Los Alamos
NATIONAL LABORATORY
EST. 1943

LA-UR-18-27987
Approved for public release; distribution is unlimited.

Title: Development of a Library for Conducting Monte Carlo Tallies on Heterogeneous Systems

Author(s): Burke, Timothy Patrick

Intended for: Report

Issued: 2018-08-21

Disclaimer:
Los Alamos National Laboratory, an affirmative action/equal opportunity employer, is operated by the Los Alamos National Security, LLC for the National Nuclear Security Administration of the U.S. Department of Energy under contract DE-AC52-06NA25396. By approving this article, the publisher recognizes that the U.S. Government retains nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or to allow others to do so, for U.S. Government purposes. Los Alamos National Laboratory requests that the publisher identify this article as work performed under the auspices of the U.S. Department of Energy. Los Alamos National Laboratory strongly supports academic freedom and a researcher's right to publish; as an institution, however, the Laboratory does not endorse the viewpoint of a publication or guarantee its technical correctness.

Development of a Library for Conducting Monte Carlo Tallies on Heterogeneous Systems

Timothy P. Burke

July 20, 2018

Abstract

A library is introduced that enables users to conduct Monte Carlo tallies on heterogeneous computing systems. The library can be built with or linked to a Monte Carlo transport code to provide tally capabilities for the project. The library provides a set of tools and an interface for constructing tallies that can be computed on CPUs or GPUs regardless of where the sample data was generated. The library can be incorporated into existing Monte Carlo transport codes at both the research and production level with minimal intrusion into the transport code. The tally library is capable of computing a variety of tallies including track-length histogram (mesh) tallies, collision-based kernel density estimators, functional expansion tallies, eigenvalue sensitivity coefficients via iterated fission probability, and sensitivities of reaction rates or adjoint-weighted quantities via differential operator sampling. The parallelization methods are discussed and speedups are shown for 2-D and 3-D problems with comparisons to both the tally library CPU implementations as well as the tally capabilities in the host transport code. Performance results and examples are shown for MCNP, MCATK, and OpenMC.

I Introduction

The inclusion of GPUs in High Performance Computing (HPC) machines is becoming increasingly common. However, Monte Carlo particle transport is poorly suited for acceleration on GPUs due to the random memory access patterns associated with cross section and geometry look-ups as well as due to the branching from logic statements within individual histories. A considerable amount of effort has been expended to circumvent or overcome these limitations, with the use of event-based Monte Carlo being one example [1, 2, 3]. However, current transport codes would have to be largely redesigned in order to use methods like event-based Monte Carlo in order to run efficiently on GPUs. While some Monte Carlo codes have had success in exporting the transport process to GPUs for specific applications, general production-level transport codes have yet to widely make use of GPUs [4, 5].

Rather than export the entire particle transport process to the GPU, it is possible to accelerate portions of the Monte Carlo code via GPUs while the main transport process is conducted on the CPU. Previous work has shown that Monte Carlo tallies can be accelerated for specific applications [6, 7, 8]. This work introduces a tally library that is a work-in-progress for computing general Monte Carlo tallies on heterogeneous architectures. The purpose of this library is two-fold. First, it enables the transfer of tally methods between Monte Carlo transport codes. Second, it enables the use of GPUs in current production-level Monte Carlo transport codes to compute general Monte Carlo tallies using the same algorithm on the CPU and GPU. Both of these capabilities are feasible since the transport process is independent from the tally process within a steady-state calculation or within a time step during a transient calculation. The library can be included in existing research or production level codes and used to conduct tallies on either the CPU or GPU regardless of where or how the sample data (collisions, particle tracks, surface-crossings, etc.) are generated.

This work serves as a developer and user guide for the tally library, as well as a proof-of-concept for creating a generic tally library that can be linked to existing transport codes that is capable of conducting tallies on heterogeneous computing systems. The tally library is written in C++ and has been linked with MCNP®[1], MCATK, and OpenMC [9, 10, 11]. Tally performance results are shown for a variety of transport problems in continuous energy, and performance limitations for GPUs is discussed. A high-level summary of the library and it's performance for 3-D mesh tallies can be found in Ref. [12].

II Library Overview

The tally library is being developed as a method to share a common a set of tools among Monte Carlo transport codes for computing tallies capable of running on heterogeneous systems. The traditional process for creating a Monte Carlo particle transport code is to have all necessary capabilities self contained within the code. This process leads to significant code duplication as many Monte Carlo codes contain similar or even identical features and routines. For example, Monte Carlo codes are often interested in tallying quantities like flux or reaction rates in a mesh or a series of cells. In order to compute these tallies, most Monte Carlo codes employ their own tally algorithms and data structures. However, this reliance on transport-code-specific data structures and types is not necessary for tallying. Since the transport process is independent from the tally routines in a Monte Carlo code, the tally capability can be removed from the Monte Carlo transport code base and put into an external library. This separability allows tallies to be computed on the GPU while the transport process is conducted on the CPU, and it enables the tally library to be written generally so that it can be used in multiple transport codes. Details on the features and capabilities in the tally library are given in Section III.

The general overview of the tally routine is as follows. Whenever a particle event is generated in the transport routine, the transport code calls on the tally library to handle the relevant information (distance, position, angle, energy, cell, etc.) required to compute the tallies. This event-specific information, referred to as a sample, can either be stored in a container and processed in batch with other samples or it can be processed immediately after it is generated. Once a pre-defined number of samples have been stored (49152 for this paper) or the end of a generation or batch has been reached, the samples are copied to the GPU for processing and then processed through any GPU or CPU tallies specified by the user. Specifics regarding the tally library implementation are given in Section IV.

To use the tally library, a transport code can either call on methods within the library directly or it can pass data to the tally library through an interface. These interfaces can be transport-code specific to reduce the burden of data transfers between the library and transport code, or they can be generic C or Fortran interfaces. The interface consists of calling initialization functions that construct the tallies and structures necessary to store the samples and conduct the tallies, functions to store and process samples and process batches of stored samples, functions to compute tally statistics, functions to handle particle splitting, and functions to print results. A description of the interface functions and their various purposes is given in Section IV.B.

In addition to developing a general library for computing tallies, this work develops tallies that are designed to operate on both CPUs and GPUs. Since the tally process is independent from the transport process the tally process can be exported to GPUs, enabling the use of GPUs for a portion of the Monte Carlo simulation. This gives the Monte Carlo simulation access to the GPU's memory and additional computing power that would otherwise sit unused. The GPU can be used to conduct basic tallies like cell and energy tallies, or it can be used to accelerate tally techniques that are expensive on the CPU, like histogram or mesh tallies, kernel density estimators, expected path length estimators, point detector estimators, or sensitivity tallies. While some of the tally algorithms are not ideal for use on GPUs, the ability to overlap tally calculation on the GPU with transport processes on the CPU can still yield speedups or memory utility benefits for many applications. Details regarding the implementation for

[1] MCNP® is a registered trademark owned by LANS, LLC, manager and operator of LANL.

heterogeneous computing environments are given in Section IV and performance results are shown in Section V.

III Tally Library Features

The tally library is capable of computing tallies in fixed-source and eigenvalue calculations, and it can accommodate any particle type. For example, Section III.B demonstrates the ability to compute photon flux in a medical phantom, and Sec. V.A shows results for neutron flux in a 3-D mesh tally. The tally library is capable of computing the following tallies:

- Track-length histogram (mesh)
- Collision estimators
- Surface crossing flux estimators
- Collision-based Kernel Density Estimators (KDEs)
- Functional Expansion Tallies (FETs)
- k_{eff} sensitivities to density or geometric perturbations
- Reaction rate sensitivities to density or geometric perturbations

All of the above tallies with the exception of reaction rate sensitivities can be executed on the GPU. As will be shown in Section V, this can result in reductions in wall-clock time compared to both CPU execution as well as compared to using the tally capabilities provided by the transport code. Furthermore, all of the above tallies can be used in combination with a generic series of filters. The library currently supports filters over energy bins, cells, and surfaces. The tally library uses batch statistics for computing uncertainties. The number of batches used for statistics is the number of cycles or generations used for an eigenvalue calculation, and is specified in the talon.xml input file for fixed-source calculations with a default value of 20 batches if not specified. The number of histories in the fixed source calculation needs to be a multiple of the number of batches, but this requirement could be relaxed. Tally visualization is done via ParaView with details and examples shown in Section III.B [13]. Tallies can be defined at run-time or at compile-time, with the input for tallies described in Section III.A.

Currently, the tally library has the capability to compute total, fission, absorption, nu-fission, and scattering reaction rates. The implementation requires the cross sections for these reaction rates to be looked up and stored on the CPU at every particle event. This cross section data is then passed to the GPU with the particle event data for processing of tallies. In the future, this need could be eliminated by re-computing only the necessary cross sections or by implementing a feature that computes tallies for every particle event rather than storing events for later scoring. A brief overview of the lesser-known tallies implemented in the library is given below.

Functional Expansion Tallies

Functional Expansion Tallies (FETs) can be used to obtain functional representations of flux profiles, dose profiles, or reaction rate densities using on a truncated series expansion of orthogonal basis functions via

$$f(x) \approx \sum_{n=0}^{N} a_n p_n(x),$$

where $f(x)$ is the reaction rate density being estimated, N is the number of polynomials used in the expansion of $f(x)$, and a_n is the coefficient corresponding to polynomial p_n. The expansion coefficient for each basis function is

estimated in Monte Carlo codes via

$$a_n = \frac{1}{W} \sum_{n=1}^{N} \sum_{c=1}^{c_n} \frac{w_{n,c} \Sigma_r \left(\boldsymbol{X}_{n,c}, E_{n,c} \right)}{\Sigma_t \left(\boldsymbol{X}_{n,c}, E_{n,c} \right)} p_n(x), \tag{1}$$

where W is the total weight being simulated, N is the total number of histories in this generation, C_n is the number of collisions in history n, $w_{n,c}$ and $E_{n,c}$ are the weight and energy of history n entering collision c, $\boldsymbol{X}_{n,c}$ is the location of the collision c in history n, Σ_r is the macroscopic cross section corresponding to reaction rate f, and Σ_t is the macroscopic total cross section. FETs have shown potential for multi-physics coupling applications [14] and can be used to obtain continuous representations of densities like dose profiles. Currently, the tally library uses separate Legendre polynomials for x, y, and z, with future work including the addition of Zernike polynomials and other multivariate polynomials. For more details on FETs in nuclear engineering applications see Refs. [15] and [16].

Kernel Density Estimators

KDEs have been studied in depth for applications to tallies in radiation transport problems [17, 18, 19]. KDEs are non-parametric estimators that obtain estimates at points in phase space. The uncertainties associated with each point's estimates are unaffected by the desired resolution (local density of tally points). Instead, their accuracy is dependent on the kernel width, or bandwidth. Due to their smoothing properties and their ability to obtain point-wise representations of underlying distributions, KDEs show potential for applications to multi-physics coupling and for obtaining estimates of distributions on high-resolution structured meshes or unstructured meshes.

Estimates at a tally point are generated by summing kernel functions evaluated for each collision site via

$$\hat{f}(\boldsymbol{x}) = \frac{1}{W} \sum_{n=1}^{N} \sum_{c=1}^{C_n} \frac{w_{n,c}}{\Sigma_t \left(\boldsymbol{X}_{n,c}, E_{n,c} \right)} K \left(\boldsymbol{x}, \boldsymbol{X}_{n,c}, \boldsymbol{h} \right),$$

where W is the total weight being simulated, N is the total number of histories in this generation, C_n is the number of collisions in history n, $w_{n,c}$ and $E_{n,c}$ are the weight and energy of history n entering collision c, $\boldsymbol{X}_{n,c}$ is the location of the collision c in history n, $\boldsymbol{x}$ is the location of the tally point, $\boldsymbol{h}$ is the vector of bandwidths in each dimension, and K is the multivariate kernel function which is typically defined via a product of univariate kernels:

$$K(\boldsymbol{x}, \boldsymbol{X}_{n,c}, \boldsymbol{h}) = \prod_{l=1}^{d} \frac{1}{h_l} k \left(\frac{x_l - X_{n,c,l}}{h_l} \right),$$

where l indicates the dimension and d is the total number of dimensions. For more details on KDEs the reader is referred to Refs. [17, 18, 19, 20].

Sensitivities

The tally library is capable of computing eigenvalue sensitivities via Iterated Fission Probability (IFP), reaction rate sensitivities via differential operator sampling (DOS), and sensitivities of adjoint-weighted quantities (e.g. the effective generation time Λ_{eff} and the effective delayed neutron fraction β_{eff}) via DOS applied to IFP. Details regarding the calculation of sensitivities to adjoint-weighted quantities via DOS applied to IFP can be found in Ref. [21]. Currently, the library is limited to calculating sensitivities due to density perturbations as well as sensitivities due to system dimension or geometric perturbations. Details regarding the method for computing sensitivities to geometric perturbations and its application to critical assemblies and reactor physics problems can be found in Ref. [21] and details regarding the method's application to shielding problems can be found in Ref. [22]. Future work includes implementing the capability to compute sensitivities to nuclide cross sections and atomic densities. Furthermore, the calculation of sensitivities via DOS is limited to the CPU since a particle's history needs to be preserved for

the calculation and the GPU tally algorithms are parallelized on a per-sample rather than a per-history basis. Additionally, the need to preserve a particle's history requires the proper handling of particle splitting events, which is implemented for fixed-source problems but is left for future work for eigenvalue problems. Furthermore, the tally library uses the most memory intensive method for computing sensitivities, with future work including the application of a sparse storage scheme to reduce the memory burden.

III.A Tally Input

Users have two options for specifying tallies: run-time specifications using predefined tallies and filters or compile-time definitions using either predefined tallies and filters or user-generated tallies and filters. The benefit of compile-time definitions is that they can be less general and thus have better performance than the run-time specified tallies. For example, logic statements to decide execution paths based on input parameters can be eliminated. Furthermore, the compiler can unroll loops whose dimensions are specified at compile-time. For instance, if it is common to compute tallies with a specific energy group filter, then the vectors usually used to store bin edge information can be changed to fixed-length arrays, and the bin edge information can be defined at compile time. Compile-time tallies are defined within the tallyDefinitions header file which contains examples of how to define tallies at compile-time.

Run-time tally specification

Tallies are specified at run-time using an XML input file. Currently, the options one can specify on a tally include

- Type: collision, tracklength, KDE, FET, sensitivity
- Dimension: 1, 2, or 3
- Filters: none, or a list of filter numbers
- Precision: single or double
- Architecture: CPU or GPU
- Scores: flux, dose, fission, nu-fission, absorption, total, scatter
- Tally-type specific entries: kde_options, fet_options, mesh, etc.

Currently the tally inputs are specified in a file is named "talon.xml" that exists in the directory where the transport code is being executed from. This could be altered to load in a file specified from the command line or specified within the transport code's input file. An example of an XML input file for a 2-D track-length mesh tally and a 2-D collision KDE with reigon based bandwidths is shown in Fig. 1.

```
<tallies>

  <tally id="1">
    <type> KDE </type>
    <dimension> 2 </dimension>
    <architecture> gpu </architecture>
    <precision> single </precision>
    <kde_options> 1 </kde_options>
    <scores> flux dose </scores>
  </tally>

  <kde_options id="1" use_optimal_bandwidths="true">
    <mesh_id> 1 </mesh_id>
    <region cells="1␣2␣3" > </region>
    <region cells="11␣22␣33" > </region>
    <global_lower_left>  −10 −10 −10 </global_lower_left>
    <global_upper_right> 10 10 10 </global_upper_right>
  </kde_options>

  <tally id="3">
    <type> tracklength </type>
    <mesh> 1 </mesh>
    <dimension> 2 </dimension>
    <architecture> cpu </architecture>
    <precision> double </precision>
    <scores> fission absorption total </scores>
  </tally>

  <mesh id="1" type="regular">
    <dimension> 1350 950 </dimension>
    <lower_left> −100 −100 </lower_left>
    <upper_right> 1250 850 </upper_right>
  </mesh>

</tallies>
```

Figure 1: Example of tally library input for a 2-D track-length mesh tally and a 2-D collision KDE with region-based bandwidths.

Filters

The tally library uses filters to prevent particles from contributing to a specific tally based on the state of the particle that generated the event. Currently, the tally library has five run-time specifiable filters. Users can filter particles by cell (or geometry node index for MCATK), energy, energy and cell, surface, or energy and surface. If a user needs additional filters, they can be created at compile time and applied to a tally with relative ease. Filters are specified at run-time via the <filters> XML tag, an example of which is shown for a collision tally in Figure 2. The tally in Figure 1 has a cell and energy filter, and an energy filter and surface filter are specified but unused. Users can also specify that no filters should be applied to a tally by excluding the <filters> tag from a tally. If so, that tally will skip the majority of logic associated with filtering samples when computing tally contributions.

```
<tallies>

  <tally id="2">
    <type> collision </type>
    <dimension> 1 </dimension>
    <architecture> cpu </architecture>
    <precision> double </precision>
    <scores> flux </scores>
    <filters> 2 3 </filters>
  </tally>

  <filter id="1" type="energy" bins="0. 10.0e6" />
  <filter id="2" type="energy" bins="0. 0.5e6 1.0e6 10.0e6" />
  <filter id="3" type="cell" bins="1 2" />
  <filter id="4" type="surface" bins="7" />

</tallies>
```

Figure 2: Example of tally library input for a collision tally with cell and energy filters.

KDEs

The KDE's accuracy is heavily influenced by the bandwidth, and thus this is the most important parameter to specify when defining a KDE. For problems with multi-modal distributions it is best to define bandwidths on a region-by-region basis. Additionally, bandwidths can be automatically chosen for a given region by computing an "optimal" bandwidth via the normal reference rule [20]. KDE regions are defined by listing the cell numbers belonging to each KDE region, with each region being a separate element (or child) within kde_options XML node as shown in Fig. 1. The tally library can compute optimal bandwidths automatically for KDE regions specified by the user, or it can compute one global optimal bandwidth for the entire problem. Furthermore, users can specify a global bandwidth that should be used instead, or they can specify bandwidths for each region defined in the problem. All of these options are handled via the kde_options node in the XML input file shown in Fig. 1.

FETs

FETs are specified in a similar manner to KDEs, with basic tally information provided in the tally XML node, and FET-specific information provided in an fet_options XML node. Current FET inputs include:

- lower_left and upper_right bounding regions for the FET which are used to normalize the function in each dimension.
- print_dimension - the resolution for printing point-wise estimates from the FET to a VTK file for visualization.
- coordinate_type - Cartesian, cylindrical or spherical. Currently only Cartesian is supported.
- n_expansion - the number of expansion coefficients to use.

```
<tallies>

 <tally id="4">
    <type> FET </type>
    <dimension> 1 </dimension>
    <architecture> cpu </architecture>
    <precision> single </precision>
    <fet_options> 1 </fet_options>
    <scores> flux </scores>
  </tally>

  <fet_options id="1">
    <lower_left> -10 -10 -1 </lower_left>
    <upper_right> 10  10  1 </upper_right>
    <print_dimension>100 100 100</print_dimension>
    <coordinate_type> cartesian </coordinate_type>
    <n_expansion> 4 </n_expansion>
  </fet_options>

</tallies>
```

Figure 3: Example of tally library input for a functional expansion tally.

Track-length tallies

Track-length tallies on a Cartesian mesh have been implemented, with tallies on a cylindrical and spherical mesh left for future work. In addition to the basic tally options, track-length tallies require the specification of a mesh id via a mesh node that determines which mesh to use for the tally. An example of a track-length tally on a Cartesian mesh is provided in Fig. 1.

Sensitivity Tallies

Currently, the library can compute sensitivities of and track-length tallies and surface tallies with cell and energy filters as well as FETs defined at runtime in the XML input file. The sensitivities of other tallies can be computed using compile-time defined tallies. The sensitivity tally is designed to enable the calculation of the sensitivity of any tally due to a perturbation defined using another generic tally. That is, the sensitivity of any tallyable quantity can be computed with relative ease using the tally library, as long as the perturbation of interest is material density or geometric sensitivities. Future work includes allowing the calculation of sensitivities to a wider range of parameters including nuclide atomic densities and cross sections.

Each sensitivity tally has its own sensitivity_options XML node to specify what sensitivities should be computed (geometric, FET, nuclide, etc.). Sensitivity tally runtime specifications are still a work-in-progress, but currently they are highly specialized for each sensitivity type. An example of the input for an energy-binned surface tally's sensitivity to a uniform expansion of a sphere using KDEs (per the method described in Ref. [21]) is given in Fig. 4.

```
<tallies>

  <tally id="5">
    <type> sensitivity </type>
    <sensitivity_options> 1 </sensitivity_options>
    <dimension> 1 </dimension>
    <architecture> cpu </architecture>
    <precision> double </precision>
    <filters> 2 1 </filters>
    <scores> flux </scores>
    <n_inactive_batches> 1 </n_inactive_batches>
  </tally>

  <filter id="1" type="surface" bins="7"/>
  <filter id="2" type="energy">
    <bins> 0.0 0.05 0.1 0.3 0.5 0.7 1.0 1.4 2.0 </bins>
  </filter>

  <sensitivity_options id="1">
    <sensitivity_type> geometry </sensitivity_type>
    <kde_options> 1 </kde_options>
  </sensitivity_options>

  <kde_options id="1"
     boundary_sensitivity="true"
     use_optimal_bandwidths="false"
     use_neighborhood = "false"
     tally_point_layout="surface">
    <origin> 0 0 0 </origin>
    <radius> 20.0 </radius>
    <global_inv_h> 0.5 </global_inv_h>
    <global_inv_h_cyl> 1 </global_inv_h_cyl>
    <global_lower_left>  -20 -20 -20 </global_lower_left>
    <global_upper_right> 20 20 20 </global_upper_right>
    <coordinate_type> spherical </coordinate_type>
  </kde_options>

</tallies>
```

Figure 4: Example of tally library input for computing the sensitivity to geometric perturbations.

III.B Tally Visualization

Tally visualization and plotting is done via ParaView by printing results to legacy VTK files [13]. ParaView is a fully-featured visualization tool capable of plotting 1-D, 2-D, and 3-D data with the ability to easily visualize time-dependent data sets. Furthermore, ParaView can view slices of 3-D datasets and it can adjust the opacity of datasets, enabling rapid visualization of results overlaid with highly-detailed models. This capability is demonstrated using a model of VIP man head and upper torso in MCNP [23]. VIP man is a representation of a phantom used in medical physics simulations consisting of 64 materials in a voxelized representation of the upper portion of a human down to the top of the lungs with each voxel consisting of 2 mm cubes and a total of approximately 1.36×10^6 voxels. The tally library was used to generate a 3-D histogram representation of the scalar photon flux in each voxel of the VIP man using 10^7 histories. The flux profiles in the center of the model are visualized using ParaView in Figures 5-6.

The voxelized representation of the material density is overlaid on the flux profiles in greyscale with a 20% opacity - providing a frame of reference for where the flux is concentrated within the phantom.

Figure 5: Flux from a 3-D track-length histogram at the center of VIP man with an overlay of material density in greyscale at 20% opacity.

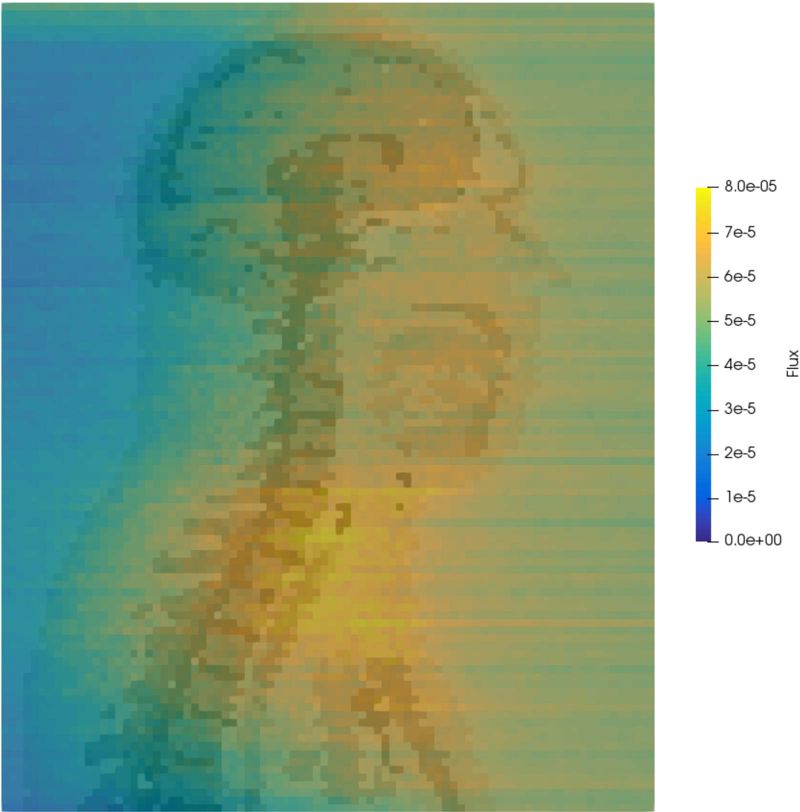

Figure 6: Profile-view of flux from a 3-D track-length histogram at the center of VIP man with an overlay of material density in greyscale at 20% opacity.

IV Tally Library Implementation

]The tally library is designed to be as minimally intrusive to the transport code as possible. As such, all tally-specific information is contained within a tally manager (TallyManager class). The tally manager is a singleton object that contains a list of all the tallies on the CPU and GPU and contains member functions for storing sample data, processing data, generating statistics, etc. Transport codes can interface with the tally manager directly or through a C-interface defined in interface.cpp. A list of the interface functions and their purpose is detailed in Table II. Future work includes developing a Fortran interface that is easier to call from Fortran-based transport codes.

The tally library is designed to collect sample data (collisions or particle tracks) from a transport code and process them in batches on either the CPU or GPU or both the CPU and GPU. A multiple of 32 samples (49152 in this work) are stored in each batch before they are processed. Whenever a tallyable particle event is generated, the transport code calls the storeCollisionData or storeTrackData function and passes in any necessary sample data (position, angle, energy, event type, etc.) for tallying. This sample data is stored in an array owned by the tally manager. Since samples are stored for later processing, cross sections associated with each sample also need to be stored for later processing. Currently, only macroscopic fission, nu-fission, absorption, elastic, and total cross sections associated with each material are stored for every sample. Future work includes allowing for nuclide-specific reaction rates and allowing for a greater variety and a variable number of cross section types to be stored with every sample.

Once the sample array is full, a deep copy is performed to send the sample data stored on the CPU to arrays located on the GPU and then the sample arrays are processed by tallies on their respective architectures. The tally manager contains vectors of tallies for each architecture. If the GPU is in use, the library begins filling a second set of sample arrays (events and cross sections) after processing all CPU tallies while the first set of arrays is being processed on the GPU. At the end of each generation, partially-filled sample arrays are processed and statistics are accumulated based on the sample contributions in the current generation.

The tally library is set up for MPI and OpenMP parallelism with acceleration on NVIDIA GPUs via CUDA. Each thread collects sample data in its own array and sends it to the GPU for later processing. The ParallelAssistant singleton contains details about the processor's rank, total number of ranks, total number of threads per rank, and how many histories the transport code expects the MPI rank to do. Currently the library assumes there is one GPU per compute node, but this assumption can be relaxed.

IV.A Tallies

Each tally type (KDE, tracklength, collision, FET, etc) is defined as a derived type from a base collision tally, which in turn is a derived from a pure virtual base class. The use of a virtual base class allows tallies of different types to be placed in the same container in the tally manager - thus allowing tallies of different types to be easily used in a single simulation. The virtual base class contains the functions defined in Table I, with each tally having the option to override or customize the base functionality inherited from the collision tally. The computeContributions function is customized for each tally and is the primary function that processes each sample list according to the tally type and what architecture the tally is defined for.

The overhead from using virtual functions is designed to be as minimal as possible. The tally functions are called at most once per sample list, or once per 49152 samples in this work. If the use of virtual functions becomes a performance bottleneck, which may be the case if it is desirable to process particle events as they are generated, then the use of virtual functions could potentially be replaced with variants from the C++ standard library in C++17.

Table I: Name and purpose of tally functions.

Virtual function name	Purpose
computeContributions	Computes contributions to the tally from a list of samples (particle events).
computeInactiveContributions	Computes contributions during inactive generations or batches for computing tally-specific quantities used in active generations or batches. Used to compute KDE optimal bandwidths and populate values needed to compute sensitivities.
initializeGeneration	Does necessary initializations for each tally at the start of every generation. For example, this calls KDE Optimal bandwidth calculation routines for the first active generation or batch. This also computes necessary quantities for sensitivity tallies. This does nothing for standard collision or track-length tallies.
accumulateTally	Accumulates contributions for a tally during inactive generations. Used to compute optimal bandwidths in the last inactive generation for KDEs.
computeTallyStatistics	Called at the end of a simulation, computes the batch-based statistics for every tally.

IV.A.1 Filters

Filters are used to bin results by cell index, energy bin, surface index, cell index and energy bin, etc. A filter simply takes a sample (or a sample list and sample index) and returns a bin index. This bin index is used to determine what index to increment in the tally results. A filter in the tally library could be over a single value like cell ID, or it could be over cell ID and energy. Thus, a filter in the tally library can be composed of a number of other filters. Filters are implemented for efficiency in the tally library. Filters are designed as a decorator pattern using generic programming. That is, each set of filters is self-contained and does not rely on virtual function calls or function pointers. Each filter type (energy, energy and cell, cell, etc) is a unique type in C++. Each tally class is templated on a filter type, and each tally contains a filter as a member variable. This is in contrast to using filters defined using a virtual base class or function pointers to filters contained elsewhere in the code. Having each tally own its filters means there is no virtual table lookup and the memory used by the filters is local to the tally. However, this method prohibits the use of caching filter results, which can give performance improvements if the same filter is used in several tallies. This no-cache design was chosen to be friendly for execution on the GPU, where each thread would need its own filter cache since each thread processes a unique particle event and different particle events are tallied simultaneously. The tally library has filters defined over cell, energy, cell and energy, surface, and surface and energy. Additional filters can be defined in the filterDefinitions.hh header file or tallyDefinitions.hh header file and used with compile-time defined tallies, or they can be added to the run-time definable filters by adding additional input parsing capabilities in the input cpp and header files.

IV.B Integration with Transport Codes

Transport codes can interact with the tally library via two options: by directly calling methods in the TallyManager singleton or by calling C-interface functions that then call methods in the TallyManager singleton. The first option allows for greater functionality since one can use classes and functions defined in the transport code and pass them into the tally library via function templates. The second method enables Fortran-based codes to use the tally library while also removing the compile-time dependencies the transport code has with regards to the tally library. This removal of dependencies enables faster compile times when editing C++ files for compile-time tally definitions since only the link phase will have to be redone - no files in the transport code will have to be recompiled. The interface functions and tally manager member functions share the same name and are listed in Table II with their corresponding purpose.

Table II: Name and purpose of interface functions called from the transport code.

function name(s)	Purpose
initializeTallies	Constructs tallies and sets values required to properly normalize tallies and communicate data across MPI ranks.
storeTrackInformation, storeCollisionInformation	Stores sample data within the tally library and sends the list of sample data for processing when the container is full.
initializeTallyGenerations	Performs tally-specific beginning-of-generation calculations. Used to compute optimal KDE bandwidths in the last inactive generation/batch, and computes and communicates any necessary information for computing sensitivities.
accumulateTallies	Accumulates the statistics for every tally - computes the sum and square of the sum for each tally result.
computeTallyStatistics	Computes the batch-based statistics for every tally and prints results - called at the end of a simulation.
bankTallyData	Banks tally data used for adjoint-weighting and differential operator sampling - called when a particle splits.
unbankTallyData	Retrieves tally data used for adjoint-weighting and differential operator sampling from the particle bank - called when a particle is removed from the bank for transport.

The library is designed to be as minimally intrusive as possible, however it still requires some information that is specified in the transport code. To that end, there exist two primary methods for transferring information from the transport code to the library. One method for C++ based transport codes is to pass function pointers to the tally library that tell the library how to look-up data like cross sections, cells, surfaces, etc. The other method is to define a set of C functions or Fortran functions with C-bindings that match the function definitions in the tally library for looking up relevant information. For example, cell-based filters require some way to match the cells specified in the tally library's input with the cells specified in the transport code's geometry. For Fortran codes, this is handled through a subroutine with C-bindings called "lookupCellIndex" that takes a cell id specified in the input and returns the cell's integer index that represents the cell within the transport code. Furthermore, KDE tally points need to know what cell they reside in for determining various KDE options. This is done through a function called "getCell" that takes a position vector and returns the cell index, material, and density associated with that point in space. A list of the interface function names and their purpose is given in Table III.

Table III: Name and purpose of C-interface functions called from the tally library.

C function name	Purpose
getCell	Given a position, returns the cell, material, and density associated with that position.
lookupCellIndex	Given the ID of a cell specified in the transport code's geometry input, returns the integer index used to refer to that cell within the transport code.
computeXS	Queries the transport code to look up the necessary cross sections for computing reaction rates specified in the tally library input.

IV.C Tallies on Heterogeneous Architectures

The tally library is designed to allow developers to write one tally algorithm implementation that can be executed on either the CPU or GPU. Since C++11 and CUDA 7.5, developers can more easily write algorithms that can be executed on both the CPU and GPU without having to explicitly use architecture-specific programming models like CUDA. Having one implementation that can be run on different architectures significantly reduces code duplication and enables rapid development of GPU-capable algorithms. While it is still true that performance benefits can

be obtained by tuning the algorithm explicitly for a given architecture, the ability to write architecture-agnostic code gives developers easy access to a base level of acceleration available on the GPU. Architecture-independent tally implementations are made possible through lambdas and templates in C++ and CUDA, managed memory in CUDA, policy-based class design, and tag dispatching. Tallies are templated with an Execution Policy, either CUDA_exec or CPU_exec, which is used to determine what architecture a tally should be executed on, how data should be allocated for use on that architecture, and for launching any architecture-specific functions.

Unfortunately, many modern C++ features and libraries like the STL are not supported within CUDA, so care must still be exercised such that algorithms can be executed on both the CPU and GPU. This lack of support stems from function calls requiring specific compiler macros (__device__) in order to be compiled for CUDA. For example, containers in the C++ standard library are not decorated with these macros, so member functions of containers in the C++ standard library like access operators cannot be called within algorithms intended for execution on the GPU. This is a significant limitation for writing new code intended for execution on heterogeneous systems as well as for adapting existing CPU-only code to GPU implementations as it often relies on methods in the standard library.

The tally library addresses this issue by using containers within Kokkos as a replacement for containers like vectors from the C++ standard template library [24]. In some cases the containers within Kokkos are insufficient or too cumbersome to use, and a modified version of the vector container from the standard library is used instead. The Tally Library's vector container uses GNU's C standard library and simply decorates the access operators as well as the size() operator with __host__ __ device__ compiler macros so that data contained within the vector can be operated on within device code without having to resort to using raw pointers. Otherwise, the tally library's vector is identical to that of the C++ standard library.

The tally library is designed to process samples lists on a per-sample basis on the GPU. Since each MPI rank or thread manages its own sample list, sample lists are processed serially on the CPU. This parallelization over samples on the GPU prohibits history-based statistics on the GPU, and it prohibits DOS-based sensitivity calculation. Future work includes developing a method for efficient history-based parallelization on the GPU.

When compiling for use on GPUs, the tally library requires a CUDA-aware MPI library. The CUDA-aware MPI library allows for data to be transferred directly between addresses residing on GPUs. If MPI is not CUDA-aware then data would have to be copied from the GPU to the CPU before the MPI operation could be performed on the data. After the MPI operation is complete, the data would potentially have to be transferred back to the GPU. Thus, using CUDA-aware MPI reduces communicaiton time and makes communication between GPUs easier to implement,

V Performance Results

The library is tested on a variety of problems ranging from 1-D slab to a quarter reactor core. CPU timing is obtained using a compute node with two Intel Xeon E5-2660 V3 processors and one NVIDIA GTX TitanX GPU with compute capability 5.3. Timing results are obtained using 18 or 20 MPI processes, as some problems ran faster with 18 MPI processes rather than 20 when the tally library was not in use. Furthermore, all timing results are obtained using the median of three separate runs. CPU timing is obtained with the library compiled for CPU-only execution as this improves the performance of the CPU tallies. This performance difference is due to the cost associated with initializing GPUs, even though they are not being used, when the library is compiled with GPU capabilities enabled as well as the cost associated with using CUDA lambdas for the parallel functors in the library. Presently, CUDA lambdas are cast to function pointers if they are used on the CPU if the library is compiled with CUDA enabled whereas they behave like normal lambdas and can be inlined when compiled for CPU-only execution.

V.A Track-length mesh tallies

3-D Mesh on 1-D Critical Slab

The performance of the tally library is demonstrated using a 3-D mesh tally on a 1-D critical slab eigenvalue problem in MCNP. Additional tallies and problems are not shown due to space constraints. A 1-D critical slab is chosen as a test problem to provide a performance benchmark using a simple continuous-energy transport process. The contrived problem consists of a 20 cm wide slab of U-235 at a density of 6 g/cm^3. Even though the transport process is relatively easy, computationally expensive tallies can be applied to this problem to demonstrate the library's performance. Since the GPU can process tallies while the CPU is conducting transport, a computationally expensive tally combined with a relatively easy transport process puts more burden on the GPU and thus gives a reasonable lower limit for the performance of the tally on the GPU.

Timing results are obtained using one compute node with two Intel Xeon E5-2660 V3 processors and one NVIDIA GTX TitanX GPU with compute capability 5.3. The code was compiled using GCC 5.4.0 and CUDA 8.0. The simulations are run with 5 inactive generations, 15 active generations, and 10^6 particles per generation. The runtime parameters were chosen to reduce the impact of MPI communication at the end of every generation, thus focusing on the performance of the tally algorithms. The results show the tally time - the increase in simulation time due to using the tally. The tally time is obtained by taking the difference of the average simulation time with and without tallies using five separate simulations for each average. CPU timing is obtained with the library compiled for CPU-only execution as this improves the performance of the CPU tallies. This performance difference is primarily due to the cost associated with using CUDA-capable lambdas in C++ for the parallel functors in the library. Presently, CUDA lambdas are cast to function pointers if they are used on the CPU if the library is compiled with CUDA enabled whereas they behave like normal lambdas and can be inlined when compiled for CPU-only execution.

Scaling with Mesh Size

Figure 7 shows the tally performance versus mesh size for a 3-D mesh tally obtained using single and double precision GPU tallies and double precision CPU tallies in the tally library along with the comparable tally in MCNP. Single precision tallies obtain approximately the same performance as double precision CPU tallies, but are consistently 1-2% slower than double precision tallies. The mesh size specified in Figure 7 indicates the number of mesh bins in each dimension, so the left-most data point uses one mesh bin and the right-most data point uses a total of 10^6 mesh bins

Runtimes for performance comparisons are obtained using 18 MPI processes on 20 compute cores due to an increase in runtime of approximately 7 s for the CPU-only calculations with and without tallies when going from 18 to 20 MPI processes. The average runtime from 5 different runs without tallies is 15.3 s for CPU-only compilation and 22.4 s for GPU + CPU compilation. The difference in runtime between CPU-only and CPU + GPU compilation when no tallies are specified is due to the time required for each MPI process to initialize the GPU, which is required even when no GPU tallies are specified. Typically, Monte Carlo simulations take on the order of minutes to days, so ignoring the extra 7 seconds of initialization time when using the GPU is justified. For this paper, all floating point sample data is stored on the CPU and GPU in double precision, but tallies are allowed to be computed in either single or double precision.

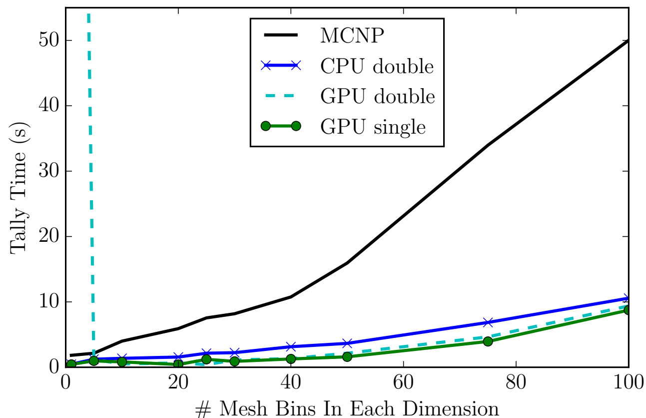

Figure 7: Tally-only time versus mesh size for 3-D mesh (histogram) tallies using the tally library and MCNP.

Figure 7 shows that the tally library out-performs the mesh tally in MCNP for all mesh sizes, with the exception of double-precision GPU tallies for small mesh sizes with less than 25 bins. Furthermore, the GPU tallies outperform the CPU tallies for all meshes with 5 or more bins in each dimension. However, the GPU tally's relative performance versus the CPU tally begins to diminish after 75 bins in each dimension. The reduced performance of the GPU tally versus the CPU tally with increasing mesh bins is due to the GPU spending the majority of its time looking up and writing to the memory addresses in the tally results array. These read/write operations cannot be coalesced due to the random nature of a particle track's location and therefore the random nature of the memory locations of the results that the particle tracks contribute scores to. Thus, it is likely the case that these read/writes to the tally results array are entirely uncoalesced, requiring 32 read/write operations per 32 contributions. An ideal algorithm on the GPU has fully coalesced read/write patterns, requiring only one read/write operation from global memory per 32 contributions. Even though this algorithm's memory access pattern is not ideal for the GPU it still performs reasonably well, reducing tally times by 1-2 seconds yielding a speedup of approximately 2 for small meshes and 1.2 for large meshes. This gain in performance would improve with a more computationally intensive transport problem where the CPU and GPU calculations would have more overlap.

Another feature seen in Fig. 7 is the poor performance of the double precision GPU tally for meshes with less than 100 bins. This is due to the lack of direct support for double precision atomic add operations on GPUs with compute capabilities less than 6.0. For GPUs with compute capability less than 6.0, double precision atomic additions have to be implemented as an atomic compare and swap, resulting in significantly worse performance. As the mesh size increases there are fewer read/write conflicts between threads, thus showing improved performance as the number of bins increases. However, eventually the double precision tally encounters the same issue as the single precision GPU tally where the read/write operations become entirely uncoalesced due to the random nature of particle tracks on the mesh, and the performance is limited by the time required to access global memory. Thus, the double precision GPU and single precision GPU tallies have similar performance for large mesh sizes.

Scaling with MPI Processes

A scaling study of performance vs. number of MPI processes was done for the GPU and CPU tallies from the library, with results shown in Fig. 8. The scaling study uses a $100 \times 100 \times 100$ mesh with 3, 5, 10, 15, and 18 MPI processes. Since MCNP uses a master-slave algorithm, the number of MPI ranks used to transport particles and compute tallies is one less than the number of MPI processes requested in the simulation. Ideal speedup is achieved when both the CPU and GPU are running at 100% - when there is a perfect balance between work the CPU is doing to generate sample data and the work the GPU is doing to compute tallies using that sample data. Thus, every additional MPI process that is added has the potential to offset this balance. For constant runtime parameters

(particles per generation, number of generations), the GPU is doing a constant amount of work regardless of the number of MPI processes specified, with the exception of additional communication and context switching between the MPI processes. Thus, every additional MPI process used in the simulation will proportionally reduce the amount of work the CPUs have to do while the amount of work the GPU has to do remains relatively constant. This has the potential to offset the balance of work between the CPUs and GPU, reducing the speedup obtained from using the GPU as seen in Fig. 8.

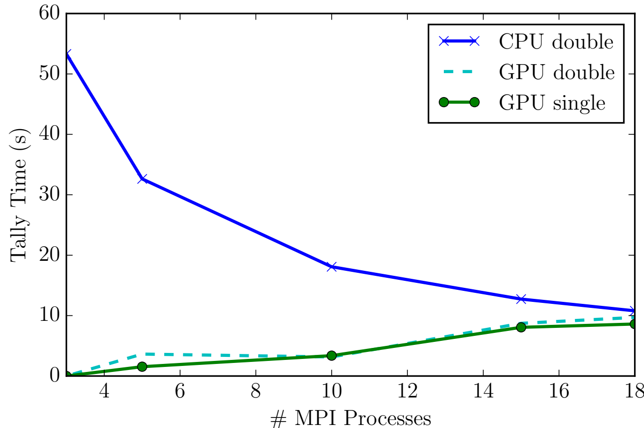

Figure 8: Tally time versus number of MPI processes for a $100 \times 100 \times 100$ mesh tally using the tally library.

Fig. 8 shows that for 3 MPI processes the GPU tally is free (within the resolution of the timing setup). However, as the number of MPI process increases the work the GPU has to do relative to the CPU increases, eventually making the tally process on the GPU only 15% faster than the tally on the CPU. Even so, if only one CPU chip (10 cores in this system) is used per GPU, a tally speedup of 6.5, or a runtime reduction of 15 s, is obtained. This is a reasonable metric as high performance computing clusters are moving towards multiple GPUs/node, with the Summit HPC machine at Oak Ridge National Laboratory containing six NVIDIA V100 GPUs and two 22-core IBM Power9 CPUs per node with 598 GB of CPU memory and 1600 GB of GPU memory and early-access version of Lawrence Livermore National Laboratory's Sierra high performance computing system containing 4 NVIDIA Pascal GPUs and 20 Power 8 cores per node [25].

Verification

The accuracy of the method and the calculation of uncertainties using batch statistics is verified by comparison to MCNP using an identical estimator. The problem setup is the same as the previous section, consisting of a 20 cm wide slab of U-235 at a density of 6 g/cm^3. Two eigenvalue simluations were run, one with 10^6 particles per generation and 15 active generations, and one with 10^5 particles per generation with 195 active generations. Results are obtained using a 1-D mesh with 10 mesh bins across the width of the slab. For both simulations the tally library produces identical averages compared to MCNP's native fmesh tally, with results agreeing within the reported precision from the meshtal output files. However, the use of batch statistics leads to differences within the reported statistical uncertainties. For both calculations the average relative 1σ uncertainty is on the order of 4×10^{-4}, however the batch statistics yield uncertainties that vary more than the history-based statistics. Table IV shows the ratio of the batch-based relative uncertainty to the history-based relative uncertainty for 15 and 195 active generations for each of the 10 mesh bins.

Table IV: Comparison of batch-based and history-based uncertainties for a 1-D mesh.

Bin #	$\frac{\sigma_{B,15}}{\sigma_{H,15}}$	$\frac{\sigma_{B,195}}{\sigma_{H,195}}$
1	1.26	1.13
2	1.01	1.03
3	0.81	1.01
4	0.88	1.03
5	1.07	0.98
6	1.52	0.97
7	1.28	0.93
8	0.88	0.99
9	0.98	1.04
10	1.17	0.98

Table IV shows that increasing the number of active batches reduces the variance of the variance, with 195 active generations giving better agreement with the history-based uncertainties. Nonetheless, the batch-based uncertainties agree within 30% of the history-based uncertainties for 15 active generations and within 15% when using 195 active generations. This disagreement could potentially be reduced by using more active generations or by subdividing each eigenvalue generation into multiple batches, thus effectively increasing the number of samples used to compute uncertainties.

VI Conclusion

While still a work-in-progress, the tally library has a large suite of features implemented including collision, track-length, KDE, FET, and sensitivities of k_{eff}, reaction rate ratios, and kinetics parameters to material density perturbations well as system dimension perturbations, as well as filters over cell, energy, and surface. The tally library can compute scalar flux, dose, leakage, and fission, absorption, nu-fission, scatter, and total reaction rates. The library is capable of computing all tallies on CPUs and GPUs, with the exception of sensitivities to reaction rate ratios and kinetics parameters due to the GPU's parallelization over samples rather than over individual histories. The library is capable of interfacing with parallel transport codes using MPI and OpenMP threading. The library has been used with Fortran and C++ based transport codes with results showing that the library is efficient at computing tallies using CPUs and GPUs, with CPU track-length tallies being more efficient than tally implementations available in the transport codes tested in this paper.

In its current form, the library lacks the capability to compute nuclide-specific reaction rates and sensitivities and its current suite of reaction rates that it can calculate is limited. This stems from the requirement of storing cross sections per particle event for later calculation in batch. The addition of the capability to compute cross sections on the GPU would reduce this memory storage burden at a cost of increased calculation time due to the need to recompute cross sections on the fly. Work is currently underway to alleviate this burden on the CPU by allowing samples to be tallied as they are generated rather than storing them in batches for later processing, thus enabling the direct use of cross sections that are already computed for use in the transport calculation to compute reaction rates.

Future Work

Future work includes enabling the calculation of nuclide-specific reaction rates and sensitivities. The development of sensitivities to nuclide-specific quantities requires a differential operator sampling method that uses a sparse storage scheme and is capable of handling histories that undergo particle splitting. Additionally, the use of GPU-specific algorithms that leverage GPU-specific memory options like constant memory, shared memory, and texture memory

will be investigated to improve the performance of computationally intensive tallies like KDEs. There are also many more capabilities that could be implemented in the tally library, including cylindrical and spherical meshes for track-length tallies, different polynomials for FETs, and history-based statistics, to name a few.

References

[1] F. B. BROWN and W. R. MARTIN, "Monte Carlo Methods for Radiation Transport Analysis on Vector Computers," *Progress in Nuclear Energy*, **14**, 269–299 (1984).

[2] S. P. HAMILTON, T. M. EVANS, and S. R. SLATTERY, "GPU Acceleration of History-Based Multigroup Monte Carlo," in "Trans. Am. Nucl. Soc.", **115** (2016).

[3] R. C. BLEILE et al., "Investigation of Portable Event-Based Monte Carlo Transport Using the NVIDIA Thrust Library," in "Trans. Am. Nucl. Soc.", **114** (2016).

[4] X. G. XU et al., "ARCHER, a new Monte Carlo Software Tool for Emerging Heterogeneous Computing Environments," *Annals of Nuclear Energy*, **82**, 2 – 9 (2014).

[5] R. M. BERGMANN et al., "Performance and accuracy of criticality calculations performed using WARP - A framework for continuous energy Monte Carlo neutron transport in general 3D geometries on GPUs," *Annals of Nuclear Energy*, **103**, 334–349 (2017).

[6] T. P. BURKE et al., "Acceleration of Monte Carlo Methods on Heterogeneous CPU-GPU Platforms Using Kernel Density Estimators," in "M&C," Jeju, Korea (April 16-20 2017).

[7] K. L. BOSSLER, "Performance of Kernel Density Estimated Mesh Tallies on GPUs," in "M&C," Jeju, Korea (April 16-20 2017).

[8] J. E. SWEEZY, "A Monte Carlo Volumetric-Ray-Casting Estimator for Global Fluence Tallies on GPUs," Los Alamos National Laboratory, LA-UR-17-22573 (2017).

[9] T. GOORLEY et al., "Initial MCNP6 Release Overview," *Nucl. Technol.*, **180**, 298 (2012).

[10] P. K. ROMANO et al., "OpenMC: A State-of-the-Art Monte Carlo Code for Research and Development," *Ann. Nucl. Energy*, **82**, 90–97 (2015).

[11] T. ADAMS et al., "Monte Carlo Application ToolKit (MCATK)," *Annals of Nuclear Energy*, **82**, 41 – 47 (2015).

[12] T. P. BURKE and F. B. BROWN, "Development of a Library for Conducting Monte Carlo Tallies on Heterogeneous Systems," in "Trans. Am. Nucl. Soc.", **119** (2018).

[13] J. AHRENS, G. BERK, and C. LAW, *ParaView: An End-User Tool for Large Data Visualization*, Visualization Handbook, Elsevier (2005), ISBN-13: 978-0123875822.

[14] M. ELLIS, D. GASTON, B. FORGET, and K. SMITH, "Preliminary Coupling of the Monte Carlo Code OpenMC and the Multiphysics Object-Oriented Simulation Environment for Analyzing Doppler Feedback in Monte Carlo Simulations," *Nuclear Science and Engineering*, **185**, *1*, 184–193 (2017).

[15] D. P. GRIESHEIMER, W. R. MARTIN, and J. P. HOLLOWAY, "Convergence properties of Monte Carlo functional expansion tallies," *Journal of Computational Physics*, **211**, *1*, 129 – 153 (2006).

[16] D. P. GRIESHEIMER, W. R. MARTIN, and J. P. HOLLOWAY, "Estimation of flux distributions with Monte Carlo functional expansion tallies," *Radiation Protection Dosimetry*, **115**, *1-4*, 428–432 (2005).

[17] K. BANERJEE and W. R. MARTIN, "Kernel Density Estimation Method for Monte Carlo Global Flux Tallies," *Nucl. Sci. Eng.*, **170**, 234–250 (2012).

[18] K. L. DUNN, "Monte Carlo Mesh Tallies based on a Kernel Density Estimator Approach," Ph.D. Thesis, University of Wisconsin–Madison (2014).

[19] T. P. BURKE, B. KIEDROWSKI, and W. MARTIN, "Kernel Density Estimation of Reaction Rates in Neutron Transport Simulations of Nuclear Reactors," *Nucl. Sci. Eng.* (2017).

[20] B. W. SILVERMAN, *Density Estimation for Statistics and Data Analysis*, Chapman and Hall, London, UK (1986).

[21] T. P. BURKE and B. C. KIEDROWSKI, "Monte Carlo Perturbation Theory Estimates of Sensitivities to System Dimensions," *Nucl. Sci. Eng.*, **189**, 199–223 (2017).

[22] T. P. BURKE and B. C. KIEDROWSKI, "Monte Carlo Estimates of Sensitivities to Geometric Perturbations for Shielding Problems," in "ANS RPSD," Santa Fe, NM (August 26-31 2018).

[23] T. GOORLEY, "MCNP Medical Physics Geometry Database," Los Alamos Unclassified Report, LA-UR-08-2468 (2008).

[24] H. C. EDWARDS, C. R. TROTT, and D. SUNDERLAND, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, **74**, *12*, 3202 – 3216 (2014).

[25] "Sierra," Https://computation.llnl.gov/computers/sierra (2018).