Algorithmic Improvements for Portable Event-Based Monte Carlo Transport Using the Nvidia Thrust Library

Ryan C. Bleile[*,†], Patrick S. Brantley[*], Matthew J. O'Brien[*], Hank Childs[†]

[*]*Lawrence Livermore National Laboratory, P.O. Box 808, Livermore, CA 94551*
[†]*Department of Computer and Information Science, University of Oregon, Eugene, OR 97403*
bleile1@llnl.gov, brantley1@llnl.gov, obrien20@llnl.gov, hank@uoregon.edu

INTRODUCTION

High performance computing environments are progressively moving towards many-core computing architectures. The Los Alamos National Laboratory Trinity machine, available in late 2016, will use both Intel Xeon Haswell processors and Intel Xeon Phi Knights Landing many integrated core (MIC) coprocessors. The Lawrence Livermore National Laboratory Sierra machine, available in 2018, will use an IBM PowerPC architecture along with Nvidia graphics processing units (GPUs). Applications that must work in this supercomputing environment must continue to adapt in order to take advantage of the diverse hardware architectures that are coming. A significant consideration is not only the performance of the application on a given platform but also the portability of the application to other platforms.

The algorithmic improvements presented in this paper build upon recently-reported work [1] on event-based Monte Carlo transport in the ALPSMC code that models particle transport in one-dimensional binary stochastic media [2]. That paper discussed the lack of available vectorization in the traditional history-based algorithm used for Monte Carlo transport and presented a data parallel event-based algorithm implemented using the Nvidia Thrust library [3] for portability. The performance of the data parallel event-based algorithm implemented using Thrust was compared to a native CUDA [4] implementation. The conclusions from that work were that the Thrust library abstraction technique caused too significant a loss in performance but that the event-based method was a viable option that should be further investigated.

In this paper, we describe algorithmic improvements to the data parallel event-based algorithm previously presented [1]. We made further algorithmic optimizations to the event-based CUDA implementation, most notably: data structure changes, a new conditional particle removal scheme in the event-based process, and the use of multiple GPUs. In addition to improvements to the algorithm, we reimplemented the Thrust version from the now further optimized CUDA version, giving a greater chance for success at a performant abstraction. Finally, we revisited our previous assumptions about the inability of the history-based method to achieve performance on vector style architectures such as the MICs and GPUs, with surprising and promising results.

ARRAYS OF STRUCTURES VERSUS STRUCTURES OF ARRAYS

One way to gain performance on Nvidia GPUs is to coalesce global memory accesses in a streaming multiprocessor (SM) [5]. SMs schedule and execute threads in lock-step groups of 32 threads called warps. Memory accesses in each warp are coalesced in order to produce fewer memory transactions overall. For 16 threads in a warp, we can pull all 16 array values into the threads with a single call into global memory if we access the array in consecutive order. Since the threads in a warp operate in lock step, if memory accesses are not coalesced, more memory transactions are needed causing the threads in a warp to stall while more memory transactions are issued.

In order to accomplish coalesced memory accesses on larger data structures, such as the particle class used in the ALPSMC C++ Monte Carlo implementation [2], a common recommendation is to transition from an array of structures (AOS) data structure to a structure of arrays (SOA) data structure [6]. This transition is important for all SIMD or vector architectures (not only GPU architectures [6]) and so makes sense as a starting point for continuing optimizations in our ALPSMC study. For ALPSMC, this transformation requires that the member variables in the particle class are separated into different arrays of those members in a parent class. This entire process can be encapsulated in a higher level interface allowing for a compile time choice to be made for which data structure option to use: AOS or SOA. Maintaining flexibility in this type of option is important when running on a diverse set of hardware, where common and important optimizations on one set of hardware might lead to performance loss on another.

OPTIMIZED PARTICLE REMOVAL SCHEME

One discovery we made while pursuing optimizations for the event-based algorithm was the significant percentage of time spent removing inactive particles from the particle list. (The event-based algorithm is described in detail in Ref. [1]; we omit a detailed description here due to space limitations.) The algorithm originally collected all particles undergoing each of the events and then performed that event on the list of particles. The last stage of the original algorithm was to always remove inactive particles (e.g. particles that were absorbed), i.e. perform material interface crossing events, zone boundary crossing events, collision events, and then remove inactive particles. Table I shows the wall clock times of each event for an example problem (Case 1a [2] with a spatial domain of 10 cm) with ten million particles for both the AOS and SOA data structure implementations. All simulations in this paper were performed on the LLNL Rzhasgpu computer described in the Numerical Results section. We can clearly see that always removing inactive particles dominates the time spent processing the events. We conjectured that if we could minimize this removal function, even at the cost of increasing compute time, we might be able to decrease the overall time spent processing events.

In order to justify removing inactive particles at all, we investigated not removing particles to show that, while

TABLE I: Wall clock times [s] for each event for a 10 million particle study using the CUDA event-based method.

	AOS Data Structure		
Event	Remove Always	Remove Never	Remove Half Size
Material Interface	0.50	4.13	0.60
Zone Boundary	0.81	4.79	0.90
Collision	1.14	5.55	1.35
Remove	2.83	3.15	0.88
Total	5.28	17.62	3.77
	SOA Data Structure		
Event	Remove Always	Remove Never	Remove Half Size
Material Interface	0.39	2.23	0.44
Zone Boundary	0.64	2.98	0.77
Collision	0.73	3.50	0.93
Remove	4.46	1.48	0.87
Total	6.22	10.19	3.01

removing particles is an expensive operation, it is more costly to operate on the full list every time. There is still some amount of time spent in the Remove stage of the algorithm because it is necessary to check if all particles have completed processing, which is the end condition for the simulation. In light of our conjecture above, we also implemented an algorithm in which we only remove inactive particles when removing them produces a significant impact on the size of the list. Following numerical experimentation, we chose to perform the remove operation if the number of inactive particles to be removed is at least half the size of the list. As a result, the maximum number of times we perform the expensive removal operation becomes log(n), where n is the size of the list. As shown in Table I, the "Remove Half Size" algorithm produces a 1.4X and 2.1X improvement in total wall clock time over the "Remove Always" algorithm for the AOS and SOA implementations, respectively. Finally, we observe that the SOA implementation produces a 1.3X improvement in total wall clock compared to the AOS implementation.

We also investigated replacing the Remove function with a sort function to understand the full effect on compute and remove times, both sorting each iteration and in the same remove half scheme described above. The sorting resulted in a significant slowdown for each method when compared to the most efficient approach: 84.9X slower when sorting each time and 3.2X slower when using the remove half scheme. Overall, increasing removal times (that includes the time for the sort) far outweighs the cost of decreasing the compute times. As a result, sorting is not effective, even when we include the new conditional sorting scheme.

NUMERICAL RESULTS

All simulations in this paper were performed on the LLNL Rzhasgpu computer that has 16 Intel Xeon Haswell 3.2 GHz host cores with 2 Nvidia Tesla K80 GPU device accelerators per node. GPU results are run in maximum batch sizes of 10 million particles on a given CUDA device, with multiple batches able to run on different devices at one time. Each Nvidia Tesla K80 appears as two devices, so there are four CUDA devices usable at a time.

Thrust and CUDA Event-Based Approach

The main conclusion from our previous paper [1] was that the event-based Monte Carlo transport algorithm is viable on GPU architectures, but the use of the Thrust library portability abstraction resulted in a significant performance penalty. With the goal of reducing this performance penalty, we reimplemented the Thrust library version of the code (that could run on CPUs and GPUs) based on the the most efficient CUDA version that incorporated the algorithmic modifications described above as well as some additional minor optimizations. This section presents the results of this work.

Figure 1 shows the results of a particle scaling study performed with the optimized event-based CUDA version compared to the original, serial, history-based version. The CUDA version has a higher initial overhead and so is less efficient than the serial history-based version at low numbers of Monte Carlo particles. As the number of Monte Carlo particles increases, the CUDA version scales in a super linear fashion. After the number of particles exceeds the batching threshold [1], the wall clock time of the CUDA version begins scaling linearly as we would expect.

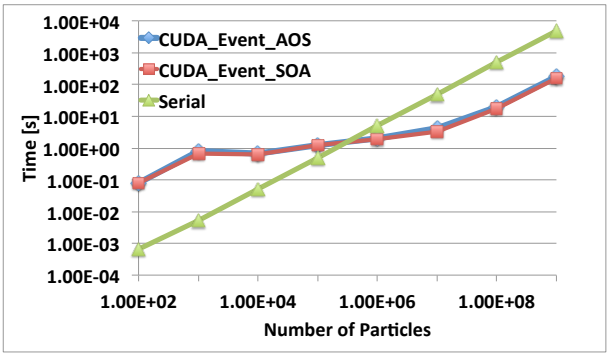

Fig. 1: Log-log plot showing wall clock times versus number of particles for the CUDA event-based algorithm compared to the original serial history-based algorithm.

Figure 2 shows the results of a particle scaling study performed with the optimized Thrust version of the ALPSMC code that was generated from the optimized CUDA version. In contrast to the results of our previous paper [1], the optimized Thrust version now exhibits the same performance characteristics as the optimized CUDA version. After making these algorithmic transformations, the Thrust version now performs slightly more efficiently than the CUDA version. This outcome demonstrates that an abstraction layer can be performant as long as the code in the abstraction layer uses the same optimizations utilized in the explicit CUDA implementation.

When we closely inspect the results from the explicit CUDA version compared with those from the Thrust CUDA

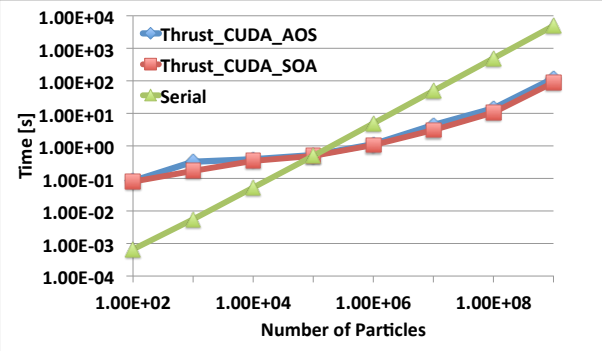

Fig. 2: Log-log plot showing wall clock times versus number of particles for the Thrust event-based algorithm running with a CUDA backend compared to the original serial history-based algorithm.

version, we see an interesting and unexpected result. Figure 3 shows the comparison of the CUDA and Thrust CUDA event-based version using the SOA data structures. The Thrust version is slightly faster than the CUDA version for all numbers of particles. We expect that this result is due to two possible factors. First, the Thrust scheduler may be launching kernels more effectively than the kernel launching scheme we implemented. Second, the memory locations of the read-only tallies and written tallies are stored with the Thrust functor which may allow Thrust to optimize what memory exists in registers or caches when the kernel launches. In the explicit CUDA version, the memory exists in either global memory or the constant memory which is already predetermined.

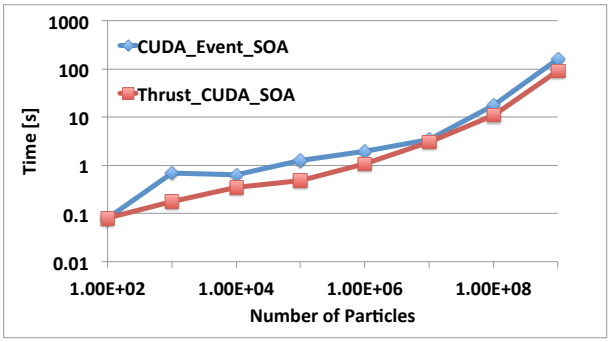

Fig. 3: Log-log plot showing wall clock times versus number of particles for the Thrust event-based algorithm running with a CUDA backend compared to the explicit CUDA event-based version.

Re-Evaluating the History-Based Method

Recent work performed by Nvidia's Anthony Scudiero [7] suggests that it may be possible to achieve performance on GPUs using a history-based Monte Carlo transport algorithm if the correct transformations are made. Additionally, since Monte Carlo transport is a memory bound problem, using a less compute-optimized approach with lower memory overhead might be a more efficient approach.

To re-evaluate the use of the history-based algorithm [1] on GPUs, we began by making changes suggested by Scudiero [7]. First, we moved those calculations that only needed to be performed once for all particles out of the single large kernel. Second, we utilized shared memory for storing the particle data structure and read-only constant memory for storing the material data (e.g. cross section values). Finally, we removed all atomic tally updates and replaced them with a shared per particle tally that is reduced to single values after the kernel is complete. The results of this work are shown in Figure 4.

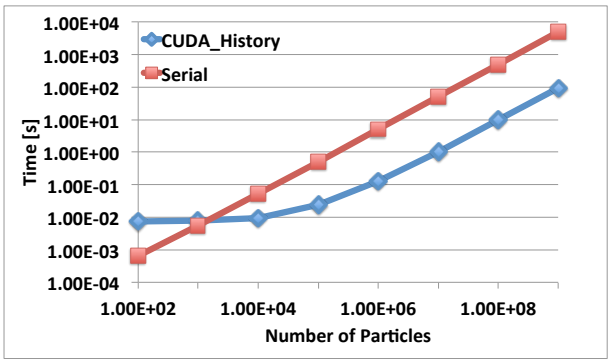

Fig. 4: Log-log plot showing wall clock times versus number of particles for the CUDA history-based algorithm compared to the original serial history-based algorithm.

The results from this test were very promising. This plot shows that the CUDA history-based algorithm has better scaling with number of particles as well as a significant performance increase at high numbers of particles. At low numbers of particles, the wall time of the CUDA version is constant, regardless of the number of particles, which is due to the overhead involved with accessing the GPU hardware. This overhead is sufficiently low (on the order of 0.01 seconds) that it should only affect problems using an unrealistically low number of particles.

Thrust and CPU

In order for the Thrust implementation to be considered portable, we must be able to demonstrate the possibility for performance of the Thrust approach on multiple platforms. For this study, we will compare the particle and processor scaling of the Thrust event-based method compiled with both the CUDA backend (for GPUs) and the OpenMP backend (for CPUs) to that of the original serial history-based method running on a single CPU core. The speedups of the event-based method compared to the original serial method are shown in Figure 5. The event-based method is slower on the CPU than the history-based method for a single thread, with a 0.5X speedup corresponding to a 2X slowdown. The Thrust event-based method with two OpenMP threads is roughly equivalent to the original history-based serial method. The Thrust event-based model reaches speedups around 5X at 16 threads when compared to the original serial history-based algorithm.

The speedups of the Thrust event-based method compared to the same Thrust event-based method run serially are shown in Figure 6. The speedups are generally as expected up to four threads but are less than expected at eight and sixteen threads. The Thrust event-based model reaches

maximum speedups around 10X compared to the Thrust model run serially.

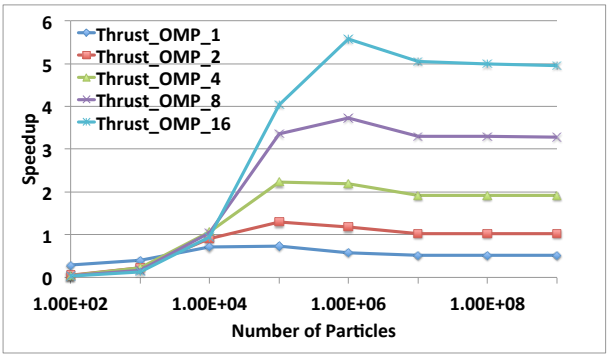

Fig. 5: Speedups versus number of particles for the event-based Thrust CPU method with 1, 2, 4, 8, and 16 OpenMP threads compared to the original serial history-based algorithm.

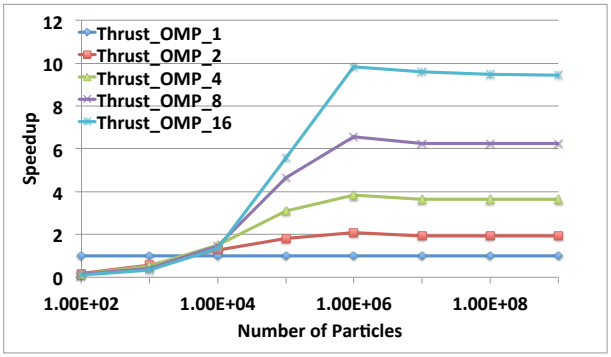

Fig. 6: Speedups versus number of particles for the event-based Thrust CPU method with 1, 2, 4, 8, and 16 OpenMP threads compared to the Thrust CPU method serially.

Results Summary

Table II shows the speedups achieved for each method running 10^9 particles on either two Nvidia Tesla K80 GPUs or a CPU with sixteen OpenMP threads. These results demonstrate that a significant amount of performance potential exists in the optimization choices that are made. The use of Thrust as a portability abstraction is not only viable but outperforms the other methods on the GPU. In addition, history-based approaches perform better or at least as well as the event-based approaches on the GPU for this problem. While the event-based Thrust OpenMP results are significantly less than optimal, they do demonstrate portability and some performance gain.

TABLE II: Maximum speedups for each approach when compared to the original history-based serial method

Method	Speedup
CUDA Event SOA	31.32
CUDA History	52.78
Thrust Event CUDA SOA	54.62
Thrust Event OpenMP SOA	5.54

CONCLUSIONS

We described the algorithm improvements we made for event-based Monte Carlo particle transport using Thrust as a portable performance abstraction. This work has shown that the event-based approach for Monte Carlo transport on GPU architectures is viable and can achieve node level speedup results that are acceptable. We have also shown that, with the same optimization choices, the Thrust abstraction layer can be just as effective as writing native CUDA. This result, enabled by the algorithmic improvements described in this paper, is in contrast to our previous conclusion that Thrust was not a viable option for an abstraction layer in Monte Carlo transport [1].

We have also demonstrated that the history-based Monte Carlo transport algorithm can perform efficiently on the GPU. As a result, the history-based approach should be investigated further for use on GPU architectures. For the Monte Carlo test code and numerical problem we investigated, we see even greater speedups with the CUDA history-based approach then we do with the CUDA event-based approach, and it required significantly fewer code modifications.

Finally, we were able to create a portable algorithm that scales with processors on a node level. While we were not able to achieve the expected 16X performance increase when applying 16 OpenMP threads to the problem, we do obtain speedups of approximately 5X. Future work could include improving the CPU OpenMP performance of the Thrust implementation.

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. Funding was provided by the LLNL Livermore Graduate Scholar Program.

REFERENCES

1. R. C. BLEILE, P. S. BRANTLEY, S. A. DAWSON, M. J. O'BRIEN, and H. CHILDS, "Investigation of Portable Event-Based Monte Carlo Transport Using the NVIDIA Thrust Library," *Trans. Am. Nucl. Soc.*, **114**, 941–944 (2016).
2. P. S. BRANTLEY, "A Benchmark Comparison of Monte Carlo Particle Transport Algorithms for Binary Stochastic Mixtures," *Journal of Quantitative Spectroscopy and Radiative Transfer*, **112**, 599–618 (2011).
3. "Thrust Web Site," (2014), https://developer.nvidia.com/Thrust.
4. "CUDA Web Site," (2014), http://www.nvidia.com/object/cuda_home_new.html.
5. NVIDIA, "CUDA C Programming Guide," (2015), version 7.5.
6. M. PHARR and W. R. MARK, "ispc: A SPMD compiler for high-performance CPU programming," *Innovative Parallel Computing (InPar)*, pp. 1–13 (2012).
7. A. SCUDIERO, "personal communication," (2016).