*PHYSOR 2018: Reactor Physics paving the way towards more efficient s*JSAND2018-2023C
Cancun, Mexico, April 22-26, 2018

METHODS FOR COMPUTING MONTE CARLO TALLIES ON THE GPU

Kerry L. Bossler

Sandia National Laboratories
P.O. Box 5800, Albuquerque, NM, 87185

kbossle@sandia.gov

ABSTRACT

Effectively using a graphics processing unit (GPU) for Monte Carlo particle transport is a challenging task due to its memory storage requirements and traditionally divergent algorithms. Most efforts in this area have focused on the entire transport process, choosing to use atomic operations or tally replication for computing tallies. This work isolates the performance of the tallies from the rest of the transport process, and studies the impact of using different approaches for tallying on the GPU. Five implementations of a photon escape tally are compared, using both single and double precision data types. Results show that replicating tallies is clearly the best option overall, if there is enough memory available on the GPU to store them. When insufficient memory becomes an issue, the best method to use depends on the size, data type, and update frequency of the tally. Global atomic updates can be a reasonable option in some cases, especially if they are infrequently used. However, there are two alternatives for general-purpose tallying that were shown to be more effective in most of the scenarios considered. These two alternatives are based on NVIDIA's warp shuffle feature, which allows 32 threads to simultaneously exchange or broadcast data, minimizing the number of atomic operations needed to get the final tally result.

KEYWORDS: Monte Carlo, GPU, Tallies, Particle Transport

1. INTRODUCTION

One task performed frequently by all the different variants of Monte Carlo particle transport codes is updating tallies. Each particle history can contribute multiple times to a variety of different tallies, ranging from simple event counters that tally occurrences of a specific event type, to more complex mesh tallies dependent on energy, angle, and time. Updating tallies on a single central processing unit (CPU) is typically done after each event of interest occurs. Making use of many CPUs requires the message passing interface (MPI) standard, which assigns multiple particle histories to each MPI process. There are two general MPI approaches used for updating tallies: either through replicating the tally structure across all the processes, or by sending event data to one or more dedicated tally servers [1]. At the end of the simulation, final tally results from all the MPI processes or tally servers are accumulated on the root process and written to output.

Compared to tallying on many CPUs, tallying on a graphics processing unit (GPU) can be even more complicated because the best approach to use depends on multiple factors. These factors include the size of the tally, the precision needed for accurate results, and the update frequency. Previous work in porting traditional Monte Carlo particle transport algorithms to the GPU have focused on the entire transport process, only briefly mentioning the impact of the tallies on the overall performance. Some authors relied on using atomic operations [2, 3, 4, 5], which lets multiple threads on the GPU access the same location in

memory without introducing any data race conflicts. Since atomic operations are serialized and can impact performance, other authors decided to replicate the tally instead [6, 7, 8]. Another study that focused on kernel density estimated mesh tallies for the GPU also chose to avoid atomic operations by assigning one mesh node to each GPU thread [9]. Unfortunately, due to the limited memory available on the GPU compared to the CPU, it is likely not feasible to replicate all the tallies that are used by production Monte Carlo particle transport codes. Therefore, given the complexity of tallying on the GPU effectively, how can an application developer make an informed decision on whether to pursue atomic operations or tally replication? This work can help application developers make that decision by comparing five methods for updating tallies on different GPU architectures.

2. NVIDIA GPU ARCHITECTURE

The general NVIDIA GPU architecture is built around a scalable array of multithreaded streaming multiprocessors (SM), each designed to execute instructions for hundreds of threads concurrently [10]. This is achieved through a single-instruction, multiple-thread (SIMT) architecture.

2.1 SIMT Architecture

Parallel work to be executed on the SIMT architecture of an NVIDIA GPU can be written using the CUDA programming model developed by NVIDIA. CUDA includes C/C++ language extensions that can be used to execute instructions on the GPU or transfer data between the CPU and the GPU. Executing instructions on the GPU is done by launching what is called a CUDA kernel. Each CUDA kernel that is launched breaks down the work into multiple thread blocks that are then distributed to all the available multiprocessors. The multiprocessors process each thread block by creating, scheduling, and executing groups of 32 threads known collectively as a warp. All threads in a warp must execute single instructions concurrently, usually on different data sets read from memory. While individual threads are allowed to branch and execute instructions independently from the others, this branch divergence forces the code to become serialized and can have a significant impact on the performance of a CUDA kernel. Therefore, the SIMT architecture of an NVIDIA GPU is operating at its optimal efficiency when there is no branch divergence within a warp.

2.2 Device Memory Hierarchy

Data used by a thread to execute an instruction must first be read from one of the many memory spaces available on an NVIDIA GPU. This memory hierarchy ranges from register memory assigned to individual threads, up to global memory that is accessible to all threads being processed by a CUDA kernel and the CPU host. A general summary of the different memory spaces is shown in Table I on the following page. Each memory space listed in Table I comes with unique advantages and disadvantages, so some thought must go into choosing the right type to use for different data accesses.

Register Memory: Since register memory is located on-chip, it provides the fastest access of all the options that are available. However, register memory is a limited resource that must be shared by all threads assigned to a multiprocessor.

Local Memory: If a CUDA kernel requires more registers than are available, then the excess data will spill over into local memory. Local memory is much slower than register memory because it is located off-chip, so it is important to prevent this from happening to obtain optimal performance.

Shared Memory: Like register memory, shared memory is also located on-chip and is a limited resource (48kB per block). The primary advantage of this memory space is that it can be used by all threads within a single block, which enables efficient communication between the threads.

Global Memory: Global memory is the largest of all the options, but is also the slowest because it is located off-chip. Most of the data that CUDA kernels need to access throughout the entire duration of the host program will be stored in global memory. A best practice for achieving optimal performance is to minimize the number of times global memory needs to be read or written by a CUDA kernel [11].

Constant Memory: Like global memory, constant memory is located off-chip. Unlike global memory, however, constant memory is cached on-chip for efficient read-only access and is also a limited resource (64 kB). Optimal efficiency when using constant memory occurs when all threads in a warp read from the same location – making it as fast as accessing register memory [11].

Texture Memory: Texture memory is another form of read-only memory located off-chip that is cached on-chip. As it was designed for graphics applications, texture memory works best when the access pattern involves spatial locality (i.e., all threads in a warp read from locations that are close to one another).

Table I. Device memory hierarchy of an NVIDIA GPU [11].

Memory Space	*Location (On/Off Chip)*	*Access*	*Scope*	*Lifetime*
Register	On	Read/Write	One Thread	Thread
Local	Off	Read/Write	One Thread	Thread
Shared	On	Read/Write	One Thread Block	Block
Global	Off	Read/Write	All Threads & Host	Host Allocation
Constant	Off	Read-Only	All Threads & Host	Host Allocation
Texture	Off	Read-Only	All Threads & Host	Host Allocation

2.3. NVIDIA GPU Options

Even though all NVIDIA GPUs use an SIMT architecture and the same device memory hierarchy, there can be some significant differences between the various cards that are available. Four NVIDIA GPUs are explored in this work: Quadro K5200, Tesla K40, Tesla K80, and Tesla P100. The three Tesla GPUs are dedicated accelerators used for scientific computing applications, whereas the Quadro K5200 was designed to provide graphics for a desktop workstation. Table II summarizes the key differences of these four GPUs.

Table II. Comparison of different NVIDIA GPU architectures.

Specification	*Quadro K5200*	*Tesla K40*	*Tesla K80*	*Tesla P100*
# GPUs per Card	1	1	2	1
CUDA Compute Capability	3.5	3.5	3.7	6.0
CUDA Cores	2304	2880	2496	3584
Streaming Multiprocessors	12	15	13	56
GPU Clock Rate	771 MHz	745 MHz	824 MHz	1481 MHz

Specification	*Quadro K5200*	*Tesla K40*	*Tesla K80*	*Tesla P100*
Single Precision TeraFLOPS	N/A	5.0	8.7[1]	10.6
Double Precision TeraFLOPS	N/A	1.7	2.9[1]	5.3
Global Memory	8125 MB	11441 MB	11441 MB	16281 MB
Memory Bandwidth	192 GB/s	288 GB/s	240 GB/s	732 GB/s
Device to Host Bandwidth	~3 GB/s	~9 GB/s	~9 GB/s	~11 GB/s

[1]Single and double precision performance for Tesla K80 is combined value for both GPUs on the card.

3. GPU METHODS FOR TALLIES

To study the impact of tallying on the GPU, a simple Monte Carlo photon transport model with an escape tally was implemented using CUDA 8.0. Since photon escape tallies only need to count the number of photons that leave the problem domain using a single integer variable, there are a wide variety of options available for implementing them on the GPU. Five of those options are explored in this work.

3.1. Photon Escape Tallies with Atomics

The first four options for implementing a photon escape tally on the GPU required some use of atomic operations, which were named global atomics, shared atomics, warp shuffle, and block reduction respectively. Source code for all four of these implementations can be found in Appendix A.

The global atomics method updates the photon escape tally using atomic operations directly in global memory whenever a photon escapes the problem domain. This method is expected to be the least efficient option, especially when a significant number of tally updates are needed. One alternative is to create a temporary tally in shared memory, where atomic operations are much more efficient because it is located on-chip. Each thread in the block would then update this temporary tally using one atomic operation in shared memory instead of global memory. After all of the threads have updated the temporary tally, an additional atomic operation is still needed to tally the contribution from each block in global memory. For a block with 128 threads, this would result in up to 128 shared atomic updates, and 1 global atomic update. Although using the shared atomics method should be more effective than the global atomics method, recall that shared memory is also a more limited resource. This means that larger tallies that do not fit into shared memory will not be able to use the shared atomics method for tallying on the GPU.

A better alternative could be to consider a CUDA feature called warp shuffle that was introduced with GPUs that have compute capability 3.x or higher. Warp shuffle allows the 32 threads in a warp to simultaneously exchange or broadcast data without using shared memory, which can be used to implement an efficient parallel reduction across the warp [12]. After each parallel reduction is complete, only one atomic operation is required per warp to add its contribution to the tally in global memory. This reduces the number of global atomic updates by a factor of 32.

Further improvement to the warp shuffle method can be made by using shared memory to perform a parallel reduction over all the warps in a block [12]. Named the block reduction method, this approach uses two warp shuffle loops. The first warp shuffle loop sums the contribution from all 32 threads in a warp, similar to the implementation of the warp shuffle method. The key difference is that the partial sum for each warp is then stored in shared memory instead of using a global atomic update. After all the threads in the block are synchronized, a second warp shuffle loop is used to combine all the partial sums into one final sum per block. For the example with 128 threads in a block, the block reduction method would only require 1 global atomic update instead of the 4 needed for the warp shuffle method.

3.2. Photon Escape Tally without Atomics

The fifth and final alternative for implementing a photon escape tally on the GPU avoided using atomic operations altogether by replicating the tally in global memory for each particle history simulated by the CPU host application. Tally replication ensures that no conflicts can occur when multiple threads representing different particle histories need to update the tally concurrently. After each particle history finishes processing and tallying its own events, the final tally result is obtained via a parallel reduction across all the replicated tallies. This parallel reduction can be done once at the end of the simulation using the thrust::reduce function, which is an algorithm available in the C++ template library called Thrust that is included with CUDA 8.0 [13].

One key disadvantage to replicating tallies instead of using atomic operations is that there may not be enough space available in global memory. This memory limitation can be minimized by using batches to process the total particle histories. Each batch of histories would need to be small enough so that all of their replicated tallies could fit in global memory. Different batches could then be scheduled to run on separate GPUs in a multi-GPU system. Another option is to use one or more GPUs as dedicated tally servers, whose sole purpose would be to receive event data and update tallies. Using GPUs as tally servers would free up global memory that would otherwise be assigned to cross sections, geometry, or other memory-intensive data needed to run a typical Monte Carlo particle transport simulation.

4. PERFORMANCE TESTS

To test the performance of the five photon escape tally implementations described in Section 3, a simple photon attenuation problem was used. Mono-energetic photons were directed into a 1D slab made up of helium, with a total cross section of 6.59936E-3 m^{-1}. The analytical solution for the fraction of photons that escape the problem domain is:

$$\frac{N}{N_o} = e^{-6.59936E-3\,x}, \tag{1}$$

where N is the number of photons that escape, N_o is the initial number of photons, and x is the thickness of the slab in meters. Expected results for three different test scenarios are summarized in Table III.

Table III. Different test scenarios used for tallying photon escape in a 1D helium slab.

Test Scenario	*Description*	*x (m)*	*N/N_o*
1	All Photons Escape	0	1.0
2	Approximately Half of the Photons Escape	100	0.5
3	No Photons Escape	10,000	0.0

The three test scenarios shown in Table III were designed to measure the impact that update frequency has when tallying on the GPU. Test scenarios 1 and 3 represent upper and lower limits on the number of times the photon escape tally needs to be updated. For test scenario 2, only half of the photons will contribute to the tally. Given that the photon attenuation is being modeled using a stochastic process, these tally updates will be performed by randomly selected threads – potentially introducing substantial branch divergence for some of the photon escape tally implementations being considered.

5. RESULTS

All three test scenarios described in Section 4 were run on the four NVIDIA GPUs described in Section 2 using 10^8 particle histories and a block size of 128. Timing data reported in the following sections are an average of ten independent runs, measuring only the contribution of the tally updates using CUDA GPU timers [11]. The Quadro K5200 GPU was installed in a Linux desktop workstation with Intel Xeon E5-2697 v3 CPUs (2.6 GHz), whereas the three Tesla GPUs were available in one of the Heterogeneous Advanced Architecture Platforms (HAAPs) at Sandia National Laboratories, named Ride [14]. Ride has three different node types: Tesla K40 GPUs with POWER8 Tuleta CPUs (3.7 GHz), Tesla K80 GPUs with POWER8 Firestone CPUs (3.9 GHz), and Tesla P100 GPUs with POWER8+ Firestone CPUs (4.0 GHz).

5.1. Quadro K5200 Results

Timing data for running the different photon escape tally implementations on the Quadro K5200 GPU is shown in Table IV. For comparative purposes, each implementation was repeated for a tally based on 32-bit integers, 64-bit unsigned integers, and a 32-bit floating-point type. The quantities in parentheses are the standard deviation for the ten independent runs used to obtain each average value.

Table IV. Timing data for five photon escape tally implementations on a Quadro K5200 GPU.

Test Scenario	*Global Atomics (ms)*	*Shared Atomics (ms)*	*Warp Shuffle (ms)*	*Block Reduction (ms)*	*No Atomics (ms)*
INTEGER TYPE (32-bit)					
1	5.48 (1.3)	7.57 (0.5)	6.64 (0.5)	9.34 (0.6)	5.26 (0.2)
2	71.0 (4.7)	34.6 (1.9)	6.58 (0.4)	9.30 (0.6)	5.22 (0.2)
3	3.44 (0.1)	4.05 (0.2)	6.12 (0.4)	9.04 (0.6)	5.31 (0.3)
UNSIGNED INTEGER TYPE (64-bit)					
1	134 (5.0)	78.1 (4.9)	7.15 (0.4)	10.4 (0.6)	7.70 (0.3)
2	69.2 (2.5)	42.9 (2.0)	7.13 (0.4)	10.4 (0.6)	7.73 (0.3)
3	3.53 (0.1)	4.08 (0.3)	7.01 (0.4)	10.6 (0.7)	7.78 (0.3)
FLOATING-POINT TYPE (32-bit)					
1	384 (4.0)	63.1 (3.8)	11.9 (< 1%)	9.07 (0.5)	5.27 (0.2)
2	197 (0.3)	34.3 (1.8)	12.6 (0.8)	9.05 (0.5)	5.26 (0.2)
3	3.61 (0.2)	4.23 (0.3)	5.96 (< 1%)	9.18 (0.6)	5.22 (0.2)

All five photon escape tally implementations based on 32-bit integers produced tally results that were consistent with Table III. In terms of performance, however, there were some significant variations. For the implementation with no atomics, all three test scenarios only took about 5 ms. This consistent performance occurs because each particle history always updates its own tally in global memory, with a 1 if it escaped, or a 0 if it got absorbed. Both the warp shuffle and block reduction methods are also consistent, since most of the operations performed are done in parallel and there are few atomic updates. In contrast, results for the global and shared atomics methods vary considerably. Both methods are very efficient for test scenario 1, which indicates that there may be some optimization being performed by the GPU when every thread is executing an atomic update with 32-bit integers. When no atomic updates are needed, such

as in test scenario 3, both the global and shared atomics methods are slightly faster than the no atomics method. The most interesting result for both global and shared atomics methods is test scenario 2. Introducing more divergence seems to result in much longer runtimes, with the global atomics method being noticeably worse than the shared atomics method. This is not surprising considering each atomic update takes longer in global memory than shared memory.

Changing from 32-bit integers to 64-bit integers should theoretically increase the time taken to perform all atomic operations, warp shuffles, and memory read/write accesses. Table IV shows that this is indeed the case, with noticeable increases in many of the test scenarios where these operations are used. The largest increases occur for test scenario 1 with the global and shared atomics methods, indicating that atomic updates for 64-bit integers are much less efficient than 32-bit integers when there is no divergence. Given that photon escape tallies are generally going to need 64-bit integers to capture total escape events correctly, these results suggest that either warp shuffle or no atomics methods should be used in a production Monte Carlo particle transport code. Note that even though the global and shared atomics methods are not as efficient as the others, they are still faster than the CPU. Processing an equivalent tally on one Intel Xeon CPU took 310 ms for 32-bit and 64-bit integers, and 350 ms for 32-bit and 64-bit floating-point values.

Although photon escape tallies can be restricted to integer data types, there are many other types of tallies that need to use floating-point values, such as particle flux or energy deposition tallies. Most GPUs support the 32-bit floating-point type with atomic updates and warp shuffle operations. When switching to 32-bit floating-point tallies, Table IV shows that the global atomics method is by far the worst option in terms of performance. The global atomics method was also the worst option in terms of accuracy, since it produced incorrect tally results for test scenarios 1 and 2. In comparison, the block reduction and no atomics methods appear to be about as efficient with 32-bit floating-point values as they were with 32-bit integers. Given that both methods rely more on data transfers than atomic updates, this is not surprising.

5.2. Tesla GPU Results

Since the Quadro K5200 GPU is primarily responsible for providing graphics for a desktop workstation, all the tally variations listed in Table IV were repeated for the three Tesla GPUs to see if having a dedicated card improves the performance. One noticeable difference was the one-time implicit initialization cost, which occurs when the first CUDA call is executed on the GPU [10]. This initialization cost for all three Tesla GPUs was less than 90 ms, whereas for the Quadro K5200 it ranged from 440 to 480 ms. With respect to the performance of updating the tallies, however, the results varied significantly. Figure 1 on the following page shows the speedup of each Tesla GPU over the Quadro K5200 for test scenario 1 based on a 32-bit integer tally.

The most performant GPU for tallying purposes is clearly the Tesla P100, with speedups ranging from 2 to 6 times greater than the Quadro K5200. This is not surprising given that the Tesla P100 has many more CUDA cores and a higher memory bandwidth. Results for the Tesla K40 were also not surprising, being faster than the Quadro K5200 for most cases, and up to 16% faster for the implementation with no atomics. The most surprising result occurred for the Tesla K80, which on paper looks as though it should slightly outperform the Quadro K5200 in terms of both compute power and data transfers. Even though the Tesla K80 consistently performed worse than the Quadro K5200 in this work, recall from Table II that each card includes two GPUs instead of one. Making use of both of those GPUs would improve the performance of the Tesla K80, making it more comparable to the Quadro K5200 and possibly even better.

Actual timing data for the Tesla P100 is shown in Table V, which also includes results for a 64-bit floating-point type. As well as being the most performant, the Tesla P100 is the only GPU considered in this work with native support for atomic updates using 64-bit floating-point values. A comparison of using single versus double precision for tallying on the GPU will be discussed further in Section 5.3.

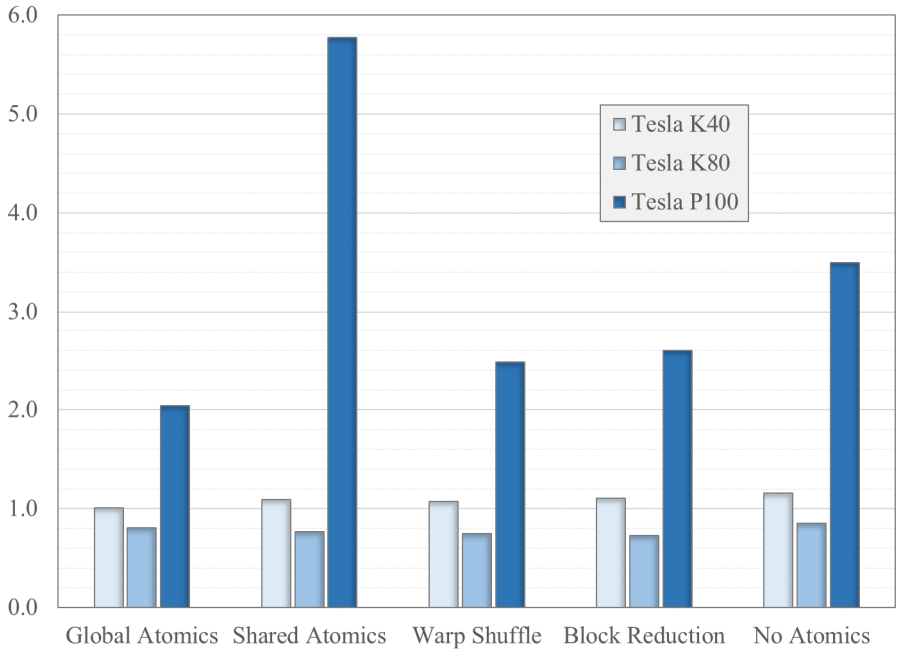

Figure 1. Speedup over Quadro K5200 for 10^8 tally updates using 32-bit integer type.

Table V. Timing data for five photon escape tally implementations on a Tesla P100 GPU.

Test Scenario	*Global Atomics (ms)*	*Shared Atomics (ms)*	*Warp Shuffle (ms)*	*Block Reduction (ms)*	*No Atomics (ms)*
INTEGER TYPE (32-bit)					
1	2.67 (< 1%)	1.31 (< 1%)	2.68 (< 1%)	3.59 (< 1%)	1.50 (< 1%)
2	2.69 (< 1%)	1.31 (< 1%)	2.68 (< 1%)	3.59 (< 1%)	1.50 (< 1%)
3	1.31 (< 1%)	1.31 (< 1%)	2.23 (< 1%)	3.54 (< 1%)	1.50 (< 1%)
UNSIGNED INTEGER TYPE (64-bit)					
1	77.0 (1.7)	92.6 (0.8)	2.68 (< 1%)	3.92 (< 1%)	2.27 (< 1%)
2	40.1 (0.5)	25.3 (0.2)	2.68 (< 1%)	3.92 (< 1%)	2.27 (< 1%)
3	1.31 (< 1%)	1.31 (< 1%)	2.40 (< 1%)	3.90 (< 1%)	2.27 (< 1%)
FLOATING-POINT TYPE (32-bit)					
1	222 (6.6)	88.2 (2.8)	7.28 (< 1%)	3.56 (< 1%)	1.50 (< 1%)
2	117 (2.9)	24.0 (0.06)	7.28 (< 1%)	3.56 (< 1%)	1.50 (< 1%)
3	1.31 (< 1%)	1.31 (< 1%)	2.23 (< 1%)	3.55 (< 1%)	1.50 (< 1%)
FLOATING-POINT TYPE (64-bit)					
1	221 (5.5)	92.3 (1.1)	7.28 (< 1%)	3.87 (< 1%)	2.27 (< 1%)
2	116 (3.7)	25.4 (0.01)	7.28 (< 1%)	3.87 (< 1%)	2.27 (< 1%)
3	1.31 (< 1%)	1.31 (< 1%)	2.40 (< 1%)	3.85 (< 1%)	2.27 (< 1%)

Table V shows that there is one interesting difference between the Quadro K5200 and Tesla P100 results, besides the obvious performance improvement. For the 32-bit integer tally, note that there is no difference in performance between test scenario 1 and 2 for the global and shared atomics methods. In fact, the shared atomics method is also the most efficient overall, even outperforming the implementation with no atomics. This shows that, at least for 32-bit integers, the Tesla P100 is less susceptible to the divergence that is caused by only some particles needing to update the tally. For all other data types, the implementation with no atomics is clearly the most performant in general and should be used wherever possible.

5.3. Single versus Double Precision

Deciding between single versus double precision often comes down to a choice between accuracy or performance. NVIDIA GPUs are generally better at processing instructions in single precision than double precision, with the Tesla K40 and K80 being three times faster, and the Tesla P100 being twice as fast. Unfortunately, many Monte Carlo tallies will need to use double precision to get acceptable accuracy in the tally results. In this work, a fixed random number seed was used to determine what photons got absorbed, which means that the number of photons that escaped should always be the same. While this was true for the integer-based tallies, when the atomic updates used 32-bit floating-point arithmetic the tally results were inconsistent between runs. For the global atomics method, the results were also completely inaccurate. Both issues were solved by using 64-bit floating-point arithmetic, which was expected to come with a penalty to performance. Table V shows that this penalty was minimal for most cases run on the Tesla P100. The only implementation with a significant increase was the no atomics method, since it relies on memory read/write accesses instead of atomic updates and warp shuffle operations.

Although only the Tesla P100 has native support for atomic operations with 64-bit floating-point values, it is possible to create an equivalent implementation for the other three GPUs using CUDA's *atomicCAS* method [10]. Figure 2 on the following page compares the performance of this *atomicCAS* method on the Quadro K5200, Tesla K40, and Tesla K80 to the native method on the Tesla P100. The implementation with no atomics is also included for reference purposes, even though it does not use atomic operations. In Figure 2, the native method for double precision atomics on the Tesla P100 is clearly much faster than the *atomicCAS* method on the other three GPUs. Performance improvements for test scenario 1 range from over 40 times faster for the shared atomics method, up to over 2000 times faster for the global atomics method. In comparison, the no atomics method was only about 3 times faster on the Tesla P100. These results suggest that double precision tallies with atomic updates should only be used on a Tesla P100 where atomic operations with 64-bit floating-point values are natively supported. Otherwise, it is probably best to use the no atomics method whenever double precision is required.

6. CONCLUSIONS

Five photon escape tally implementations were studied as potential options for tallying on the GPU. In general, it was found that avoiding atomic operations by replicating the tally is clearly the most performant option, especially when many tally updates are needed. However, replicating the tally also requires substantially more memory, which is a limited resource on the GPU. This increase in memory is acceptable for smaller tallies, but for larger tallies in a production Monte Carlo particle transport code it may not be feasible to use this option. Although there are ways to address the amount of memory needed to replicate the tallies, such as processing particle histories in batches, or using dedicated tally servers, it might be better to consider one of the other implementations that use atomic operations instead.

The naïve alternative to tally replication is to use atomic updates in global memory, which was shown to be extremely inefficient in most cases. In some cases, global atomics can be used effectively, such as if infrequent updates are needed, or when all threads update a 32-bit integer tally. For random tally updates, or data types other than 32-bit integers, then a method that limits the number of atomic operations should

always be preferred. The warp shuffle method was shown to be very effective for 32-bit and 64-bit integers, whereas the block reduction method was better for floating-point values.

The five photon escape tally implementations were also tested on four GPUs: Quadro K5200, Tesla K40, Tesla K80, and Tesla P100. Considering it was also responsible for managing graphics, the Quadro K5200 performed quite well overall, even outperforming one of the GPUs on a Tesla K80. Both the Tesla K40 and Tesla P100 were noticeably faster than the Quadro K5200, with the Tesla P100 being the clear choice when performance is important. The Tesla P100 was at least 2 to 6 times faster than all other GPUs considered in this work, and was also the only one with native support for 64-bit floating-point atomic operations.

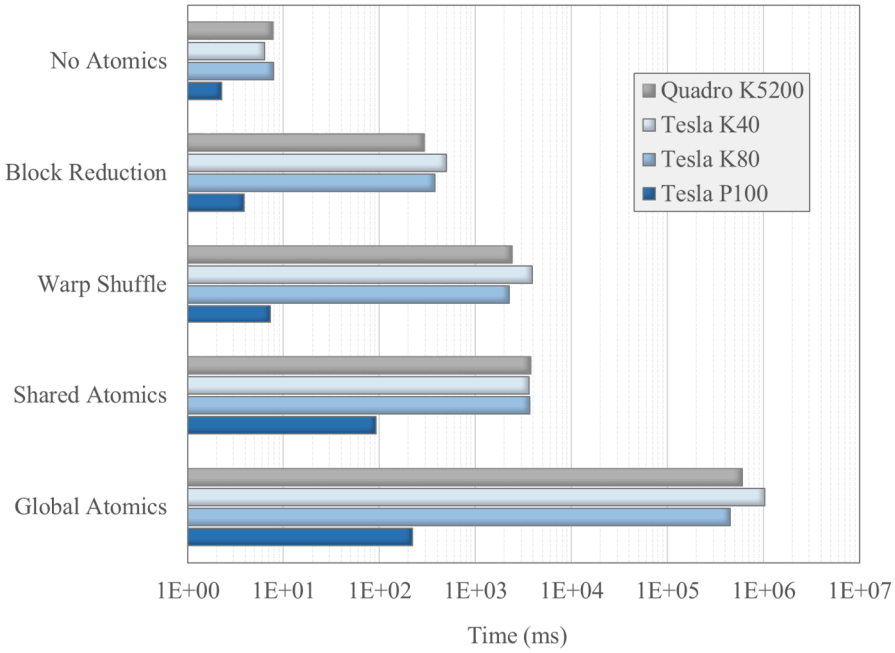

Figure 2. Timing data for 10^8 tally updates using 64-bit floating-point type.

ACKNOWLEDGMENTS

Supported by the Laboratory Directed Research and Development program at Sandia National Laboratories, a multi-mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

REFERENCES

1. P.K. Romano, A.R. Siegel, et al., "Data decomposition of Monte Carlo particle transport simulations via tally servers," *Journal of Computational Physics*, **252**, pp. 20-36 (2013).
2. 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*, **77**, pp. 176-193 (2015).
3. R.C. Bleile, P.S. Brantley, et al., "Investigation of Portable Event Based Monte Carlo Transport Using the Nvidia Thrust Library," *Trans. Am. Nucl. Soc.*, **114**, pp. 369-372 (2016).

4. D.F. Richards, R.C. Bleile, et al., "Quicksilver: A Proxy App for the Monte Carlo Transport Code Mercury," *Proceedings of 2017 IEEE International Conference on Cluster Computing*, Honolulu, Hawaii, September 5-8 (2017).
5. M. Martineau and S. McIntosh-Smith, "Exploring on-node parallelism with neutral, a Monte Carlo neutral particle transport mini-app," *Proceedings of 2017 IEEE International Conference on Cluster Computing*, Honolulu, Hawaii, September 5-8 (2017).
6. X.G. Xu, T. Liu, et al., "ARCHER, a new Monte Carlo software tool for emerging heterogeneous computing environments," *Annals of Nuclear Energy*, **82**, pp. 2-9 (2015).
7. R.C. Bleile, P.S. Brantley, et al., "Algorithmic Improvements for Portable Event-Based Monte Carlo Transport Using the Nvidia Thrust Library," *Trans. Am. Nucl. Soc.*, **115**, pp. 535-538 (2016).
8. S.P. Hamilton, T.M. Evans, and S.R. Slattery, "GPU Acceleration of History-Based Multigroup Monte Carlo," *Transactions of the American Nuclear Society*, **115**, pp. 527-530 (2016).
9. K.L. Bossler, "Performance of Kernel Density Estimated Mesh Tallies on GPUs," *Proceedings of International Conference on Mathematics & Computational Methods Applied to Nuclear Science & Engineering (M&C 2017),* Jeju, Korea, April 16-20 (2017).
10. NVIDIA Corporation, "CUDA C Programming Guide," PG-02829-001_v8.0 (2017).
11. NVIDIA Corporation, "CUDA Best Practices Guide," DG-05603-001_v8.0 (2017).
12. J. Luitjens, "Faster Parallel Reductions on Kepler," https://devblogs.nvidia.com/parallelforall/faster-parallel-reductions-kepler (2014).
13. NVIDIA Corporation, "Thrust Quick Start Guide," DU-06716-001_v8.0 (2017).
14. National Technology and Engineering Solutions of Sandia, LLC, "Advanced Systems Technology Test Beds," http://www.sandia.gov/asc/computational_systems/HAAPS.html (2017).

APPENDIX A

The implementation of the simple Monte Carlo photon transport model used in this work was split into three CUDA kernels: a source kernel, a transport kernel, and a tally kernel. The source kernel created the photons on the GPU, the transport kernel determined whether the photons escaped the domain, and the tally kernel updated the tally if the photons escaped. All tally kernel variations used in this work were required to read *nextEvent* from global memory, then update the tally in global memory if *nextEvent* was a surface crossing (i.e., 1). Updating a global tally named *d_count* using atomic operations via CUDA's *atomicAdd* method can be achieved in many different ways. The source code for four of those ways is included below.

A.1 Global Atomics

```
__device__ void globalTally(int nextEvent) {
  if (nextEvent == 1) atomicAdd(&d_count, 1);
}
```

A.2 Shared Atomics

```
__device__ void sharedTally(int nextEvent) {
  __shared__ int s_count;
  if (threadIdx.x == 0) s_count = 0;
  __syncthreads();
  if (nextEvent == 1) atomicAdd(&s_count , 1);
  __syncthreads();
  if (threadIdx.x == 0 && s_count > 0) atomicAdd(&d_count, s_count);
}
```

A.3 Warp Shuffle

```
__device__ void warpShuffle(int nextEvent) {
   // use shuffle operation to add up values in a warp
   int value = 0;
   if (nextEvent == 1) value = 1;

   for (int offset = warpSize/2; offset > 0; offset /= 2)
      value += __shfl_down(value, offset);

   // first thread in each warp adds combined value to total
   if (threadIdx.x%warpSize == 0 && value > 0) atomicAdd(&d_count, value);
}
```

A.4 Block Reduction

```
__device__ void blockReduceSum(int nextEvent) {
   // shared memory for 32 partial sums
   static __shared__ int shared[32];
   int lane = threadIdx.x % warpSize;
   int wid = threadIdx.x / warpSize;

   // each warp performs partial reduction
   int value = 0;
   if (nextEvent == 1) value = 1;

   for (int offset = warpSize/2; offset > 0; offset /= 2)
      value += __shfl_down(value, offset);

   // write reduced value to shared memory
   if (lane == 0) shared[wid] = value;
   __syncthreads();

   // read from shared memory only if that warp existed
   value = (threadIdx.x < blockDim.x / warpSize) ? shared[lane] : 0;

   // final reduce within first warp
   if (wid == 0) {
      for (int offset = warpSize/2; offset > 0; offset /= 2)
         value += __shfl_down(value, offset);
   }

   // add result for whole block to global memory
   if (threadIdx.x == 0 && value > 0) atomicAdd(&d_count, value);
}
```