Delft University of Technology
Parallel and Distributed Systems Report Series

Benchmarking Intel Xeon Phi to Guide Kernel Design

Jianbin Fang, Ana Lucia Varbanescu, Henk Sips
{j.fang,a.l.varbanescu, h.j.sips}@tudelft.nl

Lilun Zhang, Yonggang Che, Chuanfu Xu
Completed in April 2013

Report number PDS-2013-005

ISSN 1387-2109

Published and produced by:
Parallel and Distributed Systems Group
Department of Software and Computer Technology
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

Information about Parallel and Distributed Systems Report Series:
reports@pds.ewi.tudelft.nl

Information about Parallel and Distributed Systems Group:
http://www.pds.ewi.tudelft.nl/

With a minimum of 50 cores, Intel's Xeon Phi is a true many-core architecture. Featuring fairly powerful cores, two levels of caches, and a very fast interconnection, the Xeon Phi is able to achieve theoretical peak of 1000 GFLOPs and over 240 GB/s. These numbers, as well as its flexibility - it can be used as both coprocessor or a stand-alone processor - are very tempting for parallel applications looking for new performance records.

In this paper, we present four hardware-centric guidelines and a machine model for Xeon Phi programmers in search for performance. Specifically, we have benchmarked the main hardware components of the processor - the cores, the memory hierarchies, and the ring interconnect. We show that, in ideal microbenchmarking conditions, the achieved performance is very close to the theoretical one as given in the official programmer's guide. Furthermore, we have identified and quantified several causes for significant performance penalties, which are not available in the official documentation. Based on this information, we synthesized four optimization guidelines and applied them to a set of kernels, aiming to systematically optimize their performance. The optimization process is guided by performance roofs, derived from the same benchmarks. Our experimental results show that, using this strategy, we can achieve impressive performance gains and, more importantly, a high utilization of the processor.

Keywords: Performance, Microbenchmarking, Optimization.

Contents

List of Figures

List of Tables

1 Introduction

Intel Xeon Phi is the latest high-throughput architecture targeted at high performance computing, and, without a doubt, will be part of the very next generation of supercomputers that will challenge Top500[1]. To achieve its high level performance (1000 GFlops), Intel Xeon Phi [1] uses over 50 cores and 25 MB of on-chip caches. Despite the features it shares with multi-core CPUs and many-core GPUs (vectorization, SIMD/SIMT, high throughput, and high bandwidth) [2], Xeon Phi has a different architecture from all of them [3]. For example, overall cache coherency is not available on GPUs, while the ring interconnect is not to be found in either CPUs nor GPUs.

For advanced users - like most high performance computing (HPC) programmers and compiler developers are - it is essential to understand this architecture in detail, as the achieved performance depends on each of these details. For example, knowing the requirements for density and placement of threads per cores, the optimal filling of the core interconnections, or the difference in latency between the different types of memories on chip are non-trivial details that, when properly exploited, can lead to impressive performance gains.

Empirical evaluation, based on benchmarking is a recognized solution for achieving this level of understanding. Therefore, in this work, we present a suite of microbenchmarks for measuring three major architectural features, with an (expected) high impact on performance: the processing cores, the memory hierarchies, and the ring interconnect. The numbers obtained using these microbenchmarks can be subsequently used to (1) gain a deeper understanding of the hardware behavior (at times, complementary to the official specification), and (2) to establish empirical upper bounds of the achievable performance on real-life platforms.

In the second part of the paper, we present a benchmarking-based strategy for optimizing Intel Xeon Phi kernels. Thus, we present the design and iterative optimization of four different kernels (Monte Carlo, Stencil, GEMM, and SpMV - see Section 4). Our goal is to demonstrate how a kernel's optimization process can be guided by customized *performance roofs*, which identify the most immediate bottlenecks and implicitly suggest the next optimization steps. Our experimental results show significant speedups achieved for the first three kernels, and non-conclusive results for SpMV (whose performance depends on the matrix density and the usage of cache-lines). Thus, we conclude that performance roofs are a promising solution for directed (auto-)tuning of Xeon Phi applications.

To summarize, the contributions of this work are:

- We microbenchmark the Xeon Phi coprocessor in a comprehensive way (Section 3), obtaining interesting numerical results for the capabilities of its cores, memories, and interconnect.
- We present four sets of optimization guidelines and a simplified machine model of the Xeon Phi, both aimed at optimizing the process of development and tuning of applications (Section 3).
- We define the customized *performance roofs* for a given application, and we demonstrate how to calculate and use them to optimize four typical HPC kernels on the Xeon Phi (Section 4).

2 Background

2.1 Intel Xeon Phi Architecture

The Intel Xeon Phi comprises of over 50 cores (the one used in this paper belongs to the 3100 series and has 57 cores) connected by a high-performance on-die bidirectional interconnect (shown in Figure 1). In addition to these cores, there are 12 channels (supported by memory controllers) delivering up to 5.0 GT/s (240 GB/s memory bandwidth) [3]. Working as coprocessor, the many-core processor is connected to a host with special function devices such as the PCI Express system interface. Different from GPUs, a dedicated embedded Linux μOS runs on the platform.

[1] http://www.top500.org

The cores contain a 512-bit wide vector unit with the vector register files (32 registers per thread context). Each core has a 32KB L1 data cache, a 32KB L1 instruction cache, and a core-private 512KB unified L2 cache (thus 28.5MB on the die). The L2 caches are kept fully coherent with each other by the TDs (distributed duplicate tag directory), which are referenced after an L2 cache miss. The tag directory is not centralized but is broken up into 64 distributed tag directories (DTDs). Each DTD is responsible for maintaining the global coherence state in the chip for its assigned cache lines.

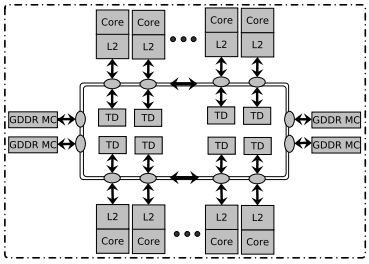


Figure 1: The Intel Xeon Phi Archiecture.

2.2 Programming on the Xeon Phi

Being an `x86` SMP-on-a-chip running Linux [4], the Xeon Phi offers the full capability to use the same tools, programming languages and programming model as an Intel Xeon processor. In particular, C users have OpenMP as well as Intel Cilk Plus. Fortran programmers can benefit from OpenMP and the added parallel features such as `DO CONCURRENT`. When dealing with both tasks and vector data, programming tools like OpenMP and MPI can be used simultaneously. In this work, we use the C language plus Intel's OpenMP implementation, and use the Intel `icc` compiler (`V2013.0.079`). Unless otherwise specified, we use the compiler option `-O3`.

There are two major approaches to involve the Intel Xeon Phi coprocessors in an application: (1) *offload mode*- the program is viewed as running on the host and offloading selected work to the coprocessor, (2) *native mode*-the program can run the coprocessor natively and independently, and can communicate with processors or other coprocessors [4]. In this work, we measure the performance of all programs with Xeon Phi working in the native execution mode.

3 Benchmarking

3.1 Processing Cores

When fully utilizing the vector processing cores, the peak instruction throughput is calculated as follows:

$FLOPS_{peak} = cores \times frequency \times lanes \times (ops/cycle)$

The Xeon Phi 3100 has 57 cores working at 1.1 GHz, and each core processes 8 double-precision data elements at a time, with maximum 2 operations (`multiply-add` or `mad`) per cycle in each lane (processing element). Therefore, the theoretical instruction throughput is 1003.2 GFlops.

To measure the instruction throughput, we run 1, 2, 4 vector (8 elements in double precision) threads on a core. During measurement, each thread performs one or a set of 2 independent `mad` and `mul` instructions for a pre-defined loop count. The loop was unrolled aggressively to hide the pipeline latency and avoid the branch overheads. The results are shown in Figure 2. Overall, we can achieve 97% of the peak instruction throughput (using 56 cores). We also have the following observations. First, when using 56 threads (one thread per core), the instruction throughput is rather low, compared with the cases when using 112 or 224 threads. This is due

to the fact that it is not possible to issue instructions from the same threads context in back-to-back cycles [3]. Thus, programmers need to run at least two threads on each core to fully utilize the hardware resources. Second, when a thread is using only one instruction stream at a time, we have to use 4 threads per core (224 threads in total) to achieve the peak instruction throughput. Furthermore, we note that the `mad` throughput is twice as large as the `mul` throughput. Thus, we conclude that for a given instruction mix, the achievable instruction throughput relies on not only the number of cores/threads, but also on the issue width (i.e., the number of independent instructions). Note that the microbenchmarks are auto-vectorized by the compiler and thus we can ensure 100% vector usage.

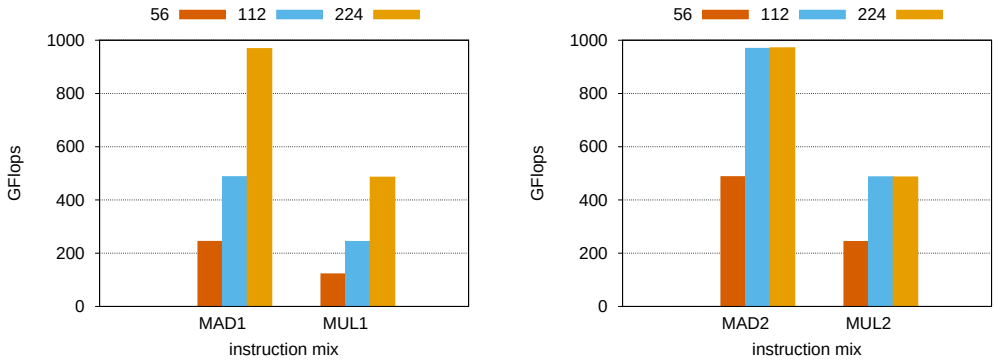


Figure 2: Arithmetic throughput using different threads and issue widths: (a) Using a single instruction; (b) Using 2 independent instructions.

Short Vector Math Library (SVML): The basic math functions such as `exp`, `log`, `sin`, and `cos` are often used in scientific computing. On the Xeon Phi, Intel provides such functions in the form of intrinsics. These math intrinsics are vector variants of corresponding scalar math operations. They take vector arguments and perform scalar math operation on each element of the source vectors. Thus, the SVML intrinsics do not have any corresponding instructions, but are the highly optimized routines. In Figure 3, we show the performance of `exp` for both the intrinsics implementations (svml and svml2) and the C library implementations (clib and clib2). We note that there are only slight performance differences for these two versions. Thus, programmers can use the C library implementations for simplicity. Further, using `base2` implementations (svml2 and clib2) can significantly decrease the execution time for the single-precision `exp`, whereas it has little effect on the double-precision one. Additionally, compared with the double-precision version of `exp`, using single-precision data elements performs 5× faster. With 512-bit wide vectors, a thread can process 16 data elements simultaneously and thus the single-precision instruction throughput is at least twice as large as that of double-precision. Therefore, we prefer using the single-precision data elements when it meets the accuracy requirement.

A final interesting phenomena is that using 57 cores leads to significant throughput degradation, which validates that Core 56 [2] should be used as a *management core* and we must avoid submitting loads onto it.

3.2 Memory Latency

3.2.1 Cache Properties

To reveal the cache properties (the latency and the structure), we use a traditional pointer chasing benchmark (like the one used in BenchIT [3]). It traverses an array A of size S by running $k = A[k]$ in an aggressively

[2] We number the processing cores starting with 0, and the cores are Core 0, Core 1, ..., Core 56, respectively.
[3] http://www.benchit.org/

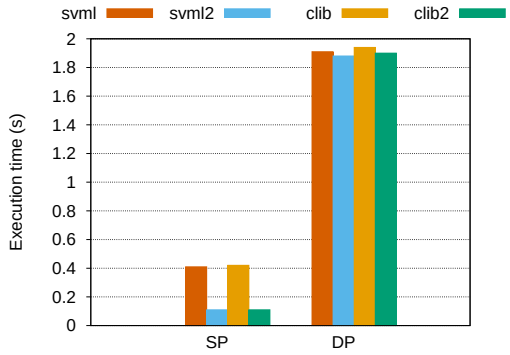


Figure 3: `exp` performance.

unrolled loop, yielding the time for one iteration. This time is dominated by the latency of the memory access. The array is initialized with a stride, i.e., $A[k] = (k + stride)\%S$. The traversal is done in one thread, and thus utilizes only one core. Therefore, the cache properties obtained here are local and belong to one core. The results are shown in Figure 4. We see that the Xeon Phi has two levels of data caches (L1 and L2). The L1 data cache is 32KB, while the L2 data caches should be smaller than 512KB. Furthermore, the accessing latency of L1 and L2 data caches is around 2.8 ns (3 cycles) and 20 ns (22 cycles), respectively. It takes around 300 ns (330 cycles) to access the data in the main memory.

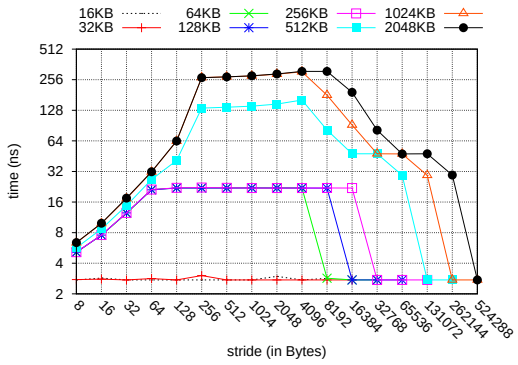


Figure 4: Average access time.

To check the cache-line size and the cache associativity at each level, we first take a closer look at the L1, when the data elements can be hold within the L2 ($\leq$256 KB). We see that when the array is not larger than 32KB (L1 size), the access time stays stable, since all the data elements can be stored in the L1 cache. When the input array is larger than 32KB, the access time increases over strides. This trend continues until the stride is 64 bytes, which shows that the threads operate the data in a batch manner, i.e., a cache-line. The measurement indicates that programmers need to access the memory in a batch manner and contiguously. When the stride is larger than 64 bytes, the accesses will experience a L1 cache miss per access, with latency staying stable. Then the average access time drops sharply when the stride equals 8192, 16384, and 32768 for array sized 64KB, 128KB, and 256KB, respectively (the access time drops to the number without L1 cache misses). At this time, we calculate that the associativity is 8 (see more details in [5]). Meanwhile, we can calculate that the L1 cache has 64 sets.

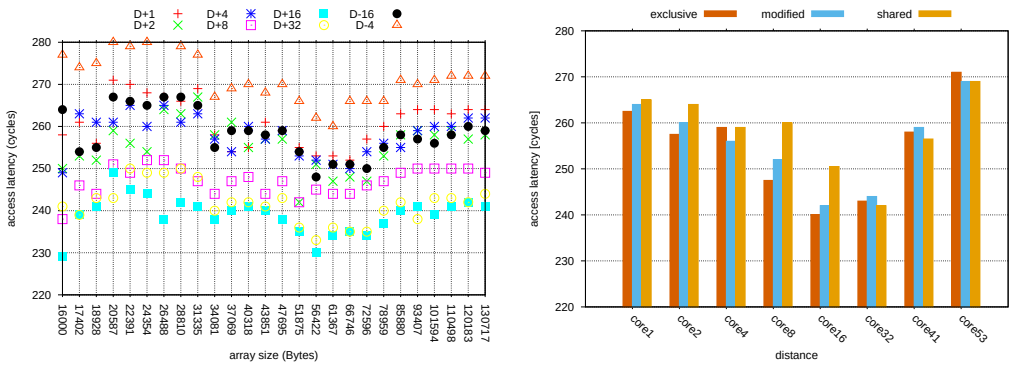
Similar observations can be found for the L2 when we vary the footprints from 256KB to 2MB (Figure 4).

In particular, we see that when the footprint is 512 KB, we experience a big latency when accessing the L2 cache. This larger latency occurs because the L2 cache on the Xeon Phi is a unified cache for both data and instructions. Thus, when designing algorithms and prefetching data into the L2 data cache, programmers need to constrain the data size within 512 KB, leaving some space for instructions.

3.2.2 Remote Cache Latency

We have illustrated the latency of local (L1/L2) caches in Section 3.2.1. In this section, we use the approach (proposed by Daniel Molka [6]) to measure the access latency of remote caches. Prior to the measurement, the to-be-transferred cache-lines are placed in different locations (cores) and in a certain coherency state (`modified`, `exclusive`, `shared`). In each measurement, we use two threads (t0, t1), with t0 pinned to Core 0 and t1 to another certain core. The latency measurement always runs on Core 0. Figure 5 shows our latency results of remote cache accesses on Xeon Phi.

In Figure 5(a), we see that, when the cache line is in `exclusive` state, the overall latencies of remote access are between 230 cycles and 280 cycles (roughly matching the results by Garea [7]), which are much larger than the local cache access but slightly smaller than the off-chip memory access (330 cycles). Further, we note that the cache latency of remote access varies inversely with the distance between the two target cores. Specifically, when the cache lines are lying on Core 16 or Core 32, we have a smaller remote cache access latency. By getting the `median` value of all the input data sets, we show the overall remote latency in Figure 5(b), which further validates the inverse relation between the latency and the core distances. We also note that there is no relationship between the remote access latency and the cache-line states (`exclusive`, `modified`, `shared`).



(a) exclusive (b) overall

Figure 5: Read latencies of Core 0 accessing the cache lines on Core 1 (D+1), Core 2 (D+2), Core 4 (D+4), Core 8 (D+8), Core 16 (D+16), Core 32 (D+32), Core 41 (D-16), and Core 53 (D-4).

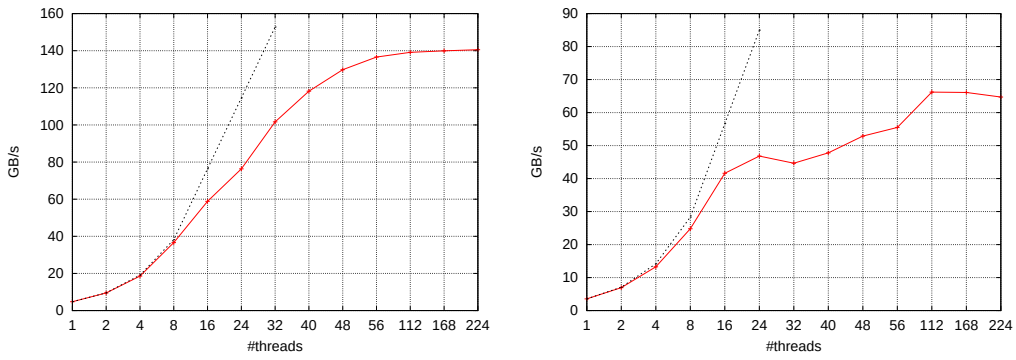
To summarize, for better performance, each core needs to keep the data in its own local cache, and avoid remote cache access and off-chip memory access. In addition, the Xeon Phi uses a private LLC (last-level cache) [8]. Specifically, if no cores share any data or code, the effective total L2 size of the chip is 28.5 MB. Whereas, if every core shares exactly the same data and code in perfect synchronization, then the effective total L2 size of the chip is 512 KB.

3.3 Memory Bandwidth

3.3.1 Off-Chip Memory Bandwidth

The Xeon Phi used in this work has 12 channels, each 32-bits wide. At up to 5.0 GT/s transfer speed, it provides a theoretical bandwidth of 240 GB/s. But **is this theoretical number achievable or how close can we get to this number?** To this end, we measure the memory bandwidth for both `read` and `write`. The `read` benchmark reads data from an array A ($b = b + A[k]$). The `write` benchmark writes a thread-specific value into an array A ($A[k] = C_t$). Note that A needs to be large enough (e.g., 1 GB) such that it cannot be held with the on-chip memory. We use different running threads from a single one to 224.

The results are shown in Figure 6. Overall, we see both the maximum bandwidths are far below the theoretical bandwidth (240 GB/s). We also note that the `read` bandwidth increases when using more threads, peaking at 140 GB/s (from using 112 threads on). On the other hand, the `write` bandwidth is not as large as the `read` bandwidth. When using 112 threads, we can obtain the maximum bandwidth (66 GB/s), whereas it drops slightly thereafter. When using more threads, we will generate more requests to memory controllers, thus making the interconnect and memory channels busier. Therefore, programmers need to launch over 2 threads per core to saturate the interconnect and the memory channels.



(a) read (b) write

Figure 6: Read and write memory bandwidth.

3.3.2 Aggregated On-Chip Memory Bandwidth

Each core on the Xeon Phi has a separate/private L1 and L2 data cache. Thus, we measure the cache bandwidth on a single core and calculate the *aggregated* cache bandwidth by multiplying the number of cores. The benchmark is similar to the one we used to measure the off-chip memory bandwidth. But it differs in that the array A is not larger than, for example, the L1 data cache (32 KB) so that we ensure the data elements are located at the corresponding cache level. We measure the bandwidth using 1, 2, 3, 4 threads on a single core. Each thread accesses a different but equally-sized cache space (starting with 1 KB).

The results are shown in Figure 7. We see that the bandwidth increases when the threads are accessing more data, and this trend goes on until the total data amount exceeds the L1 capacity. Further, using 2 threads delivers the maximum bandwidth (22 GB/s) when each thread accesses an array of 14 KB or 15 KB. Therefore, we can calculate that the aggregated L1 cache bandwidth is around 1232 GB/s.

Furthermore, it is difficult to measure the L2 bandwidth due to the presence of the L1 cache. The bandwidth depends on the memory access patterns. Specifically, when we use a L2-friendly memory access pattern, the compiler will identify the stream pattern and prefetch data to the L1 cache in time. By this, we will get a

bandwidth similar to that of the L1 cache due to the common efforts of L1 and L2. On the other hand, an unfriendly memory access will experience L1 misses and result in loading data from the L2 cache.

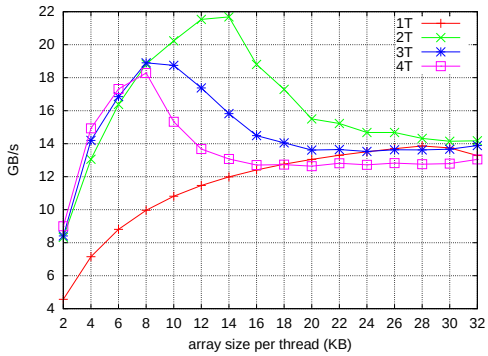


Figure 7: L1 Cache read bandwidth on a single core.

3.3.3 Results Validation

To validate our results, we use the `STREAM Benchmark` [9] to measure the memory bandwidth. To avoid the kernels (`copy`, `scale`, `add`, and `triad`) to interfere with each other, we measure their memory bandwidth separately. We average the `read` and `write` bandwidth (presented in Section 3.3.1) to calculate the `predicted` value. The bandwidth numbers (`measured` versus `predicted`) are shown in Table 1. Overall, the `measured` memory bandwidths stay around the `predicted` numbers. We also note that the `copy` bandwidth is larger than `scale` bandwidth by around 25%. When investigating the assembly code, we noticed the `copy` kernel uses `intel_fast_memcpy` which optimizes the copy operations.

Table 1: STREAM bandwidths in GB/s.

	copy	scale	add	triad
measured	120	95	112	113
predicted	103	103	115	115

3.3.4 Factors that Affect Bandwidth

ECC Effects: The Xeon Phi coprocessor supports ECC (Error Correction Code) to avoid software errors caused by naturally occurring radiation. Enabling ECC brings us more reliable data, but it also introduces extra overhead to check error bits. **How is the performance with ECC enabled and disabled?** We examined the performance differences before and after disabling ECC and show the memory bandwidth in Figure 8. After ECC was enabled, we noticed a 20%∼27% bandwidth decrease for the `STREAM` kernels.

Prefetch Effects: The L2 cache has a streaming hardware prefetcher that can selectively prefetch code, read, and RFO (Read-For-Ownership) cachelines into the L2 cache [3]. This prefetcher is enabled by default, and we use `Stanza Triad (STriad)` [10] to evaluate the efficacy of prefetching on a single core. `STriad` works by performing DAXPY (Triad) inner loop for a length L stanza before jumping k elements and then continuing on to the next L elements, until we reach the end of the array. We set the total problem size to 128 MB, and set k to 2048 words in double-precision. Each stanza data size was run 10 times, with the L2 cache flushed each

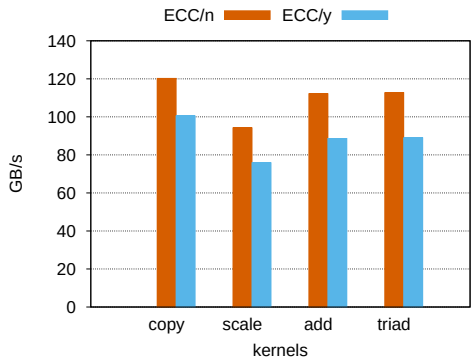


Figure 8: ECC Effects (We use 56 threads/cores and the input array is of 1GB).

time, and we averaged the performance to calculate the memory bandwidth for each stanza length. Figure 9 shows the results of the STriad experiments on the Xeon Phi. We see an increase in memory bandwidth over the stanza length. Further, it can be seen that with increasing stanza lengths, STriad performance asymptotically approaches the bandwidth of `STREAM triad`. Therefore, we conclude that non-contiguous access to memory is detrimental to memory bandwidth efficiency and thus the performance of memory-bound kernels. For these reasons, programmers should try to create the longest possible stanzas of contiguous memory accesses for better prefetching effects and larger memory bandwidths.

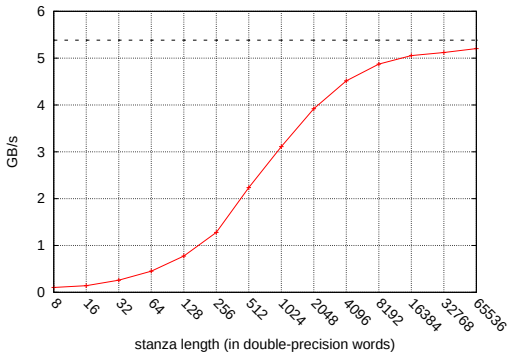


Figure 9: Performance of STriad on the Xeon Phi.

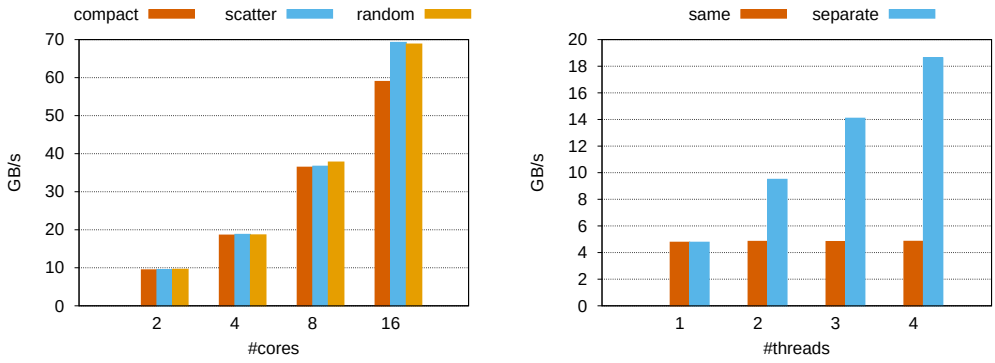
To summarize, the achieved memory bandwidth is far below the pin bandwidth (240 GB/s). We assume other limiting factors are the ring interconnect (e.g., ring stops) and channel/bank conflicts. Thus, we will use the sustainable bandwidth(s) as a reference when modeling and optimizing applications. In the following, we will investigate the ring interconnect on the Xeon Phi in memory bandwidth.

3.4 Ring Interconnect

3.4.1 Core Distribution Effects

On Intel Xeon Phi, the cores and memory controllers/modules are connected by a ring interconnect. **When multiple neighboring cores are requesting data from main memory simultaneously, will the ring become a bottleneck?** We use different ways to measure the `read` bandwidth: (1) *compact*- using multiple

cores that are located close to each other, (2) *scatter*- using multiple cores that are evenly distributed around the ring, (3) *random*- the core `IDs` are selected randomly with no repeats. The numbers are measured using 2, 4, 8, and 16 cores (Figure 10(a)). We see that the three approaches achieve similar memory bandwidths when using 2, 4, or 8 cores, but when using 16 threads, the *compact* approach suffers 15% performance decrease compared with the other approaches. This is because the neighboring cores requesting data simultaneously use the ring in an imbalanced way. Thus, when not all the cores on the ring are utilized, programmers need to distribute threads to cores in a *scatter* or *random* manner.



(a) Core Distribution (b) Thread Distribution

Figure 10: Core and thread distribution effects (we use the `read` kernel and the array size is 1 GB in size).

3.4.2 Threads Distribution Effects

On Intel Xeon Phi, each core supports four hardware threads. The question is **whether threads from the same cores can achieve similar bandwidths with the case when the threads run on separate cores?** Figure 10(b) shows that when the threads run on the same core, the bandwidth increases slightly compared with that of using one thread. On the other hand, running threads on separate cores results a linear increase in bandwidth with the number of threads. We conclude that when multiple threads on the same core are requesting data simultaneously, they will compete for the shared hardware resources such as ring stops, thus serializing the requests.

3.5 Summary

From the benchmarking results and our programming experience on the Xeon Phi, we can make the following observations and optimization guidelines:

- **High Throughput:** Our results validate that the Xeon Phi is indeed a high-throughput platform. The peak instruction throughput is achievable, but it depends on the following factors: (1) the number of threads and cores, (2) the usage of the 512-bit vectors, (3) the issue width (the number of independent instructions), (4) the instruction mix. Furthermore, once the accuracy requirements are satisfied, we prefer using single-precision data elements (D2S), and we also prefer using the `base2` math functions over other versions (FMF).
- **Memory Selection:** Accessing the local L1 cache is 8× faster than accessing the local L2 cache, which is again an order of magnitude faster than accessing the remote caches or the off-chip memory. However, the difference between a remote cache access and an off-chip memory access is relatively small (15%).

Furthermore, the remote access latency does not depend on the cache-line states. Therefore, programmers should try to use the local caches as much as possible, and avoid the remote caches and the off-chip memory.

- **Efficient Memory Access:** Each vector thread loads data from the off-chip memory in a group (chunk) manner (i.e., cache-line) on the Xeon Phi. The maximum achievable bandwidths are 140 GB/s for read operations and 66 GB/s for write operations, which are far smaller than 240 GB/s. Further, programmers need 112 (56×2) threads to issue enough memory requests so as to saturate the ring interconnect and the memory channels. We also noticed that the ECC status and prefetching can both significantly affect the bandwidths. To use the prefetcher efficiently, we prefer creating the longest possible `stanzas` of contiguous memory accesses.
- **Ring Interconnect:** All cores can be seen as *symmetric* peers. However, when the cores are not fully utilized (i.e., some remains idle), attention should be paid to the core distributions and the thread distributions. Specifically, we should distribute threads (when one thread is pinned to one core) uniformly (using thread affinity) to avoid ring traffic congestion. Moreover, when two or more threads run on the same core, the memory requests to the ring are serialized.

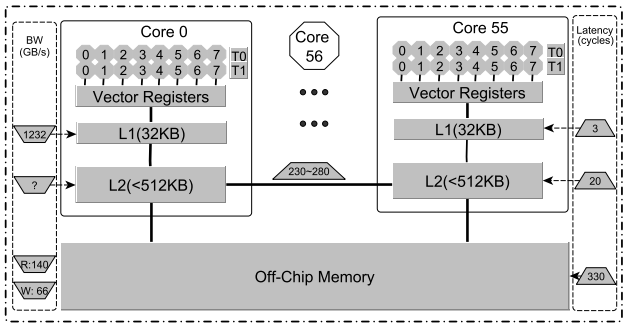


Figure 11: A machine model of Intel Xeon Phi.

Based on the microbenchmarking results, we present a machine model of the Xeon Phi from a programmers's point of view (shown in Figure 11). The machine has 57 cores, each of which contains 2 vector threads working on 8 double-precision or 16 single-precision data elements in a lock-step manner. Since the last core (Core 56) runs the μOS, we see it as a *management core* and avoid submitting tasks to this core. Each time a thread requests data elements from the main memory, it will load a cache-line (64 bytes) into its (L1/L2) cache. It will save memory bandwidth if all the processing elements of a vector thread access the data elements located in the same cache-line. Furthermore, compared with accessing local caches, remote caches and off-chip memory accesses are much more expensive. We mark the access latencies and bandwidths at different memory levels to specify the machine model.

This machine model limits itself to those architectural details that are important for performance. For example, programmers do not have to keep the ring interconnect in mind because the cores perform like they are symmetric. Consequently, it simplifies the programming. On the other hand, this model plus the optimization guidelines provides detailed information when modeling and optimizing applications, which will be illustrated in Section 4.

4 Guiding Kernel Design

In this section, we parallelize, evaluate, and optimize four different kernels in OpenMP on the Xeon Phi. We focus on the hardware-centric optimizations, guided by the customized performance roofs.

4.1 Customized Performance Roofs

For a kernel, the performance is bounded by either arithmetic computation or memory access, or both. To guide kernel design and optimization, we provide customized performance roofs for both *flops* and *bandwidth*. Then, comparing the measured performance with the roofs will indicate which one is the performance bottleneck (The one that is closer to roofs is the performance bottleneck). Thereafter, we can perform the corresponding optimizations. This process will continue till we meet the machine bounds or when we find the performance is limited by the kernel inherent characteristic. In this following, we will introduce *flops roof* and *bandwidth roof* and illustrate how to calculate them for a given kernel and context.

Flops Roof is the maximum achievable flops given a context. As we shown in Section 3.1, the maximum achievable flops depends on four factors. Taking the `mad` and `mmad` instructions for example, the flops roofs on the Xeon Phi are shown in Table 2 (`mad(2)` means using 2 independent `mad` instructions). The flops roofs are measured when the vector cores are 100% utilized. For a given kernel, the number of used cores and threads can be manually set by programmers. The instruction mix and the number of independent instruction streams (issue width) can be identified from the kernel. To determine the SIMD usage, we use the verbose information emitted by the compiler [4]. Note that the automated vectorized code is not always more efficient than the non-vectorized version.

Table 2: Flops roofs using different threads, instruction mixes, and issue widths on the Xeon Phi.

	56	112	224
mad(1)	244.37	487.65	968.74
mad(2)	487.52	969.46	971.57
mmad(1)	133.33	218.55	365.84
mmad(2)	241.60	371.74	483.10

However, for some real-world applications, the instruction mixes are complex and diverse. In particular, it is difficult to count the flops in the presence of the basic functions such as *logrithms* and *exponentials* [11]. For these kernels, we assume calling the functions one time as a basic operation like an addition or a multiplication. When measuring the flops roofs, we write a microbenchmark by removing the memory access operations from the original kernel. In this way, we can not only mimic the data dependency, but keep the control flows.

Bandwidth Roof is an overall estimate based on read/write operations from/to the off-chip memory. In Section 3.3.1, we have measured that the maximum bandwidths are 140 GB/s and 66 GB/s for read and write operations, respectively. Given a kernel and a run-time input, we can calculate the bandwidth roofs using the ratio of read and write operations. Taking GEMM for example (Section 4.4), the kernel reads two matrix, but write only one ($r : w = 2 : 1$). Therefore, we estimate the bandwidth roof to be 115 GB/s.

4.2 Monte Carlo European Option

4.2.1 Description

Monte Carlo European Option (MCEO) uses the Monte Carlo method to value an option. It samples a path to obtain the expected payoff in a risk-neutral world and then discount the payoff to current value using risk-free interest rate. For an option with a current stock price $S(0)$, we follow the equation to value the derivatives from time t to $t + \triangle t$:

$S(t + \triangle t) = S(t) \cdot exp((\mu - \frac{\sigma^2}{2}) \cdot \triangle t + \sigma \cdot \varepsilon \cdot \sqrt{\triangle t})$

where μ is the expected return in a risk neutral work, σ is the volatility, and ε is a random sample from a normal distribution. The value of each option (N) can be accumulated at the time of expiration T for enough

[4] use the `-vec-report` option.

price paths M. Totally, the amount of data from/to the off-chip memory (*bytes*) and the number of operations ($flops$) are: $bytes = 5 \times 8 \times N + 8 \times M$, $flops = 9 \times M \times N$.

4.2.2 Parallelization and Optimizations

A natural way to parallelize MCEO is to partition the options into multiple groups and run them independently on different threads. Since MCEO use the *exponential* function, we measure the flops roofs by removing the memory accesses. The performance is shown when $N = 112 \times 1024$ and $M = 2^{20}$ in double-precision (Table 3). We note that the memory usage is rather low and the naively parallelized MCEO is compute-bound. Thus, we further use the guidelines mentioned in Section 3.5 to improve its performance.

Using Simple Expressions: When using the `-O3` option, the compiler will perform vectorization automatically. However, using complex expressions will make vectorization inefficient or even impossible. In MCEO, we use a `selection` statement to avoid negative numbers: $(a > 0)?(a) : (0)$, where a is again a complex expression with exponential computations. When compiling the program, the `selection` statement can be vectorized successfully, but a is computed twice. Instead, we can first evaluate the expression a and then execute the selection statement. In this way, MCEO runs 2× faster (+SE in Table 3).

Using Single-Precision: Using single-precision data elements brings us 2× speedup on the Xeon Phi, and thus we prefer using single-precision data elements when meeting the accuracy requirements. The experimental results (+D2S in Table 3) show that switching to use the single-precision data elements improves the performance by over 3×. The number is between 2× (the theoretical speedup when switching from double-precision to single-precision) and 5× (the *exp* speedup when switching from double-precision to single-precision shown in Section 3.1).

Using Fast Function: In this kernel, the exponential function is the most time-consuming operation. As we have shown, the `base2` function performs much faster than the `base`e version on Xeon Phi. Thus, we use the *exp2* to replace *exp* to further speedup the kernel execution (+FMF in Table 3).

Table 3: MCEO Performance on the Xeon Phi

	flops (GFlops)			bandwidth (GB/s)		
	measured	roofs	%	measured	roofs	%
naive	18.78	20	93.92	0.0002	130	–
+SE	35.99	39	92.29	0.0004	130	–
+D2S	108.23	135	80.17	0.0008	130	–
+FMF	290.17	359	80.83	0.0018	130	–

4.3 Stencil Computation

4.3.1 Description

At the heart of partial difference equation (PDE) solvers are `stencils`, using iterative finite-difference techniques that sweep over a spatial grid, performing the `nearest-neighbor` computations [12–14]. In this paper, we use the regular 7-point 3D stencil kernels which can be expressed as triply nested loops *ijk* over:

$B(i,j,k) = \alpha \cdot A(i,j,k) + \beta \cdot (A(i-1,j,k) + A(i+1,j,k) + A(i,j-1,k) + A(i,j+1,k) + A(i,j,k-1) + A(i,j,k+1))$

Suppose B is $N_i \times N_j \times N_k$ in size, and A is $(N_i + 2) \times (N_j + 2) \times (N_k + 2)$. In total, the memory amount to be transferred (*bytes*) and the flops ($flops$) are (in double-precision): $bytes = 2 \times 8 \times N_i \times N_j \times N_k$, and $flops = 8 \times N_i \times N_j \times N_k$.

4.3.2 Parallelization and Optimizations

A naive way to parallelize this kernel is to add a `pragma` over the k dimension, leading to a maximum parallelism of N_k. We use 112 threads to run the stencil kernel (with each 2 contiguous threads assigned to a core) for 2 data sets (112^3 and 336^3). The instruction of this kernel is a mix of additions and multiplications. We measure that the flops roof is 487 GFlops. The overall read and write ratio is 1 : 1, and thus the estimated roof bandwidth is 103 GB/s.

The flops and memory bandwidth are shown in Table 4. We note that the stencil achieves below 10% of the machine instruction peak. When using the small data set, we can achieve 80% of the roof bandwidth, while it is only 54% for the large data set. Thus, the performance is bounded by memory access. Figure 12 shows the 3D stencil domain decomposition. The naive parallelization approach decomposes the grid in the k direction and partitions the domain into multiple panels of $N_i \times N_j \times B_k$, where B_k is determined by the number of threads (Figure 12(b)).

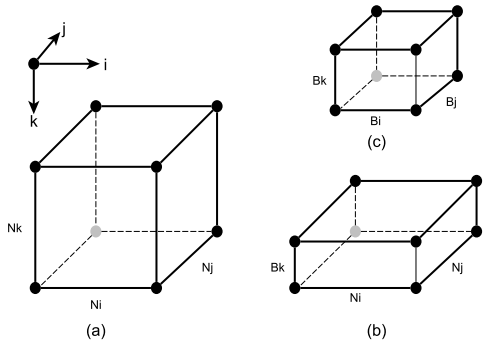


Figure 12: Stencil grid decomposition: (a) A data grid of $N_i \times N_j \times N_k$, where i is the unit stride dimension, (b) The naive OpenMP domain decomposition, (c) The cache blocking domain decomposition.

To achieve a finer-granularity tuning, we use `cache blocking` on the kernel [12–14]. Using cache blocking divides the grid into multiple blocks of $B_i \times B_j \times B_k$. In this way, we can exploit more parallelism from this kernel compared with the naively parallelized version. Further, we can better utilize the data caches by taking suitable block configurations. We enumerate all the block sizes to get the optimal performance for the blocking implementation. The stencil performance with cache blocking can be found in in Table 4. We see that the bandwidth usage is up to 90% even for the large data set. Further, we note that the optimal block configuration is $(112, 16, 8)$ and $(336, 8, 16)$ for the data set 112^3 and the data set 336^3, respectively. The results validate that we prefer accessing the memory contiguously on the Xeon Phi, i.e., keeping the unit-stride accesses as contiguous as possible. Moreover, we do not find a systematic way to select a right value for B_j or B_k.

Table 4: Stencil Performance on the Xeon Phi

		flops (GFlops)			bandwidth (GB/s)		
		measured	roofs	%	measured	roofs	%
naive	112	39.50	487	8.11	83.30	103	80.88
	336	27.37	487	5.62	55.72	103	54.10
blocking	112	48.60	487	9.98	102.50	103	99.51
	336	44.82	487	9.20	91.24	103	88.58

Listing 1: naive

```
#pragma omp parallel for shared(A, B, C, alpha,
beta, M, N, P) private(i,j,k)
for(j=0; j<M; j++){
  for(i=0; i<N; i++){

    double c = 0.0;
    for(k=0; k<P; k++){
      double a = A[k+j*P];
      double b = B[i+k*N];
      c += a * b;
    }
    C[i+j*N] = beta * C[i+j*N] + alpha * c;
  }
}
```

Listing 2: transB

```
#pragma omp parallel for shared(A, B, C, alpha,
beta, M, N, P) private(i,j,k)
for(j=0; j<M; j++){
  for(i=0; i<N; i++){

    double c = 0.0;
    for(k=0; k<P; k++){
      double a = A[k+j*P];
      double b = B[k+i*N];
      c += a * b;
    }
    C[i+j*N] = beta * C[i+j*N] + alpha * c;
  }
}
```

Listing 3: loop splitting

```
#pragma omp parallel for shared(A, B, C, alpha,
beta, M, N, P) private(i,j,k)
#pragma omp for nowait
  for(j=0; j<M; j++)
    for(i=0; i<N; i++)
      C[i+j*N] = beta * C[i+j*N];
#pragma omp for nowait
for(j=0; j<M; j++)
  for(k=0; k<P; k++)
    for(i=0; i<N; i++){
      double a = A[k+j*P];
      double b = B[i+k*N];
      C[i+j*N] += alpha * (a * b);
    }
```

4.4 GEMM

4.4.1 Description

As a building block for many other routines, the GEMM routine calculates the new value of matrix C based on the matrix-product of matrices A and B, and the old value of matrix C: $C \leftarrow \alpha AB + \beta C$, where α and β values are scalar coefficients. We suppose A, B, C are $M \times P$, $P \times N$, and $M \times N$ in size. Totally, GEMM performs $2N^3$ operations (suppose $M = N = P$). Without taking caches into account, it accesses $2N^3$ double-precision data elements from the off-chip memory. On the other hand, the number will be $3N^2$ when the data elements are used ideally in caches.

4.4.2 Parallelization and Optimizations

A naive way to parallelize this kernel is to add a `pragma` over the outer loop (Listing 1). This naive implementation uses `mad(1)` instructions with a roof of 487 GFlops. The read and write ratio is 2 : 1, and thus the roof bandwidth is around 115 GB/s (bandwidth roof2). When we consider the memory accesses without caches, the roof bandwidth is 140 GB/s (bandwidth roof1).

The GEMM performance is shown in Table 5, with $M = N = P = 4096$. We see both the flops and bandwidth usages are quite low. It is difficult to tell which one is the performance bottleneck (flops or bandwidth). When analyzing the code, we have identified two influencing factors: (1) memory accesses on B are not contiguous and the data elements need to be `gathered` from different positions, and (2) the inner-most loop includes a `reduction` operation. Thus, we first take a transposed B as input to eliminate the memory access factor (Listing 2). We see that the measured flops is 51 GFlops, and the memory bandwidth is 51 GB/s. To remove the impacts from `reduction`, we split the loop into two simple ones (Listing 3). At this time, `mmad` is the core instruction, with an instruction roof of 371 GFlops. The results show that GEMM is speeded up by 100× in flops. Therefore, GEMM is an example that has bottlenecks from both the memory access and the arithmetic instructions on the Xeon Phi. When optimizing it, programmers need to handle the data transposition and loop splitting manually, rather automatically by the compiler. Further, we measured that the maximum achievable performance is around 550 GFlops using the Intel's MKL routines on the Xeon Phi. We assume it needs further work in exploring the usage of the two levels of data caches to get higher performance.

Table 5: GEMM Performance on the Xeon Phi.

	flops (GFlops)			bandwidth (GB/s)					
	measured	roofs	%	roof1	measured1	%	roof2	measured2	%
naive	1.27	487	0.26	140	1.18	0.84	115	0.00	0.00
transB	51.00	487	10.47	140	51.13	36.43	115	0.15	0.13
ls	112.28	218	51.50	140	73.87	52.76	115	0.22	0.19

4.5 SpMV

4.5.1 Description

Given an $M \times N$ sparse matrix A and a dense vector x, we consider the sparse matrix-vector (SpMV) multiply $y \leftarrow Ax$, with y a dense result vector. A typical way of storing a sparse matrix A is the Compressed Row Storage (CRS) format [15], which stores data in a row-by-row fashion using three arrays: *col*, *val*, and *row*. The first two arrays are of size $nz(A)$, with $nz(A)$ the number of nonzeroes in A, whereas *row* is of length $M+1$. The array *col* stores the column index of each nonzero in A, and *val* stores the corresponding numerical values. The ranges $[row_i, row_{i+1})$ in those arrays correspond to the nonzeroes in the i^{th} row of A. Thus, the amount of memory that needs to be transferred (*bytes*) and the flops ($flops$) are (note that we use `integer` to store the array index and `double` to store the values): $bytes = 2 \times M \times 8 + nz(A) \times (8+4) + (M+1) \times 4$, $flops = nz(A) \times 2$.

4.5.2 Parallelization and Optimizations

We assign random positions to a given number of elements in a square matrix A, and use a function to encode these positions in compressed sparse row format. The ratio of non-zero data elements can be controlled by a parameter ra. We parallelize the SpMV kernel by letting each thread (112 threads in total) process multiple continuous rows in parallel. For all the experiments, we run the code for 3 times to warp up the TLB and to avoid the effect of lazy allocation. Then we run the SpMV with a repeat of 100 times, with caches flushed between two repeats.

Table 6: SpMV Performance on the Xeon Phi

ra	flops (GFlops)			bandwidth (GB/s)		
	measured	roofs	%	measured	roofs	%
1%	4.47	487	0.92	26.95	140	19.25
4%	6.13	487	1.26	36.81	140	26.29
10%	11.34	487	2.33	68.07	140	48.62
20%	13.88	487	2.85	83.31	140	59.51
50%	21.40	487	4.39	128.44	140	91.74
100%	23.13	487	4.75	138.81	140	99.15

The final output is averaged and shown in Table 6. SpMV uses `mad` operations and thus the roof flops is 487 GFlops. According to the read and write operations, we can calculate the roof bandwidth is 140 GB/s. From the measured performance, we see that SpMV is memory-bound. Specifically, when changing the ratio of non-zero data elements from 1% to 100% (i.e., A is a dense matrix), the off-chip bandwidth usage varies from 19% to 99%. When A is sparse (i.e., it has very few non-zero data elements), we obtain poor performance. This is because of the small bandwidth when accessing vector x. In particular, this occurs when the continuous two non-zero data elements are far from each other. In such case, a thread will load a cache-line of data elements (8 doubles) on the Xeon Phi, but use only one of them. Even through the kernel can use the vector units, the vector data from x has to be `gathered` from different positions and possibly from different cache-lines. Therefore, this low performance is determined by the kernel characteristics, which is memory-access unfriendly on Xeon Phi, and cannot be optimized by programmers manually.

5 Related Work

In this section, we present the prior work on microbenchmarking and kernel optimizations. Generally, we split the microbenchmarking work into two groups: one is on CPUs, and the other is on GPUs. In [5], the authors develop a high-level program to measure the cache and TLB for any machine. Part of our work (cache associativity calculation) is based on the approaches presented in this paper which, however, was targeting uni-core processors. Since we stepped into the multi-core era, there have been multiple studies on the multi-core CPUs. In [16], the authors report performance measurement on three multi-core processors. In addition to the execution time and throughput measurement, they provide a detailed analysis on the memory hierarchy performance and on the performance scalability between single and dual cores. Daniel Molka et al. [6] presented many fundamental details of the the Intel Nehalem microarchitecture. Their analysis is based on benchmarks to measure latency and bandwidth between different locations in the memory subsystem. We use the approach proposed by Molka to measure the access latency of remote caches.

Regarding GPUs, Volkov et al. present detailed benchmarking of the GPU memory system that reveals sizes and latencies of caches and TLB. They illustrate a couple of algorithmic optimizations, and the matrix-matrix multiply routine (GEMM) runs up to 60% faster than the vendor's implementation and approaches the peak of hardware capabilities [17]. Later, Wong et al. [18] present an analysis of the NVIDIA GT200 GPU and their measurement techniques. They used a set of micro-benchmarks to reveal architectural details of the processing cores and the memory hierarchies. Their results revealed the presence of some undocumented hardware structures. While these microbenchmarks are in CUDA and targeted NVIDIA GPUs, Thoman et al. [19] develop a set of OpenCL benchmarks targeting a large variety of platforms. Especially, they include code designed to determine parameters unique to OpenCL like the dynamic branching penalties prevalent on GPUs. They demonstrate how their results can be used to guide algorithm design and optimization, and the guided manual optimization of an example kernel results in an average improvement of 61%.

Ramos et al. [7] developed an intuitive performance model for cache-coherent architectures and demonstrated its use on Intel Xeon Phi. Their model is based on latency measurements, which match well with our latency results. In addition to the cache access latency, we have shown how we benchmark the instruction throughput, the memory bandwidth at different levels, and the interconnect performance. Thus, to the best of our knowledge, this is the first comprehensive research effort on benchmarking the Xeon Phi.

The four kernels used in our work have also been studied in prior work. In particular, Intel presents detailed steps to port Monte Carlo European Option onto the co-processor [20], but the authors did not quantify the double-precision to single-precision switch especially on the *exp*. In [12], [13], [14], the authors have examined multiple optimization techniques on the stencil kernel targeting both CPUs and GPUs. On the Xeon Phi coprocessor, we have found that `cache blocking` is still a useful optimization technique to better utilize the data caches. We also presented the SpMV performance on the Xeon Phi and reached similar conclusions with Saule [21]. With a parameter representing the number of non-zero data elements, we further and clearly illustrate the memory-bound fact on SpMV. Overall, we take these four kernels as case studies to see how to parallelize them and tune their performance from the scratch on the Xeon Phi. Most importantly, the process is guided by the customized performance roofs and we examine how to perform the optimization guidelines on them.

6 Discussion and Conclusion

Given the performance promises of Intel Xeon Phi, it is very likely to become popular in the next generation of supercomputers. Therefore, our work focused on providing key insights into the performance of this new many-core accelerator. By using a set of microbenchmarks, we characterized the three major components of this architecture - cores, memory, and interconnect - and we synthesized a set of four machine-centric optimization guidelines and a simplified machine model for facilitating kernel design and performance tuning on the Xeon

Phi.

In our instruction throughput benchmarking, we show that the achieved instruction throughput depends on the number of used cores and instantiated threads, the number of independent instruction streams, the instruction mix, and the SIMD usage. Thus, we can define and calculate *throughput roofs* to guide kernel design and computational optimizations.

The benchmarking of the memory sub-system including the two levels of caches and the off-chip memory shows that the memory access on the Xeon Phi is similar to that of multi-core CPUs. Also, we showed that accessing the local caches is an order of magnitude faster than accessing the remote caches or the off-chip memory. Regarding the memory bandwidth, we measured that the maximum `read` and `write` bandwidths are 140 GB/s and 66 GB/s, respectively. However, these bandwidths are only achievable when the accesses are contiguous and the stanzas are long enough. To estimate the *bandwidth roofs*, we make use of the maximum achievable bandwidth(s) and the ratio of read and write operations in the kernel.

The experiments performed on the ring interconnect show that the cores can be seen as *symmetric* peers. We also show that, when not all the cores are used, programmers need to place the working threads equally spread on the cores, to avoid ring traffic congestion.

To demonstrate the usability of our findings, we have selected four HPC kernels as case studies, and explored the design and tuning space guided by the performance roofs on throughput and bandwidth. Our results show that the proposed optimization strategy is functional and leads to significant performance gain (up to 100×) for regular kernels, while its impact on kernels with irregular memory accesses is limited.

Based on our experience with Xeon Phi, we believe it inherits more from traditional multi-core CPUs (than from GPUs) on both hardware characteristics and programming approach. Aimed at HPC, and backed-up by a strong compiler, the processor works well for regular, highly parallel applications. In turn, for irregular and/or data-dependent behavior, the kernels will still need manual interventions from programmers to improve the achieved performance. And while programming itself has been significantly simplified, the biggest challenge for writing well-performing Xeon Phi applications remains the identification of the proper parallelization strategy.

References

[1] Intel. Intel Xeon Phi Coprocessor. http://software.intel.com/en-us/mic-developer, April 2013. 4

[2] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. *SIGARCH Comput. Archit. News*, 38(3):451–460, June 2010. 4

[3] Intel. *Intel Xeon Phi Coprocessor System Software Development Guide*, November 2012. 4, 6, 10

[4] Intel. *An Overview of Programming for Intel Xeon Processors and Intel Xeon Phi Coprocessors*, October 2012. 5

[5] Alan J. Smith and Rafael H. Saavedra. Measuring cache and TLB performance and their effect on benchmark runtimes. *IEEE Trans. Comput.*, 44(10):1223–1235, October 1995. 7, 19

[6] Daniel Molka, Daniel Hackenberg, Robert Schöne, and Matthias S. Müller. Memory performance and cache coherency effects on an intel nehalem multiprocessor system. In *18th International Conference on Parallel Architectures and Compilation Techniques, 2009. PACT '09.*, pages 261–270. IEEE, September 2009. 8, 19

[7] S. Ramos Garea and T. Hoefler. Modeling Communication in Cache-Coherent SMP Systems - A Case-Study with Xeon Phi. 2013. Accepted at HPDC'13. 8, 19

[8] C. J. Hughes, Changkyu Kim, and Yen-Kuang Chen. Performance and energy implications of Many-Core caches for throughput computing. *Micro, IEEE*, 30(6):25–35, November 2010. 8

[9] John D. McCalpin. STREAM: Sustainable Memory Bandwidth With High Performance Computers. http://www.cs.virginia.edu/stream/, April 2013. 10

[10] Kaushik Datta, Shoaib Kamil, Samuel Williams, Leonid Oliker, John Shalf, and Katherine Yelick. Optimization and performance modeling of stencil computations on modern microprocessors. *SIAM Rev.*, 51(1):129–159, February 2009. 10

[11] Folding@home. What is a FLOP? http://folding.stanford.edu/English/FAQ-flops, April 2013. 14

[12] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, Piscataway, NJ, USA, 2008. IEEE Press. 15, 16, 19

[13] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams. An auto-tuning framework for parallel multicore stencil computations. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, April 2010. 15, 16, 19

[14] M. Christen, O. Schenk, and H. Burkhart. PATUS: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 676–687. IEEE, May 2011. 15, 16, 19

[15] Nathan Bell and Michael Garland. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, December 2008. 18

[16] Lu Peng, Jih-Kwon Peir, Tribuvan K. Prakash, Carl Staelin, Yen-Kuang Chen, and David Koppelman. Memory hierarchy performance measurement of commercial dual-core desktop processors. *Journal of Systems Architecture*, 54(8):816–828, August 2008. 19

[17] Vasily Volkov and James W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press. 19

[18] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. Demystifying GPU microarchitecture through microbenchmarking. In *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, pages 235–246. IEEE, March 2010. 19

[19] Peter Thoman, Klaus Kofler, Heiko Studt, John Thomson, and Thomas Fahringer. Automatic OpenCL device characterization: Guiding optimized kernel design. In Emmanuel Jeannot, Raymond Namyst, and Jean Roman, editors, *Euro-Par 2011 Parallel Processing*, volume 6853 of *Lecture Notes in Computer Science*, pages 438–452. Springer Berlin Heidelberg, 2011. 19

[20] Intel. Achieving High Performance on Monte Carlo European Option on Intel? Xeon Phi? Coprocessors. http://software.intel.com/en-us/articles/achieving-high-performance-on-monte-carlo\discretionary{-}{}{}, April 2013. 19

[21] Erik Saule, Kamer Kaya, and Umit V. Catalyurek. Performance evaluation of sparse matrix multiplication kernels on intel xeon phi, February 2013. 19