2015 IEEE 29th International Parallel and Distributed Processing Symposium

A Performance Analysis of SIMD Algorithms for Monte Carlo Simulations of Nuclear Reactor Cores

David Ozog and Allen D. Malony
Department of Computer and Information Science
University of Oregon
Eugene, Oregon USA
{*ozog,malony*}*@cs.uoregon.edu*

Andrew R. Siegel
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, Illinois USA
siegel@mcs.anl.gov

***Abstract*—A primary characteristic of history-based Monte Carlo neutron transport simulation is the application of MIMD-style parallelism: the path of each neutron particle is largely independent of all other particles, so threads of execution perform independent instructions with respect to other threads. This conflicts with the growing trend of HPC vendors exploiting SIMD hardware, which accomplishes better parallelism and more FLOPS per watt. Event-based neutron transport suits vectorization better than history-based transport, but it is difficult to implement and complicates data management and transfer. However, the Intel Xeon Phi architecture supports the familiar `x86` instruction set and memory model, mitigating difficulties in vectorizing neutron transport codes.**

This paper compares the event-based and history-based approaches for exploiting SIMD in Monte Carlo neutron transport simulations. For both algorithms, we analyze performance using the three different execution models provided by the Xeon Phi (offload, native, and symmetric) within the full-featured OpenMC framework. A representative micro-benchmark of the performance bottleneck computation shows about 10x performance improvement using the event-based method. In an optimized history-based simulation of a full-physics nuclear reactor core in OpenMC, the MIC shows a calculation rate 1.6x higher than a modern 16-core CPU, 2.5x higher when balancing load between the CPU and 1 MIC, and 4x higher when balancing load between the CPU and 2 MICs. As far as we are aware, our calculation rate per node on a high fidelity benchmark (17,098 particles/second) is higher than any other Monte Carlo neutron transport application. Furthermore, we attain 95% distributed efficiency when using MPI and up to 512 concurrent MIC devices.

***Keywords*-Monte Carlo, neutron transport, reactor simulation, performance, SIMD, Intel Xeon Phi coprocessor, MIC**

I. Introduction

Monte Carlo (MC) methods of neutron transport generally exhibit excellent scalability across distributed computers, but may achieve an unsatisfactory percentage of local-node peak performance, especially on new architectures. History-based methods simulate a large number, say N, of independent particles, each of which can be mapped to a dedicated processor core. This is a "pleasingly parallel" algorithm in which each thread of execution simulates approximately N/p particles, where p is the total number of processor cores. Because each group of particles progresses independently of the others, history-based MC implementations are appropriate for multi-threaded computers capable of executing different instructions in parallel (MIMD). However, in new computer systems such as the Intel Xeon Phi, the use of the *same instruction* on multiple items of data (SIMD) is necessary to obtain good performance. Accomplishing this may require significant algorithmic changes to the N/p history-based decomposition, because data must be migrated and reorganized to fully exploit coprocessor hardware.

The history-based MC algorithm is straightforward to implement on modern computer clusters using hybrid parallel programming models such as MPI+OpenMP. The performance of these simulations in distributed memory systems is relatively well understood and quite scalable [1]; however, shared memory scaling can be less intuitive because of memory contention effects and complications within modern cache memory subsystems. As computational architectures foster exascale computing capabilities, it will become more important to understand and alleviate these effects. Hardware accelerators provide ever-wider vector units that encourage the use of the SIMD instructions in shared memory. Some applications can trivially exploit the powerful capabilities of these vector-centric architectures (diffusion, finite difference method, etc.), while others may require a careful redesign of data structures and algorithms (MC transport), and an unfortunate few may not be suitable candidates for vectorization at all.

The MC transport loop over particles, as described so far, is not a good candidate for vectorization because each particle in the simulation is in some intermittent state depending on its history. At a particular simulation time, each particle may be in a code block which applies physics for either elastic/inelastic scattering, absorption, collision, or migration in a manner that is imposed by thread-specific random seeds. SIMD instructions cannot be applied across history-based particle data structures because different instructions are required at different times. However, pivotal work by Troubetzkoy [2], Brown, and Martin [3] reexamined the history-based algorithm by considering an alternate event-

1530-2075/15 $31.00 © 2015 IEEE
DOI 10.1109/IPDPS.2015.105

based simulation, in which particles are *banked* for eventual vectorization. With the present-day resurgence of vector computing on heterogeneous systems, the banking method has gained appeal [4]–[7]. Nonetheless, we are currently unaware of any full-physics implementations of the banking method in MC neutron transport codes, likely due to the difficulty in transforming history-based data structures and control flow.

To analyze SIMD opportunities in MC neutron transport, we target the Intel Xeon Phi platform, which has high vectorization potential and programming/execution model flexibility. The execution models (described in detail in section II-B) have interesting implications and consequences when comparing banking and history-based algorithms. We find that the offload execution model is quite compatible with the banking method, but less so with the history-based method. On the other hand, the native and symmetric models are compatible with the history-based method, but less so with the banking method.

The goals of this paper are to support the above claims, to quantify our expectations in terms performance, and to discuss implementation challenges and strategies with banking and history-based methods. We measure a mature MC neutron transport application, OpenMC, to determine how well the Xeon Phi's different execution models perform. The results suggest further work to be done, the most appropriate execution models for satisfying user requirements, and important implications for the future of MC physics applications in the face of ubiquitous accelerator-based clusters.

II. Background

A. The OpenMC Monte Carlo Application

Our experiments are performed within the OpenMC [8] neutron transport code, originally developed in 2011 by the Computational Reactor Physics Group at MIT. OpenMC has quickly gained traction as a mature neutronics application for conducting accurate 3D simulations of nuclear reactor geometries. It is a modern framework written in Fortran 2008 that prioritizes good software engineering practices. Furthermore, it serves as a valuable testbed for implementing novel physics, parallel algorithms, and for exploring new high performance parallel architectures. The parallel programming model typically used is MPI for distributed memory communication, and OpenMP for shared memory multi-threading.

OpenMC applies the history method, in which each particle is independently tracked by a specific thread from its birth (e.g., a fission event) to its death (e.g., an absorption event). Throughout their lifetimes, particles undergo a series of possible interactions that include elastic/inelastic scattering, fission, absorption, and other secondary reactions. The movement of particles and occurrence of reactions are dictated by the interplay between a random number

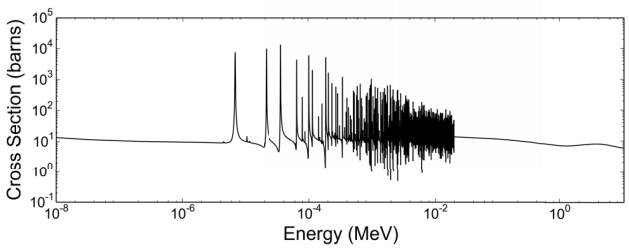

Figure 1. Cross section data for the Uranium-238 isotope

generator (RNG) and experimentally informed *neutron cross section data*. For each of the above reactions, there exist continuous energy cross section values, denoted σ_r, where r is a reaction specific subscript.

We now describe three importance concepts in MC neutron transport simulation that are relevant to this work:

1) Neutron Cross Section Data: An example of the total cross section data for the Uranium-238 isotope is in Figure 1. When a particle enters a material, all the nuclides therein contribute to the total macroscopic cross section, Σ_t. Algorithm 1 shows the calculation of Σ_t, done by each OpenMP thread of execution. The cross section data, which is interpolated to a fine grid across the energy domain, is replicated on each compute node and read-only.

Algorithm 1 Calculation of the macroscopic cross section

Input: $\vec{x}_p$ (location of particle p) and E_p (its energy)
  $m \leftarrow$ `lookup_material(` $\vec{x}_p$ `)`
  $u \leftarrow$ `find_energy_index(` E_p `)`
  for all n such that n is a nuclide in m **do**
    $\Sigma_t \leftarrow \Sigma_t$ + `atomic_density(` n `)` * $\sigma_r(n, u)$
  end for

Reactor simulations typically contain hundreds of nuclides and several reaction types, and it has been shown that computation of Σ_t for all particles is the primary bottleneck [1]. Often, the table lookup of σ_r results in a last-level cache miss, and is therefore entirely memory bound. Because of the large amount of time spent doing cross section lookups, we focus our initial efforts on the vectorization of this bottleneck (discussed further in section III-A below).

2) Sampling with an RNG: After calculation of the cross section values, the RNG becomes central to the determination of particle interactions. To see this, consider the calculation of distance to the next collision in MC neutron transport. Given the probability density function as a function of distance, $p(l)$, we integrate to obtain the cumulative distribution function for the distance to the next collision, d:

$$\int_0^d p(l)\, dl = \int_0^d \Sigma_t e^{-\Sigma_t l}\, dl = 1 - e^{-\Sigma_t d}$$

where Σ_t is the total macroscopic cross section of the material. By inversion transform sampling:

$$d = -\frac{\ln(1-\xi)}{\Sigma_t} = -\frac{\ln \xi}{\Sigma_t} \tag{1}$$

where ξ is a random number on the interval $[0, 1)$.

Other examples of RNG influence include the determination of whether or not an absorption reaction occurs and the value of the scattering angle, μ. An absorption reaction occurs if

$$\xi\sigma_t(E) < \sigma_a(E) - \sigma_f(E)$$

where the cross sections σ_t, σ_a, and σ_f correspond to the total, absorption, and fission cross sections, respectively. The cosine of scattering angles is determined by

$$\mu = 2\xi - 1$$

where μ is the cosine of the scattering angle.

3) Thermal effects and unresolved resonances: In order to attain high accuracy, OpenMC incorporates sophisticated physics treatments that are inherently difficult to vectorize. For instance, thermal motion effects occur because target nuclei have non-zero velocity, and the effects are accounted for on-the-fly. Thermal binding effects may also occur because of crystalline properties of the target nuclei. To account for such properties, OpenMC incorporates $S(\alpha, \beta)$ table lookups into the calculation of the cross sections, and makes subtle changes to outgoing elastic and inelastic scattering angles and energies. Another complication arises in the unresolved resonance range (URR), which resides in the high energy regime where resonances cannot be resolved experimentally (for example, around 10^{-2} MeV in Figure 1). Calculations in this range are not based on discrete cross section values but rather on probability tables [9]. The $S(\alpha, \beta)$ and URR routines inherently contain a large number of conditional expressions, or code branchings. Conditionals complicate implementation of the banking method, because groups of particles corresponding to each code branch need to be separated. This is typically done by replacing the conditionals with appropriate gather/scatter, compress/decompress, and bit-controlled vector operations.

B. The Xeon Phi: Architecture and Execution Models

The Intel Xeon Phi coprocessor, based on the Intel Many Integrated Core (MIC) architecture, is a PCIe attached device specialized for highly parallel computation. The device specifications and details are thoroughly described elsewhere [10]. There are 3 primary modes of execution supported by the MIC. Each mode has different implementation strategies and performance implications, especially in the context of Monte Carlo neutron transport calculations. They are the *offload*, *native*, and *symmetric* modes:

- **Offload** - The MIC assists the host with designated kernel regions. Offloading computation implicitly incurs PCIe latency and bandwidth costs (a major issue when offloading cross section lookups, even with a persistent energy grid and overlap via asynchronous transfer).
- **Native** - The MIC executes as an isolated Linux system. Serial performance is exacerbated by the relatively low core clock speed of the MIC, and applications are limited to on-board memory, no larger than 16 GB.
- **Symmetric** - Computation is shared between the host and MIC, and coordination occurs via message passing. Load balancing between host and MIC becomes an issue for maintaining synchrony (see section III-B3).

This paper analyzes all 3 models within the context of OpenMC by means of performance measurements, implementation constraints, implications for generality and extendability, and overall scalability.

III. Experimental Measurements

Our micro-benchmark experiments utilized resources from the Joint Laboratory for Systems Evaluation (JLSE) provided by the Argonne Leadership Computing Facility and the Mathematics and Computer Science department at Argonne National Laboratory. Specifically, we used the Jenny, Ruth, and Lucie compute nodes. The nodes contain 2 CPU sockets, each with an Intel E5-2687W processor (16 total cores, each 2-way hyper-threaded), clocking at 3.40 GHz, and a total of 64 GB of DDR3 RAM. Two Intel Xeon Phi 7120a coprocessor (MIC) devices connect to the PCIe 2.0 x16 bus. Each MIC device has 61 cores (4-way hyper-threaded) with a 1.238 GHz clock and 16 GB of RAM.

The micro-benchmarks in section III-A compare the single-node performance of the banking and history-based methods, whereas section III-B measures distributed-memory scaling performance of MPI+OpenMP in the full-featured OpenMC application. We conducted these experiments on the Stampede system at the Texas Advanced Computing Center (TACC). Each host CPU has 2 Xeon E5-2680 processors (8 cores total with 2-way hyperthreading), each with a 2.6 GHz clock and 32 GB of DDR3 RAM. 1,024 nodes contain 1 PCIe-attached SE10P MIC coprocessors (with 61 cores each running at 1.1 GHz), and 384 hosts have 2 MICs each (please note this in Figure 6). The interconnect is an FDR Mellanox InfiniBand network. Unless otherwise specified, all MIC experiments were run with 244 total OpenMP threads in native/symmetric mode and 240 threads in offload mode (leaving 1 core dedicated to data transfer). All applications are compiled with the Intel compiler version 14.0.1 with `-O3` optimizations.

The input geometry to the OpenMC application is the Hoogenboom-Martin benchmark [11] which provides a common model for performance analysis of Monte Carlo simulations of full-core reactors. The model consists of a pressurized water reactor core with 241 identical fuel assemblies (each 21.42 x 21.42 cm). Each assembly consists of a 17 by 17 lattice of fuel pins including 24 control rod guide tubes and an instrumentation tube. A thin cladding composed of natural zirconium surrounds each fuel pin. In the original Hoogenboom model, 34 different nuclides make up the nuclear fuel: a mixture of actinides, minor actinides, and key fission products. Throughout our experiments, we

label the 34-nuclide model as "H.M. Small". A similar model, labelled "H.M. Large", consists of a more accurate representation of fuel containing 320 different nuclides.

In section III-A below, we consider two micro-benchmarks that quantify the potential benefits of the banking method when vectorizing across particles in offload mode. In section III-B, we show performance results for a full-featured history-based implementation of OpenMC in native mode, then discuss the load balancing strategies for distributed scaling in symmetric mode.

A. OpenMC Micro-benchmarks

The feasibility of banking a large number of particles to be offloaded to the Xeon Phi is evaluated by considering two micro-benchmarks:

1) *Cross Section Lookups*: OpenMC is fully initialized, and particles are *banked* just before a cross section lookup is required. Then, after banking N particles, we measure the time to compute all N cross sections.
2) *Sampling Collision Distances*: After computing all N cross sections from micro-benchmark #1, we measure the time to compute all N distances to the next collision event (as in Equation (1)).

In benchmark #1, we show that vectorization of the cross section loop over nuclides is possible and compare calculation rates of the MIC with the host CPU. In micro-benchmark #2, we present optimizations in random number generation and the use of vector intrinsic functions to improve performance. We then consider the overhead involved with banking particles, and postulate about the performance of offloading in a complete implementation.

1) Cross Section Lookups in Offload Mode: In this micro-benchmark, we modified OpenMC to bank a particle into shared memory just before a cross section lookup would normally take place. Once a sufficient number of particles are in the global bank, a loop over *all* particles accomplishes the lookups at once. This drastically changes the control flow of the current implementation of OpenMC, and dismantles most of its functionality. However, since the lookup function is the bottleneck region of OpenMC, measuring this modified version and treating it as a micro-benchmark is quite informative.

Algorithm 2 shows the overall procedure for this micro-benchmark. We made a series of optimizations (not shown) to improve the performance of the lookups on the MIC. The most important was the transformation of arrays of Fortran derived types into single isolated arrays. This is a common optimization ("array of structs to struct of arrays", or AoS to SoA) that is particularly important on the MIC, and examples of its implementation and analysis are discussed in many sources [10], [12]. Other relatively minor optimizations include data alignment to 64-byte boundaries and the manual tuning of prefetch distances.

Algorithm 2 Banked Cross Section Lookup

```
1:  Initialize OpenMC (geometry, energy grid, nuclides, etc)
2:  #omp parallel for
3:  for i ← 0 to total_particles do
4:     initialize_particle(i);
5:     bank_particle(i);
6:  end for
7:  synchronize_bank();
8:  #omp parallel for
9:  for i ← 0 to banked_particles do
10:    Locate E_p on Unionized Grid;
11:    #pragma simd
12:    for n ← 0 to num_nuclides do
13:       Calculate macroscopic cross section;
14:    end for
15: end for
```

Figure 2 shows the number of lookups per second on the MIC for two different implementations: one with banking of particles and one without. Both take the H.M. Large benchmark as the input problem and apply the unionized energy grid algorithm [13], which reduces the number of grid searches by creating a point-wise union of the reaction cross section values across all nuclides. Using the instrumentation API of the TAU parallel performance system [14] to establish static timers, the average lookups per second were gathered for both the banking method and the original history-based method. For the banking method, the timer starts in the main thread at entry to the loop across the particle bank and stops at the loop's exit. For the history-based method, TAU collects the average overall time spent in the `calculate_xs()` routine, which calculates the total macroscopic cross section for a single particle in a particular material at some moment of the simulation.

In order to accomplish vectorization across the particle bank, we vectorized the inner loop over nuclides within a material (line 12 of Algorithm 2), as opposed to the outer loop over banked particles (line 9). Although it is possible to force vectorization across the outer loop with a `#pragma`

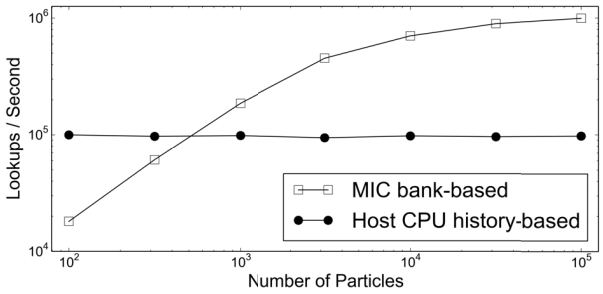

Figure 2. Cross section lookup rates for banking (MIC) and history (CPU) methods in micro-benchmark #1 on a JLSE compute node (H.M. Large).

Algorithm 3 (Naive) - Sample distance to next collision

Input: Array X with N cross section values
Output: Array D with N distances to next collision

```
for i ← 0 to iters do
  #pragma omp parallel for
  for j ← 0 to N do
    float r = rand_r()/RAND_MAX;
    D[j] = -log(r) / X[j];
  end for
end for
```

simd directive, this results in relatively worse performance (likely because the bounds of the inner loop vary with the different materials). This is an important observation because it implies that the same vectorization can be accomplished in a history-based implementation. It was also necessary to remove the blocks that handle $S(\alpha, \beta)$ and URR calculations to achieve vectorization. For fairness of comparison, the same blocks of code were removed from the original version of the host CPU code. With the MIC version, we see a speedup on the order of 10x using the banking method. While this is promising, future work must incorporate the $S(\alpha, \beta)$ and URR calculations to attain the best simulation accuracy. This will require more careful hand-coding to handle the high amount of conditional branching in the code.

2) Sampling Distances and Tracking: The second micro-benchmark examines a ubiquitous method in MC transport: sampling the distance to the next collision as a particle travels through a particular (homogeneous) material. In fact, this is the next step in OpenMC after a cross section lookup. The goal of this micro-benchmark is to calculate the distance d_j, as determined by Equation (1), for every banked particle ($j = 1 \ldots N$). By vectorizing the calculation of sampled distances across a banked collection of particles, we can better understand performance of the banking method in a full-featured implementation.

As in the previous section, we measure the time spent looping through all N particles in the bank. In this case, however, the loop is repeated a number of times (10^4 in our experiments) to mimic the generation loop and to get more consistent measurements. Algorithm 3 shows a naive OpenMP implementation in which each thread is assigned some number of particles in the bank. Since no scheduling directive is present, load balancing is deferred to Intel's OpenMP runtime. A random number is chosen for each particle using a C standard library call to the reentrant rand_r() function.

Algorithm 4 is an optimized version of Algorithm 3. There are two primary differences; the first being the use of Intel's Math Kernel Library (MKL) to generate random numbers using the Vector Statistical Library (VSL) interface. Lines 5-8 show that an array of size N (10^7 in our experiments) is initialized with uniformly random floats on the interval

Algorithm 4 (Optimized) - Sample distance to collision

Input: Array X with N cross section values
Output: Array D with N distances to next collision

```
1:  float R[nstreams][N/nstreams];
2:  VSLStreamStatePtr strm[nstreams];
3:  for i ← 0 to nstreams do vslNewStream(&strm[i]) end for
4:  for i ← 0 to iters do
5:    #omp parallel for
6:    for k ← 0 to nstreams do
7:      vsRngUniform(strm[k], N, R[k], 0.0, 1.0);
8:    end for
9:    #omp parallel for schedule(guided)
10:   for (j = 0; j < N; j+=16) do
11:     __m512 v1, v2, v3, v4, v5, v6;
12:     v1 = _mm512_load_ps(R + j);
13:     v2 = _mm512_load_ps(X + j);
14:     v3 = _mm512_log_ps(v1);
15:     v4 = _mm512_div_ps(v3, v2);
16:     v5 = _mm512_set1_ps(-1.0);
17:     v6 = _mm512_mul_ps(v4, v5);
18:     _mm512_store_ps(& D[j], v6);
19:   end for
20: end for
```

$[0, 1]$ via the VSL random number generator API. This API is highly optimized because it uses vectorized mathematical functions. VSL also allows for parallel random number generation with multiple threads using "skip ahead" or "leap frog" *streams*. This enables threads to initialize different sections of the output array, R. Our experiments use the VSL_BRNG_MT2203 random number generator set.

The second optimization involves the use of vector intrinsic functions to perform the SIMD operations manually. Since Equation (1) is a very simple mathematical operation, it is easy to utilize the appropriate vector intrinsic functions for 512-bit wide instructions across the loop. These correspond to lines 12-18 where each _m512 is a vector register consisting of 16 floating point elements, each 4 bytes long. Since this kernel is a memory bound computation (the X, D, and R arrays are typically too large to fit entirely into L2 cache), optimal prefetch distances are required to get the best performance. The #pragma noprefetch prohibits the compiler from guessing prefetch distances, and _mm_prefetch() calls specify exact distances that were carefully tuned for this MIC and kernel combination (not shown in Algorithm 4 for brevity). The _mm_malloc() function is used to allocate dynamic memory with 64-byte boundary alignment (also not shown for brevity).

Table I compares the execution times of three different implementations: the naive approach from Algorithm 3, an optimized version (#1) with VSL random number generation (no vector intrinsics), and the full optimization (#2) from Algorithm 4. The results show that initialization of random numbers can drastically inhibit performance when not using vector-optimized methods, especially on the MIC. Also, the use of vector intrinsics can further improve performance, even for relatively simple computational loops.

	Naive	Optimized-1	Optimized-2
Implementation	time(s)	time(s)	time(s)
CPU - 32 threads	412	40.6	36.6*
MIC - 122 threads	8,243	21.0	18.9

Table I
AVERAGE TIMES FOR THE SAMPLING DISTANCE MICRO-BENCHMARK WITH $iters = 10^4$ AND $N = 10^7$ ON A JLSE NODE.

3) Offload Overhead Considerations: The measurements from the two micro-benchmarks above only include the time spent in the loop over banked particles. However, we also need to consider the overhead time spent banking particles and shipping the associated data between device and host. This section compares time measurements with respect to the banking of the particles, the transferring of data over the PCIe bus, and the computing of cross sections.

Using a combination of Intel offload reports (by setting the `OFFLOAD_REPORT` environment variable to 3) and TAU timers, we gathered the information in Table II. The time to bank particles on the host is considerably lower than on the MIC because it is a write-intensive memory operation that is not vectorized. The transfer time to the MIC is the most expensive operation, which stresses the importance of overlapping computation with asynchronous data transfer. The time to transfer the relatively large energy grid is considerably higher (approximately 1 second for every 5 GB), but that cost is only paid during initialization, and is amortized with large numbers of batches and/or generations per batch.

	H.M. Small	H.M. Large
Operation	time(s)	time(s)
banking (host)	4 ms	4 ms
banking (MIC)	21 ms	34 ms
transfer time (PCIe)	460 ms	2,210 ms
bank size transferred	496 MB	2.84 GB
energy grid size transferred	1.31 GB	8.37 GB
compute bank cross sections (MIC)	17 ms	101 ms

Table II
AVERAGE TIME AND SIZES (PER ITERATION) FOR BANKING PARTICLES AND OFFLOADING TO THE MIC WITH 10^5 PARTICLES

Figure 3 shows the asymptotic behavior of the ratio of offload time to generation time as the number of particles is increased. For each group, the average generation time (the time to simulate all particle histories) of an iteration is normalized to 1.0, and all other times are reported as a ratio of that value. As the number of particles increases, the relative cost of each operation shows a different trend: offloading bank data over the PCIe bus decreases, calculating cross sections on the host increases, and calculating cross sections on the MIC decreases. This behavior suggests that it is indeed possible to gain performance by offloading cross section lookups to the MIC, despite the banking overheads, if simulating sufficiently many particles - above 10,000.

*Using the corresponding 256-bit AVX vector intrinsics

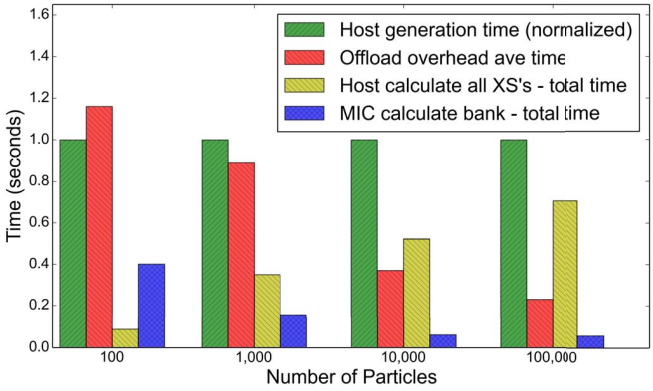

Figure 3. Time comparison between banking particles on the CPU and offloading to the MIC (H.M. Small). Yellow corresponds to the history method on the CPU, and blue/red to the banking method on the MIC. All times are normalized with respect to the host generation time (green).

B. Full-Physics, History-Based Transport on the MIC

The performance improvements presented in the previous section are promising, but leave something to be desired. In order to accomplish vectorization more easily, we removed $S(\alpha, \beta)$ and URR physics calculations from the micro-benchmarks. Furthermore, restructuring OpenMC to bank particles before lookups drastically changes the overall control flow, and will require a substantial development effort to accomplish a complete implementation. The impressive speedups and asymptotic trends of offload overhead are encouraging and suggest a full Xeon Phi implementation with banking may be worthwhile. However, one significant advantage of programming on the Xeon Phi is that little porting effort is required to run *entire* applications on the coprocessor. This section explores what performance we can expect with all physics in OpenMC included and without any banking. We consider two different execution models: native mode and symmetric mode, as described in section II-B.

Figure 4 shows an excerpt of a TAU comparison profile between an OpenMC execution on the MIC in native mode and on the host CPU. The application was largely unchanged to run on the MIC. The primary optimization was a switch from a manual sum reduction across global tallies to an OpenMP based sum reduction. Also, some OpenMP critical sections were replaced with relatively lightweight atomic sections, and alignment of key data structures was forced to lie on 64-byte boundaries. All these changes also improved the performance on the host CPU, and Figure 4 includes all optimizations on both architectures.

The top three routines in Figure 4 all contribute to the cross section lookups, where `calculate_xs()` is the

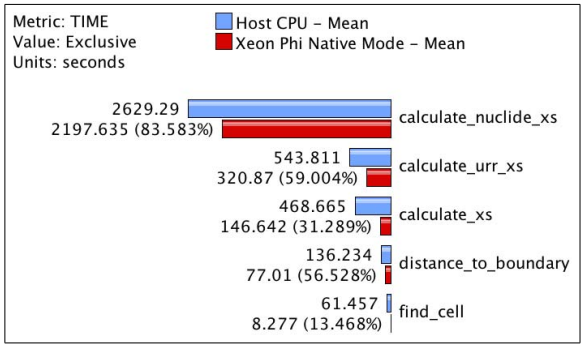

Figure 4. A TAU Profile comparison between the host CPU execution and the MIC (in native mode) after optimizations on a JLSE node. The input was an H.M. Large benchmark with 10^7 particles.

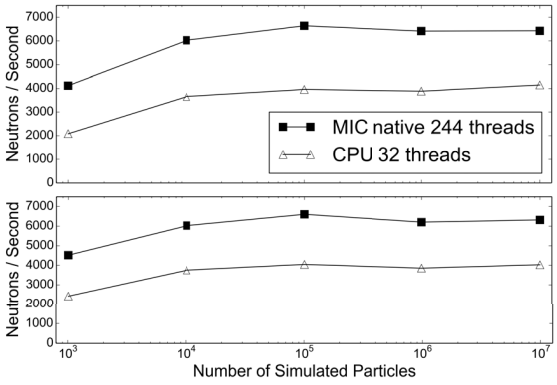

Figure 5. Calculation rate (neutrons per second) in OpenMC for inactive batches (top) and active batches (bottom) on a JLSE compute node (with the H.M. Large benchmark).

parent function. The main feature to notice is the MIC performance is better than the original on these bottleneck functions. Furthermore, the *total* simulation time on the host was 96 minutes whereas the total time on the MIC was 65 minutes (for a 1.5x speedup).

1) Native Mode on a Single Compute Node: An important metric for evaluating the performance of an OpenMC execution is the number of simulated neutrons per second, or the *calculation rate*. We desire a high rate because many particles per batch are required to obtain meaningful statistics, and many batches are needed to assure source convergence. Figure 5 compares the calculation rate between the host CPU and the MIC using the H.M. Large model and several different numbers of particles. All simulations use the unionized energy grid algorithm [13]. The top half of Figure 5 shows the calculation rate for inactive batches, which do not contribute to tallies. The bottom half shows active batches, in which tallies are accumulated. We run inactive batches because the fission source distribution and eigenvalue do not converge immediately, so there is little reason to tally initially. In general, the performance differs between inactive and active batches, but in the default H.M. benchmark, there is little distinction.

The above measurements show that the MIC consistently simulates neutrons at a rate 1.5 to 2 times faster than the CPU, and the highest rates occur with at least 10^5 particles per node. We define the ratio of calculation rates as

$$\alpha = \frac{\text{CPU calculation rate}}{\text{MIC calculation rate}} \tag{2}$$

From the data in Figure 5, we see that α is consistent when simulating at least 10^4 particles: $\alpha_{\text{i}} = 0.61 \pm 0.02$ for inactive batches and $\alpha_{\text{a}} = 0.62 \pm 0.01$ for active batches. In general, α differs between active and inactive batches, particularly if user-defined tallies are collected throughout phase space. In our experiments, however, only the default global tallies are considered (total collisions, absorptions, and track-lengths), and are relatively less expensive.

The node runs out of main memory on both the CPU and the 16 GB MIC when simulating between 10^7 and 10^8 particles. On the 8 GB MIC, between 10^6 and 10^7 particles can be simulated. While this is a serious restriction, the issue can be ameliorated by distributing the computation to more compute nodes, which is the subject of the next section.

2) Symmetric Mode with MPI + OpenMP: When using MPI, symmetric mode is easily applied by building two different binaries - one for the host CPU and one for the MIC device - then executing them simultaneously with `mpirun` or `mpiexec`. Although it is possible to run 1 or more MPI processes per core on the MIC, most applications find the best performance using a relatively small number of processes and an appropriate number of threads [15] (for example, 2 processes and 122 threads or 4 processes and 61 threads, etc.). After our optimizations, OpenMC attains the best performance when running a single MPI process per MIC device with 244 threads.

With little extra effort, OpenMC can be run in symmetric mode by using the host and native binaries from the previous section. However, OpenMC assigns work to MPI processes statically: the total number of particles is split into p groups, where p is the number of MPI processes. The results from Figure 5 establish that the MIC and CPU have quite different calculation rates, so load imbalance naturally occurs when running the application in this manner.

The "Original" column in Table III shows that the performance when running the CPU + MIC combination is 16% less than the ideal calculation rate (10,691 neutrons/second). The CPU + 2 MIC combination is 32% less than ideal rate (17,332 neutrons/second). The next section tackles this load balancing issue.

3) Load Balancing Symmetric Mode: A simple way to address the problem above is by imposing a static load balancing scheme [12]. If there are n_{total} particles, the goal is to determine the best number of particles to assign to each

	Original	Load Balanced
Implementation	rate	rate
CPU	4,050	N/A
MIC	6,641	N/A
CPU + MIC	8,988	10,068
CPU + 2 MICs	11,860	17,098

Table III
AVERAGE CALCULATION RATES (NEUTRONS PER SECOND) FOR H.M. LARGE (ACTIVE BATCHES WITH 10^5 PARTICLES) ON A JLSE NODE. IN THE LOAD BALANCED COLUMN, $\alpha = 0.62$.

MIC, n_{mic}, and the best number of particles to assign to each CPU, n_{cpu}. At any given scale, there are p_{mic} processes running on MIC devices and p_{cpu} processes running on host CPUs where

$$p_{mic}n_{mic} + p_{cpu}n_{cpu} = n_{total}.$$

Solving this equation for both n_{mic} and n_{cpu} and using the fact that $n_{cpu}/n_{mic} = \alpha$ gives

$$n_{mic} = \frac{n_{total}}{p_{mic} + p_{cpu}\alpha} \quad \text{and} \quad n_{cpu} = \frac{n_{total}\alpha}{p_{mic} + p_{cpu}\alpha}. \quad (3)$$

This α is essentially the same quantity from Relation (2). For our H.M. Large experiment with 10^7 particles, choosing $\alpha = 0.62$ estimates that $n_{mic} =$ 6,172,840 and $n_{cpu} =$ 3,827,160 for a single-node execution. Using these quantities, the performance is only about 6% less than the ideal calculation rate, as shown in the "Load Balanced" column in Table III.

The strong scaling plot in Figure 6 shows OpenMC performance when using the simple static load balancing method above. In this experiment, we simulated the H.M. Large model with 10^7 total particles at several different scales. Because JLSE only has 3 nodes with MICs, we switch to the Stampede cluster at TACC. The Stampede cluster has 1,024 nodes with 1 MIC per node and 384 nodes with 2 MICs per node. This is why the 1-MIC curve in Figure 6 scales to 2^{10} nodes, but the 2-MIC curve does not.

Figure 6 shows near perfect distributed strong scaling. For example, at 128 nodes (17,664 total cores), the simulation time is 95% of the expected ideal based on the 4 node measurement (which is the smallest allotment that will fit 10^7 particles, due to the relatively limited amount of MIC memory). The extra 5% can easily be accounted for by considering the extra communication required at 128 nodes, which is not included in our simple ideal estimate.

The tailing of the 1-MIC curve at 1,024 nodes is expected based on the measurements in Figure 5. Because $\alpha = 0.42$ on Stampede (with 10^6 particles), Equation (3) predicts 6,643 particles assigned to the MIC and 3,122 to the CPU. With such a low number of particles, however, our calculation rate has decreased and α has changed. The effect is not seen in the "CPU only" curve because we are still safely simulating about 10^4 particles per node.

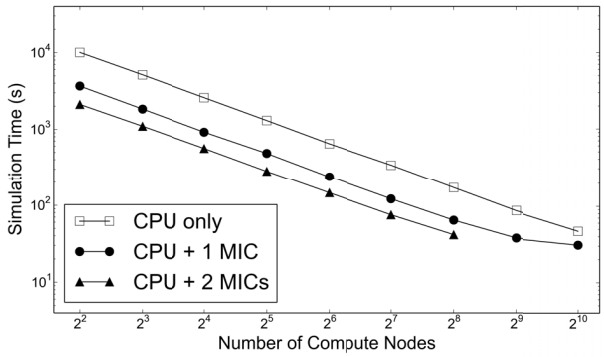

Figure 6. Strong scaling of the H.M. Large simulation with $N = 10^7$ on the TACC Stampede Cluster.

At this point, the simple fix is to increase the number of particles per node (to weak scale the application). A weak scaling plot (for which n_{mic} and n_{cpu} are held constant while increasing scale) is shown in Figure 7. Weak scaling exhibits greater than 94% efficiency at all scales up to 128 compute nodes on Stampede. We expect that weak scaling will maintain high efficiency at much larger scales, at least until the performance of MPI reduction begins to decline*.

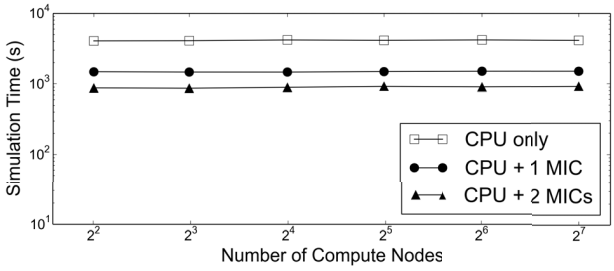

Figure 7. Weak scaling of the H.M. Large simulation with $N = 10^6$ per node on the TACC Stampede Cluster.

IV. RELATED WORK

A. SIMD in MC Transport

Related proof-of-concept work explores the possibility of porting MC neutron transport methods to SIMD architectures, in particular on GPUs [4]–[6]. Similar to the cross section lookup micro-benchmark above, gathering particles for vectorization purposes takes quite some effort to fully implement, and often synthetic data is used, or stripped down codes are presented [6]. While some work has been done on the Xeon Phi coprocessor [7], we are not aware of any thorough analysis of the alternative execution models, especially of the native and symmetric modes. As we have seen, the primary advantage of these modes of execution

* We are confident that the weak scaling curve will remain flat out to 2^{10} nodes and will be conducting experiments on Stampede to validate this claim.

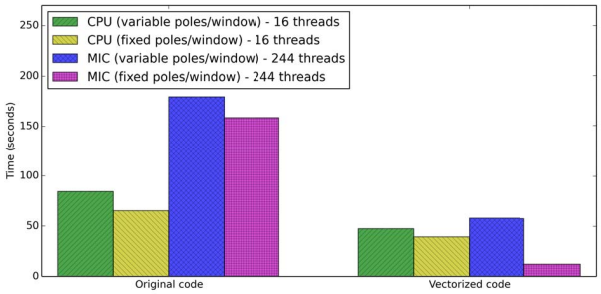

Figure 8. Execution time for RSBench implementations on Stampede

is that full-physics implementations can be analyzed on the Xeon Phi with relatively little development effort; and in the case of OpenMC, performance is considerably better than on the host.

B. Multipole Expansion of Cross Section Data

A particularly exciting area of related research is in the multipole expansion of cross section data [16], [17]. The primary motivation for this work is to incorporate the temperature dependence of nuclear cross section data into MC simulation. Applying temperature dependence with the standard table lookup approach requires an astoundingly large amount of data that is impractical to replicate. Instead of doing table lookups, the multipole expansion method calculates a summation over poles (i.e. singularities), each with a different Faddeeva function evaluation [17]. Not only does this enable temperature dependence at remarkably low memory cost, but it potentially turns a memory-bound problem into a compute-bound problem. This could be beneficial for OpenMC because it is bottlenecked by the cross section lookups.

In order to study the performance behavior of multipole expansion, a proxy application called RSBench was developed by Tramm and Siegel [18]. On a host CPU Xeon processor, previous work found that RSBench can achieve twice the FLOP rate as the classical lookup approach. Initial comparison with experiments on the MIC are encouraging after assuring vectorization and fixing the number of poles per window. Figure 8 compares the execution time of the original code with a vectorized version on Stampede. Future work will involve exploring the viability of whether multipole expansion data can be prepared to have a constant number of poles per window.

V. Future Work

The primary component missing from our banking-based implementation is the inclusion of the $S(\alpha, \beta)$ and URR routines. Accomplishing this will require care to assure that the loop over nuclides remains vectorized. In the above implementations, we favor automatic vectorization by the compiler, but the use of vector intrinsics are a viable option to assure the best possible performance. However, there are drawbacks: Fortran vector intrinsics currently do not exist, requiring the extra hurdle of linking with C/C++ and possibly switching back and forth from row to column-major memory layouts. Also, the use of intrinsics potentially obfuscates the meaning of the code, which detracts from OpenMC's pleasantly high readability.

Our history-based implementation of OpenMC in native/symmetric mode can be further optimized. For instance, the AoS to SoA transformation improved the performance of micro-benchmark #1, but this optimization has not yet been fully incorporated into OpenMC. Full conversion will require a more extensive development effort and verification for correctness than with the micro-benchmark. Furthermore, if vectorization of the $S(\alpha, \beta)$ and URR routines can be accomplished, then SoA will surely improve performance.

In symmetric mode, the calculation rate varies little between batches (especially when separating inactive from active), so the value of α_i and α_a can be estimated accurately from only a single inactive and active batch, respectively. This implies that α can be determined at runtime by setting it to $1/p$ on the first batch, and using the measured calculation rates to determine an appropriate α for subsequent batches. Some modification of OpenMC's internal data structures are required to adapt at runtime, and we are currently implementing this feature.

OpenMC's on-node performance and off-node scalability in symmetric mode are promising, especially in light of the future directions of Xeon Phi architectures. Intel has announced Knights Landing [19], in which there will no longer be a required PCIe connection between host and co-processor. Up to 72 cores can be attached directly to the host CPU socket, and out-of-order execution will be supported, resulting in a possible automatic $\sim$3x single thread speedup over Knights Corner for most applications. Since on-package memory will be up to 16 GB, similar to our experiments above, and out-of-order execution surely plays a crucial role in the cross section lookup calculations, we may potentially observe far better performance on Knights Landing systems in symmetric mode.

Finally, an interesting future direction is analyzing energy expenditures in MC neutron transport. Host-attached devices, such as MIC and GPU devices, show excellent performance per watt. Power measurement and management tools, such as the Running Average Power Limit (RAPL) interface, `micsmc`, and PAPI's `micpower` component provide a means for comparison of host and device energy performance. Future work will include these energy measurements and incorporate those into an analysis of the trade-off between time to solution and energy expenditure.

VI. Conclusion

We have discussed two techniques for doing vector processing on the Intel Xeon Phi in the context of the OpenMC

neutron transport application. The first approach is to bank large collections of particles, then apply vectorized loops across each collection. This is congruent with the Xeon Phi's *offloading* execution model, in which data must be transferred to and from devices for maximum performance. The second approach is to vectorize loops within the history-based code for each individual particle. This was prototyped in *native* mode, then scaled out with MPI + OpenMP in *symmetric* mode*.

This paper highlights the important trade-off between particle banking and history-based methods in MC neutron transport applications. The performance gained when banking particles for vectorization is potentially very high: our micro-benchmarks show between 2x and 10x speedups. However, a full banking-based implementation is difficult to achieve, and complications arise when transforming data structures to be vector compatible, overlapping computation with asynchronous data transfers, determining which code regions to vectorize/offload, etc. The history-based method is far more straightforward to implement, and potential for effective vectorization exists, especially when there is a large number of nuclides per material. The MIC is able to execute a full-physics, full-core reactor simulation of the H.M. Large benchmark about 1.6 times faster than the host, and 4 times faster when balancing load between the CPU and 2 MICs. As far as we are aware, our per-node calculation rate on the H.M. Large benchmark (17,098 particles/sec) is higher than any other MC neutron transport application. Furthermore, we achieve 95% of the ideal simulation time at scales up to 39,424 cores using 512 MICs and CPUs in parallel.

ACKNOWLEDGMENTS

D. Ozog is supported by the Department of Energy Computational Science Graduate Fellowship (DOE CSGF) program under contract DE-FG02-97ER25308. This research used resources of the Argonne Leadership Computing Facility and Laboratory Computing Resource Center at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357. The authors acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing HPC resources that have contributed to the research results reported within this paper. URL: http://www.tacc.utexas.edu. The research at the University of Oregon was supported by grants from the U.S. Department of Energy, Office of Science, under contracts DE-FG02-07ER25826, DE-SC0001777, and DE-FG02-09ER25873.

REFERENCES

[1] A. Siegel, K. Smith, P. Romano, B. Forget, and K. Felker, "Multi-core Performance Studies of a Monte Carlo Neutron Transport Code," *International Journal of High Performance Computing Applications*, vol. 28, no. 1, pp. 87–96, Feb. 2014.

[2] E. Troubetzkoy, H. Steinberg, and M. Kalos, "Monte Carlo Radiation Penetration Calculations on a Parallel Computer," *Trans. Am. Nucl. Soc.*, vol. 17, p. 260, 1973.

[3] F. B. Brown and W. R. Martin, "Monte Carlo Methods for Radiation Transport Analysis on Vector Computers," *Progress in Nuclear Energy*, vol. 14, no. 3, pp. 269 – 299, 1984.

[4] Liu, Du, Ji, Xu, and Brown, "A comparative study of history-based versus vectorized Monte Carlo methods in the GPU/CUDA environment for a simple neutron eigenvalue problem," *SNA + MC 2013*, p. 04206, 2014.

[5] Bergmann, Ryan M. and Vujić, Jasmina L., "Optimization of Monte Carlo Algorithms and Ray Tracing on GPUs," *SNA + MC 2013*, p. 04212, 2014.

[6] F. A. van Heerden, "A Coarse Grained Particle Transport Solver Designed Specifically for GPUs," *Transport Theory and Statistical Physics*, vol. 41, pp. 80–100, 2012.

[7] Liu, Tianyu, George Xu, X, and Carothers, Christopher D., "Comparison of Two Accelerators for Monte Carlo Radiation Transport Calculations, NVIDIA Tesla M2090 GPU and Intel Xeon Phi 5110p Coprocessor: A Case Study for X-ray CT Imaging Dose Calculation," *SNA + MC 2013*, p. 04205, 2014.

[8] P. K. Romano and B. Forget, "The OpenMC Monte Carlo particle transport code," *Annals of Nuclear Energy*, vol. 51, no. 0, pp. 274 – 281, 2013.

[9] L. B. Levitt, "The probability table method for treating unresolved neutron resonances in Monte Carlo calculations," *Nucl. Sci. Eng*, vol. 49, pp. 450–457, 1972.

[10] J. Jeffers and J. Reinders, *Intel Xeon Phi Coprocessor High Performance Programming*. Morgan-Kaufmann, 2013.

[11] J. Hoogenboom, W. Martin, and B. Petrovic, "Monte Carlo performance benchmark for detailed power density calculation in a full size reactor core," *Nuclear Energy Agency*, 2009.

[12] A. Vladimirov and V. Karpusenko, *Parallel Programming and Optimization with Intel Xeon Phi Coprocessors*. Colfax International, 2013.

[13] J. Leppänen, "Two practical methods for unionized energy grid construction in continuous-energy Monte Carlo neutron transport calculation," *Annals of Nuclear Energy*, vol. 36, no. 7, pp. 878–885, 2009.

[14] S. Shende and A. Malony, "The TAU parallel performance system," *International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, May 2006.

[15] K. Antypas, "Preparing Your Application for Advanced Manycore Architectures," presented at the DOE CSGF HPC Advanced Topics Workshop, Washington, D.C. (2014).

[16] R. Hwang, "A Rigorous Pole Representation of Multilevel Cross Sections and Its Practical Applications," *Nuclear Science and Engineering*, vol. 96(3), pp. 192–209, 1987.

[17] B. Forget, S. Xu, and K. Smith, "Direct Doppler broadening in Monte Carlo simulations using the multipole representation," *Annals of Nuclear Energy*, vol. 64, pp. 78 – 85, 2014.

[18] J. R. Tramm and A. R. Siegel, "Performance Analysis of a Reduced Data Movement Algorithm for Neutron Cross Section Data in Monte Carlo Simulations," 2014.

[19] R. Hazra, "Accelerating Insights... In the Technical Computing Transformation," presented at the *International Supercomputing Conference*, Leipzig, Germany (2014).

*The source code is freely available online:
https://github.com/davidozog/openmc/tree/mic-mpi-load-balanced