI [2], had to be updated

to work on GPUs for Mercury. We investigated several issues such as memory management; history- vs. event-based tracking; fat vs. thin threading; CPU/GPU load balancing; variable replication in tallies; and loop speedups in the initialization/finalization stages of the calculation.

2.1. Code Sharing

Significant overlap exists in the computer science infrastructure needed to write a neutron transport Monte Carlo code and an IMC thermal x-ray photon transport code. To avoid duplication of effort and to enable porting of both codes to advanced computing architectures, we have merged our neutron Monte Carlo effort and our IMC effort. In terms of lines of code, approximately 80% is shared between the Mercury Monte Carlo particle transport code and the new Imp IMC thermal photon code [3].

Features that are common to both particle MC and IMC include sourcing particles; having domain decomposition/domain replication/dynamic load balancing; ray tracing to the zone boundary; buffering and communicating particles when they cross domain/processor boundaries; determining that asynchronous particle streaming communication has completed; reducing tallies over domain replicas; and having a general source/tally/variance reduction infrastructure. All of these operations are agnostic as to the type of particle being transported. Specific physics for each particle type is not part of the shared infrastructure. This means that most of the changes necessary to run and be performant on the GPU can be made once, in a single code base.

2.2. Memory Model

While several options exist for handling memory, our team has initially chosen to use managed memory. This allows the GPU to access memory that is allocated on the host. The operating system will move data as needed between GPU and CPU memory. Pointers to managed memory are valid on both the CPU and GPU. The downsides of managed memory are that allocation with cudaMallocManaged is significantly slower than ordinary malloc, function pointers are not valid on both CPU and GPU, and care must be taken to avoid needless data motion between the CPU and GPU. To mitigate these issues, we reduced the number of memory allocation calls by consolidating data structures. We are also working to limit data motion by moving as many loops as possible to the GPU thus reducing the need for data migration. Finally, we are integrating the Umpire [5] pooled memory manager to reduce the cost of memory allocations. Early testing with Umpire has not yet shown any major speedups, mainly due to past re-writes of the code to be more memory friendly. If Umpire had been available earlier, it may have saved time in refactoring our code to have fewer memory allocation calls.

2.3. Porting GIDI To CUDA

GIDI is a new nuclear data library developed at LLNL [2]. A companion library, MCGIDI, handles cross section lookup and collision physics for Monte Carlo. Both libraries are written in C++ and use virtual functions. Unfortunately, objects with virtual functions are not copied correctly from the CPU to the GPU by managed memory. We resolved this issue by overloading the new operator and writing serialize / deserialize functions that work with memory placement. Nuclear data objects (or any objects with virtual functions) can be serialized into a simple data stream and copied to the GPU. On the GPU, the data is deserialized using placement new and some helper functions to determine memory placement. This approach results in only one real call for memory allocation for the data with placement new used to call constructors and set up virtual pointers.

2.4. Mini-Apps

Two mini-apps were used to initially scope out issues and algorithms. The first and simplest was a small Monte Carlo test code of approximately 1k lines called ALPSMC that was used to investigate history- versus event-based tracking algorithms on the GPU [6,7]. The event-based algorithm seeks to reduce divergence by grouping particles undergoing the same events at the cost of some kind of sorting process. After initial performance optimizations, the history- and event-based algorithms performed comparably on the GPU [7]. Hamilton et al. reached similar conclusions for multigroup Monte Carlo [8] but found the need to use event-based algorithms for continuous energy Monte Carlo to obtain improved performance [9].

We also developed Quicksilver [10], a 7k line C++ code that supports MPI, OpenMP, and CUDA. Quicksilver models a time-dependent, fixed-sourced transport problem on a 3-D mesh with artificial multigroup nuclear data. Quicksilver was used to investigate whether the entire tracking loop could be executed as a single big GPU kernel as well as to test a fat vs. thin thread approach. The fat thread model has a separate container of particles for each thread, and data races are managed by replicating any data structure that could possibly be changed by multiple threads simultaneously. Thin threads share a particle container, and data races are managed with atomics. Quicksilver showed acceptable performance for the big kernel approach. It also demonstrated that the thin thread model did work better on the GPU and could perform well on the CPU with minimal losses in speed.

2.5. Big Kernel

GPU kernels are usually small, computationally intense loop bodies. Because particle tracking has no such loops, we turned instead to a big kernel strategy. In this approach, the entire tracking loop (approximately 80,000 lines of code reachable through function calls), is converted to a single GPU kernel. Each particle is handled by a single GPU thread. The thread follows the particle through collisions, facet crossings and other events until the particle is complete for this cycle. This approach complements our existing CPU threading strategy well, but it introduces several issues such as the need to port many high-level routines, increased divergence, and making compiler optimizations more difficult. This approach also uses a large number of GPU registers which limits the number of in-flight particles that the GPU can process and also causes registers spills which means more memory traffic which slows down the code.

2.6. CPU / GPU Load Balancing

MPI based load balancing has existed in Mercury for a long time. Originally, the unit of work was a particle segment, and it was assumed that all processors were equally efficient at computing particle segments. The heterogenous nature of CPU/GPU platforms challenges this assumption. In response, we now also incorporate the relative particle processing rate of the CPUs and GPUs using the previous cycle's processing rate to help predict the work that computational unit can do the next cycle. The goal is that all CPU threads end at the same time as GPU threads end. All CPUs and all GPUs are assumed to process particles at the same rate.

2.7. Variable Tally Replication

In the fat thread model, each MPI rank replicated its tallies to maintain one copy per OpenMP thread. With GPUs, there are too many threads to replicate many of our tallies, so we use atomics to avoid data races. Unfortunately, this can lead to slowdowns when many threads attempt to update the same tally. Our solution is to allow each tally to have a variable number of replications. For example, the scalar flux tally is a tally over particle type, cell, material, and energy group. This tally does not need to be replicated many times since it is relatively uncommon for threads to need to access the same value at the same time. In contrast, the tally of the total number of overall segments is a single integer that is accessed by every thread for every particle movement. For this small tally we can easily afford several replicas to reduce contention.

2.8. Initialization / Finalization Speedups

Once the particle tracking loop was initially ported to GPUs, new bottlenecks became apparent in the initialization and finalization sections of the code. Work was done to move particle sourcing and tally accumulation loops onto the GPUs, accomplishing two objectives. First, it gives more work to the GPUs, and second it keeps data on the GPU and avoids shuffling data back and forth. Most of this work was done by constructing a loop macro that can be used to loop over OpenMP threads or GPU threads.

2.9. GPU Porting Conclusions

Putting all this together, we have succeeded in creating an initial port of the Mercury and Imp codes that can run on the Sierra GPU architecture. In order to stay versatile, we allow for running in many configurations across nodes. We support mixing MPI on CPU cores with OpenMP threads on CPU cores as well as with up to 4 GPUs per node. Each GPU is currently assigned to an MPI CPU core. In the future, we may allow for more versatility by allowing multiple CPU cores to launch kernels on the same GPU or a CPU core to drive multiple GPUs. Significant work goes into trying to search for optimal configurations as the code evolves.

Since we are using the big kernel approach, we have about 80 thousand lines of code that can be run on the GPU device. This spans roughly a thousand functions. We split apart the collision routine between Imp and Mercury to help speed up Imp. In Imp, every collision is some form of scattering in which the incoming particle can be changed to an outgoing particle without creating temporary storage for particles. For Mercury, particles may disappear or increase in a collision. In addition, there is a lot more data for dealing with all possible nuclear reactions in Mercury.

3. RUNTIME SPEED COMPARISONS

With all of these initial porting changes, we ran the codes through an example problem to determine speed changes. We compare the total runtime, particle tracking time and initialize / finalize times for a node of a CTS-1 machine versus a node of a Sierra equivalent machine. Table I shows the results for Mercury running a problem with a critical sphere of uranium in water with 4 million particles and continuous energy nuclear data. We see an overall speedup of 1.20X versus a CTS-1 node. If we just look at running on the P9 CPU cores versus enabling GPUs, we see a speedup of 1.48X. Since most of the time is spent tracking particles (in the Tracking Time column), the slowness of initialization and finalization does not play too much into this problem. However, this can dominate for other problems.

Table I. Mercury Runtimes: Godiva in Water, CG, continuous energy

Resources	CPU / GPU	Total Time (minutes)	Tracking Time (minutes)	Init / Final Time (minutes)
CTS-1	36 cores	1.43	1.28	0.16
P9	40 cores	1.76 (0.82X)	1.54 (0.83X)	0.22 (0.72X)
V100	4 GPUs	1.76 (0.81X)	1.59 (0.81X)	0.18 (0.89X)
V100+P9	4 GPUs + 36 cores	1.19 (1.20X)	0.88 (1.45X)	0.31 (0.50X)

Table II shows results for running Imp on the crooked pipe test problem [11] with 20 million simulation particles per time step and a final simulation time of 10^{-6} seconds. It shows a much improved overall 2.9X speedup. And the speedup of P9 versus P9 with GPUs is 3.7X. Imp appears to be more efficient running on the GPU than Mercury mostly due to the handling of collisions.

Table II. Imp Runtimes: Crooked Pipe with $20x10^6$ MC photons per time step

Resources	CPU / GPU	Total Time (minutes)	Tracking Time (minutes)	Init / Final Time (minutes)
CTS-1	36 cores	23.67	21.83	1.67
P9	40 cores	30.33 (0.78X)	27.33 (0.80X)	1.72 (0.97X)
V100	4 GPUs	9.05 (2.62X)	7.52 (2.90X)	1.55 (1.08X)
V100+P9	4 GPUs + 36 cores	8.17 (2.9X)	6.37 (3.43X)	0.68 (2.44X)

4. CONCLUSIONS AND FUTURE WORK

We have developed an initial port of the Mercury and Imp codes that can run on the Sierra GPU architecture. We are continuing to port Mercury and Imp to the Sierra GPUs and are beginning to focus on optimizing performance. Most of the current work is focused on understanding our performance data. There are a lot of issues to look at and evaluate. And when one issue is discovered and addressed, another one that has already been analyzed may rise to the top as the next big issue.

Since we are using managed memory, data may move between the CPU and GPU with relative coding ease, but at a cost to performance. We are going to continue to try to keep data either on the CPU or on the device for as long as possible. Some of our initialization and finalization routines may loop over data on the CPU. We plan to recode these to be GPU routines if the data is currently on the GPU.

In some of our performance analysis, register pressure and spilling over shows up as a major reason for the current slow performance. In addition, since we are using the big kernel approach, we call many functions which increases the thread stack size. Currently, these constraints limit us to about 256 threads in-flight per SM. In response to these issues, we are going to investigate breaking our big kernel into several smaller

ones. As we subdivide our big kernel, we will use profiling tools to see what routines should be divided further. This should naturally lead to some event-based splitting of loops.

ACKNOWLEDGMENTS

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. P. S. Brantley, R. C. Bleile, S. A. Dawson, M. S. McKinley, M. J. O'Brien, M. Pozulp, R. J. Procassini, D. Richards, S. M. Sepke, and D. E. Stevens. "Mercury User Guide: Version 5.12." *Lawrence Livermore National Laboratory Report LLNL-SM-560687 (Modification #15)* (2018).
2. M.-A. Descalle, "Implementation Of The GNDs Format For Evaluated And Processed Nuclear Data", *PHYSOR 2018*, Cancun, Mexico, April 22-26 (2018).
3. P. S. Brantley, N. A. Gentile, M. A. Lambert, M. S. McKinley, M. J. O'Brien, J. A. Walsh, "A New Implicit Monte Carlo Thermal Photon Transport Capability Developed Using Shared Monte Carlo Infrastructure," *M&C 2019*, Portland, Oregon (2019).
4. "TOP500.org," https://www.top500.org/lists/2018/11/ (2018).
5. D.A. Beckingsale, M. Mcfadden, and R. D. Hornung, "Umpire: Integration with Other Software Technologies," *Lawrence Livermore National Laboratory Report LLNL-TR-764330* (2018).
6. R. C. Bleile, P. S. Brantley, S. A. Dawson, M. J. O'Brien, H. Childs, "Investigation of Portable Event-Based Monte Carlo Transport Using the NVIDIA Thrust Library," *Trans. Am. Nuc. Soc.*, **volume 114**, pp. 941-944, (2016). On USB.
7. R. C. Bleile, P. S. Brantley, M. J. O'Brien, H. Childs, "Algorithmic Improvements for Portable Event-Based Monte Carlo Transport Using the Nvidia Thrust Library," *Trans. Am. Nuc. Soc.,* **volume 115**, pp. 535-538, (2016). On USB.
8. S. P. Hamilton, S. R. Slattery, 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).
9. S. P. Hamilton, T. M. Evans, S. R. Slattery, "Continuous-Energy Monte Carlo Neutron Transport on GPUs in Shift," *Trans. Am. Nuc. Soc.*, **volume 118**, pp. 401-403, (2018). On USB.
10. D. Richards, P. Brantley, S. McKinley, M. O'Brien, "Quicksilver: A Mini-App for Mercury," *LLNL Technical Report LLNL-SM-668536* (2015).
11. F. Graziani and J. LeBlanc. "The Crooked Pipe Test Problem." Lawrence Livermore National Laboratory Report UCRL-MI-143393 (2000).