• Please send the proof to Tianyu Liu  
  email: liut10@rpi.edu  
  phone: 518-428-9567  
  address: 400 McChesney Ave Ext  
  APT 21-5  
  Troy, New York, 12180

• Total number of pages: 31

• Total number of figures: 3

• Total number of tables: 6
Optimizing the Monte Carlo neutron cross-section construction code, XSBench, for MIC and GPU platforms

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

Rensselaer Polytechnic Institute, Troy, New York, 12180, USA

Email address: xug2@rpi.edu (X. George Xu)

Draft by Rensselaer Radiation Measurement and Dosimetry Group (RRMDG)
Abstract

XSBench is a proxy application used to study the performance of nuclear macroscopic cross-section data construction — usually the most time-consuming process in Monte Carlo neutron transport simulations. In this paper we report on our experience in optimizing XSBench to Intel multi-core 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 (H-M) 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 coprocessor 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, CUDA
1. 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. Due to 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 U.S. For example, Trinity at LANL, Cori at NERSC and Aurora at ANL will be based on Intel’s Knights Landing (KNL) and Knights Hill (KHL) MICs, while Summit at ORNL and Sierra at LLNL will use Nvidia’s future-generation Volta GPUs.

In nuclear engineering there has been a strong interest in applying the 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, 2, 3]. The transport methods implemented in these studies range from simple history-based and vectorized one-energy models, to vectorized continuous-energy model. Recently, OpenMC has been accelerated on MICs for a full-core PWR transport simulation [4].

The 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 DOE has launched the “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 a ground for the two-way communication between the software and hardware developers [5]. The proxy applications aim to encapsulate the performance characteristics of several scientific computing applications, while providing a simpler code base that is easier to analyse [5].

One such proxy application called XSBench was developed by Tramm and Siegel at Argonne National Laboratory (ANL) [6, 7]. XSBench represents the most time-
consuming computation task within Monte Carlo neutron transport — the construction of macroscopic cross-section, which accounts for 33% or more of the total computing time in MCNP [8] and 85% in OpenMC [6, 9]. The original XSBench is a parallel OpenMP code written in C for the traditional multi-core CPUs. It facilitates the performance study of CPU-only systems by collecting bandwidth, 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 will be described in section 2.4. It was reported that the code on a K40 GPU was \( \sim 5.6 \) times faster than that on a 10-core Ivy Bridge CPU.

This paper is the first to optimize XSBench to the MIC and CPU platforms, and thereby to make a fair comparison between the 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 paper also describes how the code was ported to the GPU according to Scudiero [10]. In addition, this paper summarizes our preliminary test of the “hash-based energy lookup” algorithm [9] on the MIC device.

2. 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.

2.1. The Original XSBench Algorithm

There are three data categories in XSBench [6], shown in Figure 1: nuclide grid, unionized grid and material data. Firstly, 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 5
contiguous double-precision data value: the total cross-section $\sigma_t$, the scattering cross-section $\sigma_s$, the absorption cross-section $\sigma_a$, the fission cross-section $\sigma_f$ and the average number of fission neutrons $\nu$. 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. Secondly, 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. Thirdly, the material data consist of several arrays, including the number of nuclides each material contains, the 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 pseudo-random 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 the appendix algorithm 1. 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 (step 3 to 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 (step 6 to 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 (step 14 to 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 between 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 paper 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 $\sim 5.7$ GB 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.

2.2. 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-KB L1 data cache and a 512-KB L2 cache. The L2 cache is coherent across the 60 cores [14], and is inclusive of the data in 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.

**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 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 non-blocking, 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 is 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 1. The type of prefetch is determined by 3 factors: the 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 is evicted, the prefetch type is called Non-Temporal Access (NTA). Exclusiveness refers to the act of invalidating the same data that appear in other level of caches or cache 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 4 loops
produced the optimal performance.

**Vectorization.** On the MIC coprocessor, a total of 8 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 which 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 (step 3∼6). Then the actual prefetch operation is performed. For a specific nuclide, we prefetch two 64-byte blocks, corresponding to the 8 double-precision data for the lower energy (6 actual nuclear data plus 2 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 4 prefetch requests are needed (step 7∼10). In the interpolation step (13∼15), vector operations are used to calculate 5 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 1 isotope to process for the current loop, (2) only 1 isotope to prefetch for the next loop, (3) 2 isotopes to process for the current loop and 1 to prefetch for the next loop. The conditional statements avoid false prefetch, which may pollute cache and incur performance penalty.

**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 4 threads with consecutive IDs are bound to the 4 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 MB huge page feature was used, reducing Translation Lookaside Buffer (TLB) miss. This feature is currently turned on by default by the MIC’s onboard OS as well as our host OS.

2.3. 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.

Firstly, the prefetch distance and type are different from MIC. On the CPU platform, the performance appeared less sensitive to these two factors, shown in the next section. 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 MICs’. The data cannot be prefetch directly to L1 cache. Besides, although type T1 and T2 do exist, they are in fact equivalent to T0. Secondly, the SIMD instructions are different from MIC. On the Westmere CPU, the Streaming SIMD Extensions 4 (SSE4) instructions operate on 128-bit registers (2 double-precision data) at a time, while on the Haswell CPU, the Advanced Vector Extensions 2 (AVX2) instructions operate on 256-bit registers (4 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).

2.4. Tuning XSBench to the GPU Platform

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 reference
[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, 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 ap-
plied 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 Exe-
cution (PTX) [18] — GPUs intermediate programming language. The interface is asm
volatile(‘‘prefetch.global.[type] [%0];’’:‘‘1’’(addr)). However, a perfor-
mance degradation by up to ∼50% was observed. The preliminary result and analysis is
given in the next section. In addition, we did not use the “fuel sort” method mentioned
in reference [10]. This method moves the cross-section construction for the fuel mate-
rial 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.

**Cache.** The GPUs have two levels of cache, a per-SM L1 cache whose size is configurable, and a 1536-KB 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 being explicitly mentioned in reference [10]. (3) L1 is expanded to 48 KB through the CUDA runtime-API [23].

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

2.5. Programming Environment and Hardware Specifications

The compiler used for the CPU, MIC and the 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 pseudo-random 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 2. On the CPU, hyperthreading was enabled. On the MIC, each physical core by default permits 4 hardware threads, so a total of 240 threads were used. On the GPU, Error-Correcting Code (ECC) function was turned on, and the turbo function was enabled, which can dynamically overclock the device as long as the temperature and power allows. It is worth noting that the HPC Top-500 ranking has been using the number of SMs as the core count [28]. This is more reasonable and less misleading than quoting the number of SPs only, given that the functionality of a GPU SP is not comparable to that of a CPU core.
3. Results and Discussions

3.1. Compute Performance

The performance of XSBench on different computing platforms is summarized in Table 3. On each platform, the performance of the tuned code is better than the original implementation. The speedup factor was found to be $1.24\times$ on the Westmere CPU, $1.53\times$ on the Haswell CPU, $2.31\times$ on the KNC MIC and $5.98\times$ 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-KB L1 cache. If the size is kept as the default 16 KB, the performance gain over the original code on the GPU will reduce from $5.98\times$ to $2.97\times$.

Reference [10] reported that the GPU code performed 11,955,176 lookups per second on the K40 GPU. Considering that [10] applied an additional “fuel sort” method not adopted in our study, and that this method contributes to less than 10% performance improvement, our result is in good agreement with those reported in [10] for the tuned GPU program. Furthermore, [10] reported 1,321,943 lookups per second on a 6-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 4. 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 35% edge over the MIC.

3.2. 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 5. Here we used the fully optimized code as the baseline (suppose the performance is $P_0$ in lookups/sec). 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.

3.3. Performance Study of the GPU

We experimented with PTX-based prefetch on the GPU with different prefetch distance and type, but observed a performance degradation to varying degrees, as shown in Figure 3. One simple explanation [29] is that the GPU has thousands of in-flight threads, and that the overhead of thousands of outstanding memory request 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 the CPU and MIC [19, 20]. Consequently the GPU is more prone to cache pollution. Profiling on the cache behavior can shed light on this problem, as suggested by [19, 20]. Figure 3 illustrates part of the profiling results. When the data are prefetched to L2 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 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 L1 hit rate was observed. More detailed quantitative analysis needs to be done in the future.

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

The hash-based energy lookup method [9] currently implemented in MCNP 6.1.1 [30] is a memory saving alternative to the double indexing method used in XSBench. For H-M large problem, in XSBench the unionized energy grid and its associated index array require \( \sim 5.4 \) GB memory, while in MCNP a much smaller index array is needed (\( \sim 10.8 \) MB). 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 indicates that the binary search takes up \( \sim 36\% \) 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 T2, vectorization enabled. The result is shown in Table 6. On the MIC the performance improvement is only \( \sim 12\% \), 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 \(\sim 6\%\). However, this method and the above optimizations do not add up. When used together, a slight degradation of \(\sim 3\%\) was even observed. Tuning the hash-based energy lookup appears to be a challenge on the MIC.

4. Conclusions

We have conducted an in-depth optimization study on the proxy neutronics application, XSBench, for three hardware platforms — the 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\(\times\) faster on the CPU, 2.25\(\times\) on the MIC and 5.98\(\times\) 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\(\times\) and 2.20\(\times\) respectively.

5. Acknowledgments

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


Figure 1: Diagram of XSBench data structure.
Figure 2: Impact of prefetch distance and type over performance on the KNC MIC and Haswell CPU.
Figure 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/sec), the hit rate in L1 cache for global loads, and the hit rate at L2 cache for all read requests from L1 cache.
Table 1: Common prefetch types on the MIC and CPU. NTA=Non-Temporal Access. [31, 14, 32]

<table>
<thead>
<tr>
<th>Processor</th>
<th>Mnemonic</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIC</td>
<td>NTA</td>
<td>Load data to L1 and L2 cache, mark it as NTA</td>
</tr>
<tr>
<td></td>
<td>T0</td>
<td>Load data to L1 and L2 cache</td>
</tr>
<tr>
<td></td>
<td>T1</td>
<td>Load data to L2 cache only</td>
</tr>
<tr>
<td></td>
<td>T2</td>
<td>Load data to L2 cache only, mark it as NTA</td>
</tr>
<tr>
<td></td>
<td>ENTA</td>
<td>Exclusive version of NTA</td>
</tr>
<tr>
<td></td>
<td>ET0</td>
<td>Exclusive version of T0</td>
</tr>
<tr>
<td></td>
<td>ET1</td>
<td>Exclusive version of T1</td>
</tr>
<tr>
<td></td>
<td>ET2</td>
<td>Exclusive version of T2</td>
</tr>
<tr>
<td>CPU</td>
<td>NTA</td>
<td>Loads data to L2 and L3 cache, mark it as NTA</td>
</tr>
<tr>
<td></td>
<td>T0</td>
<td>Loads data to L2 and L3 cache</td>
</tr>
</tbody>
</table>
Table 2: Hardware specifications (HT=hyperthread, SM=streaming multiprocessor, SP=streaming processor).

<table>
<thead>
<tr>
<th>Processor</th>
<th>Microarchitecture</th>
<th>Core count</th>
<th>Base clock [GHz]</th>
<th>Price [USD]</th>
<th>Launch date [quarter-year]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel X5650</td>
<td>Westmere</td>
<td>6 (12 HTs)</td>
<td>2.66</td>
<td>996</td>
<td>1-2010</td>
</tr>
<tr>
<td>Intel E5-2697 v3</td>
<td>Haswell</td>
<td>14 (28 HTs)</td>
<td>2.6</td>
<td>2,702</td>
<td>3-2014</td>
</tr>
<tr>
<td>Intel 5110P</td>
<td>Knights Corner</td>
<td>60 (240 HTs)</td>
<td>1.053</td>
<td>2,437</td>
<td>4-2012</td>
</tr>
<tr>
<td>Nvidia K40</td>
<td>Kepler</td>
<td>15 SMs (2880 SPs)</td>
<td>0.745</td>
<td>3,300</td>
<td>4-2013</td>
</tr>
</tbody>
</table>
Table 3: Performance of XSbench for H-M large problem.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Code</th>
<th>Performance [10^6 lookups/sec]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Westmere CPU</td>
<td>Original</td>
<td>1.26</td>
</tr>
<tr>
<td></td>
<td>Tuned</td>
<td>1.83</td>
</tr>
<tr>
<td>Haswell CPU</td>
<td>Original</td>
<td>3.07</td>
</tr>
<tr>
<td></td>
<td>Tuned</td>
<td>4.64</td>
</tr>
<tr>
<td>Knights Corner MIC</td>
<td>Original</td>
<td>3.38</td>
</tr>
<tr>
<td></td>
<td>Tuned</td>
<td>7.58</td>
</tr>
<tr>
<td>Kepler GPU</td>
<td>Original</td>
<td>1.71</td>
</tr>
<tr>
<td></td>
<td>Tuned</td>
<td>10.2</td>
</tr>
</tbody>
</table>
Table 4: Speedup factors of the tuned codes.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Westmere CPU</td>
<td>Baseline</td>
</tr>
<tr>
<td>Haswell CPU</td>
<td>2.54×</td>
</tr>
<tr>
<td>Knights Corner</td>
<td>4.14× 1.63× Baseline</td>
</tr>
<tr>
<td>Kepler GPU</td>
<td>5.59× 2.20× 1.35×</td>
</tr>
</tbody>
</table>
Table 5: Impact of different optimization techniques on the MIC.

<table>
<thead>
<tr>
<th>Optimization</th>
<th>Performance difference</th>
<th>Reason for performance improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data prefetch</td>
<td>43.0%</td>
<td>Reduce cache misses</td>
</tr>
<tr>
<td>Loop unrolling</td>
<td>29.8%</td>
<td>Improve instruction level parallelism</td>
</tr>
<tr>
<td>Huge page</td>
<td>51.2%</td>
<td>Reduce TLB misses</td>
</tr>
<tr>
<td>Vectorization</td>
<td>11.0%</td>
<td>SIMD</td>
</tr>
<tr>
<td>Thread affinity</td>
<td>6.7%</td>
<td>Reduce cache misses</td>
</tr>
</tbody>
</table>
Table 6: Performance of XSbench adapted to the hash-based energy lookup algorithm.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Code</th>
<th>Performance [10^6 lookups/sec]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Westmere CPU</td>
<td>Original</td>
<td>0.768</td>
</tr>
<tr>
<td>Knights Corner</td>
<td>Original</td>
<td>2.82</td>
</tr>
<tr>
<td></td>
<td>Tuned</td>
<td>3.16</td>
</tr>
</tbody>
</table>
Appendix A. Algorithm

Algorithm 1: Original XSbench algorithm.
Algorithm 2: Optimized algorithm for MICs and CPUs. For illustration purpose, 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.