NUCLEAR SCIENCE AND ENGINEERING · VOLUME 185 · 232–242 · JANUARY 2017

DOI: http://dx.doi.org/10.13182/NSE16-33

Technical Note

Optimizing the Monte Carlo Neutron Cross-Section Construction Code XSBench for MIC and GPU Platforms

Tianyu Liu,* Noah Wolfe, Christopher D. Carothers, Wei Ji, and X. George Xu

Rensselaer Polytechnic Institute, Troy, New York, 12180

Received January 31, 2016
Accepted for Publication April 8, 2016

Abstract — *XSBench is a proxy application used to study the performance of nuclear macroscopic cross-section data construction, which is usually the most time-consuming process in Monte Carlo neutron transport simulations. In this technical note we report on our experience in optimizing XSBench to Intel multicore central processing units (CPUs), many integrated core coprocessors (MICs), and Nvidia graphics processing units (GPUs). The continuous-energy cross-section construction in the Monte Carlo simulation of the Hoogenboom-Martin large problem is used in our benchmark. We demonstrate that through several tuning techniques, particularly data prefetch, the performance of XSBench on each platform can be desirably improved compared to the original implementation on the same platform. It is shown that the performance gain is 1.46× on the Westmere CPU, 1.51× on the Haswell CPU, 2.25× on the Knights Corner (KNC) MIC, and 5.98× on the Kepler GPU. The comparison across different platforms shows that when using the high-end Haswell CPU as the baseline, the KNC MIC is 1.63× faster while the high-end Kepler GPU is 2.20× faster.*

Keywords — *XSBench, MIC (Xeon Phi), GPU.*

Note — *Some figures may be in color only in the electronic version.*

I. INTRODUCTION

Recent years have seen an increased application of accelerators in high-performance computing (HPC). Two mainstream accelerators are many integrated core coprocessors (MICs) (also known as Xeon Phi) by Intel and graphics processing units (GPUs) by Nvidia. Because of their high-energy efficiency (compute performance per watt), MICs and GPUs are being utilized in the next-generation supercomputers at several national laboratories in the United States. For example, Trinity at Los Alamos National Laboratory, Cori at National Energy Research Scientific Computing Center, and Aurora at Argonne National Laboratory (ANL) will be based on Intel's Knights Landing and Knights Hill MICs, while Summit at Oak Ridge National Laboratory and Sierra at Lawrence Livermore National Laboratory will use Nvidia's future-generation Volta GPUs.

In nuclear engineering there has been a strong interest in applying accelerators to reduce the time required to perform Monte Carlo radiation transport simulations for reactor criticality problems. Several studies have evaluated the performance of GPUs on neutron transport.[1–3] The transport methods implemented in these studies range from simple history-based and vectorized one-energy models to vectorized continuous-energy models. Recently, OpenMC has been accelerated on MICs for a full-core pressurized water reactor transport simulation.[4]

Accelerator-based heterogeneous architecture is considered one of the candidates for exascale systems that are expected to arrive in the beginning of the next decade. To maximize the efficient use of these proposed computing systems, the U.S. Department of Energy (DOE) has launched a co-design initiative to engage the software and hardware developers in a collaborative effort. To facilitate that effort, DOE has defined a group of proxy applications as the groundwork for two-way communication between software and hardware developers.[5] The proxy applications aim to encapsulate the performance characteristics of sev-

*E-mail: liut10@rpi.edu

eral scientific computing applications while providing a simpler code base that is easier to analyze.[5]

One such proxy application called XSBench was developed by Tramm et al.[7] at ANL. XSBench represents the most time-consuming computation task within Monte Carlo neutron transport, i.e., the construction of the macroscopic cross section, which accounts for 33% or more of the total computing time in MCNP (Ref. 8) and 85% in OpenMC (Refs. 6 and 9). The original XSBench is a parallel OpenMP code written in C for the traditional multicore central processing units (CPUs). It facilitates the performance study of CPU-only systems by collecting bandwidth, floating-point operations per second (FLOPS), and scalability metrics.[6]

There has been only one study on optimizing XSBench to the accelerator platforms, conducted by Scudiero at Nvidia.[10] Several important tuning techniques including loading outside inner loop, using LDG intrinsics, unrolling outer loop, and fuel sort were introduced to enhance the performance of the GPU-based XSBench code. These methods are described in Sec. II.D. It was reported that the code on a K40 GPU was approximately 5.6 times faster than that on a ten-core Ivy Bridge CPU.

This technical note is the first to optimize XSBench to the MIC and CPU platforms and thereby to make a fair comparison among CPU, GPU, and MIC—all computing devices are evolving rapidly in HPC nowadays. The optimization effort is mainly focused on the methods of hiding memory latency. For completeness, this technical note also describes how the code was ported to the GPU according to Ref. 10. In addition, we summarize our preliminary test of the hash-based energy lookup algorithm[9] on the MIC device.

II. MATERIALS AND METHODS

This section describes the basic data structure and algorithm used in XSBench as well as the optimization techniques for the CPU, MIC, and GPU platforms.

II.A. The Original XSBench Algorithm

There are three data categories in XSBench,[6] shown in Fig. 1: nuclide grid, unionized grid, and material data.

First, the nuclide grid is an array of structures (AOS). It has $N_E \cdot N_n$ elements, where N_E is the average number of energy grid points per nuclide and N_n is the number of nuclides. Each element structure is composed of five contiguous double-precision data values: the total cross section σ_t, the scattering cross section σ_s, the absorption cross section σ_a, the fission cross section σ_f, and the average number of fission neutrons ν. For a certain nuclide, the belonging elements are sorted in ascending order of the energy grid points. There is an additional array of pointers with N_n elements. They store the address of the first element (i.e., nuclide grid point with lowest energy) for each nuclide. This array is used to conveniently access nuclide grid elements given the nuclide index and energy index.

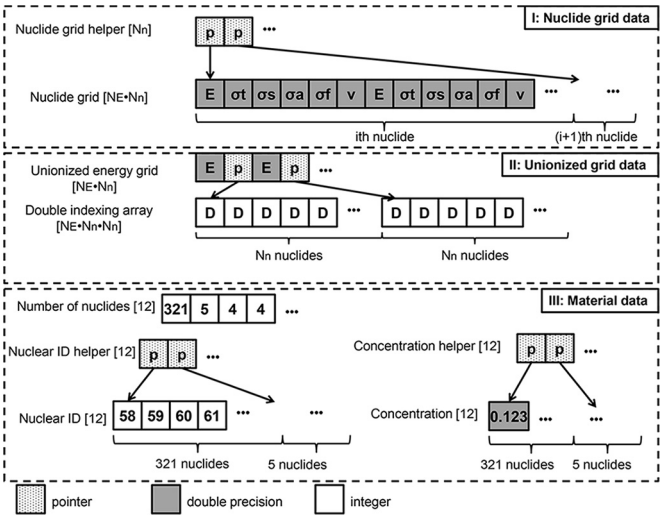

Fig. 1. Diagram of XSBench data structure.

Second, the unionized grid is also an AOS with a length of $N_E \cdot N_n$. It is a result of combining and reordering all the energy grid points from all the nuclides in the nuclide grid. Each element structure has two components: a double-precision energy grid point and a pointer. For each energy grid point, the paired pointer points to an index array, which provides the indices of that energy on the nuclide grid for all the nuclides.[11] This data layout enables a fast cross-section construction algorithm called double indexing.[12] All the index arrays are placed contiguously in the linear memory. In total, the index array has $N_E \cdot N_n \cdot N_n$ elements.

Third, the material data consist of several arrays, including the number of nuclides each material contains, the identifier (ID) of each nuclide in that material, the corresponding concentration, and the array of pointers designed to facilitate the access to the nuclide ID and concentration arrays.

At runtime, as a preprocessing step, XSBench generates the nuclide grid using pseudorandom numbers. The code then performs the computational kernel: macroscopic cross-section construction (also called lookup). The steps of performing a single instance of construction are listed in Algorithm 1 in the Appendix. The code is parallel where multiple OpenMP threads execute their own instances of Algorithm 1 concurrently. To make it clear, different threads are not used in a single evaluation of Algorithm 1. The energy and material data are randomly chosen, followed by a single binary search procedure in the unionized energy grid to find the index i_E

of the given energy E (step 1). Then, the program iterates over all the nuclides contained in the given material. Each iteration is composed of three steps. First (steps 3, 4, and 5), the nuclide ID i_n and concentration are loaded from the system main memory; the locations of the lower energy $N[0]$ and higher energy $N[6]$ are determined based on i_n and i_E using the double-indexing method.[12] Second (steps 6 through 13), the cross-section data are loaded from the system main memory, and the microscopic cross section at energy E is calculated by interpolation. Third (steps 14 through 18), the per-nuclide cross section is weighted by the concentration, and the result is accumulated.

By design, XSBench supports several cross-section construction problems. The major difference among these problems is the number of nuclides in the fuel and the average number of energy grid points per nuclide and thus the memory space required to hold these data. In this technical note we focus on the cross-section construction for the Hoogenboom-Martin (H-M) large problem that entails 321 nuclides in the fuel, 11 303 energy grid points per nuclide, and ~5.7-Gbyte memory usage.[13]

As an aside, XSBench focuses on the performance study of the cross-section construction process and ignores particle tracking, reaction simulation, and tally. It thus generates cross-section data using random numbers for practical convenience instead of using actual nuclear data libraries. Some other data are also artificially generated, including the energy and material concentration data. The rest come from the actual H-M large problem.

II.B. Tuning XSBench to the MIC Platform

The coprocessor used in this study is Intel's first-generation MIC, also known as Knights Corner (KNC). It is built upon the legacy Pentium cores. Each core has a 32-kbyte L1 data cache and a 512-kbyte L2 cache. The L2 cache is coherent across the 60 cores[14] and is inclusive of the data in the L1 cache. In addition, each core supports 4 hardware threads and has 32 entries of 512-bit-long vector registers per hardware thread. Our optimization techniques are described next.

II.B.1. Prefetch

The KNC core implements the in-order execution. This indicates that the processor is prone to stall when the data operands are unavailable and are being fetched from the MIC's on-board dynamic random access memory (DRAM). It is often recommended[15] that more than one thread be launched per core to help hide the memory access latency. This method, however, is not enough for XSBench as the memory load operations are so frequent that the hardware threads may all encounter memory stall.

To overcome this problem, we adopted the software-based data prefetch.[16] In simplest terms, this technique loads the data from the far memory (DRAM) to the near memory (cache) ahead of the computation where the data will be used. More concretely, when in the i'th iteration of a for-loop, one may issue a nonblocking, prefetch request to load the data to the cache that will be used later in the j'th ($j \geq i$) iteration. The difference $j - i$ is called prefetch distance. It has a direct impact on code performance. If the distance is too short, the j'th iteration may be reached too soon before the data are ready and the memory stall still exists. If the distance is too long, the cache residency time increases, and the effective cache size is reduced.[17] Very often in this case, the prefetched data may evict previous useful data or be evicted by the subsequent data from the cache. On the MIC (as well as most of the modern CPUs), each prefetch is able to load a 64-byte chunk of data to the cache. In our code, data prefetch is implemented using the intrinsic function in the C language. The interface is `_mm_prefetch(addr, type)`, where `addr` is the starting address of the data and `type` is the prefetch type. This interface is applicable to both the MICs and CPUs.

Another essential factor to be considered is the prefetch type. The MICs allow data to reside in the cache in various ways, as summarized in Table I. The type of prefetch is determined by three factors: location, temporality, and exclusiveness. Location includes L1 (because L2 is an inclusive cache, prefetch to L1 implies to L2) and L2 cache only. Temporality refers to whether to retain or evict the data upon the first use. If the data are evicted, the prefetch type is called nontemporal access (NTA).

TABLE I

Common Prefetch Types on the MIC and CPU*

Processor	Mnemonic	Purpose
MIC	NTA	Load data to L1 and L2 cache; mark it as NTA
	T0	Load data to L1 and L2 cache
	T1	Load data to L2 cache only
	T2	Load data to L2 cache only; mark it as NTA
	ENTA	Exclusive version of NTA
	ET0	Exclusive version of T0
	ET1	Exclusive version of T1
	ET2	Exclusive version of T2
CPU	NTA	Load data to L2 and L3 cache; mark it as NTA
	T0	Load data to L2 and L3 cache

*References 14, 31, and 32.

Exclusiveness refers to the act of invalidating the same data that appear in other levels of caches or caches of other cores. We empirically set the prefetch distance and type to 1 and T2, respectively.

Meanwhile, we manually performed loop unrolling, a well-known method to improve instruction-level parallelism. "Unrolling the loop by 2" means that the two adjacent iterations are combined into one and the cross sections of two nuclides of the same material are constructed together. In our test, it was determined that unrolling four loops produced the optimal performance.

II.B.2. Vectorization

On the MIC coprocessor, a total of eight double-precision data values can be manipulated in parallel through the 512-bit vector registers. Although XSBench is primarily memory bound, it was found that vectorizing the cross-section calculation still noticeably improves the performance. The vectorization is implemented using MIC-specific intrinsic functions. To use these functions, it is required that the data be aligned to 64-byte memory addresses. For the nuclide grid that is an AOS, this means both the initial memory address and the address of every element structure in the array should be multiples of 64. Therefore, each element structure was padded with two dummy double-precision data.

The optimized algorithm is illustrated in Algorithm 2. First, preparations are made to determine the address from which the data need to be prefetched (steps 3 through 6). Then, the actual prefetch operation is performed. For a specific nuclide, we prefetch two 64-byte blocks, corresponding to the eight double-precision data for the lower energy (six actual nuclear data plus two dummy data) and those for the higher energy. Because the MIC's cache line size is also equal to 64 bytes, one instance of prefetch suffices to give us all the nuclear data for a certain energy. It follows that when two nuclides are handled at the same time, a total of four prefetch requests are needed (steps 7 through 10). In the interpolation step (steps 13, 14, and 15), vector operations are used to calculate five cross sections in parallel. To increase the arithmetic throughput, step 15 is performed using the Fuse Multiply Add (FMA) instruction to combine two separate instructions—the addition and multiplication—into one. For simplicity, some conditional statements used to handle special cases are not shown in Algorithm 2. These cases include, for instance, (1) only one isotope to process for the current loop; (2) only one isotope to prefetch for the next loop; (3) two isotopes to process for the current loop and one isotope to prefetch for the next loop. The conditional statements avoid false prefetch, which may pollute cache and incur performance penalty.

II.B.3. Other Technical Considerations

First, the OpenMP's thread affinity was set to the "balanced" pattern, whereby 240 threads are evenly distributed among the 60 cores and every four threads with consecutive IDs are bound to the four hardware threads (i.e., logical cores) on the same physical core. This improves cache utilization because otherwise the hardware is free to migrate threads across the cores and cause cache misses. Second, the 2-Mbyte huge page feature was used, reducing translation lookaside buffer (TLB) miss. This feature is currently turned on by default by the MIC's on-board operating system (OS) as well as our host OS.

II.C. Tuning XSBench to the CPU Platform

In our laboratory we have two CPUs, a 6-core CPU based on the Westmere microarchitecture and a 14-core CPU based on the Haswell microarchitecture. The cores of both CPUs implement out-of-order execution. When a memory stall occurs, the processor may proceed with other instructions that are not dependent on the data being loaded. This hardware mechanism relaxes the memory limitations but cannot replace the software tuning. The methods to optimize XSBench on the MIC are all applicable to the CPU platform with a couple of minor differences.

First, the prefetch distance and type are different from MIC. On the CPU platform, the performance appeared less sensitive to these two factors, shown in Sec. III.B. We chose 13 and NTA, respectively, in our test. The optimal loop unrolling level was found to be 6. It should be pointed out that the prefetch types of Xeon CPUs are distinct from the MICs'. The data cannot be prefetch directly to L1 cache. Besides, although types T1 and T2 do exist, they are in fact equivalent to T0. Second, the single instruction, multiple data (SIMD) instructions are different from MIC. On the Westmere CPU, the Streaming SIMD Extensions 4 (SSE4) instructions operate on 128-bit registers (two double-precision data) at a time, while on the Haswell CPU, the Advanced Vector Extensions 2 (AVX2) instructions operate on 256-bit registers (four double-precision data).

As a general remark, the prefetch and vectorization methods can potentially be applied to other time-consuming subroutines of a Monte Carlo program beyond cross-section construction alone. The requirement is that these subroutines should have similar structure to cross-section construction: tight for-loop (i.e., loop with relatively

few instructions in it, so that the prefetched data still remain in the cache when they are needed by the processor in subsequent iterations), sufficient number of iterations (so that the prefetch distance can be increased to the optimal value), and vectorizable operations (so that they can be combined into single instructions).

II.D. Tuning XSBench to the GPU Platform

II.D.1. Prefetch

The GPUs adopt warps—a mechanism different from the MICs and CPUs—to hide memory latency. A warp is composed of 32 threads. It implements the Single Instruction, Multiple Thread (SIMT) model, where all the threads within a warp execute the same instruction. The branches caused by conditional statements such as `if-else` are gracefully handled by the hardware at the cost of instruction replay. On a GPU, hundreds of warps can be launched together. When one warp encounters memory stall, other warps may take over the hardware and be processed. We followed Ref. 10 to fetch data more efficiently for each warp. According to the loading outside inner loop method,[10] for a specific nuclide, all of the 12 double-precision data (6 for the lower energy, 6 for the higher energy, and no padding applied) are fetched from the GPU DRAM to the register before computation, using the large data type `double2`. Furthermore, according to the unrolling outer loop method,[10] loop unrolling is applied such that every iteration handles two nuclides. Thus, a total of 24 double-precision data are fetched before computation. This technique applies to the data that will be used by the computation in the same iteration and can be seen as a form of data prefetch with zero distance. Besides, we tested the actual GPU prefetch instructions with positive prefetch distance. The prefetch is implemented as Parallel Thread Execution[18] (PTX)—GPUs' intermediate programming language. The interface is asm `volatile("prefetch.global.[type] [%0];"::"l"(addr))`. However, a performance degradation by up to ~50% was observed. The preliminary result and analysis are given in Sec. III.C. In addition, we did not use the fuel sort method mentioned in Ref. 10. This method moves the cross-section construction for the fuel material to the first warp of a thread block for divergence reduction using the GPU shared memory as the buffer.[19,20] Further investigation is still needed to see whether and how this method can be applied to the traditional history-based Monte Carlo transport code, where a certain particle is always attached to one thread regardless of what material it may enter, and frequently swapping all the particle attribute data between threads may be costly to perform.

II.D.2. Cache

The GPUs have two levels of cache: a per-SM L1 cache whose size is configurable and a 1536-kbyte global L2 cache.[21] In addition, the K40 GPU has a separate, per-SM texture cache for read-only data. By default, however, the K40 GPU only uses L2 for load operations on the global memory.[22] We took the following steps to make the most of GPU caches. (1) According to the "using LDG intrinsics" method,[10] the texture cache is enabled—by using the compiler intrinsic `__ldg()`—to cache the load of cross-section data. (2) L1 is enabled to cache the load of all other data by applying `-dlcm=ca` flag to the compiler. This step was also taken by Scudiero,[20] although not explicitly mentioned in Ref. 10. (3) L1 is expanded to 48 kbytes through the CUDA runtime application programming interface[23] (API).

II.D.3. Execution Configuration

To perform N lookups for XSBench (or more generally to simulate N particles for a Monte Carlo code) on a GPU, one needs to properly distribute the task by selecting the number of threads per block T_B and the number of blocks per grid b. This procedure is called execution configuration. To select T_B we followed the common practice: Query the compilation statistics and obtain the number of registers each thread will consume and then search an Excel spreadsheet developed by Nvidia called CUDA occupancy calculator[24] for a T_B value that maximizes the GPU occupancy.

To select b, a common way is to let each thread handle one task, i.e., $t = N$, where t is the total number of threads to be launched on the GPU. As a result, $b = \frac{t}{T_B} = \frac{N}{T_B}$. However, this approach has the disadvantage of large cumulative thread overhead. The solution is to perform more than one task per thread, and consequently, $b = \frac{N}{T_B \cdot M}$, where M is the number of tasks per thread. The value M should be carefully determined. If it is too large, the blocks of threads launched will be too small and underutilize the GPU resource throughout the run time. On the other hand, if it is too small, the GPU will suffer the tail effect.[25] To simplify the task distribution and improve load balancing, we adopted the persistent thread method.[26] The key concept is that b is now also chosen from the CUDA occupancy calculator such that the total t threads saturate the GPU resource exactly. All the threads are launched at the same time and persist throughout the GPU program. They are effectively treated as if they are physical hardware threads.[26] The convenience is that $M = \frac{N}{t}$, which is identical to the task distribution approach on the CPU or MIC. Using the persistent thread method, it can be determined that $b = 15$ and $T_B = 512$, so

TABLE II

Hardware Specifications

Processor	Microarchitecture	Core Count	Base Clock (GHz)	Price (US$)	Launch Date (quarter-year)
Intel X5650	Westmere	6 (12 hyperthreads)	2.66	996	1-2010
Intel E5-2697 v3	Haswell	14 (28 hyperthreads)	2.6	2702	3-2014
Intel 5110P	KNC	60 (240 hyperthreads)	1.053	2437	4-2012
Nvidia K40	Kepler	15 SMs (2880 SPs)	0.745	3300	4-2013

that at runtime the total number of resident threads for the GPU-based XSBench code is $t = bT_B = 7680$.

II.E. Programming Environment and Hardware Specifications

The compiler used for the CPU, MIC, and GPU host code was Intel icc version 15.0.6. The compiler for the GPU device code was nvcc version 6.5. All the codes were compiled with `-O1` optimization level. Any compiler option that is likely to trade accuracy for performance was avoided. The parallel Xorshift pseudorandom number generator ported from Nvidia's cuRAND library[27] was used in the parallel cross-section construction kernel.

The hardware specifications are listed in Table II. On the CPU, hyperthreading was enabled. On the MIC, each physical core by default permits 4 hardware threads, so a total of 240 threads was used. On the GPU, the error-correcting code (ECC) function was turned on, and the GPU Boost function was enabled, which can dynamically overclock the device as long as the temperature and power allow. It is worth noting that the HPC Top-500 ranking has been using the number of streaming multiprocessors (SMs) as the core count.[28] This is more reasonable and less misleading than quoting the number of streaming processors (SPs) only, given that the functionality of a GPU SP is not comparable to that of a CPU core.

III. RESULTS AND DISCUSSIONS

III.A. Compute Performance

The performance of XSBench on different computing platforms is summarized in Table III. On each platform, the performance of the tuned code is better than the original implementation. The speedup factor was found to be 1.24× on the Westmere CPU, 1.53× on the Haswell CPU, 2.31× on the KNC MIC, and 5.98× on the Kepler GPU. The in-order MIC cores are more sensitive to the memory stall than the out-of-order CPU cores, so the effect of data prefetch is more remarkable on the MIC. The tuned GPU code benefits in part from the expanded 48-kbyte L1 cache. If the size is kept as the default 16 kbytes, the performance gain over the original code on the GPU will reduce from 5.98× to 2.97×.

Reference 10 reports that the GPU code performed 11 955 176 lookups/s on the K40 GPU. Considering that Ref. 10 applies an additional fuel sort method not adopted in our study and that this method contributes to <10% performance improvement, our result is in good agreement with those reported in Ref. 10 for the tuned GPU program. Furthermore, Ref. 10 reports 1 321 943 lookups/s on a six-core Sandy Bridge CPU with hyperthreading enabled. This is 18% slower than our tuned code on the one-generation-older Westmere CPU. It is believed that in their study the CPU code was likely not optimized.

To make a more reasonable and fair comparison in this study, the speedup factor of the tuned code on one platform over the other is reported in Table IV. The code on the state-of-the-art Haswell CPU has significantly better performance than the three-generations-older Westmere CPU. The KNC MIC is surprisingly 63% faster than the Haswell CPU. The Kepler GPU appears to be the fastest computing device in our test, having a 35% edge over the MIC.

TABLE III

Performance of XSBench for H-M Large Problem

Processor	Code	Performance (10^6 lookups/s)
Westmere CPU	Original	1.26
	Tuned	1.83
Haswell CPU	Original	3.07
	Tuned	4.64
KNC MIC	Original	3.38
	Tuned	7.58
Kepler GPU	Original	1.71
	Tuned	10.2

TABLE IV

Speedup Factors of the Tuned Codes

Processor	Speedup		
Westmere CPU	Baseline		
Haswell CPU	2.54×	Baseline	
KNC MIC	4.14×	1.63×	Baseline
Kepler GPU	5.59×	2.20×	1.35×

III.B. Performance Study of the MIC and CPU

Figure 2 shows how the performance varies with the prefetch distance and type. A CPU core has excellent single-thread performance due to its high frequency and out-of-order execution mechanism. Prefetching multiple iterations ahead of computation thus reduces unnecessary cache residency time and improves latency hiding. The performance in this study appears somewhat independent of the prefetch distance and type when the former parameter exceeds 5. In contrast, on the MIC, prefetching just for the next iteration is opportune. Any greater prefetch distance results in performance degradation. Prefetching to the coherent L2 cache alone brings the best performance, regardless of the temporality and exclusiveness of the data, while prefetching to L1 slows the code in all cases, suggesting a larger overhead involved in this process.

The optimization techniques reported here have different impact levels on MIC's performance, as seen from Table V. Here, we used the fully optimized code as the baseline (suppose the performance is P_0 in lookups per second). We switched off each type of optimization individually (suppose the performance now becomes P_i) and calculated the value $(P_0 - P_i)/P_0 \times 100\%$. The default huge page feature offered by the recent OS on the MIC was found useful in reducing the TLB misses. The results suggest that among all software-based approaches, data prefetch stands out as the most efficient on the MIC.

III.C. Performance Study of the GPU

We experimented with PTX-based prefetch on the GPU with different prefetch distance and type but observed performance degradation to varying degrees, as shown in Fig. 3. One simple explanation[29] is that the GPU has thousands of in-flight threads and that the overhead of thousands of outstanding memory requests may offset the advantage of prefetch. Additionally, the cache is a scarce resource on the GPU. The cache size per thread is significantly lower than that of the CPU and MIC (Refs. 19 and 20). Consequently, the GPU is more prone to cache pollution. Profiling on the cache behavior can shed light on this problem, as suggested by Refs. 19 and 20. Figure 3 illustrates part of the profiling results. When the data are prefetched to L2

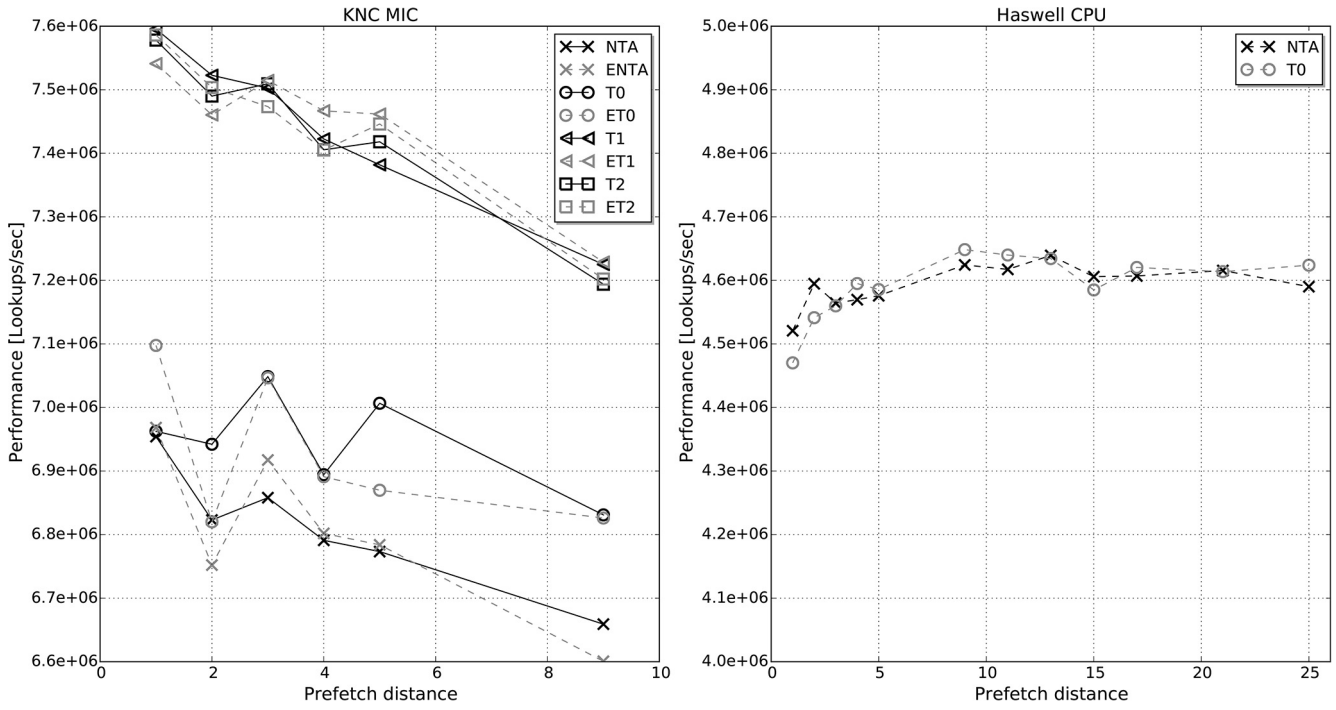

Fig. 2. Impact of prefetch distance and type over performance on the KNC MIC and Haswell CPU.

TABLE V

Impact of Different Optimization Techniques on the MIC

Optimization	Performance Difference	Reason for Performance Improvement
Data prefetch	43.0%	Reduce cache misses
Loop unrolling	29.8%	Improve instruction level parallelism
Huge page	51.2%	Reduce TLB misses
Vectorization	11.0%	SIMD
Thread affinity	6.7%	Reduce cache misses

cache only, as the prefetch distance increases, more useful data are evicted from L2, and the hit rate at L2 for all read requests from L1 becomes lower, causing more accesses to the long-latency DRAM. When the data are prefetched to both the L1 and L2 cache, the same problem occurs, and useful data are also evicted from the L1 cache in addition, making the performance even worse. It should be pointed out that some of the profiling results are not fully understood. For instance, for the L2-only case, a slight increase in the L1 hit rate was observed. More detailed quantitative analysis needs to be done in the future.

III.D. Preliminary Test of Hash-Based Energy Lookup on the MIC

The hash-based energy lookup method[9] currently implemented in MCNP 6.1.1 (Ref. 30) is a memory-saving alternative to the double-indexing method used in XSBench. For the H-M large problem, in XSBench the unionized energy grid and its associated index array require ~5.4 Gbytes of memory, while in MCNP a much smaller index array is needed (~10.8 Mbytes). The trade-off is the performance, specifically the cost of performing additional binary search operations for each nuclide of a material. We adapted XSBench to the hash-based energy lookup method. The profiling results of the CPU code indicate that the binary search takes up ~36% of the loop in time.

To see how this change affects the optimization effectiveness, we did the following preliminary test: data prefetch distance 0, type T2, loop unrolling level 2, vectorization enabled. The result is shown in Table VI. On the MIC the performance improvement is only ~12%, much less than that of XSBench. The reason is the binary search inevitably causes cache pollution and interferes with data prefetch.

We also experimented data prefetch directly on the binary search itself. The while loop of the binary search contains a branch structure that determines whether to raise the lower bound or lower the upper bound, depending on whether the given energy is greater than the energy grid point at the mid index. Access to the energy grid point causes memory latency. We therefore added two prefetch operations immediately before the branch structure to prefetch two energy grid points for the next iteration, corresponding to two possible cases where the lower bound or upper bound becomes the mid index in the next iteration. Because only one 8-byte energy data out of two 64-byte cache lines will be actually used, this method puts more pressure on the cache. It was found that with this method alone, the code became faster by ~6%. However, this method and the above optimizations do not add up. When used together, a slight degradation of ~3% was even observed. Tuning the hash-based energy lookup appears to be a challenge on the MIC.

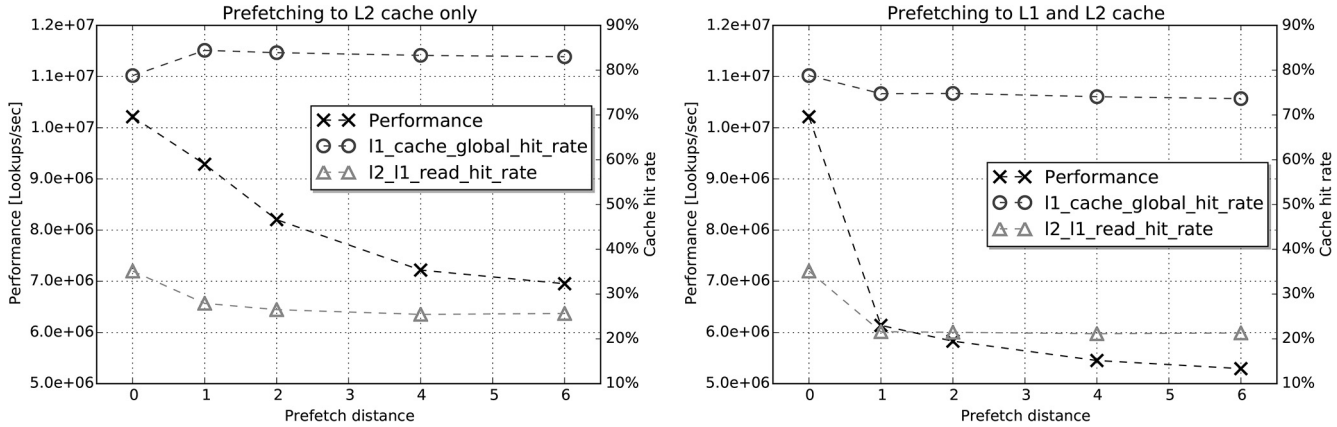

Fig. 3. Performance degradation of the GPU code is observed when using PTX-based prefetch. Three sets of data are shown here as a function of the prefetch distance: the performance (lookups per second), the hit rate at L1 cache for global loads, and the hit rate at L2 cache for all read requests from L1 cache.

TABLE VI

Performance of XSBench Adapted to the Hash-Based Energy Lookup Algorithm

Processor	Code	Performance (10^6 lookups/s)
Westmere CPU	Original	0.768
KNC	Original	2.82
	Tuned	3.16

IV. CONCLUSIONS

We have conducted an in-depth optimization study on the proxy neutronics application XSBench for three hardware platforms: CPU, MIC, and GPU. The following conclusions can be drawn. (1) Data prefetch is crucial to all the platforms, including the CPU. The cross-section construction problem is memory bound. One has a uniform optimization goal on different platforms—to hide the memory access latency. Our results suggest that data prefetch is an effective technique to achieve this goal. (2) To fully exploit the hardware, manual tuning must be performed. In general, both the GPU and MIC provide good API supports that make direct porting straightforward, but the performance gain may be disappointingly marginal. A decent improvement can be enabled by platform-specific code tuning. Compared to the original code, the optimized XSBench is found to be 1.51× faster on the CPU, 2.25× faster on the MIC, and 5.98× faster on the GPU. (3) The Intel KNC MIC and high-end Nvidia Kepler GPU outperform the high-end Intel Haswell CPU. The difference, however, is not as large as we had expected. The speedup factors are 1.63× and 2.20×, respectively.

APPENDIX

ALGORITHMS

Algorithm 1: Original XSBench Algorithm

input : energy E, material index i_{mat}
output: 5 macroscopic cross-sections $\Sigma[5]$

find index i_E of E in unionized energy grid using binary search
for $j \leftarrow 0$ **to** $p-1$ **do** // p--number of isotopes in i_{mat}th material
  // load arrays in main memory to local var
  $i_n \leftarrow M[i_{mat}, j]$ // M--material array
  $c \leftarrow C[i_{mat}, j]$ // C--concentration array
  find the location of N[0] and N[6] according to i_n and i_E
  // calculate macroscopic cross-section by interpolation
  $E_{low} \leftarrow N[0]$ // load energy from main memory
  $E_{high} \leftarrow N[6]$ // load energy from main memory
  $f \leftarrow \frac{E_{high}-E}{E_{high}-E_{low}}$ // calculate interpolation coefficient f
  $\sigma[0] \leftarrow N[7] - f \cdot (N[7] - N[1])$ // calculate σ_t
  $\sigma[1] \leftarrow N[8] - f \cdot (N[8] - N[2])$ // calculate σ_s
  $\sigma[2] \leftarrow N[9] - f \cdot (N[9] - N[3])$ // calculate σ_a
  $\sigma[3] \leftarrow N[10] - f \cdot (N[10] - N[4])$ // calculate σ_f
  $\sigma[4] \leftarrow N[11] - f \cdot (N[11] - N[5])$ // calculate ν
  // accumulate per-nuclide result
  $\Sigma[0] \leftarrow \Sigma[0] + \sigma[0] \cdot c$
  $\Sigma[1] \leftarrow \Sigma[1] + \sigma[1] \cdot c$
  $\Sigma[2] \leftarrow \Sigma[2] + \sigma[2] \cdot c$
  $\Sigma[3] \leftarrow \Sigma[3] + \sigma[3] \cdot c$
  $\Sigma[4] \leftarrow \Sigma[4] + \sigma[4] \cdot c$
  $j \leftarrow j + 1$ // increment counter
end

Algorithm 2: Optimized Algorithm for MICs and CPUs

For illustration purposes, here, the loop unrolling level is set to 2, and the additional conditional statements used to handle special cases and avoid false prefetch are not shown. The optimizations are concentrated on reducing memory latency and increasing vectorization intensity. The underlying double-indexing search method remains the same with the original XSBench.

input : energy E, material index i_{mat}
output: 5 macroscopic cross-sections $\Sigma[5]$

```
1  ...
2  for j ← 0 to p − 1 do
       // make preparation for prefetch
3      i_n ← M[i_mat, j + d] // for (j + d)th nuclide
4      find the location of N[0] and N[8] according to i_n and i_E
5      i_n ← M[i_mat, j + 1 + d] // for (j + 1 + d)th nuclide
6      find the location of N[0] and N[8] according to i_n and i_E

       // perform the actual prefetch operation
7      prefetch N[0] ∼ N[7] for (j + d)th nuclide
8      prefetch N[8] ∼ N[15] for (j + d)th nuclide
9      prefetch N[0] ∼ N[7] for (j + 1 + d)th nuclide
10     prefetch N[8] ∼ N[15] for (j + 1 + d)th nuclide

       // load arrays from main memory or cache for jth nuclide
11     ...
       // repeat for j + 1th nuclide

       // calculate macroscopic cross-section by interpolation for
          jth nuclide
12     E_low[0 : 7] ← N[0] // build vector
13     E_high[0 : 7] ← N[8]
14     f[0 : 7] ← (E_high[0:7] − E[0:7]) / (E_high[0:7] − E_low[0:7])
15     σ[0 : 7] ← N[8 : 15] − f[0 : 7] · (N[8 : 15] − N[0 : 7])
       // repeat for j + 1th nuclide

       // accumulate per-nuclide result for jth nuclide
16     Σ[0 : 4] ← Σ[0 : 4] + σ[1 : 5] · c
       // repeat for j + 1th nuclide

17     j ← j + 2 // increment counter by 2 due to loop unrolling
18 end
```


Acknowledgments

We thank T. Scudiero at Nvidia for the informative discussion on the XSBench GPU optimizations. This study was performed using hardware donated by Nvidia and Intel.

References

1. A. G. NELSON, "Monte Carlo Methods for Neutron Transport on Graphics Processing Units Using CUDA," PhD Thesis, The Pennsylvania State University (2009).

2. T. LIU et al., "A Comparative Study of History-Based Versus Vectorized Monte Carlo Methods in the GPU/CUDA Environment for a Simple Neutron Eigenvalue Problem," *Proc. Joint Int. Conf. Supercomputing in Nuclear Applications (SNA) and the Monte Carlo (MC) Method (SNA + MC 2013)*, Paris, France, October 27–31, 2013, paper 04206, EDP Sciences (2014).
3. R. M. BERGMANN and J. L. VUJIĆ, "Algorithmic Choices in WARP—A Framework for Continuous Energy Monte Carlo Neutron Transport in General 3D Geometries on GPUs," *Ann. Nucl. Energy*, **77**, 176 (2015); http://dx.doi.org/10.1016/j.anucene.2014.10.039.
4. D. OZOG et al., "Full-Core PWR Transport Simulations on Xeon Phi Clusters," *Proc. Joint Int. Conf. Mathematics and Computation (M&C), Supercomputing in Nuclear Applications (SNA) and the Monte Carlo (MC) Method*, Nashville, Tennessee, April 19–23, 2015, American Nuclear Society (2015).
5. M. HEROUX, R. NEELY, and S. SWAMINARAYAN, "ASC Co-Design Proxy App Strategy," LA-UR-13-20460, LLNL-TR-592878, Sandia National Laboratories, Lawrence Livermore National Laboratory, Los Alamos National Laboratory.
6. J. TRAMM and A. R. SIEGEL, "Memory Bottlenecks and Memory Contention in Multi-Core Monte Carlo Transport Codes," *Proc. Joint Int. Conf. Supercomputing in Nuclear Applications (SNA) and the Monte Carlo (MC) Method (SNA + MC 2013)*, Paris, France, October 27–31, 2013, paper 04206, EDP Sciences (2014).
7. J. R. TRAMM et al., "XSBench—The Development and Verification of a Performance Abstraction for Monte Carlo Reactor Analysis," *Proc. PHYSOR 2014—The Role of Reactor Physics Toward a Sustainable Future*, Kyoto, Japan, September 28–October 3, 2014.
8. D. B. PELOWITZ et al., "MCNP6 User's Manual Version 1.0," Los Alamos National Laboratory (2013).
9. F. B. BROWN, "New Hash-Based Energy Lookup Algorithm for Monte Carlo Codes," *Trans. Am. Nucl. Soc.*, **111**, 659 (2014).
10. T. SCUDIERO, "Monte Carlo Neutron Transport: Simulating Nuclear Reactions One Neutron at a Time," *Proc. GPU Technology Conf. (GTC) 2014*, San Jose, California, March 24–27, 2013, Nvidia (2014).
11. P. ROMANO et al., "Release Notes for OpenMC 0.4.0" (2014); http://mit-crpg.github.io/openmc/ (current as of Jan. 31, 2016).
12. J. LEPPÄNEN, "Two Practical Methods for Unionized Energy Grid Construction in Continuous-Energy Monte Carlo Neutron Transport Calculation," *Ann. Nucl. Energy*, **36**, 7, 878 (2009); http://dx.doi.org/10.1016/j.anucene.2009.03.019.
13. J. TRAMM, "XSBench: The Monte Carlo Macroscopic Cross Section Lookup Benchmark" (2014); https://github.com/ANL-CESAR/XSBench (current as of Jan. 31, 2016).
14. "Intel Xeon Phi Coprocessor System Software Developer's Guide," Intel (2014).
15. M. BARTH et al., "Best Practice Guide Intel Xeon Phi v1.1" (2014); http://www.prace-ri.eu/best-practice-guide-intel-xeon-phi-html/ (current as of Jan. 31, 2016).
16. T. C. MOWRY, M. S. LAM, and A. GUPTA, "Design and Evaluation of a Compiler Algorithm for Prefetching," *ACM Sigplan Notices*, **27**, 62 (1992); http://dx.doi.org/10.1145/143371.143488.
17. H. G. CRAGON, *Memory Systems and Pipelined Processors*, Jones & Bartlett Learning, Boston, Massachusetts (1996).
18. "Parallel Thread Execution ISA Version 4.1," Nvidia (2014).
19. T. SCUDIERO, "Memory Bandwidth Bootcamp: Beyond Best Practices," *Proc. GPU Technology Conf. (GTC) 2015*, Silicon Valley, California, March 17–20, 2015, Nvidia (2015).
20. T. SCUDIERO, Personal Communication, Nvidia (2015).
21. "Whitepaper—Nvidia's Next Generation CUDA Compute Architecture: Kepler GK110/210," Nvidia (2014).
22. "Tuning CUDA Applications for Kepler," Nvidia (2014).
23. "Nvidia CUDA Runtime API (CUDA 6.5)," Nvidia (2014).
24. "CUDA Occupancy Calculator," Nvidia (2014).
25. P. MICIKEVICIUS, "GPU Performance Analysis and Optimization," *Proc. GPU Technology Conf. (GTC) 2012*, San Jose, California, March 14–17, 2012, Nvidia (2012).
26. K. GUPTA, J. A. STUART, and J. D. OWENS, "A Study of Persistent Threads Style GPU Programming for GPGPU Workloads," *Innovative Parallel Computing (InPar)*, pp. 1–14, IEEE (2012).
27. "cuRAND Library," Nvidia (2014).
28. H. MEUER et al., "Top 500 List" (Nov. 2014); http://www.top500.org/lists/ (current as of Jan. 31, 2016).
29. J. LEE, H. KIM, and R. VUDUC, "When Prefetching Works, When It Does Not, and Why," *ACM Trans. Archit. Code Optim.*, **9**, *1*, 2 (2012); http://dx.doi.org/10.1145/2133382.2133384.
30. T. GOORLEY, "MCNP 6.1.1—Beta Release Notes," LA-UR-14-24680, Los Alamos National Laboratory (2014).
31. "Intel Xeon Phi Coprocessor Instruction Set Architecture Reference Manual," Intel (2012).
32. "Intel 64 and IA-32 Architectures Software Developers Manual Volume 1: Basic Architecture," Intel (2015).