LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-TR-781441

Thin-Threads: An Approach for History-Based Monte Carlo on GPUs

R. C. Bleile, P. S. Brantley, H. H. Childs

July 15, 2019

Disclaimer

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

Thin-Threads: An Approach for History-Based Monte Carlo on GPUs

Ryan Bleile*†, Patrick Brantley*, David Richards*, Shawn Dawson*,
Michael Scott McKinley*, Matthew O'Brien*, Hank Childs†
* Lawrence Livermore National Laboratory
Livermore, CA 94550
† University of Oregon
Eugene, OR 97403

***Abstract*—A graphics processing unit (GPU) has become a core technology for modern supercomputers. Applications that once ran on supercomputers are being forced to make significant changes to their designs to utilize these new machines. This paper introduces the concept of Thin-Threads as a method for history-based Monte Carlo transport applications on GPUs. The key principles behind Thin-Threads are light memory usage and communication and managing data race issues via atomics. We show that we can achieve a 10x speedup when moving from the traditional method to Thin-Threads on GPUs. Additionally, we demonstrate the viability of the Thin-Threads model at scale for GPU and CPU platforms.**

***Index Terms*—GPU, Monte Carlo, particle transport**

I. Introduction

Particle transport problems describe the ways in which particles move through and interact with materials or structures. In a real world system, attempting to compute every individual particle in order to understand aggregate effects is not computationally feasible. Instead, the Monte Carlo method is used to solve these problems by aggregating particle behaviors through statistical sampling and particle weighting. Specifically, Monte Carlo methods use pseudo-random numbers to sample from probability distributions, which are used to model the likelihoods of various events that may occur when particles collide with atoms in a material.

In a Monte Carlo transport problem, each statistical particle represents some number of physical particles, thus allowing the computation to be bounded by available computational resources. Given the inherent statistical nature of this problem, running more particles increases the fidelity of a simulation and reduces statistical uncertainties. High-performance computers are often used to boost the number of computations that can be performed to solve larger scale problems and/or achieve higher fidelity solutions.

Trends in high-performance computing are affecting best practices for Monte Carlo particle transport problems. Supercomputers are increasingly more powerful, enabling greater fidelity when performing a simulation. However, the nodes of these supercomputers increasingly contain more cores, with each individual core being less powerful. Example architectures that reflect this trend are the Intel Xeon Phi and NVIDIA GPUs.

In this paper we introduce Thin-Threads (defined in Section III-B), a new threading approach for Monte Carlo particle transport problems. While elements of Thin-Threads have appeared in previous research, our contribution lies in combining these elements, providing a thorough description of implementation, and evaluating its efficacy. Additionally, we look at new methods for overlapping computation and communication using Thin-Threads. Finally, we show that the Thin-Threads approach is capable of outperforming the traditional "Fat-Threads" (defined in Section III-A) approach, up to three times faster on CPUs and ten times faster on GPUs for certain workloads.

II. Related Work

Monte Carlo transport methods can be used to solve a variety of problems. Each of these problems introduces different complications that need to be addressed to efficiently execute on the GPU. This section will highlight the variety of the work done in this area, by explaining the primary objectives of the applications used for each study.

The work related to this paper was conducted in the Quicksilver proxy application [1]. Quicksilver solves the Monte Carlo particle transport problem by using distributed particle streaming and a multigroup energy nuclear data energy representation. Quicksilver originally implemented threading through a Fat-Threads model, described in Section III-A. An initial implementation of the Thin-Threads model was added to Quicksilver in order to provide a feasible method for GPU computing. A discussion of the process that led to Thin-Threads as well as the key features of the OpenMP 4.5 and CUDA implementations are presented by Richards et. al [16].

Quicksilver is a proxy application of the full production code, Mercury [2]. Mercury uses distributed memory particle streaming as well as domain replication to scale across nodes. Additionally, it also uses both continuous energy and multigroup energy cross sections. Mercury implemented threads using OpenMP and the Fat-Threads threading model [9]. Recently, Mercury has implemented the Thin-Threads threading model discussed in this paper. Quicksilver was originally developed to model Mercury's call tree and memory usage patterns for streaming multigroup problems.

ALPS MC [7] is a one dimensional Monte Carlo transport research application. ALPS MC solves a 1-D problem in a binary stochastic medium, using simple physics. It was originally written as a serial application and is only useful as a model for intra-node performance. ALPS MC implemented the simplest form of the Thin-Threads approach to gain a threading model that was capable of running on GPUs. The research focused on the choice of particle tracking algorithms, comparing history-based and event-based approaches as well as optimizing the stages of these algorithms [8].

XSBench [20] is a proxy application for OpenMC [17]. XSBench specifically implemented the continuous energy nuclear data lookup portion of a Monte Carlo transport problem. Initial testing discovered that a significant portion, upwards of 80% of the total runtime, of OpenMC was spent performing continuous energy lookups. In response, an application for solving that specific issue was developed to understand the impact of different design choices on that model.

Profugus [4] is a multigroup proxy application designed to represent Shift MC [14]. Profugus solves the criticality eigenvalue problem using a multigroup energy spectrum. In a recent paper [11], Hamilton et. al. presented data comparing history- and event-based

tracking algorithms on GPUs. This discussion focused on intra-node performance and techniques to improve the history- and event-based algorithms. A simple Thin-Threads model is used, modeled after the approach discussed in ALPS MC. This work did not include a streaming particles feature, so MPI scaling was accomplished through domain replication. Using domain replication, particles only need to be sent to other ranks during load balancing.

Other groups have looked into Monte Carlo transport on GPUs as well. In a 2014 thesis, Bergmann discusses the efforts made to WARP for continuous energy Monte Carlo on GPUs [6]. This work focused on utilizing GPUs for solving the internal ray-tracing problem on GPUs by using the NVIDIA ray tracing library OptiX [12]. Some event-based tacking was added to facilitate the ray-tracing as a unique element of the simulation. Another group developed separate versions of their Monte Carlo package (ARCHER [21]) for each hardware platform, including NVIDIA GPUs, the Intel Xeon Phi, and CPU systems. ARCHER solves the photon transport problem primarily but also solves eigenvalue problems for neutron transport. This work primarily focused on intra-node performance and challenges general to making Monte Carlo GPU compatible.

Monte Carlo is a broad class of algorithms encompassing many fields of study. Outside the realm of particle transport algorithms, other groups have looked into GPUs for many of these methods as well. In a work by Alawneh et al. [5], Monte Carlo methods are used to compute the sea ice loads using a design based on offloading many small kernels to the GPU to perform the compute heavy workloads. Work by Szalkowski at al. [19] focused on the distributed parallel generation of random numbers for multidimensional integration, while Stpiczynski et al. [18] emphasized vectorized approaches for CPUs, GPUs, and MIC architectures for this same problem. Monte Carlo algorithms are all similar in that they use random numbers to accomplish a task. The key difference of these works from our own is the size of the kernels used/needed to accomplish those tasks. In Monte Carlo integration, generating adequate random numbers can be a time consuming process that dominates the compute time. For particle transport algorithms, the time spent generating random numbers can often be ignored as it is a minor element in the full algorithm. Many applications take advantage of the high throughput and compute capabilities of the GPUs in a way that particle transport algorithms cannot. Even with these differences, the innovations made by these groups provide lessons that we can learn from and potentially adopt for our own work.

This paper expands upon the previous work of each of these groups. Every group implemented concepts that share similarities with the Thin-Threads model discussed here. By explicitly defining this threading model we can facilitate more discussion on this topic. Additionally, this provides a minimum yet efficient starting point for others trying to begin working on Monte Carlo transport problems on GPUs.

III. Threading Models

To solve Monte Carlo particle transport problems, millions to billions of particles need to be processed. Parallel computing is necessary to process this number of particles in a reasonable amount of time. Monte Carlo particle transport problems are embarrassingly parallel, since the unit of work — a particle — is completely independent of all others. As supercomputer architectures have shifted to increased parallelism within a node, adding parallelization through threading has become increasingly common and necessary.

There are two major approaches to solving Monte Carlo particle transport problems: history-based and event-based. The work presented in this paper applies the Thin-Threads threading model to the history-based Monte Carlo transport problem. With the history-based tracking algorithm, individual particle histories are tracked until a predetermined amount of particles has been simulated. These particles are processed one at a time, until there are no more particles left to process. For the event-based tracking algorithm, particles are continually regrouped by the event they will process next. With this algorithm, each event group is processed in parallel before needing to regroup particles again.

History-based Monte Carlo particle transport applications generally divide work into three distinct sections: cycle initialize, cycle tracking, and cycle finalize. These three sections are described in pseudocode in Figure 1. Cycle initialize and cycle finalize are both relatively small and straight forward. Cycle initialize handles setting up inputs, such as sourcing particles, and doing variance reduction calculations. Cycle finalize handles reducing output data, such as tallies collected during tracking. Cycle tracking is the core of the code, containing the large majority of the functionality and physics. The work done during cycle tracking is almost entirely contained within a loop over particles. Inside the loop, each particle computes which event it will do next, via sampling probability distributions and using random numbers to make decisions. Then the particle executes its given event, e.g., moving through the mesh, colliding with the background material, etc. Particles continue to do this two-step process — compute distances then apply the nearest event — until they reach an end condition, such as absorption, or census.

```
cycle_init() {
  source in particles
  population control
}

cycle_tracking() {
  for all particles {
    do {
      compute distance to census
      compute distance to facet
      compute distance to reaction
      do segment with shortest distance
      increment tallies
    } until census, absorbed, escaped
  }
}

cycle_finalize() {
  reduce all tallies
}
```

Fig. 1: Pseudocode for the three major phases of a history-based Monte Carlo transport code.

Parallelization usually occurs over the “for all particles” loop in cycle_tracking(). Traditionally, particles are split across threads in groups, providing each thread with its own unique chunk of work to complete. We refer to this as the “Fat-Threads” approach, which we describe in more detail in Section III-A. An alternative approach is for threads to share a chunk of particles, with each thread operating on a single particle within the collection of particles. We refer to this as the Thin-Threads approach, which we describe in Section III-B.

A. Traditional Fat-Threads Approach

*1) **Overview**:*

A Fat-Threads threading model is one where all potential data races are handled through replication of data structures. This allows each thread to work completely independently of one another. Each thread is assigned its own collection of particles to work on, and all output tally and buffer type data structures are replicated. Replicating tally data can be non-trivial, as tally data structures exist in multiple forms: tallies for a single value over the whole problem, tallies for each element in the problem, and tallies for each material in the problem. Each of these tallies requires different amounts of memory to store their data. This model, in combination with a load balancing algorithm, was shown to scale well on CPU platforms in production applications [13].

Quicksilver implements Fat-Threads in a typical fashion. Its fundamental unit of work is advancing a particle, its primary data element is the particle, and its data structure for a particle contains roughly 200 bytes of information. Particles are stored in "particle vaults," which is a container class for grouping particles together and defining functions on sets of particles. At the highest level in the data structure, there is a "particle vault container" (PVC) that can hold a changing number of particle vaults, as well as shared data between vaults. Finally, each rank is given a PVC to organize its workload, and each thread associated with that rank is then given a particle vault from the PVC.

In Monte Carlo transport problems, distributed-memory parallelism is commonly used to split up large geometries into separate domains across ranks. Separate geometric domains adds the need for particles to be communicated across ranks as they move through the geometry. In the Fat-Threads model, particles are communicated asynchronously across ranks when needed by the cycle tracking function. When a rank runs out of particle vaults to give to threads, that rank can receive a buffer of particles from another rank, and then fill up new particle vaults, continuing the current cycle. In addition, threads can perform the send and receives themselves as they fill buffers or need more work.

This model for running particles on ranks and threads works well on CPU platforms, by maintaining data locality in a thread and removing the need to deal with data races between threads. The communication cost of sending particles to different ranks is almost completely masked by the computation of particles on each rank, since the computation of particles on each rank occurs while particles are in flight. Additionally, particle vaults become an obvious organization structure for dealing with load balance, providing a flexible infrastructure for running threads.

*2) **Barriers on GPUs**:*

There are two primary concerns with the Fat-Threads model — memory footprint and communication from accelerators.

With respect to memory footprint, the issue is that the Fat-Threads model is likely to use too much memory on GPU devices. When switching from a CPU to GPU platform, the number of threads per rank goes from tens of threads (at most) to thousands of threads or more. If data structures continued to be replicated in the same manner on a GPU platform, providing each GPU thread with its own data structures to read from or write to, then available memory would quickly run out. This is a concern even if a code extends GPU memory via paging in memory from the host. Even given infinite access to host memory, GPU architectures would struggle from a complete lack of coalesced memory access and a need to constantly page-in data, resulting in an inability to get acceptable performance.

With respect to communication from accelerators, the fundamental issue is the lack of MPI functionality from a GPU device. The GPU cannot make the same MPI function calls that a CPU can during particle tracking. This is particularly problematic for the Fat-Threads model, since it relies heavily on the use of asynchronous communication to move particles from rank to rank while computation is being done. Since the MPI calls cannot be made while processing particles, new methods for communicating particles across boundaries must be investigated.

Between these two issues, the Fat-Threads model appears to be incongruent with GPU architectures.

B. Thin-Threads

*1) **Overview**:*

Thin-Threads have multiple beneficial properties for history-based Monte Carlo on GPUs. First, Thin-Threads are threads that are light on memory usage and communication. Second, Thin-Threads handle all potential data races directly, primarily through use of atomics. This model allows for a larger number of threads to be callable at once, reducing the memory footprint when threading. Threads primarily work independently, although there is some interaction via their shared atomic operations. Third, Thin-Threads do not access MPI or other forms of inter-node communication directly. Instead, Thin-Threads employ a batching and asynchronous communication model.

Overall, Thin-Threads adapt to modern HPC architectures, in that:

- They are lightweight, in order to match decreases in single thread performance.
- Their communication management is aligned with current restrictions (i.e., MPI communication is not possible, or it is possible but not performant).
- Its design accounts for the currently popular use of accelerators, specifically in achieving overlap in communication and computation.

Figure 2 outlines pseudocode for a new cycle tracking function for the Thin-Threads approach.

```
cycle_tracking() {
  while( !done ){
    for each batch {
      Do Kernel
      Do MPI Send
      Do MPI Receive
      Clean Extra Vaults
    }
    test for done
    if( !done )
      Collapse Vaults
  }
}
```

Fig. 2: Pseudocode for batching control flow in the Thin-Threads approach. Do Kernel refers to launching the cycle tracking kernel. Clean Extra Vaults refers to the process of ensuring there is adequate space for the next kernel launch. Collapse Vaults refers to the process of reducing the particles in the particle vault container into the minimum number of vaults required to contain them.

While the basic concept of Thin-Threads is relatively straightforward, it requires significant attention to detail in implementation.

The implementation details are described in depth in the following sections.

*2) **Basic Implementation Details**:*

There are two primary tasks required to implement the Thin-Threads model. The first task is to make the tracking loop thread-safe. This requires adding atomics for writing to output tally data and modifying the particle container data structure to allow for threaded reading and writing. The second task is to remove all MPI from within the tracking loop. This requires adding a replacement MPI model after the tracking loop, as well as additional MPI buffers that get filled during the tracking loop. This MPI model is asynchronous and provides the groundwork for a batching model.

*3) **Implementation Details - Batching Model**:*

We built the batch model around three key concepts. First, memory is allocated from the host side, since memory allocations on the GPU are typically slow and limited to device-only memory. Second, the number of particle vaults in the PVC must be capable of being changed dynamically, i.e., we can add particle vaults to a PVC if needed. The number of particles a single rank may see cannot be known in advance and so we must have a flexible system to allow for new particles to be added. Third, we cannot access the MPI region of the code from within the main body of the tracking loop. All MPI must be handled outside the main body of tracking, although we still need a way to handle particles that need to be communicated. Each of these three key concepts are discussed further in the remainder of this section.

In order to satisfy the first key concept, avoiding new data allocation on GPUs, we determined that the number of particles within a given particle vault needs to be fixed. This allows a particle vault to define the group of particles that will execute together in a kernel. A side effect of the fixed vault size is the need for an extra buffer for managing particles created during tracking, since we cannot know the actual number of particles any given cycle will produce. In order to guarantee there is enough space in the extra buffer, we pre-allocate enough particle vaults to handle the case where every particle undergoes the maximum production in a reaction. This can be determined through a heuristic calculation as long as any particle that produces new particles for computation is also added to the new particle list (i.e., its computation is postponed) to guarantee the size of the extra particle list is bounded. The extra vaults and postponing computation of a particle ensures that we will not need to allocate new data during the tracking kernel.

The second key concept, dynamically changing the number of particle vaults in a PVC, must work within the context of the first key concept, data allocations only from the host (i.e., not from the GPUs). To accomplish this, we designed a host-side data structure (the PVC, which is on the host only) that (1) can dynamically change sizes, and (2) always contains enough memory for each kernel on the device (through the particle vaults it contains). More details on specific data structure choices are explained in Section III-B4.

In order to satisfy the third key concept, no MPI communication during particle tracking, all MPI was removed from the tracking loop itself. Instead, when a particle leaves a given rank's domain during the tracking loop, it is placed in a buffer. After the tracking loop finishes, the host inspects this buffer and performs the appropriate communication. The size of this buffer has a clear upper bound, since it cannot exceed the fixed batch size in a single particle vault, i.e., the number of particles needing to be sent via MPI will not exceed the number of particles we are tracking in each batch. In terms of implementation, we create an index list of particles in the kernel which identifies the particles that need to be communicated

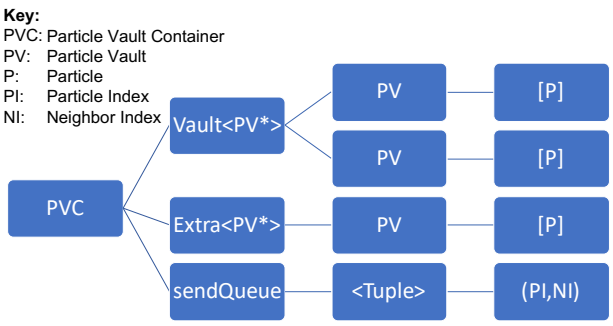


Fig. 3: A visual representation of the Particle Vault Container (PVC) data structure.

via MPI, as well as to which neighbor they need to be sent. This simple tuple of data can be generated quickly in the kernel, allowing for faster compute times, at the cost of needing to loop over the index of particles later, on the host, and copying them into MPI buffers. This method has so far not shown itself to be performance critical, spending orders of magnitude less time than the actual kernel compute times.

*4) **Implementation Details - Data Structures**:*

The previous subsection (III-B3 defined three key concepts for the Thin-Threads batch model. One of these key concepts enabled growth in the number of particles stored on a given rank. There are two reasons that particle growth on a rank can occur: through reactions in cycle tracking or through receiving particles via MPI communication.

When a new particle is created, it needs to be added into a particle vault. Of course, in our scheme, the GPU cannot allocate new memory. Our solution is to allocate extra particle vaults prior to executing the tracking kernel, and then have the kernel add new particles to these extra particle vaults as it executes. These new particles can then be considered for future processing. Therefore, the particle vault container must not only be dynamic in size, but also must allow direct access for passing in batches to kernels. We use a vector (from the C++ Standard Template Library) of particle vault pointers to handle these requirements. By making a vector of pointers, our particle vault container can change sizes through a two step process: allocating the pointer (built into the vector class) and then allocating the particle vaults to which the pointers point (custom allocation function). In addition, it allows us to re-organize the vaults in the container as necessary, such as swapping an empty vault for a filled one (which becomes as easy as swapping two pointers instead of needing to perform a deep copy). Figure 3 details the new structure for the particle vault container to enable this new work flow.

After each iteration of kernel launch followed by MPI communication, our algorithm cleans up the extra vaults and combines newly received and newly created particles. This process creates new batches for use in future iterations.

*5) **Implementation Details - Control Flow**:*

Figure 2, which appeared earlier in the Thin-Threads overview section, describes how the Thin-Threads model incorporates batching into the control flow. Its control flow allows for overlapping computation with communication, and lets us recover lost performance on CPUs, compared to Fat-Threads model. The Thin-Threads control flow works as follows. First, a vault is taken out of the particle vault container and sent into the Kernel (Do Kernel). Second, any particles that need to be sent are pulled into MPI buffers based on the values

in the send queue tuples, particles index in vault, and neighbor rank index (Do MPI Send). Third, the host checks to see Whether or not any particles need to be received (Do MPI Receive). Fourth, newly created particles and received particles are condensed into particle vaults that are then added to the PVC. The extra particle vaults are populated again with all empty vaults (Clean Extra Vaults). Once this process has been completed, the data structures are ready to handle another pass through this process, i.e. a filled vault is ready for kernel launches, and extra vaults are ready to receive new particles.

One significant element of this application is the need to run on the CPUs and GPUs through a single source code base. To achieve this, a simple execution policy model was established which allowed for ranks to determine what form the kernel would take. The amount of replicated code for each policy available was reduced to a single function call inside each kernel or for loop. This means that each policy only needs to define the parameters necessary to launch the kernel, or run the for loop. The available policies are Serial, OpenMP 2.0, OpenMP 4.5, and CUDA. The use of macros around language specific functions, such as atomics, allows each of these methods to run through the same code on CPUs as well as GPUs. Additionally, this execution policy model will be the basis of future work, where we can explore the use of CPUs and GPUs at the same time.

IV. Thin-Threads Performance Studies

This section describes the results from studies performed on Lawrence Livermore's IBM/Nvidia GPU test platform, Ray. This machine uses two IBM Power8 CPUs and four Nvidia Pascal P100 GPUs per node. The IBM Power8 CPU has 10 cores and can run up to 8 threads per core. That said, our best performance comes from running threaded CPU runs with four threads per core, and so we only use four of the eight threads in our experiments.

A Monte Carlo particle transport workload is defined by two major factors — the types of reactions and likelihood of mesh facet crossings. For the types of reactions, the key elements are the cross section and material information. For the likelihood of mesh-facet crossings, the key elements are the mesh layout and decomposition. While the elements defining the types of reaction are defined by the underlying physics, the elements defining the likelihood of mesh-facet crossings can be varied. Therefore, our performance study varies the elements behind mesh-facet crossings (mesh layout and decomposition).

For the material and cross section information, we considered the Godiva in water [10] problem. Specifically, we replicated the ratios of particle streaming to collisions, as well as the ratios of the types of reactions that occur in the collisions.

For the mesh-facet crossings, we defined the size of the mesh elements so that the likelihood of events is roughly equal (i.e., so the occurrences of mesh facet crossing and collision events are balanced). The problem defines a Cartesian mesh of 10x10x10 mesh elements per rank (one decomposition element) in a rectangular, doubling, scaling pattern. For example, one rank would use [10x10x10] mesh elements, where as two ranks would use [20x10x10] mesh elements, and four ranks would use [20x20x10] elements. Given the simplicity of running problems in a rank per GPU mode we opted to use four ranks for the base problem and define one node worth of performance as the result of running on four P100 Pascal GPUs. In order to maintain a fair comparison when running on CPUs, we opted to also use four ranks per node and use OpenMP threading to fully utilize a node. At four threads per core and five cores per rank, the CPU data was generated using four ranks with twenty threads per rank.

In terms of runtime per cycle, our goal was to pick workloads that reflected real world problems. On the one hand, runtimes that are very short would not reflect real world problems (and also skew analysis). On the other, long runtimes, while more common in practice, limit the number of tests we could perform. Overall, we decided to consider runtimes of approximately two seconds per cycle. To accomplish this, we opted to run one million particles per rank, which completes in roughly two seconds per cycle on a GPU. Given four ranks per node as our baseline, we ran four million particles per node and scale accordingly during scaling studies.

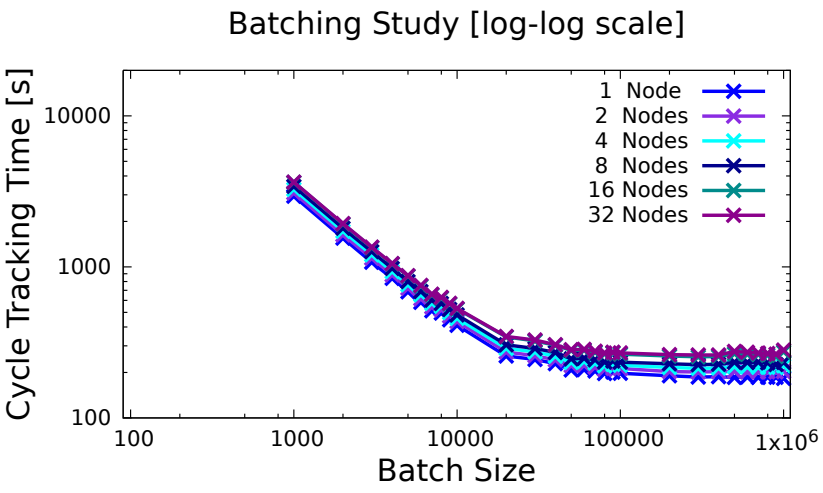
A. Effect of Batch Size on Performance

In this section, we analyze the effect batch size has on the overall performance for Thin-Threads. Batch size has multiple, potential impacts on performance. First, it determines the number of threads that can be running simultaneously on a rank, which has a profound impact on the performance of threading. Second, it allows for different amounts of computation to overlap with communication, providing a tunable knob for optimizing MPI. Finally, batch size choices also determine the number and size of memory allocations that need to occur, which should be minimized in this setting. The results for this section are plotted in Figures 4(a) and 4(b). In these figures, batch size is plotted on the x-axis and runtime in seconds on the y-axis; with respect to performance, lower is better. The experiments performed were somewhat asymmetric: our minimum batch size was 100 for the CPU and 1000 for the GPU. We had to increase the minimum batch size for the GPU, since batch sizes of 100 did not complete within a reasonable amount of time, due to not utilizing the GPU adequately.

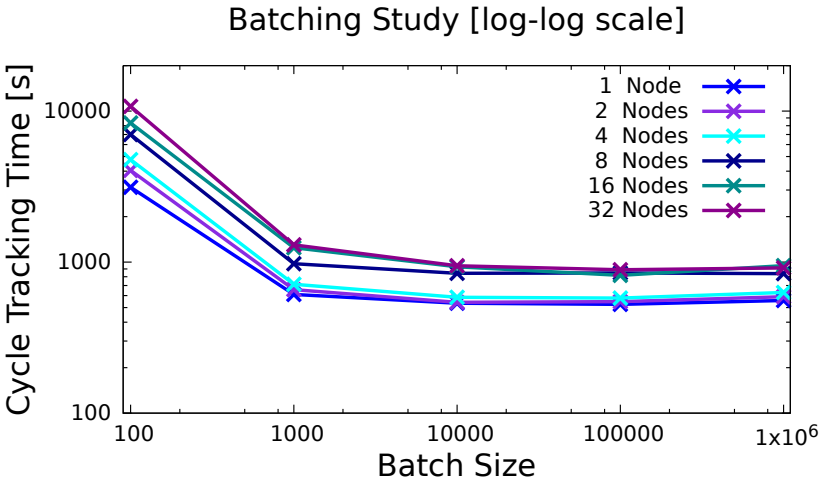
The effect of batch size on the availability of threads has profound performance implications. This is especially true on GPU architectures, as the batch size determines the kernel size of the particle vaults. Large kernels are needed to efficiently utilize all of the cores on GPU hardware. Figure 4(a) clearly shows the trend of increased performance (decreased runtime) as the batch size increases. The trend in runtime decreases linearly as we increase batch size, up until the GPU hardware is adequately saturated. Once GPU has enough work (at around a batch size of 50,000), the performance benefit plateaus. Batch sizes above 50,000 provide similar performance, reducing the need to find a specific value for optimum performance. At higher batch sizes (approaching one million), the curve trends up slightly, most likely due to there being less MPI overlap occurring in that regime.

Figure 4(b) shows the performance trends on CPU architectures at different scales. The trends for CPUs have a similar shape to GPUs. The primary difference between the two architectures is that the maximum performance (lowest runtime) point for CPUs occurs much earlier than it does for GPUs. Both sets of results show a decrease in performance (increase in runtime) as when batch sizes become much smaller than the total number of particles. For CPUs, our results consider batch sizes as small as 100. Increasing the batch size to 1000 results in almost an order of magnitude increase in performance.

Another interesting point is that this trend shows nearly identical performance at different scales, meaning that even at poor batch sizes for GPU performance the MPI weak scaling is still managing well. This is true on CPUs as well when running 4 ranks per node. In our previous experience, not shown here, we witnessed negative side effects to running with batch sizes that were too large when run at large scale (thousands of ranks).



(a) GPU



(b) CPU

Fig. 4: These plots show weak scaling studies of cycle tracking time versus batch size, for 1 to 32 Nodes. This data shows batch size has considerable impact on performance. For GPU runs (sub-figure (a)), the optimum batch size is 300,000 particles per batch. For CPU runs (sub-figure (b)), 100,000 particles per batch was the optimum size, although the performance differences for batch sizes over 1000 were much smaller. The most important takeaway from these plots are the trends across all nodes, rather than the line corresponding to a single configuration of nodes.

B. Weak Scaling Efficiency Comparisons

In this next phase of results, we consider two topics: weak scaling and comparison to Fat-Threads. Our experiments here incorporated the optimum batch sizes from the previous phase of results (Section IV-A).

Table I lists the results from a weak scaling study (1 node to 32), comparing the same configuration for Thin-Threads with GPUs, Thin-Threads with CPUs, and Fat-Threads on CPUs. The entries in each table are the actual runtimes. This table highlights the added benefits of the Thin-Threads model, especially at this scale, as even the CPU results show improvement over the original Fat-Threads model.

The data is most representative of real-world workloads at higher node counts. With low node counts, each node's domain has fewer neighbors, which means less time is spent doing communication. For example, with four nodes, each node's domain has only two neighbors. As the nodes counts get higher and higher, then most of the nodes will have six neighbors (+/-X, +/-Y, +/-Z). In particular, slowdowns in performance can be seen at 8 and 16 nodes, as nodes at these levels of concurrency have more neighbors than smaller concurrencies. Specifically, at 16 nodes and 64 ranks has ranks that needs to send and receive messages with up to six neighbors. We can see this effect on the weak scaling data, especially for the Thin-Threaded CPU code, as the complexity of the MPI increases the runtime increases to match but settles again at a new steady value.

Table I shows that, for 32 nodes and 128 ranks, the GPUs running Thin-Threads are 3.4× faster than the CPUs running Thin-Threads and 10.4× faster than this same configuration of CPUs running Fat-Threads. We consider this performance to be very successful in the context of Monte Carlo particle transport. Since the Monte Carlo particle transport algorithm is not bound by the resources that the GPU makes readily available (compute and streaming memory throughput), it is inherently difficult to achieve significant GPU

TABLE I: Weak scaling results for Thin-Threads on CPUs and GPUs, compared to the weak scaling results for Fat-Threads for the same configuration. Time is listed in seconds. A batch size of 300,000 was used for the Thin-Thread+GPU runs and 100,000 was used for the Thin-Thread+CPU runs. There was no batching in the Fat-Thread code. All models were run with four ranks, and the CPU runs used 20 threads per rank.

Nodes / Ranks	Thin (GPU) [s]	Thin (CPU) [s]	Fat (CPU) [s]
1 / 4	1.866e+02	5.247e+02	6.788e+02
2 / 8	2.013e+02	5.470e+02	8.531e+02
4 / 16	2.130e+02	5.777e+02	1.998e+03
8 / 32	2.250e+02	8.482e+02	2.380e+03
16 / 64	2.537e+02	8.166e+02	2.327e+03
32 / 128	2.610e+02	8.902e+02	2.725e+03

TABLE II: Efficiency data from the weak scaling study. Basic parallel efficiency is given by comparing to single node performance. Relative efficiency is given by comparing to previous size performance (i.e., 2 nodes efficiency is tracking time as [1 Node / 2 Nodes], whereas 4 node efficiency is tracking time as [2 Nodes / 4 Nodes]).

Nodes	Thin (GPU)		Thin (CPU)		Fat (CPU)	
Ranks	Eff. 1 Node	Rel. Eff.	Eff. 1 Node	Rel. Eff.	Eff. 1 Node	Rel. Eff.
1 / 4	100%	—	100%	—	100%	—
2 / 8	92.69%	92.69%	95.92%	95.92%	79.56%	79.56%
4 / 16	87.61%	94.51%	90.83%	94.69%	33.97%	42.70%
8 / 32	82.93%	94.67%	61.86%	68.11%	28.52%	83.95%
16 / 64	73.55%	88.69%	64.25%	103.9%	29.17%	102.3%
32 / 128	71.49%	97.20%	58.94%	91.73%	24.92%	85.39%

performance. Instead, it is bound by memory latency and filled with branching divergent paths, both of which are identifiable as significant limiting factors with this algorithm on GPUs.

Table II shows the scaling efficiency up to 32 nodes. This can be calculated directly from Table I. The efficiency is calculated using a single node as a baseline. This table shows that the GPU maintains a weak scaling value of just over 70% efficiency at 32 nodes compared to using just one node. On a CPU platform, this is just under 60% for Thin-Threads and only 25% for Fat-Threads. This drop in performance on CPU platforms is in part due to the greater sensitivity that the CPU performance is showing to the added MPI complexity of higher scales, as well as the fact that 4 ranks, with 20 threads per rank, is not the optimum CPU layout for this machine.

Table II also shows the relative efficiency of scaling, for each increase in node count. This table highlights a number of interesting points about the scaling pattern. That said, some of the effects are due to the relationship between node count and problem size. As we increase the problem by a factor of 2, we are doing so only in one dimension at a time. At 16 nodes we have a perfect cube for problem dimensions [4x4x4], where as at 8 or 32 nodes we have a rectangular problem domain instead ([4x4x2] and [8x4x4], respectively). This informs some of the findings of the table. First, not all of the scales are slower. Specifically, on the CPU runs, the 16 nodes experiments shows better performance than the 8 or 32 node runs. This is most likely a side effect of load balancing in the scaling study itself. Second, there are a few definite points where efficiency drops dramatically compared to the previous scale. This highlights a step in complexity, as the subsequent scales do not continue to drop dramatically.

TABLE III: Figure of Merit and efficiency data from weak scaling runs on Vulcan, with 4 ranks per node and 16 threads per rank [15]. Efficiency against the 1 node runs and the relative efficiency for each step are shown. Relative efficiency is calculated in the same way as described in Table II.

Nodes	FOM [seg/sec]	Eff. 1 Node	Rel. Eff.
1	2.068e+06	100%	–
2	4.018e+06	97.15%	97.15%
4	7.622e+06	92.14%	94.85%
8	1.443e+07	87.22%	94.66%
16	2.773e+07	83.81%	96.08%
32	5.447e+07	82.31%	98.21%
64	1.072e+08	81.00%	98.40%
128	2.124e+08	80.24%	99.07%
256	4.216e+08	79.64%	99.25%
512	8.359e+08	78.95%	99.13%
1024	1.665e+09	78.63%	99.59%
2048	3.314e+09	78.25%	99.52%
4096	6.600e+09	77.92%	99.58%
8192	1.301e+10	76.80%	98.56%
16384	2.612e+10	77.09%	100.38%
24576	3.909e+10	76.91%	99.77%

An important take away from this efficiency data is that the Thin-Threaded model exhibits promising scaling behavior. This data shows the viability of this approach and that under these circumstances Thin-Threads performs best. That said, Monte Carlo particle transport problems can have irregular performance behaviors, and a more comprehensive study at higher node counts and more workloads could be useful.

C. Weak Scaling on BGQ

This section describes results on a Lawrence Livemore's Vulcan machine, which uses the BGQ architecture. Table III shows the scaling data we gathered. The performance data comes from a similar workload as was run in Section IV-B, with the only significant difference being less particles per node. This change was necessary since a node of BGQ is less performant than a GPU node on Ray.

Our data in this section is presented with respect to a figure of merit (FOM), specifically how many segments per second each problem ran on average. One advantage to considering results with respect to the FOM is that a doubling in resources should produce a doubling in the FOM. This is represented as percent efficiency — 100% efficiency means double the nodes led to a doubling of the FOM.

This data shows that the Thin-Threads solution scales well up to the entirety of the Vulcan portion of the Sequoia supercomputer. While we see higher efficiency on Vulcan than on Ray, this is most likely due to the nature of each machine. The BGQ system is designed from the ground up to minimize per-node variation in performance and has advanced networking features allowing codes to scale efficiently. Ray does not have these advantages — it has variation in performance per node (since it has power-based CPU clock throttling) and it has a simpler network architecture. Since Ray is a test bed machine, it is likely that some of our efficiency loss comes from the unoptimized network setup, or clock speed throttling resulting from using more and more of the system. Despite these differences, we see similar performance patterns between the two systems.

Comparing the CPU Thin-Threaded results on Ray with the CPU results on Vulcan, we can see that the same pattern of decreased efficiency at low scales with a leveling out of performance as we increase the scale. Given the similarity in these data sets we believe that a larger system could expect similar scaling performance even at much higher scales.

V. Conclusion

In this paper we demonstrated the effectiveness of the Thin-Threads approach for history-based Monte Carlo particle transport problems on GPUs. Additionally, Thin-Threads have also shown a degree of portability as both CPU and GPU forms of this approach have proved to be performant. On GPU platforms we achieved about 3× greater performance over the Thin-Threads CPU model and about 10× greater performance over the Fat-Threads CPU model.

One reason the Thin-Threads approach was effective was the inclusion of an asynchronous MPI batching model. The batching scheme presented in this paper has the added benefit of being a tunable parameter. This means that for problems where MPI is a dominating factor for performance, finding a good batch size could provide a starting point for optimizing performance. In some cases, we also noticed that the batch size was not a significant factor in performance. In these cases, as long as the batch size provided adequate parallelism, other factors dominated performance aside from time spent in MPI, therefore, overlapping computation with communication had little effect. Even in these cases, however, providing enough parallelism is an important factor and so it is important to determine a good batch size for the hardware. Through our experiences on Ray and Vulcan we saw that batch sizes of 100,000 or more worked well for GPU platforms and batch sizes greater than one thousand worked well for CPU platforms.

An important aspect of our study on Thin-Thread performance was the parallel efficiency when scaling up to large numbers of nodes. Specifically, we wanted to evaluate the performance of our new Batching+Asynchronous MPI approach. On Vulcan we showed that we could maintain nearly perfect relative efficiency (most being at or greater than 99%) and an overall parallel efficiency of greater than 75% on 24 thousand nodes (98304 ranks) when compared to a single node. On Ray we found we could maintain relative efficiencies in the 90% range after the initial dip around 8 nodes, and maintained a greater than 70% efficiency at 32 nodes on GPUs.

The performance and scalable efficiency of the Thin-Threads approach provides the basis to move forward in developing a GPU version of the full production application, using a single code base and threading model for both the CPU and GPU. The Thin-Threads model was developed inside of the Quicksilver mini-app available on Github [3]. Future work for this model will be its implementation in Mercury, the full production application Quicksilver is based on. Additional plans include large scale GPU runs on Sierra as well researching methods for running in a hydrid CPU+GPU batch processing mode.

References

[1] Co-design at lawrence livermore national lab: Quicksilver. Lawrence Livermore National Laboratory (LLNL), Livermore, CA (United States), https://codesign.llnl.gov/quicksilver.php, accessed: 2017-07-07

[2] Mercury web site, https://wci.llnl.gov/simulation/computer-codes/mercury, accessed: 2017-06-22

[3] Quicksilver. a proxy app for the monte carlo transport code, mercury. llnl-code-684037, https://github.com/LLNL/Quicksilver, version: 83ade89

[4] A set of radiation transport mini-applications used for performance optimization on hpc systems., http://ornl-cees.github.io/Profugus, accessed: 2018-04-05

[5] Ayubian, S., Alawneh, S., Richard, M., et al.: Implementation and performance of a gpu-based monte-carlo framework for determining design ice load. In: 2017 International Conference on High Performance Computing & Simulation (HPCS). pp. 109–116. IEEE (2017)

[6] Bergmann, R.: 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)

[7] Bleile, R., Brantley, P., Dawson, S., O'Brien, M., Childs, H.: Investigation of portable event-based monte carlo transport using the nvidia thrust library. Trans. Am. Nucl. Soc. (114), 941–944 (2016)

[8] Bleile, R., Brantley, P., O'Brien, M., Childs, H.: Algorithmic improvements for portable event-based monte carlo transport using the nvidia thrust library. Trans. Am. Nucl. Soc. (115), 535–538 (2016)

[9] Brantley, P., Dawson, S., McKinley, M., O'Brien, M., Stevens, D., Beck, B., Jurgenson, E., Ebbers, C., Hall, J.: Recent advances in the mercury monte carlo particle transport code. In: International Conference on Mathematics and Computational Methods Applied to Nuclear Science & Engineering (2013)

[10] Cullen, D.E., Clouse, C.J., Procassini, R., Little, R.C.: Static and dynamic criticality: are they different? Tech. rep., Lawrence Livermore National Lab., Livermore, CA (US) (2003), report: UCRL-TR-201506

[11] Hamilton, S.P., Slattery, S.R., Evans, T.M.: Multigroup monte carlo on gpus: Comparison of history-and event-based algorithms. Annals of Nuclear Energy **113**, 506–518 (2018)

[12] Ludvigsen, H., Elster, A.C.: Real-time ray tracing using nvidia optix. In: Eurographics (Short Papers). pp. 65–68 (2010)

[13] O'Brien, M.J., Brantley, P.S., Joy, K.I.: Scalable load balancing for massively parallel distributed monte carlo particle transport. In: Proceedings of International Conference on Mathematics and Computational Methods Applied to Nuclear Science & Engineering (M&C 2013), Sun Valley, Idaho. vol. 45, pp. 647–658 (2013)

[14] Pandya, T.M., Johnson, S.R., Davidson, G.G., Evans, T.M., Hamilton, S.P.: Shift: a massively parallel monte carlo radiation transport package. In: Proc. ANS MC2015?Joint International Conference on Mathematics and Computation (M&C), Supercomputing in Nuclear Applications (SNA) and the Monte Carlo (MC) Method. pp. 19–23 (2015)

[15] Richards, D., Bleile, R.: Quicksilver, summary version 1.03. Tech. rep., Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States) (2018)

[16] Richards, D.F., Bleile, R.C., Brantley, P.S., Dawson, S.A., McKinley, M.S., O'Brien, M.J.: Quicksilver: A proxy app for the monte carlo transport code mercury. In: Cluster Computing (CLUSTER), 2017 IEEE International Conference on. pp. 866–873. IEEE (2017)

[17] Romano, P.K., Forget, B.: The openmc monte carlo particle transport code. Annals of Nuclear Energy **51**, 274–281 (2013)

[18] Stpiczyński, P.: Vectorized algorithm for multidimensional monte carlo integration on modern gpu, cpu and mic architectures. The Journal of Supercomputing **74**(2), 936–952 (2018)

[19] Szałkowski, D., Stpiczyński, P.: Using distributed memory parallel computers and gpu clusters for multidimensional monte carlo integration. Concurrency and Computation: Practice and Experience **27**(4), 923–936 (2015)

[20] Tramm, J.R., Siegel, A.R.: Memory bottlenecks and memory contention in multi-core monte carlo transport codes. Annals of Nuclear Energy **82**, 195–202 (2015)

[21] Xu, X.G., Liu, T., Su, L., Du, X., Riblett, M., Ji, W., Brown, F.B.: An update of archer, a monte carlo radiation transport software testbed for emerging hardware such as gpus. Transactions of the American Nuclear Society **108**, 433–434 (2013)