M&C 2017 - International Conference on Mathematics & Computational Methods Applied to Nuclear Science & Engineering, Jeju, Korea, April 16-20, 2017, on USB (2017)

Performance of Kernel Density Estimated Mesh Tallies on GPUs

Kerry L. Bossler

Sandia National Laboratories: P.O. Box 5800, Albuquerque, NM, 87185, kbossle@sandia.gov

Abstract – *Although kernel density estimated (KDE) mesh tallies are capable of approximating distributions like particle flux with less variance than conventional mesh tallies, this improvement in precision comes with an increased computational cost per score. However, given that KDE mesh tallies are both compute-intensive and highly data-parallel, this increased computational cost can be reduced by taking advantage of a Graphics Processing Unit (GPU). This work compares a GPU implementation of a mesh tally based on the KDE integral-track estimator to an equivalent implementation designed for a single Central Processing Unit (CPU). Using a common abstract framework, both implementations consist of a setup stage, a compute stage, and a finalize stage. A detailed analysis on the performance of all three stages is presented. Results show that the compute stage of the KDE integral-track mesh tally is very efficient on the GPU, with one set of scores for over 10 million nodes being processed in 58 ms. Taking into consideration the increased cost for setup and finalize stages, tallying scores for 1000 particle tracks for over 10 million nodes is expected to be about 100 times faster on the GPU than a single CPU.*

I. INTRODUCTION

Kernel density estimated (KDE) tallies are a viable option for solving both criticality [1, 2] and fixed-source [3] Monte Carlo radiation transport problems. A few different approaches to using KDE tallies for transport purposes have been explored over the last few years – including the original KDE collision and track estimators [1], the KDE integral-track estimator [3], and the Mean Free Path KDE collision estimator [2]. Although KDE tallies are capable of approximating distributions like particle flux with less variance than conventional tallies, this improvement in precision comes with an increased computational cost per score. The KDE integral-track estimator in particular requires that at least one integration is performed for every particle track expected to contribute to the tally.

Previous work has shown that the most time-consuming task of a mesh tally based on the KDE integral-track estimator is defining the neighborhood region, which identifies all of the mesh nodes that might contribute a non-zero score for each particle track [3]. One alternative to defining a neighborhood region is to compute scores at every mesh node in parallel on a Graphics Processing Unit (GPU). Even though this would result in numerous trivial scores being computed, GPUs have thousands of threads that can execute the same instruction on different data sets very efficiently. This makes the GPU an ideal architecture for processing a KDE integral-track mesh tally, since each score is computed independently of the others.

GPUs have already been considered for an improved version of the Mean Free Path KDE collision estimator, which reported overall speedups ranging between 1.6 and 5.0 [4]. Rather than focusing on a heterogeneous computing environment like the previous work, this paper explores the effectiveness of tallying on the GPU in detail. A KDE integral-track mesh tally was developed for the GPU using NVIDIA's CUDA toolkit v7.5 [5]. This GPU version is compared to an equivalent implementation designed for a single Central Processing Unit (CPU).

II. KDE MESH TALLIES

KDE tallies can be used to estimate particle flux at all nodal coordinates (x, y, z) on some input mesh. Each estimate of the particle flux is obtained by averaging contributions from N particle histories that experience a sequence of C_i distinct events:

$$\hat{\varphi}(x,y,z)=\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{C_i}\varphi_{ic}(x,y,z). \qquad (1)$$

The contribution to the particle flux for the i^{th} history and c^{th} event, denoted φ_{ic}, can be computed using particle tracks with a track estimator, or collisions with a collision estimator. As with conventional mesh tallies, contributions computed using a track estimator usually produce more precise results for the same number of particle histories. The KDE integral-track estimator computes a score for each particle track by performing an integration over the path length S traveled along the track:

$$\varphi_{ic}(x,y,z)=w_{ic}\int_0^{d_{ic}}\frac{1}{h_x}K_x\frac{1}{h_y}K_y\frac{1}{h_z}K_z dS. \qquad (2)$$

In Equation 2, w_{ic} is the particle weight, and d_{ic} is the track length for the c^{th} particle track that is processed from the i^{th} history. The integrand is a product of one-dimensional kernel functions K_x, K_y, and K_z, whose shape and size will determine how many nodal coordinates produce non-zero scores. The size of the kernel functions is determined by the bandwidth vector (h_x, h_y, h_z). For the purposes of this work, the kernel functions K_x, K_y, and K_z are all of the form:

$$K_x = \frac{3}{4}\left(1-\left(\frac{x-X_o-uS}{h_x}\right)^2\right), \quad (3)$$

where (X_o, Y_o, Z_o) and (u, v, w) are the origin and unit direction of the particle respectively. Equation 3 is based on the Epanechnikov kernel [6], which will only be non-zero on the domain:

$$\left|\frac{x-X_o-uS}{h_x}\right| \le 1. \quad (4)$$

III. GPU ARCHITECTURE FOR KDE MESH TALLIES

The NVIDIA GPU architecture consists of a scalable array of multithreaded streaming multiprocessors (SM), each able to execute hundreds of threads concurrently using a single-instruction, multiple-thread (SIMT) approach [5]. Threads that are assigned to an SM are processed in groups of 32, which is known as a warp. If at least one thread in that warp needs to execute one or more different instructions, due to an if-statement or loop condition, then it must wait until all other 31 threads have first completed their instructions. Optimal performance in processing a warp can therefore only occur when there is no branch divergence among its 32 threads. This means that the GPU is best suited to highly data-parallel applications that execute an identical sequence of instructions on multiple data sets.

Computing each score for a KDE integral-track mesh tally is generally more compute-intensive than it is for a conventional mesh tally. However, these computations are also highly data-parallel due to the fact that each mesh node can be considered independently of the others. Being both compute-intensive and highly data-parallel makes the KDE integral-track mesh tally an ideal candidate for taking advantage of GPU architecture.

Before any scores can be computed for a tally using a specific algorithm on a GPU, there are two important concepts that should first be considered: how to assign work to threads, and what device memory options are available for efficient read/write access. Both can have a noticeable impact on performance. A brief overview of these concepts in the context of the KDE integral-track mesh tally is below.

1. Assigning Work to Threads

Since a score for each mesh node can be computed independently of the others, the most obvious choice for assigning work to threads is to use one thread per mesh node. Each warp processed on the GPU would therefore compute scores for a group of 32 mesh nodes at a time. Ensuring that each thread is only responsible for one mesh node means that it will only ever need to access one element in the data structure used for accumulating scores. No write contention will ever occur as a result of multiple threads trying to add a score to the same memory location, which can, at its worst, force the code being executed on the GPU to become serialized.

2. Device Memory Options

After determining how to assign work to threads on the GPU, the next step is to consider how the data will be stored that is needed for computing scores. For the purposes of this work, only register memory, global memory, and constant memory were used.

Register memory is located on-chip and therefore provides the fastest read/write access times. However, there are a fixed number of registers available per SM that must be shared by all of its assigned threads. Individual threads can only access registers that were allocated to them, which is determined by the compiler. These registers will likely contain most of the local variables needed for computing a score before it is added to the tally.

Global memory is the largest memory space on the GPU, but also has the slowest read/write access times because it is located off-chip. Variables stored in global memory last for the duration of the host program, and are accessible to all threads. This is the only memory space with enough room to store nodal coordinates for a large mesh, as well as the tally data structure used for accumulating its corresponding scores.

Like global memory, constant memory is located off-chip. There are two primary differences between global memory and constant memory. First, the size of the constant memory space is only 64 KB for current NVIDIA GPU architectures. Second, data stored in constant memory is cached on-chip for efficient read-only access. If all threads in a warp read from the same location, then constant memory can be as fast as accessing register memory [7]. This makes constant memory useful for variables such as the bandwidth vector and particle track data that are invariant and used by all threads.

IV. GPU VS. CPU IMPLEMENTATION

A KDE integral-track mesh tally using Equations 1 through 4 was implemented so that all scores could be computed using double precision on either a GPU or CPU. Both GPU and CPU versions share a common abstract framework written in C++11 that consists of three distinct stages: setting up the problem, computing scores, and finalizing tally results. Although they share a common abstract framework, there are some significant differences in the GPU version that affects its performance compared to the CPU version. The similarities and differences in the setup, compute, and finalize stages are highlighted in the following sections.

1. Setup Stage

Setting up the problem for both GPU and CPU versions involves extracting the nodal coordinates from the input mesh, creating the output file for storing tally results, and initializing the estimator used for accumulating tally scores. Note that the output file, written in the Exodus II binary format [8], stores a full copy of the input mesh and will also store the final accumulated tally results. This setup stage is only executed once.

Unlike the CPU version, the GPU version must perform additional work to copy read-only data to its different memory spaces, and to initialize the intermediate tally array in global memory that it uses for accumulating scores. This additional work increases the time needed to complete the setup stage. Read-only data that is copied to the GPU during the setup stage includes the nodal coordinates added to global memory, and the bandwidth vector added to constant memory.

2. Compute Stage

After the setup stage is complete, the next stage involves converting particle track data into tally scores by evaluating Equation 2 using 4-point Gaussian quadrature. This compute stage is therefore executed multiple times, once per particle track. The quadrature points needed to use the 4-point Gaussian quadrature method are stored in the constant memory space on the GPU.

Before even attempting to evaluate Equation 2 for a specific mesh node, both GPU and CPU versions first check to see if it is expected to produce a non-zero score for the particle track. Lower and upper integration limits for path length S are computed by using Equation 4 to determine minimum and maximum values in all three dimensions x, y, and z. Only nodes that result in some overlap between these three intervals and the particle track will actually continue in evaluating Equation 2. Note that this is not the same as using a neighborhood region, which reduces the set of nodes before checking for valid integration limits.

Although GPU and CPU versions of the KDE integral-track mesh tally use exactly the same algorithm, there are two primary differences in their implementations that impact their respective compute stages. The first difference is that the GPU version needs to copy the particle track data into its constant memory space prior to performing any calculations on that data. The second difference, which is more significant, is how the scores are accumulated. Whereas an intermediate tally array is used to accumulate scores for the GPU version, the CPU version adds scores directly into a key-value data structure. This data structure uses the mesh node ID as the key, and its accumulated score (if any) as the value. As a result, the CPU version can obtain nodal coordinates for computing scores by mesh node ID as needed, rather than copying them all upfront during the setup stage like the GPU version.

3. Finalize Stage

Finalizing tally results involves normalizing all of the accumulated tally scores by the number of particle histories, and writing those normalized scores to the output file. Like the setup stage, the finalize stage is only executed once and the GPU version takes longer because it must perform additional work. Additional work that is performed during the finalize stage includes transferring the accumulated scores from the intermediate tally array on the GPU back to the CPU, and then converting those scores into the same key-value data structure used by the CPU version. Any memory resources allocated on the GPU must also be freed.

V. PERFORMANCE TESTS

The performance of both versions of the KDE integral-track mesh tally was tested to determine the effectiveness of using GPUs for tallying scores. Since the smallest unit of work is attempting to compute a score at every mesh node, only one fixed particle track and bandwidth vector were used for all performance tests:

- $(X_o, Y_o, Z_o) = (-0.2, 0.2, 1.0)$
- $(u, v, w) = (0.0, -0.8, 0.6)$
- $w_{ic} = 0.5$
- $d_{ic} = 2.0$
- $(h_x, h_y, h_z) = (0.1, 0.1, 0.1)$

In addition to the above data, a pre-determined set of nodal coordinates (x, y, z) must also be defined to evaluate Equation 2. Typically, these nodal coordinates would be extracted from some known input mesh. However, it was decided to use non-physical mesh representations for these performance tests to more accurately assess the strengths and weaknesses of using the GPU for KDE integral-track mesh tally calculations. The four non-physical mesh representations chosen for this analysis are described below. A varying number of nodes were considered for each representation to reflect different valid mesh configurations, ranging from a single element with 8 nodes, up to a 215x215x215 element grid with 10,077,696 nodes.

Mesh 1: All Nodes Compute Score

The maximum amount of work performed per particle track occurs when every node computes a non-zero score. Mesh 1 was designed to measure this maximum by making the coordinates of all nodes equal to (-0.2, 0.1, 1.1).

Mesh 2: All Nodes Compute No Score

In contrast to Mesh 1, Mesh 2 was designed to measure the minimum amount of work performed per particle track, which was done by changing the coordinates of all nodes to

(0.0, 0.0, 0.0). This forces the check based on Equation 4 to always fail so that Equation 2 is never evaluated.

Mesh 3: Alternate Nodes Compute Different Score

If different nodes compute different scores, there should be no noticeable impact on performance for either the GPU version or the CPU version. To show that this is indeed the case, Mesh 3 was designed so that all even-numbered nodes had the coordinates (-0.2, 0.1, 1.1), and all odd-numbered nodes had the coordinates (-0.15, -0.1, 1.1).

Mesh 4: Alternate Nodes Compute No Score

While different nodes computing different scores should not impact performance, if half of the nodes compute no score then there should be a noticeable difference. Therefore, Mesh 4 was designed so that all even-numbered nodes had the coordinates (-0.2, 0.1, 1.1), and all odd-numbered nodes had the coordinates (0.0, 0.0, 0.0).

VI. RESULTS

All performance tests were run on a desktop Linux workstation with Intel Xeon E5-2697 v3 (2.60 GHz) CPUs and one NVIDIA Quadro K5200 GPU. Timing results reported in this section are an average of ten independent runs obtained using std::chrono::steady_clock from the C++11 standard, unless otherwise noted.

Table I and II below contain summaries of the timing results for Mesh 1 with 1331 and 10,077,696 nodes respectively. As expected, the general trend for both cases is that the setup and finalize times take longer for the GPU version compared to the CPU version, but the compute times are noticeably faster. During the compute stage, processing scores for 1331 nodes was about 20 times faster on the GPU, and processing scores for 10,077,696 nodes was about 250 times faster.

Table I. Timing Results for Mesh 1 with 1331 Nodes

STAGE	CPU (ms)	GPU (ms)
Setup	9.78	341.51
Compute	1.93	0.094
Finalize	0.81	1.71
Total	**12.52**	**343.31**

Table II. Timing Results for Mesh 1 with 10,077,696 Nodes

STAGE	CPU (ms)	GPU (ms)
Setup	7.81×10^{4}	7.87×10^{4}
Compute	1.46×10^{4}	5.80×10^{1}
Finalize	0.51×10^{4}	1.07×10^{4}
Total	$\mathbf{9.78\times10^{4}}$	$\mathbf{8.94\times10^{4}}$

Although the compute stages for both 1331 and 10,077,696 nodes are noticeably faster on the GPU, note that the overall performance for one particle track is highly dependent on the impact of the setup and finalize stages. Taking the timing results of these two stages into consideration, the case with 10,077,696 nodes is only 9% faster on the GPU compared to the CPU, and the case with 1331 nodes is actually 27 times slower.

Given that the compute stage is the only one that is repeated, the GPU version should rapidly start to outperform the CPU version as more and more particle tracks are tallied on a large mesh. For example, tallying 100 particle tracks on a mesh with 10,077,696 nodes should result in a total speedup of around 16, and tallying 1000 particle tracks should result in a total speedup of around 100. Even a smaller mesh with 1331 nodes will eventually start outperforming the CPU version. At least 182 particle tracks must be tallied for the GPU version to achieve similar performance, but tallying 1000 particle tracks would be about 4 times faster overall.

One fact that is important to note with these results is that they only measure the performance of the tallying process. Running a full Monte Carlo simulation with particle tracking in a heterogeneous computing environment introduces additional factors that will reduce the impact that using the GPU has on the overall performance. The most significant of these factors is the timing of the communication between the GPU and the CPU. In the work on the improved Mean Free Path KDE collision estimator, groups of 50,000 collisions were created on the CPU before being transferred to the GPU for tallying [2]. This means that the GPU has to wait until enough collisions are available before it can do anything. Although particle tracking and tally processing can be overlapped, the time required to create the collisions might be much higher than the time it takes to process them on the GPU.

Results for the full range of mesh representations considered in this work are discussed in more detail below for the setup, finalize, and compute stages.

1. GPU vs. CPU Setup and Finalize Stages

As shown in Tables I and II, the setup and finalize stages for 1331 and 10,077,696 nodes take longer for the GPU version than the CPU version due to the additional work that needs to be done. A visual comparison of the cost of this additional work is shown in Fig. 1, which displays the ratio of the GPU to CPU setup and finalize times for the full range of nodes considered with Mesh 1. Similar results also occur for Mesh 2, 3, and 4, which indicates that there is a fixed cost for both setting up and finalizing the problem based on the number of nodes.

For the setup stage, Fig. 1 shows that the ratio of the GPU to CPU times varies significantly with the number of nodes. Cases with more than 10^{6} nodes appear to have equivalent setup times because the dominant factor for both versions is the creation of the output file. On the other end, cases with fewer than 10^{4} nodes take much longer to set up

for the GPU version. This increased setup time is caused by the implicit initialization of the GPU [5], which occurs when the first CUDA function is called to copy the bandwidth vector into constant memory. The initialization overhead for the Quadro K5200 GPU was about 300 ms, which is more noticeable for meshes with fewer nodes because it takes much less time to create the output file.

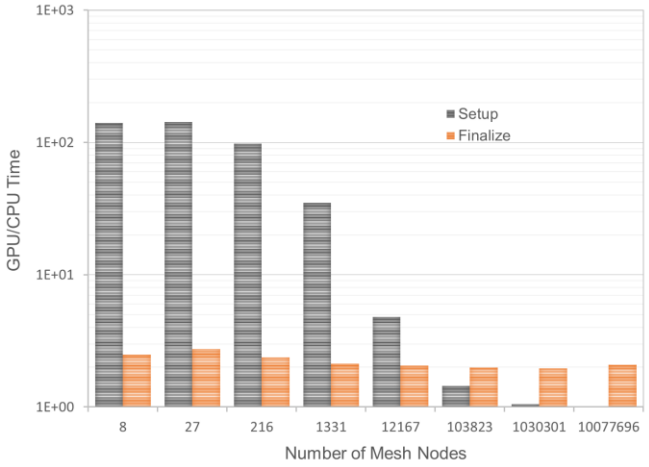

Fig. 1. Ratio of GPU to CPU setup and finalize times.

After output file creation and GPU initialization overhead, most of the GPU version's remaining setup time is due to tasks that depend on the number of nodes.[1] Fig. 2 and Fig. 3 compare the relative cost of these tasks for 1331 and 10,077,696 nodes respectively. All tasks executed on the GPU were timed using CUDA events [5], but the total time used to compute the percentages shown in Fig. 2 and Fig. 3 was obtained using std::chrono::steady_clock. The "Other" category refers to tasks executed only on the CPU, which is equal to the difference between the total time and the sum of all the tasks executed on the GPU. This difference is primarily caused by creating the zero-valued vector on the CPU for initializing the intermediate tally array on the GPU.

Fig. 2 and Fig. 3 show that the number of nodes in the mesh has a significant impact on the relative cost of each task. The first task performed is to free GPU resources that may have been previously allocated for a different mesh tally. This task is negligible for 10,077,696 nodes, and also only accounts for 3% of the total time for 1331 nodes. The next task performed is to allocate space in the global memory on the GPU for storing nodal coordinates and the intermediate tally array. While this is the most time-consuming task for 1331 nodes at 62%, it only accounts for 1% of the total time for 10,077,696 nodes. The most time-consuming task executed on the GPU for 10,077,696 nodes is copying the nodal coordinates into its global memory, which accounts for almost half of the total time. In both cases, "Other" is also significant, even though it represents tasks that are executed on the CPU and not the GPU.

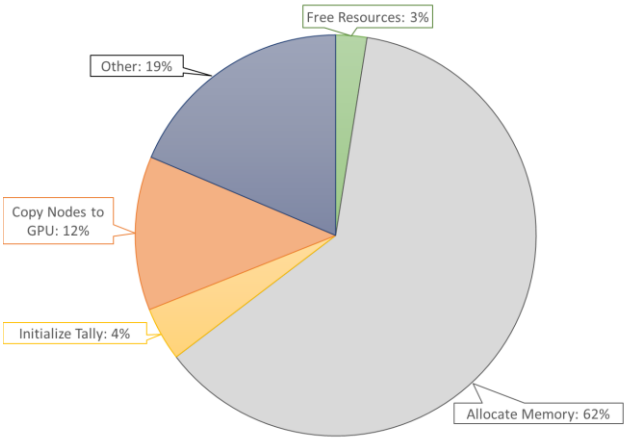

Fig. 2. GPU task breakdown during the setup stage for a mesh with 1331 nodes

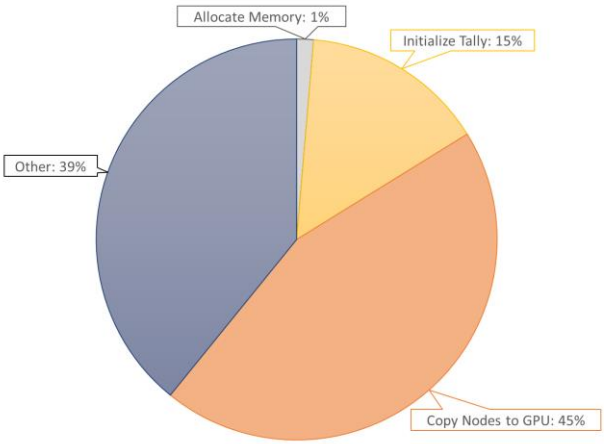

Fig. 3. GPU task breakdown during the setup stage for a mesh with 10,077,696 nodes.

In contrast to the setup stage, Fig. 1 shows that the ratio of GPU to CPU times for the finalize stage does not vary much with the number of nodes. All cases took the GPU version about twice as long to normalize tally results and write those results to the output file. Although the finalize stage for the GPU version is longer than it is for the CPU version, this increase is not predominantly caused by tasks executed on the GPU.

As an example, the additional time needed to finalize the GPU version for 10,077,696 nodes was 0.56×10^4 ms. Only 13 ms (~0.2%) of this time was used to transfer final scores back to the CPU. Even less was needed to free GPU resources (~0.01%). The remaining time was used to convert the intermediate tally array into the key-value data structure on the CPU that is written to the output file.

For 1331 nodes, the percentage of all the tasks executed on the GPU during the finalize stage increases to about 20%. These results indicate that one area for improvement

in the GPU version would be to write results directly from the intermediate tally array that was copied back to the CPU. However, this would also reduce the level of abstraction in the code because different functionality would be needed for the GPU version over the CPU version.

2. GPU vs. CPU Compute Stage

Although both GPU and CPU versions use the same algorithm to calculate scores, their performance can behave very differently as the number of nodes increases. Timing results for the compute stage versus the number of nodes are plotted in Fig. 4 for Mesh 1 and 2, and in Fig. 5 for Mesh 3 and 4. Compared to the CPU version, which varies linearly for all mesh representations, the performance of the GPU version is fairly constant until 1331 nodes. This represents the overhead for performing calculations on the GPU, which is different from the overhead for initializing the GPU.

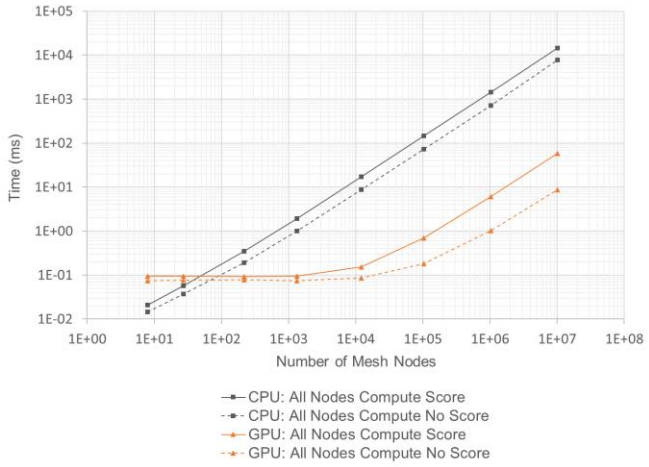

Fig. 4. Compute time vs. number of nodes for Mesh 1 and 2.

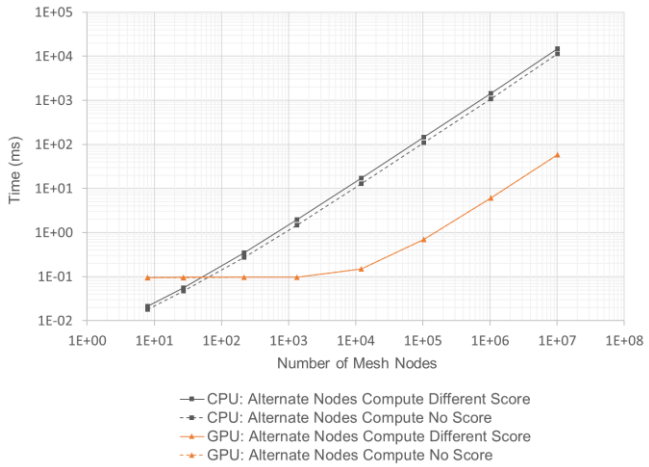

Fig. 5. Compute time vs. number of nodes for Mesh 3 and 4.

In order to offset the overhead for computing scores on the GPU and maximize its efficiency, there must be enough work for it to perform. One metric used to measure the efficiency of a GPU calculation is occupancy, which is the ratio of the number of active warps being processed on an SM to the maximum number of possible active warps [7].

For the Quadro K5200 GPU, the maximum number of possible active warps per SM is 64 [5]. So to achieve an occupancy of 100%, 64 warps must be processed concurrently on each SM. Due to register usage per thread, however, the GPU version of the KDE integral-track mesh tally can only process 48 active warps concurrently per SM, which results in a peak possible occupancy of 75%. Table III summarizes the impact of the number of nodes on the achieved occupancy for Mesh 1 reported by the NVIDIA profiler from the CUDA toolkit. Similar values were also reported for Mesh 2, 3, and 4. The number of warps processed refers to how many warps were needed to account for the given number of nodes. Since these calculations were done in blocks of 128 threads, the minimum number of warps that will be processed is 4. Reducing this to blocks of 64 or 32 threads changes the peak possible occupancy to 50% and 25% respectively.[2]

Table III. Impact of Number of Nodes on Achieved Occupancy of GPU for Mesh 1

NODES IN MESH	ACHIEVED OCCUPANCY (%)	WARPS PROCESSED
8	1.9	4
27	1.7	4
216	5.5	8
1331	6.0	44
12,167	34.2	384
103,283	68.9	3228
1,030,301	72.5	32,200
10,077,696	72.4	314,928

Table III shows that achieved occupancy is significantly impacted by the number of nodes in the mesh for which scores are computed. Given that each SM can process up to 48 active warps (i.e., 1536 threads), meshes with fewer than 1536 nodes do not produce enough work to make efficient use of the GPU. This corresponds to low achieved occupancy values, such as only 6% for the case with 1331 nodes. In contrast, meshes with substantially more than 1536 threads are able to keep all SMs on the GPU busy enough so that the achieved occupancy values are much closer to the theoretical value of 75%. A saturation point seems to occur after around a million nodes, indicating that the GPU has reached its optimal efficiency.

Once there are enough nodes to keep the GPU busy in terms of occupancy, the performance of the GPU version of the KDE integral-track mesh tally also appears to vary linearly with the number of nodes. Note, however, that there is a greater difference between Mesh 1 and Mesh 2 when compared to the CPU version, which represents the minimum and maximum performance expected for processing one particle track. As a result, there is more

potential for improving the performance when many nodes compute no score.

While Fig. 4 highlights the minimum and maximum performance expected, Fig. 5 shows what happens when alternate nodes execute different tasks. Two cases were considered. The first case, Mesh 3, computed different scores for alternate nodes. Given that the performance of Mesh 1 and Mesh 3 is similar, neither the GPU nor the CPU version pay any penalty for computing different scores. The second case, Mesh 4, computed no score for alternate nodes, which should, in theory, improve performance. Fig. 5 shows that this is only true for the CPU version.

Although only half of the nodes compute a score for Mesh 4, its performance for the GPU version was about the same as Mesh 1 and Mesh 3. As a result, there is no performance benefit over the case where all nodes compute a score. This result can be explained by understanding how the GPU works in terms of warp efficiency, which is a metric that can be used to measure the level of divergence of a GPU calculation. Table IV summarizes the impact of the number of nodes on warp efficiency for Mesh 1, 2, 3, and 4. A lower warp efficiency means that the GPU calculations that were performed experienced greater divergence.

Table IV. Impact of Number of Nodes on Warp Efficiency

NODES IN MESH	WARP EFFICIENCY (%)			
	Mesh 1	Mesh 2	Mesh 3	Mesh 4
8	26.3	32.4	24.4	15.9
27	84.6	85.8	78.6	50.9
216	96.4	96.5	89.1	55.9
1331	99.0	99.1	91.5	57.4
12,167	99.8	99.8	92.2	57.8
103,283	100.0	100	92.3	57.9
1,030,301	100.0	100	92.4	57.9
10,077,696	100.0	100	92.4	57.9

Mesh 4 consistently had a lower warp efficiency than all other mesh representations, which means that alternate nodes computing no score introduces noticeable divergence into the GPU calculations. This divergence arises from the fact that Equation 2 is only evaluated if it has valid integration limits based on Equation 4. Half of the threads in each warp have to wait until the other half compute a score, which means it takes just as long as the case where all nodes compute a score. Mesh 4 is a worst-case scenario that shows why it is important to reduce divergence in order to maximize GPU performance. For this example, divergence can be practically eliminated by reordering the nodes so that the first half all compute a score and the second half compute no score. However, the effectiveness of using this approach depends on how long it takes to sort the nodes.

In addition to highlighting how much divergence exists in Mesh 4, there are two other interesting results shown in Table IV. The first is that Mesh 3 had slightly lower warp efficiency than Mesh 1, which suggests that computing different scores at alternate nodes introduces some minor divergence. The most likely cause of this divergence is the calculation of the integration limits for Equation 2, since different nodal coordinates typically produce different integration limits. This calculation requires evaluation of two small if-statements for assigning the lower and upper values. For the particle track data used with Mesh 3, it is actually only the y-dimension that causes the divergence, given that it executes both if-statements for one node, but only one if-statement for the other.

The second interesting result from Table IV is that cases with fewer nodes progressively report lower warp efficiency, and therefore greater divergence. For 8 and 27 nodes this divergence is to be expected because not all of the threads in the warp will be actively executing instructions. Rearranging nodes will have no impact on this type of divergence. In this situation, adding more nodes is required to increase the number of warps that are processed with all 32 threads actively executing instructions.

VII. POTENTIAL IMPACT FOR CONVENTIONAL MESH TALLIES

Even though this work focused on the performance of the KDE integral-track mesh tally on GPUs, there are some important lessons learned that could be useful in developing an effective GPU implementation for a conventional mesh tally. The conventional mesh tally approach is to define each mesh cell as a histogram bin. Particle tracks can then be apportioned into those bins in two ways: either deterministically or statistically. The deterministic method involves adding the exact length of the track that intersects with each mesh cell to its corresponding tally. The statistical approach is to choose one or more random points along uniform strata of the track, then add this uniform portion to the tally for the mesh element(s) in which those points are located.

Unlike the KDE integral-track mesh tally, which is highly data-parallel, the conventional mesh tally most likely needs to use different implementations for the GPU and the CPU to take advantage of the different architectures. For example, both the deterministic and statistical method for tallying tracks requires that a point-in-cell search be performed. On the CPU, a simple implementation for this point-in-cell search could be to linearly check every mesh cell until the one in which the point is located is found. If threads were assigned to particle tracks on the GPU, performing a linear search like this could introduce a significant amount of divergence.

Instead of assigning threads to particle tracks, a better alternative could be to assign them to mesh cells. Like the mesh nodes needed for a KDE integral-track mesh tally, data describing those mesh cells would be copied to the GPU during the setup stage, their scores would be tallied during the compute stage on the GPU, and their final results would be transferred back to the CPU during the finalize stage. Ensuring that each thread is only responsible for one

mesh cell has a couple of advantages. The first advantage is that no thread will execute loops during a linear search of the mesh cells with a different number of iterations than other threads. The second advantage is that there will be no write contention when adding scores to the intermediate tally array, since each mesh cell would be assigned its own element in that array.

If threads are assigned to mesh cells, then the conventional mesh tally can also use the same approach as the KDE integral-track mesh tally for determining device memory usage. One key difference is that the conventional mesh tally will need to store more mesh data in the global memory, such as minimum and maximum coordinates for each cell in a structured mesh. This will require twice the amount of space as the nodal coordinates used by the KDE integral-track mesh tally.

VIII. CONCLUSIONS

The performance of the KDE integral-track mesh tally on the GPU was compared to the CPU for four different mesh representations, based on a varying number of nodes. Results show that computing scores on the GPU instead of the CPU could eliminate the need for defining a neighborhood region. Although eliminating the concept of the neighborhood region does increase the number of trivial scores that are computed, the SIMT architecture of the GPU processes these scores very efficiently because the algorithm is highly data-parallel. For example, scores for one particle track were computed on the GPU for over 10 million nodes in 58 ms, which is about 250 times faster than they were computed on a single CPU.

Computing scores on the GPU does come with an increased cost with respect to setup and finalize times, as well as some overhead for launching jobs from the CPU. In some cases, especially for meshes with fewer nodes that do not tally many particle tracks, this overhead for using the GPU may be too high. However, given that most Monte Carlo simulations tally millions or billions of particle tracks, the increased cost for using the GPU rapidly becomes less significant. Taking the setup and finalize costs into consideration, tallying 1000 particle tracks for over 10 million nodes is expected to be about 100 times faster on the GPU than a single CPU.

Improvements in the GPU and/or CPU versions of the KDE integral-track mesh tally would likely impact the performance gains reported by this work. Specifically, adding a neighborhood region search to the CPU version would substantially reduce the amount of trivial scores that it computed, which would result in a fairer comparison. Alternatively, an equivalent parallel version could be designed for use with a high performance computing cluster. However, using a GPU for tallying purposes would still be an attractive option considering it requires fewer resources and frees up the CPUs for other calculations that need to be performed during a Monte Carlo simulation.

Future work on using GPUs for tallying purposes will compare implementations of the conventional mesh tally based on the statistical method for apportioning tracks into mesh cells. This statistical method is expected to be easier to implement for the GPU than the deterministic method. Using lessons learned from this work, as a first attempt each thread will represent one mesh cell. All mesh cells will then simultaneously check to see if different points on the particle track are located within their domain.

ENDNOTES

[1]This excludes copying the bandwidth vector to the GPU.
[2]Based on CUDA occupancy calculator from CUDA toolkit.

ACKNOWLEDGMENTS

Supported by the Laboratory Directed Research and Development program at Sandia National Laboratories, a multi-mission laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

REFERENCES

1. K. BANERJEE and W. R. MARTIN, "Kernel Density Estimation Method for Monte Carlo Global Flux Tallies," *Nuclear Science and Engineering*, **170**, 234 (2012).
2. T. P. BURKE, B. C. KIEDROWSKI, and W. R. MARTIN, "Mean Free Path Based Kernel Density Estimators for Capturing Edge Effects in Reactor Physics Problems," *Proc. M&C 2015*, Nashville, Tennessee, April 19-23, American Nuclear Society (2015).
3. K. L. DUNN, "Monte Carlo Mesh Tallies based on a Kernel Density Estimator Approach," Ph.D. Thesis, University of Wisconsin-Madison (2014).
4. T. P. BURKE, B. C. KIEDROWSKI, W. R. MARTIN, and F. B. BROWN, "GPU Acceleration of Kernel Density Estimators in Monte Carlo Neutron Transport Simulations," *Trans. Am. Nucl. Soc*., **115**, 531 (2016).
5. NVIDIA CORP., "CUDA C Programming Guide," PG-02829-001_v7.5 (2015).
6. B. SILVERMAN, *Density Estimation for Statistics and Data Analysis*, Chapman and Hall, London, England (1986).
7. NVIDIA CORP., "CUDA Best Practices Guide," DG-05603-001_v7.5 (2015).
8. G. D. SJAARDEMA, L. A. SCHOOF, V. R. YARBERRY, "EXODUS II: A Finite Element Data Model," SAND92-2137, Sandia National Laboratories (2006).