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

Acceleration of Monte Carlo Methods on Heterogeneous CPU-GPU Platforms Using Kernel Density Estimators

[1]Timothy P. Burke, [1]Brian C. Kiedrowski, [1]William R. Martin, and [2]Forrest B. Brown

[1]*Department of Nuclear Engineering and Radiological Sciences, University of Michigan, Ann Arbor, MI,*
[2]*Los Alamos National Laboratory, P.O. Box 1663, Los Alamos, NM 87545*
tpburke@umich.edu

Abstract - *GPUs are used to accelerate Kernel Density Estimators (KDEs), an alternative tally scheme to histograms (mesh tallies), for Monte Carlo radiation transport simulations. The KDE routines are exported to the GPU while the CPU continues to do particle transport, with particle event information passed in batch to the GPU. The algorithm is optimized for the GPU, resulting in a reduction in GPU compute kernel runtimes by a factor of 2.9. Speedups of 2-5 are obtained for a quarter-assembly of 16×16 pincells and 1.35-1.65 for a 2-D pincell. Results show that most of the additional cost of the KDE routines can be completely hidden by exporting the tally process to the GPU. The methodology developed in this paper can be applied to accelerate other compute-intensive portions of Monte Carlo algorithms using heterogeneous computing.*

I. INTRODUCTION

Monte Carlo radiation transport calculations are often run on computer clusters due to their large computational cost. Monte Carlo codes are often considered embarrassingly parallel and are readily parallelized on compute nodes containing multi-core CPUs via MPI and threading. The independent nature of particle histories allows for the simulation to be parallelized by dividing the total number of particle histories being simulated among each thread in each MPI process.

While this method of parallelization works well for clusters of multi-core CPUs, traditional Monte Carlo codes are not well suited for parallelization on GPUs. GPUs are often included in compute nodes as a means of accelerating performance for benchmarks and vectorizable algorithms but are usually unused when compute nodes are used to run Monte Carlo neutron transport simulations. Due to warp divergence from the routine use of data-dependent conditional statements and the random access of memory, it is difficult to accelerate Monte Carlo neutron transport codes with GPUs. Thus, Monte Carlo algorithms have to be re-written specifically for implementation on GPUs; a barrier of entry that is often too significant for developers to pursue using GPUs.

While a large body of research has been generated for creating Monte Carlo algorithms that can run on GPUs, e.g. [1, 2], there is still no production-level Monte Carlo neutron transport code capable of using GPUs. Rather than use GPUs for the main Monte Carlo particle transport algorithm, GPUs can be used to accelerate compute-intensive vectorizable portions of the code while the main transport process is conducted on the CPU. This paper describes the acceleration of Kernel Density Estimators (KDEs) via heterogeneous computing with GPUs, but the same methodology can be applied to other computationally expensive vectorizable algorithms in Monte Carlo codes. This work uses a modified version of OpenMC 0.6.0 [3] and exports the KDE tally routines onto a GPU to obtain problem-dependent speedups ranging between 1.35 and 5.0 for the problems studied in this paper. For the problems studied in this paper, accelerating the KDE routines on the GPU eliminates most of the additional cost of the estimator, making KDEs essentially free to use.

II. BACKGROUND & THEORY

1. KDE

KDEs have recently been explored for use in Monte Carlo radiation transport simulations as an alternative to histogram tallies for capturing spatially-resolved quantities such as neutron flux and reaction rates [4, 5]. Histogram tallies suffer from large uncertainties when detailed spatial resolution of the quantity of interest is required; increasing the resolution of the underlying mesh increases the uncertainty in each histogram bin. KDEs obtain estimates of underlying densities on user-defined points, with the uncertainty at those points being independent of the desired spatial spatial resolution. Thus, KDEs show potential for obtaining smooth estimates of spatially-resolved quantities with reduced variance when compared to a histogram.

However, this ability that enables KDEs to obtain lower uncertainties per particle history is more computationally expensive than using traditional histogram bins on a structured mesh. When using KDEs, it is not unlikely for the active tally runtime to increase by an order of magnitude compared to using a comparable histogram tally. Significantly more floating-point operations are required for KDEs for each collision or particle track than a histogram tally since a single event can contribute to the scores of multiple tally points and the function evaluations required to compute those scores are more computationally intense than a simple histogram tally. Even so, it is possible to reduce this additional cost by exporting the KDE tally routines onto GPUs.

The fractional approximate Mean Free Path (aMFP) [6, 7] and cylindrical MFP KDE [7, 8] are accelerated in this paper. The collision KDE for estimating reaction rate densities is defined as

$$\hat{f}(\mathbf{x}) = \frac{1}{N} \sum_{i=1}^{N} \sum_{c=1}^{c_i} \frac{w_{i,c}\Sigma_r(\mathbf{x}, E)}{\Sigma_t(\mathbf{X}_{i,c}, E)} K(\mathbf{x}, \mathbf{X}_{i,c}, \mathbf{h}), \tag{1}$$

where N is the number of histories, c_i is the number of collisions in history i, Σ_r is the macroscopic cross section for the reaction of interest (set to 1 when estimating the scalar flux), Σ_t is the total macroscopic cross section, E is the energy of

the particle entering the collision, $w_{i,c}$ is the weight of particle i prior to collision c, K is the multivariate kernel function, $\mathbf{x}$ is the location of the tally point, $\mathbf{X}_{i,c}$ is the location of collision c in history i, and $\mathbf{h}$ is the bandwidth vector containing the bandwidths in each dimension. The energy dependence of the macroscopic cross sections will be suppressed in the notation the remainder of the paper for clarity. The fractional aMFP kernel function is defined as

$$K(\mathbf{x}, \mathbf{X}_{i,c}, \mathbf{h}) = \prod_{l=1}^{d} \frac{\Sigma_t(\mathbf{x})^{1/d}}{\overline{\Sigma}_t^{1/d} h_l} k\left(\frac{\Sigma_t(\mathbf{x})^{1/d}\left(x_l - X_{l,i,c}\right)}{\overline{\Sigma}_t^{1/d} h_l}\right), \quad (2)$$

where d is the number of dimensions, h_l is the bandwidth in dimension l, $\overline{\Sigma}_t$ is the flux-weighted average macroscopic cross section defined in Eq. (9), k is the univariate kernel function, and $x_l - X_{l,i,c}$ is the distance between the tally point and collision site in dimension l.

The cylindrical KDE was developed to better capture distributions in cylindrical geometries by using kernel functions that use the distance between the collision site and tally point in cylindrical coordinates rather than Cartesian coordinates. For reactor physics problems, the cylindrical MFP KDE uses the number of MFPs between the particle event and tally point in the radial dimension. The 2-D cylindrical MFP kernel function is defined as

$$K(\mathbf{x}, \mathbf{X}_{i,c}, \mathbf{h}) = Ck\left(\frac{\int_{r_{i,c}}^{r} \Sigma_t(r')dr'}{\overline{\Sigma}_t h_r}\right) k\left(\frac{\theta - \theta_{i,c}}{h_\theta}\right), \quad (3)$$

where C is the normalization coefficient

$$C = \frac{1}{\overline{\Sigma}_t h_r h_\theta} \frac{\Sigma_t(r_{i,c})^2}{\left(r_{i,c}\Sigma_t(r_{i,c}) - \int_{r}^{r_{i,c}} \Sigma_t(r')\mathrm{d}r'\right)}. \quad (4)$$

The details the KDE implementation for Monte Carlo eigenvalue calculations is described in Section 2.

In order to conduct the fractional aMFP KDE and the cylindrical MFP KDE the kernels in Eqs. (2) and (3) are evaluated for every tally point that falls within range of the kernel centered on the collision site. For problems where a high resolution is desired this can cause a significant increase in runtime compared to a comparable histogram tally. Since the KDE in Eq. (1) requires computing the score from each collision site to multiple tally points, it is amenable to vectorization and thus calculation on a GPU. The implementation and optimization of the KDE algorithm in Eq. (1) on GPUs is described in Section III.

2. KDE Implementation

The Epanechnikov kernel [9]

$$k(u) = \frac{3}{4\sqrt{5}}\left(1 - \frac{u^2}{5}\right), |u| \leq \sqrt{5}, \quad (5)$$

is used in this paper for all kernel functions k other than the azimuthal kernel due to its finite support region and optimal variance and bias properties. The Cartesian bandwidths are calculated via the optimal bandwidth formula [4, 7, 10]

$$h_l = \left(\frac{4}{(2+d)N}\right)^{1/(4+d)} \sigma_l, \quad (6)$$

where N is the expected number of collisions across all of the active batches and

$$\sigma_l = \left[\sum_{i=1}^{N} \frac{w_{i,l}}{\Sigma_t(\mathbf{X}_i)} X_{i,l}^2 - \left(\sum_{i=1}^{N} X_{i,l} \frac{w_{i,l}}{\Sigma_t(\mathbf{X}_i)}\right)^2\right]^{1/2} \times \left(\sum_{i=1}^{N} \frac{w_{i,l}}{\Sigma_t(\mathbf{X}_i)}\right)^{-1/2} \quad (7)$$

The radial bandwidth is calculated using Eq. (6) but using σ_r instead of σ_l calculated via

$$\sigma_r = \left[\sum_{i=1}^{N} \frac{w_{i,l}}{\Sigma_t(\mathbf{X}_i)} r_i^2 - \left(\sum_{i=1}^{N} r_i \frac{w_{i,l}}{\Sigma_t(\mathbf{X}_i)}\right)^2\right]^{1/2} \times \left(\sum_{i=1}^{N} \frac{w_{i,l}}{\Sigma_t(\mathbf{X}_i)}\right)^{-1/2} \quad (8)$$

The sample collisions required for calculating σ, N, and $\overline{\Sigma}_t$ are done in the last inactive batch of the eigenvalue calculation. The expected number of collisions in the active batches N is estimated by multiplying the number of collisions that occurred in the last inactive batch by the total number of active batches. Region-based bandwidths are used, with each KDE region having its own bandwidth and average cross section calculated via

$$\overline{\Sigma}_t = \frac{\int_0^\infty \int_\Gamma \Sigma_t(\mathbf{x})\phi(\mathbf{x})\mathrm{d}V\mathrm{d}E}{\int_0^\infty \int_\Gamma \phi(\mathbf{x})\mathrm{d}V\mathrm{d}E}, \quad (9)$$

where Γ is the volume defining a given KDE region.

III. GPU IMPLEMENTATION AND OPTIMIZATION

1. GPU Architecture

In order to accelerate algorithms on the GPU the underlying architecture must be considered. GPUs are based on Single Instruction, Multiple Thread (SIMT) architecture and are composed of hundreds to thousands of threads per GPU. During execution, the threads in a GPU are divided into groups of 32 threads, called warps, with all threads in a warp either performing the same instruction or sitting idle while the other threads in the warp execute the instruction. Warp divergence occurs when a warp encounters a data-dependent conditional branch, causing the execution of the different paths to be serialized resulting in a reduction in the performance of the algorithm.

The major memory components on a GPU accessible to developers are global memory and shared memory. Global memory is used to transfer memory between the GPU and CPU and can be accessed from all warps on the GPU, however it has a latency of approximately 400-800 clock cycles for the

GPU used in this paper. The SIMT architecture of the GPU requires threads within a warp to access contiguous sections of global memory. Thus, if all 32 threads in a warp require data from different places in global memory then 32 different loads from global memory will be required. However, if all threads in a warp require data located contiguously in global memory then this load from global memory can be coalesced into a single read from global memory if each piece of data is 32-bits or smaller. While the GPU is capable of hiding latency when accessing global memory by switching between warps with its streaming multiprocessors (more details can be found in [11]), minimizing the number of loads from global memory typically leads to improved performance.

Shared memory is fast on-chip memory that is local to each block of threads and has a latency approximately 100 times lower than that of global memory. Furthermore, shared memory does not have contiguous-data access restrictions like global memory. Thus, shared memory can be leveraged to store data that is often used and shared among warps. Furthermore, the less restrictive access requirements allow for efficient parallel reductions within a block of threads.

This work uses CUDA streams for asynchronous communication between the CPU and GPU as well as for concurrent execution of different GPU functions (called kernels) and concurrent execution of GPU kernels and memory copies between the CPU and GPU. Commands placed in a CUDA stream execute sequentially and commence execution without any further input from the CPU.

The SIMT-architecture makes the traditional parallelization methods of Monte Carlo codes intractable on a GPU due to the data-dependent conditional statements causing warp divergence as well as the random access of memory when looking up continuous-energy cross sections preventing efficient access of memory. However, the KDE algorithms are vectorizable and can be exported to the GPU for acceleration.

2. Algorithm Implementation

The GPU KDE algorithm is implemented in a modified version of OpenMC using CUDA C [11] and uses C Bindings to link the CUDA C code to the main Fortran program in OpenMC. The algorithm is designed to have simultaneous execution of the tally process on the GPU and the transport process on the CPU. Flowcharts describing the layout of the CPU and GPU portions of the GPU algorithm are shown in Figures 1 and 2 respectively. Rather than tally scores directly, the CPU collects information in one of two sets of sample arrays during the transport process. The CPU fills one set of sample arrays with all necessary collision and cross section information for a pre-set number of collisions (50,000 in this paper) and then copies the data asynchronously to the GPU. The CPU then puts the KDE GPU kernels into the same CUDA stream as the memory copy and then immediately returns and begins transporting particles and populating a second set of sample arrays. Once the CPU has finished filling the second array, it asynchronously sends the data to the GPU and waits until the first set of sample arrays has been received by the GPU before re-filling the first set of arrays. This process repeats until the end of the batch, when the partial set of collisions is sent to the GPU for calculation of scores. Once the GPU finishes processing the last set of sample arrays the GPU sends the tally data to the CPU and the CPU uses the normal MPI processes for combining tally data across multiple processors on multiple compute nodes.

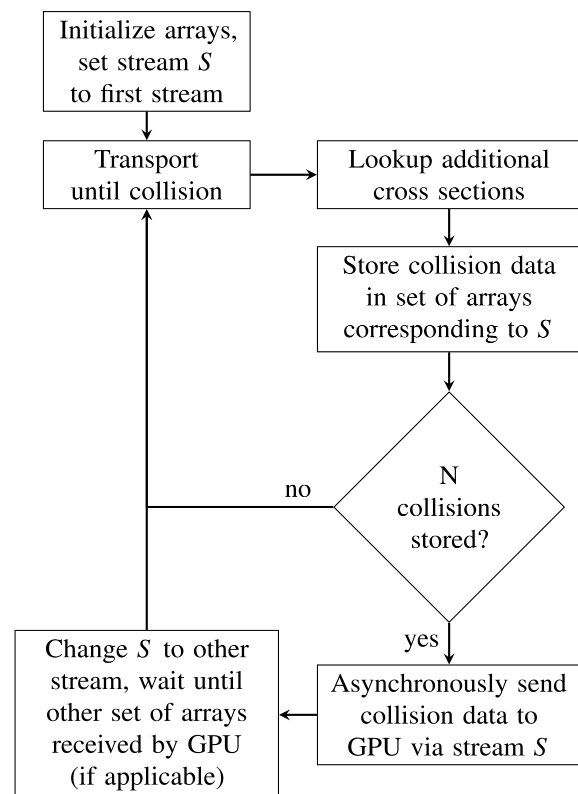

Fig. 1. Flowchart depicting CPU process for collecting tally data to send to the GPU.

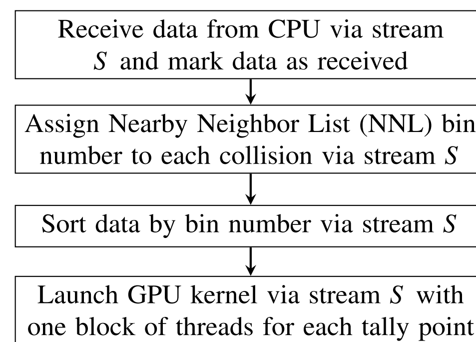

Fig. 2. Flowchart depicting GPU process for receiving and processing tally data from CPU.

Once the GPU has received the first array, it sorts the data for better memory coalescence and the KDE GPU kernel is launched. The kernel uses 64 threads per block, with one block per tally point. Each block uses the 64 threads to loop over the collisions in the sample array and compute each collision's score at that block's tally point. Each thread keeps its own sum of scores in shared memory in order to prevent serialization

of threads via atomic operations when incrementing the tally point's score. After looping over all collisions within a set, the scores from individual threads are reduced using shared memory and the total score to a tally point from all collisions within the sample array is added to the sum in global memory. This approach minimizes the number of atomic operations necessary since each block of threads is incrementing the score of different tally points. However, this approach also scales linearly with the number of tally points in the simulation and could prevent the efficient simulation of large systems with high resolution.

The algorithm was designed for a high performance computer cluster where each compute node has its own GPU(s). The cores on each compute node communicate only with their local GPU, and the tally results are combined with the usual MPI processes. Since each processor is only communicating with its local GPU, this algorithm scales with MPI to multiple cores across multiple compute nodes without a reduction in the GPU's performance.

To increase the speed of this algorithm, a Nearby Neighbor List (NNL) was created on the GPU similar to that on the CPU. Rather than loop over all collisions, the finite support of the KDE kernel necessitates only looping over collisions that contribute non-zero scores to the tally point. The NNL uses a mesh to divide the simulation domain into bins equal to the maximum kernel support length in each dimension ($h\sqrt{5}$ for the Epanechnikov kernel in Eq. (5)). Each collision is assigned a key based on on its position in the neighborhood mesh. The index of each particle (the particle's initial location in the collision array) is sorted based on its key value using the CUDA UnBound (CUB) library's radix sort algorithm. The collision data are then rearranged based on their key and index values so that the data pertaining to collisions occurring within the same neighborhood bin are located contiguously in memory. The use of a NNL is crucial for KDE algorithm performance, with speedups of over 800 obtained for the GPU algorithm on a 2-D pincell problem with a grid of 120×120 tally points.

While the NNL works well for the distance-based KDEs, the MFP KDE algorithm introduces complications. The MFP KDE either expands or contracts the bandwidth in space, and thus the maximum kernel support length in each dimension becomes an energy-dependent quantity that changes with each collision. This results in multiple bins being searched in each dimension, reducing the efficiency of the calculation. Furthermore, since the maximum kernel support length is cross section dependent, each material cross section would have to be computed in order to determine the maximum kernel support length for a given collision. This complication can be eliminated by limiting the value of the macroscopic total cross section in the argument of the kernel function and the normalization coefficient in Eqs. (2) and (3), denoted as $\Sigma_{t,k}(x)$ in the following equation, such that

$$\Sigma_{t,k}(x) = \begin{cases} \Sigma_t(x) & \Sigma_t(x) > \overline{\Sigma}_t \\ \overline{\Sigma}_t & \text{otherwise} \end{cases} \tag{10}$$

Using Eq. (10) ensures that if the cross section is below the average cross section for a region, the MFP kernel reverts to the distance-based kernel. This effectively creates a maximum spatial bandwidth for the MFP KDE and ensures that collisions can only score to tally points in adjacent NNL bins. This approximation does not adversely affect the accuracy of the simulation since using a smaller bandwidth reduces the bias in the estimate at a cost of increased variance. Furthermore, the use of Eq. (10) removes the need to compute all material cross sections after each collision, further increasing the efficiency of the algorithm.

Additionally, a local cross section lookup routine is implemented to further enhance the speed of GPU cylindrical MFP KDE algorithm. Since the KDE allows for collisions to contribute to reaction rates across material interfaces, additional cross sections must be looked up after each collision. While this additional lookup can be done on-the-fly for the CPU algorithm when using a maximum kernel support length, the GPU algorithm requires that all cross sections are computed prior to sending data to the GPU. However, the majority of collisions do not occur near material interfaces, and thus looking up additional cross sections is often unnecessary. Since the cylindrical MFP KDE is used on problems with cylindrical geometry, it is straight-forward to implement an algorithm that determines if additional cross sections need to be computed for a given collision. This algorithm results in a speedup of 1.3 for the cylindrical MFP KDE on the 2-D pincell problem shown later in this paper.

3. GPU Optimization

Several improvements were made throughout the design of the algorithm to reduce compute times. The optimization strategies tested in this paper are changing the collision data structures from an Array of Structures (AoS) to a Structure of Arrays (SoA), leveraging shared memory, and switching from double-precision to single-precision calculations. The optimizations were tested on an NVIDIA Tesla M2090 GPU using the fractional aMFP KDE on a 2-D pincell problem with 120×120 tally points and 50,000 collisions per set of sample arrays. GPU compute kernel speedups relative to the base implementation with an NNL are shown in Table I. Kernel compute times were determined using the NVIDIA Visual Profiler [12], with the compute kernel time of 35.6 ms for the base implementation using an array of structures. While the times obtained from the profiler are not as accurate as using CUDA events to obtain timing information, the profiler was found to be sufficiently accurate for determining relative performance improvements.

The choice of data structure has a significant impact on

TABLE I. Compute kernel speedups for 50,000 collisions and 120×120 tally points on a 2-D pincell problem.

Design Iteration	Relative Kernel Speedup
Array of structures	1
Structure of arrays	1.6
Shared memory w/ SoA	2.0
Single-precision w/ SoA & shared memory	2.9

the compute kernel performance. When using an AoS the data of the structure is placed sequentially in memory. This leads to a loss in efficiency based on the size of the structure. For example, the usual data used to score a set of collisions includes the colliding particles' position, angle, energy, cell, material, and weight. If the collisions' location in the x dimension is being requested by the threads in a warp then an additional 6 floating point numbers and two integers (56 bytes in double-precision) exist in memory between the x positions accessed by neighboring threads. This leads to a total of 16 loads from global memory being required (for 128 byte loads) in order for a warp of 32 threads to access the x positions of 32 collisions stored contiguously in memory. Using an SoA enables coalesced memory accesses since the collisions' data is stored in separate arrays. For example, the set of collisions passed to the GPU now contains one collision data structure containing separate arrays for each piece of data in the original structure. Thus, for the threads to access the x positions of 32 different collisions now only 2 loads from global memory are required for double-precision data. This reduction in loads from global memory as a result of using an SoA results in a compute kernel speedup of 1.6.

Leveraging the shared memory on the GPU provides additional speedup. Storing often-used data from global memory in shared memory at the start of each compute kernel provides an additional speedup of 25%.

Switching from double-precision to single-precision also increases performance by 45%. Changing to single-precision has two effects. First, it reduces the amount of memory required by each thread, enabling a coalesced load of floating-point numbers to be read in with one access to global memory rather than two. Furthermore, the GPU has a faster clock speed for single precision floating-point operations, with the NVIDIA Tesla M2090 having a single-precision clock speed that is twice that of double-precision operations. Overall, optimizing the base algorithm with an NNL produces a compute kernel speedup of 2.9.

IV. RESULTS

1. Fractional Approximate MFP KDE

The GPU algorithm with the fractional aMFP KDE was tested on a quarter-assembly of pincells, depicted in Fig. 3. Each pincell is comprised of a cylinder of 3% enriched UO_2 surrounded by water with a pin diameter of 0.7 cm and a lattice pitch of 1 cm. Absorbers of B_4C replace several of the fuel squares and are shown in black. Continuous-energy cross sections from the ENDF/B-VII.0 [13] cross section library are used for all materials. Region-based bandwidths are used with one KDE region assigned to each pincell. The boundary kernel method is used at external reflecting boundaries [14]. The use of the boundary kernel does not introduce warp divergence since each thread in a warp is calculating scores to the same tally point, thus all threads in a warp will use the boundary kernel routines if the tally point requires them. The simulation was run with 200,000 particles per batch, 2,000 total batches with 100 inactive batches. The simulations were run using two Tesla M2090 GPUs with 16 MPI processes on two eight-core Intel Xeon E5-2670 processors while the histogram results use 16 MPI processes on the CPU. Reference histogram results were collected on a structured grid of 120×120 bins with KDE tally points placed at the center of each bin. The flux obtained using the fractional aMFP KDE is shown in Fig. 4 with the C/E (KDE/histogram) comparison of the flux distributions shown in Figure 5. Speedups and runtime statistics for the assembly of pincells were acquired with 120×120 tally points and 240×240 tally points.

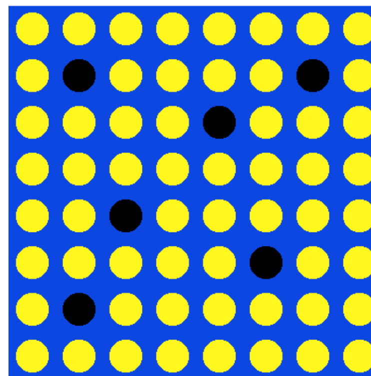

Fig. 3. Depiction of the assembly of pincells.

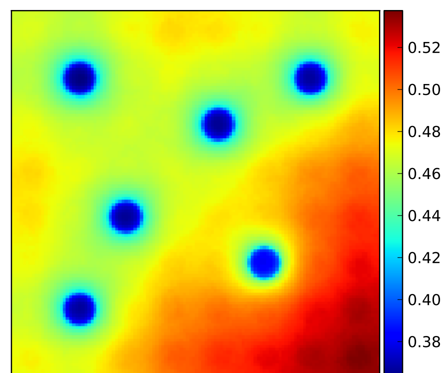

Fig. 4. Flux distribution from the fractional aMFP KDE for the assembly of pincells.

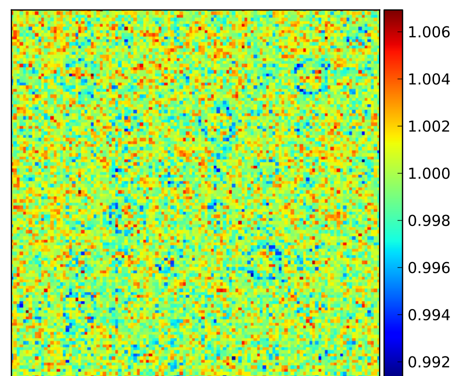

Fig. 5. C/E comparison between flux distributions obtained from the fractional aMFP KDE and reference histogram for the assembly of pincells.

Figures 4 and 5 show that the fractional aMFP KDE is capable of accurately capturing the flux and fission distributions in reactor physics problems. The difference between the flux distribution obtained from the fractional aMFP KDE and the reference histogram is less than 1% for all tally points. While there is a pattern of disagreement around the control rods, it is primarily due to comparing point-wise histogram results with volume-average KDE results in regions of steep flux gradients which would reduce by increasing the tally resolution. The Figure of Merit (FOM) for the fractional aMFP KDE is 3.3 times higher than that of the collision histogram tally with the FOM calculated using

$$\text{FOM} = \frac{1}{\frac{1}{N}\sum_i^N \left(\frac{\sigma_i}{\phi_i}\right)^2 T}, \tag{11}$$

where N is the number of histogram bins or tally points, σ_i/ϕ_i is the relative uncertainty at tally point or bin i, and T is the time spent in the active batches.

The maximum speedup achievable by using GPUs occurs when the entire KDE tally can be computed on the GPUs without the CPUs having to wait on the GPUs. The speedup obtained using the GPU algorithm with 120×120 tally points is relatively constant for a varying number of MPI processes, with a final speedup of 1.8 when using 16 MPI processes for both single-precision and double-precision calculations. The single-precision results agree with the double-precision results to six significant digits. In this scenario the KDE tally is essentially free with the exception of looking up additional cross sections after every collision. Speedups of 5 and 3.7 are obtained for 240×240 tally points when using 8 and 16 MPI processes, respectively. This speedup when using 240×240 tally points is greater than that obtained using 120×120 tally points due to the greater tally point density and thus a greater problem difficulty for the CPU KDE algorithm. The decline in speedup when switching from 8 to 16 MPI processes indicates that the GPUs are saturated with work when using 16 MPI processes. This shows that there is a limit to the speedup obtainable on the GPU before adding more work (more processors) per GPU becomes detrimental.

2. Cylindrical MFP KDE

The acceleration of the cylindrical MFP KDE algorithm is tested using a CASL 2-D pincell benchmark problem [15] comprised of a cylinder of 3.1% enriched UO_2 with radius 0.4096 cm surrounded by Zircaloy-4 cladding and 1300 ppm borated water with a lattice pitch of 1.26 cm and reflecting boundary conditions. The cladding is blended in with the gap and extends from 0.4096 cm to 0.475 cm, with the density of the Zircaloy adjusted to 5.77 g/cm^3 to preserve the amount of Zircaloy in the problem when the gap is explicitly modeled. This model was previously used to verify the cylindrical MFP KDE in [8] with results compared to a histogram tally. The performance comparison here uses the same mesh to generate the KDE tally points, with KDE tally points placed at the center of the bins in a cylindrical mesh constructed using 8 uniform bins in the azimuthal dimension and a radial mesh of 20 bins from 0 to 0.3896 cm, 200 bins from 0.3896 to 0.4096 cm, 5 bins from 0.4096 to 0.475 cm, and 20 bins from 0.475 to 0.63 cm. The fission and absorption distribution comparisons between the cylindrical MFP KDE and the reference histogram originally shown in [8] are recreated here for illustrative purposes in Figures 6-8 with the distributions shown on top and the C/E values shown below. Simulations for timing were conducted using 1,000,000 particles per batch, 10 inactive batches and 20 total batches using 16 MPI processes on two eight-core Intel Xeon E5-2670 processors and two Tesla M2090 GPUs. The speedup results are not significantly affected by using more batches; the only effect is to reduce the bandwidth according to Eq. (6).

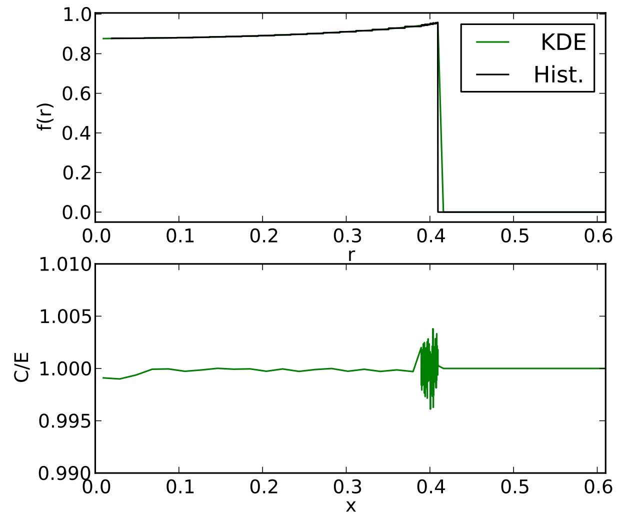

Fig. 6. Fission distribution comparison between the cylindrical MFP KDE and histogram for the pincell problem along the 22.5° azimuth.

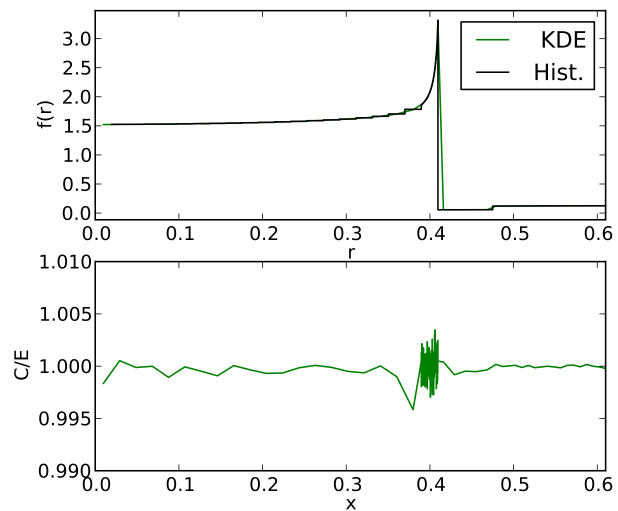

Fig. 7. Absorption distribution comparison between the cylindrical MFP KDE and histogram for the pincell problem along the 22.5° azimuth.

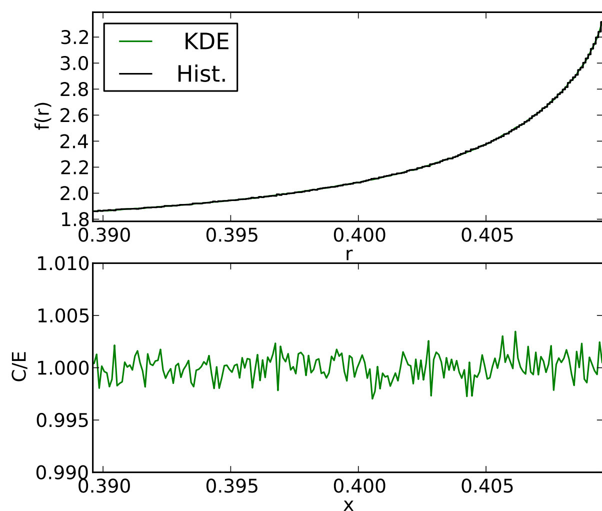

Fig. 8. Absorption distribution comparison in the rim region (200 μm into the edge of the fuel) between the cylindrical MFP KDE and histogram for the pincell problem along the 22.5° azimuth.

Figures 6-8 show that the cylindrical MFP KDE agrees with the reference histogram within 0.5% at all tally points, with the cylindrical MFP KDE capable of accurately capturing the rim effect at the fuel-clad interface. Furthermore, the CPU and GPU cylindrical MFP KDE algorithms agree within eight significant digits at all tally points, with the GPU algorithm producing a speedup of 1.35. The simulation with the GPU tally produces an inactive tally calculation rate of 76,500 particles per second and an active calculation rate of 68,000 particles per second for a total simulation time of 300 seconds. Doubling the number of tally points in the radial dimension results in a speedup of 1.65, with the simulation with the GPU tally again producing an active tally calculation rate of 68,000 particles per second and a total simulation time of 300 seconds. This again shows that the additional cost of the cylindrical MFP KDE is almost entirely hidden when conducted on the GPU for the problem sizes studied in this paper. The difference between the inactive and active tally calculation rates is primarily due to the calculation of additional cross sections for the KDE algorithm.

V. CONCLUSIONS & FUTURE WORK

A method for using heterogeneous computing to accelerate computationally intensive tally algorithms in Monte Carlo radiation transport problems was introduced and tested on a quarter-assembly problem and a pincell problem in continuous energy. The methodology shown in this paper can be used to accelerate computationally expensive algorithms via GPUs in Monte Carlo codes without having to re-write large portions of the code base. Speedups for problems shown in this summary range between 1.35 and 5, depending on the number of MPI processes and the number and density of tally points. Using the GPU to accelerate the KDE routines makes the algorithm essentially free for the cylindrical MFP KDE on the 2-D pincell problem and the fractional aMFP KDE on the quarter-assembly problem with 120×120 tally points, with the exception of having to compute additional cross sections. It should also be noted that increasing the difficulty of the transport problem, for example by simulating more complicated geometries, would allow even more computationally expensive routines to be hidden through execution on the GPU.

Future work includes further improving the collision KDE algorithm. Since the current algorithm scales linearly with the number of tally points, this may prevent conducting high resolution tallies in large systems. Redesigning the algorithm to scale with tally point density would enable such simulations. Additionally, the impact of using newer GPUs on performance could be investigated. The GPUs in this paper do not allow concurrent execution of kernels launched from different MPI processes; newer GPUs with CUDA compute capability 3.5 do not have this restriction and could potentially improve performance. Furthermore, the track-length KDE has yet to be developed for the GPU. While it is more difficult to have coalesced memory accesses when using track-length KDEs, each particle track requires more calculations to compute the score to a tally point compared to a collision thus potentially enabling the GPU to hide the cost of additional global memory fetches. Additionally, future work includes extending the heterogeneous computing methodology developed in this paper to other compute-intensive portions of Monte Carlo codes.

VI. ACKNOWLEDGEMENTS

This material is based upon work supported in part by the National Science Foundation Graduate Research Fellowship under Grant No. DGE 1256260 and by the US DOE/NNSA Advanced Scientific Computing program and was made possible through the use of Los Alamos National Laboratory's high-performance computing resources.

REFERENCES

1. T. LIU, N. WOLFE, C. D. CAROTHERS, W. JI, and X. G. XU, "Status of Archer - A Monte Carlo Code for the High-performance Heterogeneous Platforms Involving GPU and MIC," Nashville, TN (April 19-23 (2015)).
2. R. M. BERGMANN, J. L. VUJIC, E. GREENSPAN, P. F. PETERSON, R. N. SLAYBAUGH, and P. O. PERSSON, "The Development of WARP - A Framework for Continuous Energy Monte Carlo Neutron Transport in General 3D Geometries on GPUs," Ph.D. Thesis, University of California - Berkeley (2014).
3. P. K. ROMANO and B. FORGET, "The OpenMC Monte Carlo Particle Transport Code," *Ann. Nucl. Energy*, **51**, 274–281 (2013).
4. K. BANERJEE, "Kernel Density Estimator Methods for Monte Carlo Radiation Transport," Ph.D. Thesis, University of Michigan (2010).
5. K. L. DUNN, "Monte Carlo Mesh Tallies based on a Kernel Density Estimator Approach," Ph.D. Thesis, University of Wisconsin–Madison (2014).
6. T. P. BURKE, B. C. KIEDROWSKI, and W. R. MARTIN, "Approximate Mean Free Path Based Kernel Density Esti-

mators for Reaction Rates in Reactor Physics Problems," in "PHYSOR," Sun Valley, ID (May 1-5 2016).

7. T. P. BURKE, "Kernel Density Estimation Techniques for Monte Carlo Reactor Analysis," Ph.D. Thesis, University of Michigan (2016).
8. T. P. BURKE, B. C. KIEDROWSKI, and W. R. MARTIN, "Cylindrical Kernel Density Estimators for Monte Carlo Neutron Transport Reactor Physics Problems," in "Trans. Am. Nucl. Soc.", **115** ((2016)).
9. V. A. EPANECHNIKOV, "Nonparametric Estimation of a Multidimensional Probability Density," *Theor. Probab. Appl.*, **14**, 153–158 (1969).
10. B. W. SILVERMAN, *Density Estimation for Statistics and Data Analysis*, Chapman and Hall, London, UK (1986).
11. "CUDA C Programming Guide," (2015), http://docs.nvidia.com/cuda/cuda-c-programming-guide.
12. "Nvidia Visual Profiler," (2015), http://www.developer.nvidia.com/nvidia-visual-profiler.
13. M. CHADWICK ET AL., "ENDF/B-VII.0: Next Generation Evaluated Nuclear Data Library for Nuclear Science and Technology," *Nuclear Data Sheets*, **107**, *12*, 2931 – 3060 (2006), evaluated Nuclear Data File ENDF/B-VII.0.
14. M. C. JONES, "Simple Boundary Correction for Kernel Density Estimation," *Statistics and Computing*, **3**, 135–146 (1993).
15. "VERA Core Physics Benchmark Progression Problem Specfications," (2014), www.casl.gov/docs/CASL-U-2012-0131-004.pdf.