# Test-Driving Intel Xeon Phi*

Jianbin Fang
TU Delft, the Netherlands
j.fang@tudelft.nl

Henk Sips
TU Delft, the Netherlands
h.j.sips@tudelft.nl

Lilun Zhang
NUDT, China
llzhang@nudt.edu.cn

Chuanfu Xu
NUDT, China
xuchuanfu@nudt.edu.cn

Yonggang Che
NUDT, China
ygche@nudt.edu.cn

Ana Lucia Varbanescu
UvA, the Netherlands
a.l.varbanescu@uva.nl

## ABSTRACT

Based on Intel's Many Integrated Core (MIC) architecture, Intel Xeon Phi is one of the few truly many-core CPUs - featuring around 60 fairly powerful cores, two levels of caches, and graphic memory, all interconnected by a very fast ring. Given its promised ease-of-use and high performance, we took Xeon Phi out for a test drive. In this paper, we present this experience at two different levels: (1) the microbenchmark level, where we stress "each nut and bolt" of Phi in the lab, and (2) the application level, where we study Phi's performance response in a real-life environment. At the microbenchmarking level, we show the high performance of five components of the architecture, focusing on their maximum achieved performance and the prerequisites to achieve it. Next, we choose a medical imaging application (Leukocyte Tracking) as a case study. We observed that it is rather easy to get functional code and start benchmarking, but the first performance numbers can be far from satisfying. Our experience indicates that a simple data structure and massive parallelism are critical for Xeon Phi to perform well. When compiler-driven parallelization and/or vectorization fails, programming Xeon Phi for performance can become very challenging.

## Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

## Keywords

Experience with Xeon Phi, Microbenchmarking, Performance Analysis, Optimization.

*Part of the work was done by Jianbin Fang during an internship, funded by the Natural Science Foundation of China under Grant No.11272352, at NUDT in January 2013.


*ICPE'14,* March 22–26, 2014, Dublin, Ireland.

http://dx.doi.org/10.1145/2568088.2576799.

## 1. INTRODUCTION

Intel Xeon Phi (Phi) is the newest high-throughput architecture targeted at high performance computing (HPC). Without a doubt, Phi will be part of the very next generation of supercomputers that will challenge TOP500[1].

To achieve its theoretical high performance (around 1 TFlop), Intel Xeon Phi [15] uses around 60 cores and 30 MB of on-chip caches, and relies on traditional many-core features like vector units or SIMD/SIMT, high throughput, and high bandwidth [17]. It adds to that some "unconventional" features, such as the overall L2 cache coherency and the ring interconnect, all for the sake of performance and usability.

By taking Phi as a black-box with over 200 hardware threads, we ran Leukocyte Tracking (a medical imaging application [21]) on it. We found that (1) the sequential application (a single thread) on Phi runs about 5× slower than the same sequential execution on a "traditional" multi-core processor, and (2) that the Phi version scales only up to 40 threads (Figure 13, more details in Section 5). To explain this (observed) performance behavior, as well as to eventually improve it, we require a deeper understanding of the architecture and its parameters.

Moreover, previous experiences with massively parallel high performance platforms such as NVIDIA GPUs or the Cell-/BE showed that a trade-off between performance and ease-of-use is necessary: "simple" programming often leads to disappointing performance [22,27]. Therefore, given Phi's promise of breaking this pattern, this work focuses on a test drive of the platform: we have conducted a two-stage empirical study of the Xeon Phi, stressing its high-performance features both in isolation (aiming to quantify their maximum achievable performance), and in the real-life case-study (aiming to understand its regular performance).

To this end, we have implemented and used dedicated microbenchmarks - gathered in a suite called *MIC-Meter* [2] - to measure the performance of four key architectural features of Xeon Phi: the processing cores, the memory hierarchies, the ring interconnect, and the PCIe connection. Following these experiments "in isolation", we propose a conceptual model of the processor that facilitates the performance analysis and optimization of the real-life case-study.

Such a thorough evaluation can benefit two different classes of Phi users: the *experts*, who are interested in in-depth architectural knowledge, and the *production users*, interested

[1]In June 2013, two Xeon Phi supercomputers - TIANHE-2 from NUDT and STAMPEDE from TACC - were ranked first and sixth in TOP500: `http://www.top500.org.`
[2]`https://github.com/haibo031031/mic-meter`

in simple and effective ways to use processors. For expert users - like most high performance computing (HPC) programmers and compiler developers are - knowing the requirements for density and placement of threads per cores, the optimal utilization 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 significant performance gains. For production users, a simplified view of the Xeon Phi machine is mandatory to help exploring different parallelism strategies. Such a model is simplified view of the machine, including the most important functionality and performance constraints.

The main contributions of our work are as follows:

- We present our hands-on experience achieved while microbenchmarking the Xeon Phi (Section 3). This experience also leads to interesting numerical results for the capabilities of Phi's cores, memories, interconnects (i.e., the ring and the PCIe).
- We synthesize four essential platform-centric performance guidelines, aimed at easing the development and tuning of applications for the Xeon Phi (Section 4).
- We propose a conceptual model of Phi (*SCAT*), which strips off the performance irrelevant architectural details, presenting the programmers with a simple, functionality-based view of the machine (Section 4).
- Using a case study (leukocyte tracking), we analyze the application and optimize it, discussing the lessons to be learned from this experience (Section 5).

# 2. BENCHMARKING INTEL XEON PHI

In this section, we introduce Intel Xeon Phi - with its novel features and typical programming models, and we present our benchmarking methodology.

## 2.1 The Architecture

Intel Xeon Phi has over 50 cores (the version used in this paper belongs to the 5100 series and has 60 cores) connected by a high-performance on-die bidirectional interconnect (shown in Figure 1). In addition to these cores, there are 16 memory channels (supported by memory controllers) delivering up to 5.0 GT/s [12]. When working as an accelerator, Phi can be connected to a host (i.e., a device that manages it) through a PCI Express (PCIe) system interface - similar to GPU-like accelerators. Different from GPUs, a dedicated embedded Linux $\mu$OS (version: `2.6.38.8`) runs on the platform.

Each core contains a 512-bit wide vector unit (VPU) with 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. In total, a 60-core machine has a total of 30MB of L2 cache on the die. The L2 caches are kept fully coherent by the hardware, using DTDs (distributed tag directories), which are referenced after an L2 cache miss. Note that the tag directory is not centralized, but split up into 64 DTDs, each getting an equal portion of the address space and being responsible for maintaining it globally coherent. Another special feature of Xeon Phi is the fast bidirectional ring interconnect. All connected entities use the ring for communication purposes, using special controllers called *ring stops* to insert requests and receive responses on the ring.

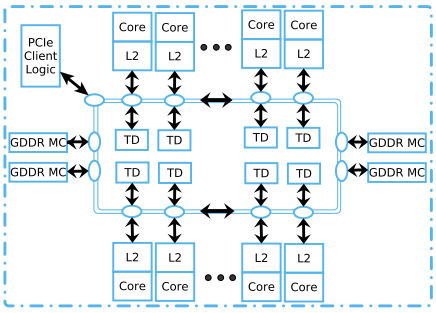



**Figure 1: The Intel Xeon Phi Architecture.**

The novelties of the Xeon Phi architecture relate to five components : (1) the vector processing cores, (2) the on-chip memory, (3) the off-chip memory, (4) the ring interconnect, and (5) the PCIe connection. As these are the features that differ, in one way or another, from a typical CPU - vectors are wider, there are many more cores, cache coherency and shared memory are provided with low penalty for 60 or more cores, and a ring interconnect holds tens of agents that can interchange messages/packets concurrently -, we focus our benchmarking efforts on these features.

## 2.2 Programming

In terms of usability, there are two ways an application can use Intel Xeon Phi: (1) in *offload mode* - the main application is running on the host, and it only offloads selected (highly parallel, computationally intensive) work to the coprocessor, or (2) in *native mode* - the application runs independently, on the Xeon Phi only, and can communicate with the main processor or other coprocessors [13] through the system bus. In this work, we benchmark Xeon Phi in both modes.

Finally, to program applications on Xeon Phi, users need to capture both functionality and parallelism. Being an `x86` SMP-on-a-chip architecture, Xeon Phi offers the full capability to use the same tools, programming languages, and programming models as a regular Intel Xeon processor. Specifically, tools like Pthreads [5], OpenMP [2], Intel Cilk Plus [1], and OpenCL [24] are readily available. Given the large number of cores on the platform, a dedicated MPI version is also available. In this work, all the experiments we present are programmed using C/intrinsics/assembly with OpenMP/Pthreads; we also use Intel's `icc` compiler (`V13.1.1.163`).

## 2.3 MIC-Meter

We show our MIC-Meter in Figure 2. The goal of our benchmarking is two-fold: to show how the special capabilities of Xeon Phi can and should be measured, to quantify the performance of this novel many-core architecture, and eventually to identify the impacting factors. To this end, we choose a microbenchmarking approach: we measure each capability in isolation, under variable loads, and we quantify its performance in terms of both latency-oriented and throughput-oriented metrics.

Simply put, *latency* is the time required to perform an operation and produce a result. As latency measurement focuses on a single action from its beginning to its end, one needs to isolate the operation to be measured and use

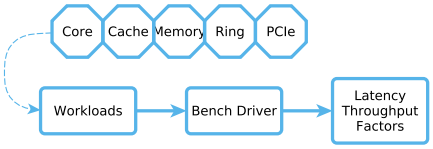



**Figure 2: The MIC-Meter Overview.**

a highly accurate, non-intrusive timing method. Alternatively, we can measure a long enough sequence of operations with an accurate timer, and estimate latency per operation by dividing the measured time by the number of operations. In this paper, latency measurements are done with a single thread (for individual operations) or two threads (for transfer operations) with Pthreads. All latency benchmarks are written in C (with inline assembly).

*Throughput* is the number of (a type of) operations executed in a given unit of time. As higher throughput means better performance, microbenchmarking focuses on measuring the *maximum achievable throughput* for different operations, under different loads; typically, the benchmarked throughput values are slightly lower than the theoretical ones. Thus, to measure maximum throughput, the main challenge is to build the workload such that the resource that is being evaluated is fully utilized. For example, when measuring computational throughput, enough threads should be used to fully utilize the cores, while when measuring memory bandwidth, the workload needs to have sufficient threads to generate enough memory requests. For all the throughput measurements in this paper, our multi-threaded workloads are written in C and OpenMP.

We note that the similarities between Phi and a regular multi-core CPU allow us to adapt existing CPU benchmarks to the requirements of Xeon Phi. In most cases, we use such "refurbished" solutions, that prove to serve our purposes.

# 3. EMPIRICAL EVALUATION

In the following sections, we present in detail the MIC-Meter and the results for each of the components: (1) the vector processing cores, (2) the on-chip and off-chip memory, (3) the ring interconnect, and (4) the PCIe connection.

## 3.1 Vector Processing Cores

We evaluate the vector processing cores in terms of both instruction latency and throughput. For latency, we use a method similar to those proposed by Agner Fog [7] and Torbjorn Granlund [9]: we measure instruction latency by running a (long enough) sequence of dependent instructions (i.e., a list of instructions that, being dependent on each other, are forced to be executed sequentially - *an instruction stream*).

The same papers propose a similar approach to measure throughput in terms of *instructions per cycle (IPC)*. However, we argue that a measurement that uses all processing cores together, and not in isolation, is more realistic for programmers. Thus, we develop a `flops` microbenchmark to explore the factors for reaching the theoretical maximum throughput on Xeon Phi (Section 3.1.2).

### 3.1.1 Vector Instruction Latency

Xeon Phi introduces 177 vector instructions [11]. We roughly divide these instructions into five classes [3]: mask instructions, arithmetic (logic) instructions, conversion instructions, permutation instructions, and extended mathematical instructions.

The benchmark for measuring the latency of vector instructions is measuring the execution time of a sequence of 100 vector operations using the same format: $zmm1 = op(zmm1, zmm2)$, where $zmm1$ and $zmm2$ represent two vectors and $op$ is the instruction being measured. By making $zmm1$ be both a source operand and the destination operand, we ensure the instruction dependency - i.e., the current operation will depend on the result of the previous one.

For special classes of instructions - such as the `conversion` instructions `vcvtps2pd` and `vcvtpd2ps` - we have to measure the latency of the conversion pair ($zmm2 = op12(zmm1)$; $zmm1 = op21(zmm2)$) in order to guarantee the dependency between contiguous instructions (i.e., it is not possible to write the result of the conversion in the same source operand, due to type incompatibility). Similarly, we measure the latency of extended mathematical instructions such as `vexp223ps` and `vlog2ps` in pairs, to avoid overflow (e.g., when using 100 successive `exp()`'s).

The interesting results for vector instruction latency are presented in Table 1. With these latency numbers, we know how many threads or instruction streams we need to hide the latency on one processing core.

**Table 1: The vector instruction latency (in cycles).**

| Instruction | Category | Latency |
|---|---|---|
| kand, kor, knot, kxor | mask instructions | 2 |
| vaddpd, vfmadd213pd, vmulpd, vsubpd | arithmetic instructions | 4 |
| vcvtdq2pd, vcvtfxpntdq2ps, vcvtfxpntps2dq, vcvtps2pd | convert instructions | 5 |
| vpermd, vpermf32x4 | permutation instructions | 6 |
| vexp223ps, vlog2ps, vrcp23ps, vrsqrt23ps | extended mathematical instructions | 6 |

### 3.1.2 Vector Instruction Throughput

The Xeon Phi 5100 has 60 cores working at 1.05 GHz, and each core can process 8 double-precision data elements at a time, with maximum 2 operations (`multiply-add` or `mad`) per cycle in each lane (i.e., a vector element). Therefore, the theoretical instruction throughput is 1008 GFlops (approximately 1 TFlop). But **is this 1 TFlop performance actually achievable?** To measure the instruction throughput, we run 1, 2, 4 threads on a core (60, 120, and 240 threads in total). During measurement, each thread performs one or two instruction streams for a fixed number of iterations: $b_{i+1} = b_i \ op \ a$, where $i$ represents the iteration, $a$ is a constant, and $b$ serves as an operand and the destination. The loop was fully unrolled to avoid branch overheads. The microbenchmark is vectorized using explicit intrinsics, to ensure a 100% vector usage.

[3]Note that we choose not to measure the latency of memory access instructions because the latency results are highly dependent on the data location(s).

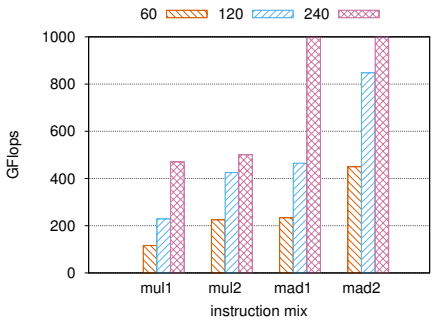



**Figure 3: Arithmetic throughput using different numbers of threads (60, 120, 240), different instruction mixes (`mul` versus `mad`), and issue widths (using one and two independent instruction streams).**

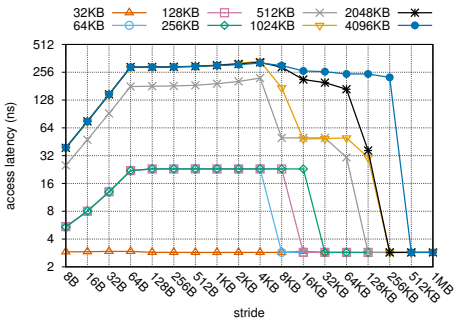



**Figure 4: Average memory latency when changing strides and datasets. The x-axis is logarithmic and it represents the pointer chasing stride.**

The results are shown in Figure 3. We note that the peak instruction throughput - i.e., one vector instruction per cycle (1TFlops in total) - can be achieved when using 240 threads and the multiply-add instruction. As expected, the `mad` throughput is twice larger than the `mul` throughput. Further, two more observations can be added. First, when using 60 threads (one thread per core), the instruction throughput is low compared with the cases when using 120 or 240 threads. This is due to the fact that it is not possible to issue instructions from the same thread context in back-to-back cycles [12]. Thus, programmers need to run at least two threads on each core to be able to fully utilize the hardware resources. Second, when a thread is using only one instruction stream, we have to use 4 threads per core (240 threads in total) to achieve the peak instruction throughput. This is because the latency of an arithmetic instruction is 4 cycles (Table 1), and we need no less than four threads to totally hide this latency (i.e., fill the pipeline bubbles [10]). To comply, programmers need to either use 4 threads per core or have more independent instruction streams.

To summarize, for a given instruction mix (`mul` or `mad`), the achievable instruction throughput depends not only on the number of cores and threads, but also on the issue width (i.e., the number of independent instruction streams). We also benchmarked the EMU (extended math unit) and see [6] for more details.

## 3.2 Memory Latency

Available benchmarks, such as `BenchIT` [25] and `lmbench` [18] use *pointer-chasing* to measure the on-chip and off-chip memory access latency. This approach has the advantage of not only determining the latency itself, but also exposing the differences between consecutive layers of a memory hierarchy (i.e., different layers of caches and main memory will have significantly different latencies). Thus, we use a similar approach to measure the latency for an Xeon Phi core (i.e., the latency for accessing local caches and main memory - see Section 3.2.1).

When more than two cores communicate, measuring latency is complicated. For this, Daniel Molka et al. proposed an approach to quantify cache-coherency effects [19]. In our work, we adapt this approach to Xeon Phi using the correct memory fences and cache flushing instructions [4].

### 3.2.1 Access Latency on a Single Core

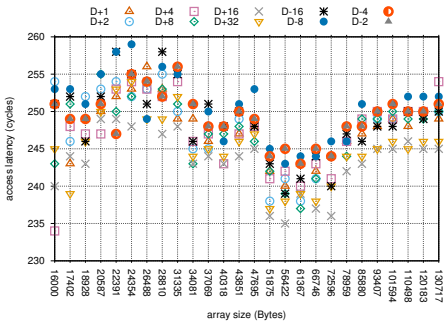
To reveal the local access latency, we use a *pointer-chasing* benchmark similar to those used by `BenchIT` and `lmbench`. Essentially, the application traverses an array $A$ of size $S$ by running $k = A[k]$ in a fully unrolled loop. The array is initialized with a *stride*, i.e., $A[k] = (k + stride)\%S$. By measuring the execution time of the traversal, we can easily obtain an estimate of the average execution time for one iteration. This time is dominated by the latency of the memory access. The traversal is done in one thread and utilizes only one core. Therefore, the memory 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 2.87 ns (3 cycles) and 22.98 ns (24 cycles), respectively. With a stride of 64 bytes, Xeon Phi takes 287.51 $\sim$ 291.18 ns (302 $\sim$ 306 cycles) to finish a data access in the main memory (when the dataset is larger than 512KB). We note that when traversing the array in a larger stride (e.g., 4KB), the latency of accessing data in off-chip memory is slightly larger. This is because the contiguous memory accesses fall into different pages. Furthermore, we can observe (from the upper trend) that threads operate the data in a batch manner, i.e., a 64-byte cache-line. Information about cache associativity can also be seen in Figure 4 (see [23] for the calculation approach).

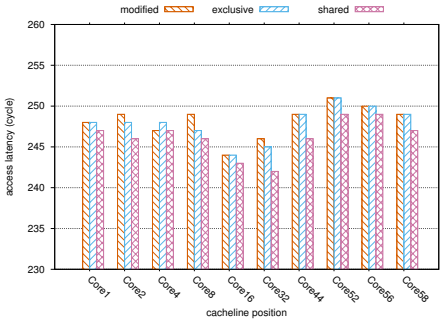
### 3.2.2 Remote Cache Latency

We have illustrated our measurements and results for memory latency on a single core in Section 3.2.1. In this section, we focus on measuring remote cache latency. For these measurements, we use an approach based on that proposed for a traditional multi-core processor by Daniel Molka [19]. Our setup is built as follows: prior to the measurement, the to-be-transferred cache-lines are placed in different locations (cores) and in a certain coherency state (`modified`, `exclusive`, or `shared`). In each measurement, we use two threads

[4]Since Xeon Phi has no `mfence` or `clflush`, we need to change the benchmark by searching and replacing them with equivalent instructions.




(a) Modified




(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 44 (D-16), Core 52 (D-8), Core 56 (D-4),and Core 58 (D-2).**

(T0, T1), with T0 pinned to Core 0 and T1 pinned on another core (Core $X$). The latency measurement always runs on Core 0, transferring a predefined number of cache lines from Core $X$ to Core 0.

Figure 5 shows our results for remote cache accesses latency on Xeon Phi. In Figure 5(a), we see that when the cache line is in `modified` state, the overall latency of remote access averages around 250 cycles, which is much larger than the local cache access latency (by an order of magnitude) but still smaller than the off-chip memory access latency (by 17%). By getting the *median* value of all the input data sets (up to 128 KB), we get the overall remote latency shown in Figure 5(b). We note no relationship between the remote access latency and the cache-line states, except that accessing remote `shared` cachelines takes a few less cycles. This is because in whichever state a cacheline is, when a core accesses it, a transfer is needed from a remote core (different from a traditional multi-core CPU with cores sharing the last-level cache). Furthermore, Xeon Phi adopts the `MOESI` cache coherence protocol [12] to share a cacheline before writing it back, and thus Figure 5(b) shows no penalty of writing data back. In [6], our experiments have shown that there is a relation between the latency and the core distances on an older version of the Xeon Phi (namely, 31S1P), but this effect seems to have disappeared on the newer Xeon Phi 5110.

## 3.3 Memory Bandwidth

McCalphin's stream benchmark [16] includes a memory bandwidth benchmark and presents results for a large number of high-end systems. However, his solution is based on a combination of both read and write operations. In this paper, we want to separate *reads* and *writes* so as to quantify the impacting factors. In `BenchIT`, Daniel Molka et al. presents a solution to measure bandwidth in a similar way with that of latency measurement (see Section 3.2.2). His microbenchmark requires compiler optimizations to be disabled (i.e., the code should be compiled with the `-O0` option), thus disabling the software prefetching on Xeon Phi. As a result, this measurement will underestimate bandwidth. In this section, we present our own `OpenMP` implementation of a memory bandwidth microbenchmark, considering hardware/software prefetching, streaming stores, ECC effects and off-chip/on-chip differences.

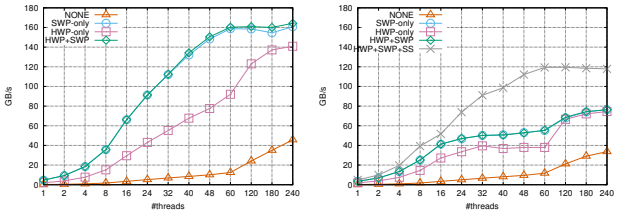
### 3.3.1 *Off-Chip Memory Bandwidth*

The Xeon Phi used in this work has 16 memory channels, each 32-bits wide. At up to 5.0 GT/s transfer speed [5], it provides a theoretical bandwidth of 320 GB/s. But **is this theoretical bandwidth really achievable in real cases?** To answer this question, we use separate benchmarks to measure the memory bandwidth for both `read` and `write` operations. The `read` benchmark reads data from an array $A$ ($b = b + A[k]$). The `write` benchmark writes a constant value into an array $A$ ($A[k] = C$). Note that $A$ needs to be large enough (e.g., 1 GB) such that it cannot fit in the on-chip memory. To avoid the impact of "cold" TLBs, we start with two "warm-up" iterations of the benchmarks, before we measure a third one. We use different numbers of running threads - from 1 to 240.

Our results are shown in Figure 6 (HWP+SWP) (we plot the median value of ten runs of the benchmarks). Overall, we see that the maximum bandwidth for both `read` and `write` is far below the theoretical peak of 320 GB/s. Moreover, both the `read` and `write` memory bandwidth increases over the number of threads - which happens because when using more threads, we can generate more requests to memory controllers, thus making the interconnect and memory channels busier. Thus, if aiming to achieve high memory bandwidth, programmers need to launch enough threads to saturate the interconnect and the memory channels. Figure 6(a) shows that the `read` bandwidth peaks at 164 GB/s, achievable with using 60 threads or more (pinning at least one thread to a core). However, we can obtain the maximum `write` bandwidth (76 GB/s, as seen in Figure 6(b)) only when using 240 threads. In general, the `write` bandwidth is around half of the `read` bandwidth. This happens because Xeon Phi implements a write-allocate cache policy and the original content has to be loaded into caches before we overwrite it completely. To avoid the memory bandwidth waste, programmer can use *streaming stores* [6] on Xeon Phi [14]. We see that using *streaming store* instructions speeds-up write operations up to 1.7 times (Figure 6(b):HWP+SWP+SS), with memory write bandwidth now peaking at 120 GB/s. Thus, programmers must consider using *streaming stores* to optimize the memory bandwidth.

[5] GT/s stands for Giga Transfers per second.

[6] Streaming stores do not require a prior cache line read for ownership (RFO) but write to memory "directly".




(a) read (b) write

**Figure 6: Read and write memory bandwidth.**

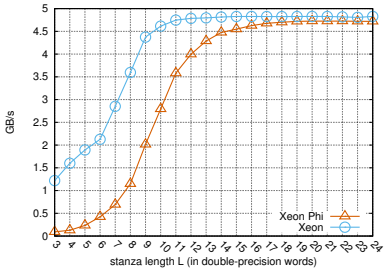



**Figure 7: Performance of STriad on the Xeon Phi (the x-axis is in log scale and the results on Xeon are normalized to those on Xeon Phi).**

**Prefetch Effects:** Xeon Phi supports both hardware prefetching (HWP) and software prefetching (SWP). The L2 cache has a streaming hardware prefetcher that can selectively prefetch code, read, and RFO (Read-For-Ownership) cachelines into the L2 cache [12]. Figure 6 shows the memory bandwidth of four different configurations: no prefetching, HWP or SWP only, or both. When disabling both HWP and SWP, the memory bandwidth is low (45 GB/s for reading and 33 GB/s for writing). With only SWP, we already achieve similar memory bandwidth to that achieved when enabling both of them. This similarity indicates that the hardware prefetcher will not kick in when software prefetching performs well. Furthermore, enabling only HWP delivers about half of the bandwidth achieved when enabling only SWP (the bandwidth is roughly 1.9× smaller, on average).

To further evaluate the efficiency of prefetching on Xeon Phi, we use the Stanza Triad (`STriad`) [4] benchmark with a single thread. `STriad` works by performing a DAXPY (Triad) inner loop for a length $L$ stanza, then jumps over $k$ elements, and continues with the next $L$ elements, until reaching the end of the array. We set the total array size to 128 MB, and set $k$ to 2048 double-precision words. For each stanza, we ran the experiment 10 times, with the L2 cache flushed each time, and we calculate median value of the 10 runs to get the memory bandwidth for each stanza length. Figure 7 shows the results of the STriad experiments on both Xeon Phi and a regular Xeon processor (Intel Xeon E5-2620). We see an increase in memory bandwidth over stanza length $L$, and we note it eventually approaches a peak of 4.7 GB/s (note that this is achieved per core). Further, we see the transition point (from the bandwidth-increasing state to the bandwidth-stable state) appears earlier on Xeon than on Xeon Phi. Therefore, we conclude that non-contiguous access to memory is detrimental to memory bandwidth efficiency, with Xeon Phi showing more restrictions on the stanza length when prefetching data than the regular Xeons. To comply, programmers have to create the longest possible stanzas of contiguous memory accesses, improving prefetching and memory bandwidth.

**ECC Effects:** The Xeon Phi coprocessor supports ECC (Error Correction Code) to avoid software errors caused by naturally occurring radiation. Enabling ECC adds reliability, but it also introduces extra overhead to check for errors. We examined the bandwidth differences with and without disabling ECC. With ECC disabled, we noticed a 20% to 27% bandwidth increase [6]. Note that all the experiments in this paper are performed with ECC enabled. Furthermore, the new $\mu$OS kernel on Phi adds support of the transparent huge pages (THP) functionality, which is enabled by default and often improves application performance without any code or environmental changes.

### 3.3.2 *Aggregated On-Chip Memory Bandwidth*

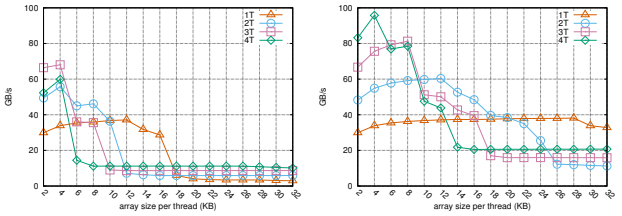
The available on-chip memory bandwidth is always essential in performance tuning and analysis. So, **how large is the on-chip memory bandwidth that can be achieved?** To answer this question, we measure the cache bandwidth on a single core [7] and calculate the *aggregated* cache bandwidth by multiplying it with the number of cores. We first use a set of `vmovapd` instructions to measure the native `read` or `write` bandwidth. Our results show that the L1 access (read or write) throughput is 64 bytes per cycle. Thus, the aggregated L1 bandwidth is 4032 GB/s for `read` or `write`. Then we measure the maximum achieved bandwidth from programmers' point of view for `scale1` ($O[i] = a \times A[i]$), `scale2` ($O[i] = a \times O[i]$), `saxpy1` ($O[i] = a \times A[i] + B[i]$), and `saxpy2` ($O[i] = a \times A[i] + O[i]$) operations. To avoid overheads from the high-level code, we use intrinsics in the kernel code. We also disable the software prefetching due to the fact that the data is located in caches after warming up.

The results are shown in Figure 8. We see that the maximum achieved bandwidth on a core is 73 GB/s, 96 GB/s, 52 GB/s, 69 GB/s for `scale1`, `scale2`, `saxpy1`, `saxpy2`, respectively. The bandwidth of `scale2` and `saxpy2` is 1.3× larger (than `scale1` and `saxpy1`, respectively) because the data cache allows a read/write cache-line replacement to happen in a single cycle [8]. The L1 bandwidth on a single core could be larger when further unrolling the loops or better scheduling instructions for each dataset. The aforementioned numbers are achieved by unrolling the loops 16 times without changing the assembly code.

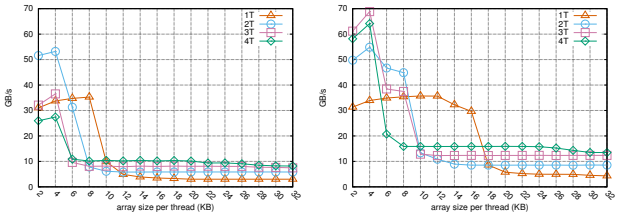
Furthermore, it is difficult to exactly measure the L2 bandwidth due to the presence of the L1 cache. The bandwidth depends on the memory access patterns. Specifically, when we use a L2-friendly memory access pattern, the compiler will identify the stream pattern and prefetch data to the L1 cache in time. By this, we will get a much larger bandwidth due to the common efforts of L1 and L2. On the other hand, an unfriendly memory access will experience many L1 misses

[7]Note that we choose not to measure the inter-core communication bandwidth because we assume that cache-line transfers occur rather scattered, and not in a large volume. Thus, the measurement of inter-core (remote) access latency is of greater use.

[8]`http://software.intel.com/en-us/articles/intel-xeon-phi-core-micro-architecture`

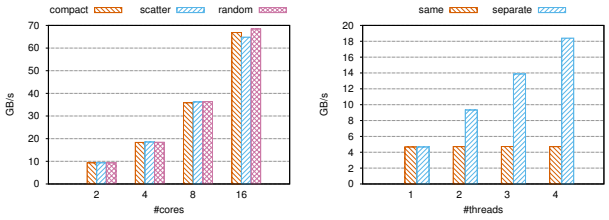



(a) scale1 bandwidth. (b) scale2 bandwidth.




(c) saxpy1 bandwidth. (d) saxpy2 bandwidth.

**Figure 8: Cache bandwidth on a single core.**




(a) Core Distribution (b) Thread Distribution

**Figure 9: Core and thread distribution effects (we use the `read` kernel and the array size is 1 GB).**

and result in cache thrashing. Our benchmarking results are obtained when disabling the software prefetching. When using 4 threads on a core, we notice a bandwidth of 11 GB/s, 20 GB/s, 10 GB/s, 16 GB/s for `scale1`, `scale2`, `saxpy1`, `saxpy2`, respectively (Figure 8).

## 3.4 Ring Interconnect

On Xeon Phi, the cores and memory controllers are interconnected in a bi-directional ring. When multiple threads are requesting data simultaneously, shared components like the ring stop or DTDs can become performance bottlenecks. In order to check this hypothesis, and its eventual performance impact, we use *thread affinity* to fix threads on cores, and we run the bandwidth microbenchmarks to quantify potential bandwidth changes (in GB/s) for different thread-to-core mapping scenarios.

### 3.4.1 Core/Thread Distribution

First, we measure the `read` memory bandwidth by distributing threads onto separate cores in three different patterns: (1) *compact* - the cores are located close to each other, (2) *scattered* - the cores are evenly distributed around the ring, and (3) *random* - the core `IDs` are selected randomly with no repeats. The bandwidths are measured using 2, 4, 8, and 16 cores and the results are presented in Figure 9(a). We see that the three approaches achieve very similar memory bandwidths. Thus, the cores around the ring are *symmetric*

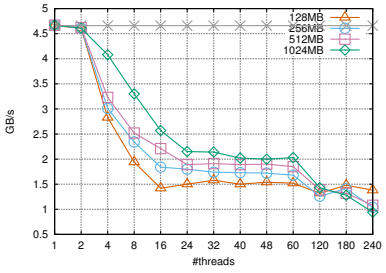



**Figure 10: The memory bandwidth when the threads read the same memory space.**

on Xeon Phi, and the distance between has practically no impact on the achieved bandwidth.

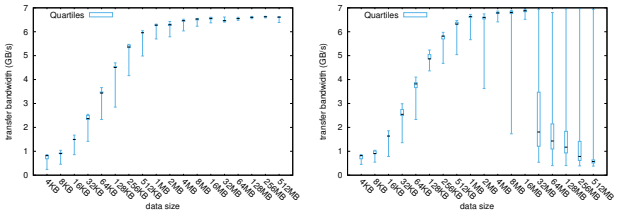
Second, as each Xeon Phi core supports up to four hardware threads, we investigate whether there is any impact on bandwidth if the threads are all gathered on the same core (thus, less interconnect traffic) or distributed among different cores. Figure 9(b) shows that when the threads run on the same core, the bandwidth stabilizes at 4.7 GB/s. We also note that running threads on separate cores results in a linear bandwidth increase with the number of threads. We conclude that when multiple threads on the same core request data simultaneously, they will compete for the shared hardware resources (e.g., the ring stops), thus serializing the requests. On the bright side, the threads located on the same core share cache data and have faster data accesses (see Section 3.2).

### 3.4.2 Accessing Shared-Data

Section 3.4.1 focuses on the achieved bandwidth when threads access separate memory spaces. In this section we investigate **what is the bandwidth when different threads access the same memory space simultaneously?** We expect that the bandwidth would resemble that obtained by a single thread, assuming the memory requests are served by *broadcasting*. Figure 10 presents the measured bandwidth, showing that the `read` bandwidth decreases over the number of threads until 24 (or 16). Thereafter, the bandwidth is constant around 1.5-2.0 GB/s (i.e., one third of the single thread bandwidth). When using more threads than cores, the bandwidth drops even further. This behavior is different from the linear increase trend (shown in Figure 9(a)) seen when accessing separate memory spaces. We assume the bottleneck lies in the simultaneous access to the DTDs. Therefore, for bandwidth gain, applications should strive to keep threads accessing different parts/cachelines of the shared memory space (for as much as possible), to avoid the effects of contention at the interconnect level.

## 3.5 PCIe Data Transfer

When used as a coprocessor, Xeon Phi is connected via PCIe to a host (e.g., a traditional CPU). When offloading computation to the Xeon Phi, the tasks and the related data need to be transferred back and forth between the two processors. As seen for GPUs [**?**], these transfers can be expensive in terms of overall application performance. Thus, we have designed a benchmark to measure the data transfer bandwidth. To do so, we use the `offload` pragma (specifying `in` and `out` for the transfer direction) to transfer datasets




(a) Host ⟹ Phi (b) Phi ⟹ Host

**Figure 11: Achieved data transfer bandwidth (over PCIe) between a host and an Xeon Phi.**

of different sizes from host to Xeon Phi and back. The transferred data is allocated with a 4KB alignment, for optimal DMA performance [12].

The achieved bandwidth between host and Xeon Phi is presented in Figure 11 (we report the results over 1000 times). We note that the bandwidth increases with data size, and it is relatively stable for different runs, for both directions. However, for data transfers larger than 32 MB, the Phi to host bandwidth shows a large variation, with the `median` bandwidth value decreasing sharply (up to 6 times!). The reasons for this large variance are still under investigation.

# 4. SCAT: AN XEON PHI MODEL

We compare our results with the information provided by the Intel Software Development Guide (SDG) in Table 2. We note that we did improve on the content of the official data: instruction latency data, local and non-local memory access bandwidth and latency data, an interconnect study, and a PCIe offload evaluation. We also have the following key observations, which lead to optimization guidelines.

*High Throughput:* 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 threads/core occupancy, (2) the utilization of the 512-bit vectors, (3) the issuing width (i.e., the number of independent instruction streams), (4) the instruction

**Table 2: A comparison with the data in SDG ('N/A' stands for "not available" in SDG).**

| Metric | SDG | Measured |
|---|---|---|
| VPU | | |
| Latency | general statement | cycles/instruction |
| Throughput | 1008 GFlops | 1008 GFlops |
| EMU evaluation | general statement | quantified |
| L1 Cache (32KB) | | |
| Latency (local) | 1 cycle | 3 cycles |
| bandwidth (local) | N/A | R=64B/c;W=64B/c |
| L2 Cache (<512KB) | | |
| Latency (local) | 11 cycles | 24 cycles |
| Bandwidth (local) | N/A | quantified |
| Latency (remote) | N/A | 250 cycles |
| Off-chip memory | | |
| Latency | N/A | 302 cycles |
| Bandwidth | 320 GB/s | R=164GB/s;W=76GB/s |
| Prefetching | general statement | quantified |
| ECC factor | general statement | quantified |
| Interconnections | | |
| Ring Traffic Contention | N/A | ring stops, DTDs |
| PCI Express Bandwidth | N/A | up to 7 GB/s |

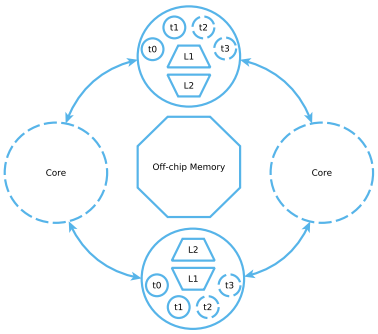



**Figure 12: The SCAT model of Intel Xeon Phi.**

mix. Furthermore, single-precision data leads to better performance for math-intensive kernels.

*Memory Selection:* Accessing the local L1 cache is 8 times 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 (17%). Furthermore, the remote access latency does not depend on the cache-line state.

*Efficient Memory Access:* Data is read and written from/to the off-chip memory in cache lines (64 bytes). The maximum achievable bandwidth is 164 GB/s for `read` operations and 76 GB/s for `write` operations - a lot lower than the theoretical peak of 320 GB/s. With `streaming store` instructions, the `write` bandwidth can increase up to 1.7 times. Further, programmers need many threads (at least 60 - one per core) to issue enough memory requests to saturate the ring interconnect and the memory channels. The hardware and software prefetching can improve bandwidth; their efficiency increases with the length of the `stanzas` of contiguous memory accesses. Finally, disabling ECC leads to an average of 20% increase in bandwidth.

*Ring Interconnect:* All cores can be seen as *symmetrical* peers, and the distance between cores has little impact on performance. However, memory requests from threads running on the same core are serialized, provided that the bandwidth reaches 4.7 GB/s. Furthermore, when threads (on different cores) are accessing the same data, the simultaneous access to the DTD leads to bandwidth loss.

Overall, we believe our results are complementary to the SDG, and, being backed up by more practical guidelines, be of added value for programmers using this platform.

**SCAT Model:** Based on the numbers and the observations, we attempt to build a simple view of the Xeon Phi, providing production users with a platform model for reasoning about parallel algorithm design and performance optimization. Figure 12 shows the machine model for Xeon Phi. The machine has 60 symmetrical cores, each of which contains 1/2(/3/4) vector threads working on 8 double-precision or 16 single-precision data elements in a lock-step manner. `Family threads` (threads suited in the same core) differ from `remote threads` (threads suited in another core) in that they share and compete local resources. Furthermore, compared with accessing local caches, remote caches and off-chip memory are slow (see the numbers in Table 2). We

summarize the model as *SCAT* (symmetric cores and asymmetric threads).

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 symmetrically. On the other hand, the threads on the same core share and compete the shared resources, putting up `asymmetry` and impelling us to take care of `thread affinity`. Therefore, this platform model captures the key performance features of the processor, ensuring good performance with relatively low programming effort (i.e., using high-level programming tools).

# 5. LEUKOCYTE TRACKING

In this section, we focus on our case-study application, Leukocyte Tracking. Specifically, we aim to evaluate the gap(s) between the achieved performance of the application and the performance indicated by the microbenchmarks.

Leukocyte Tracking is a medical imaging application which detects and tracks rolling leukocytes (white blood cells) in vivo video microscopy of blood vessels. The velocity of rolling leukocytes provides important information about the inflammation process, which aids biomedical researchers in the development of anti-inflammatory medications [21].

In the application, cells are detected in the first video frame and then tracked through subsequent frames [21]. Tracking accounts for around 90% of the total execution time and thus we focus on this procedure. Tracking is accomplished by first computing, in the area surrounding each cell, a Motion Gradient Vector Flow (MGVF) matrix. The MGVF is a gradient field biased in the direction of blood flow, and it is computed using an iterative Jacobian solution procedure. After computing the MGVF, an active contour is used once again to refine the shape and determine the new location of each cell. Unfortunately, leukocyte tracking is computationally expensive, requiring more than four and a half hours to process one minute of video. Boyer et al. have translated the tracking algorithm from Matlab to C and OpenMP [3].

## 5.1 Performance Analysis

Without any code changes, we compile and run the kernel on both Phi and SNB (Intel Xeon E5-2620, a dual 6-core processor with hyper-threading disabled), and show their performance in Figure 13. We see that, on SNB, the execution time decreases when increasing the number of threads. On Phi, the execution time decreases when the number of threads is less than 40. Using more than 40 threads brings no further performance gain. Overall, we note that the performance on Phi (with 40 threads) is 2× worse than that on SNB (with 12 threads), while the sequential execution of the same application (i.e., running on a single thread) on Xeon Phi is 5× slower than on SNB.

To further understand these results, we analyze the overall performance by taking both parallelism and per-thread performance into account, and focus on two aspects: (1) the single-thread performance and (2) scalability. The analysis includes the interactions between kernel characteristics and processor features.

**Single thread:** When tracking a leukocyte, we use 18 data structures/matrix (1 input sub-image, 1 motion gradient vector field, 8 neighbours to store intensity differences, and 8 neighbours to store the heaviside value). For the given input dataset, each matrix has 41 × 81 elements (in double-precision). In total, tracking a leukocyte needs 467 KB (18 × 41 × 81 × 8), which is smaller than the size of a local L2 cache (Figure 12 and Table 2). Thus, the iterative Jacobian solver will work intensively on tracking a leukocyte with data located on-chip, and the tracking speed is not limited by the memory access.

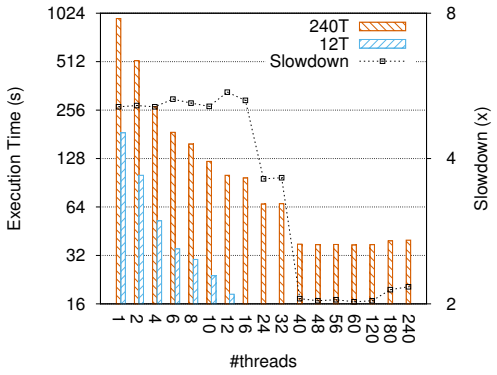



**Figure 13: The initial performance results of Leukocyte Tracking on an Xeon Phi processor (240 threads) and an Xeon processor (12 threads).**

As for computation without vectorization, a thread on Phi (working at 1.05 GHz and issuing an instruction every two cycles, see Section 3.1) runs 4× slower than one on SNB (working at 2.0 GHz and issuing instructions every cycle). With vectorization, the difference is lowered to roughly 2×. In our practical experience, the single thread performance on Phi is around 5× worse than that on SNB, an indication that vectorization is not applied on both platforms. Indeed, the compiler reports an auto-vectorization failure (consequently, only 12.5% of the SIMD lanes are used).

**Scalability:** Figure 13 shows that the performance on Phi varies little when using over 40 threads. Through code analysis, we observed that parallelization is performed over the number of leukocytes. As the number of leukocytes from the input datasets is 36, increasing the number of threads to more than 36 brings no performance gain. In other words, the kernel parallelism does not match Phi's massive hardware parallelism (36 << 240, see Figure 12). On the other hand, SNB has only has 12 threads, showing much better scalability. To fully utilize the hardware resources on Phi, we *must* increase the paralellism of the application.

## 5.2 Performance Optimization

### 5.2.1 *Vectorizing the Kernel*

When tracking a leukocyte, the kernel loops over a fixed-sized portion of a frame (a sub-image with 41×81 pixels). A typical loop is shown in Figure 14 ($m = 41$, $n = 81$). As we have mentioned, the compiler fails to vectorize this code due to the assumption of data dependency (the original code uses `pointers to pointers` and dereferencing the data structure is too complex for the compiler to automate).

We note that enabling vectorization for these cases requires an intervention from the programmer. The typical approach for manual vectorization is to add low-level intrinsics in the high-level C code, thus specifically instructing the compiler to use the vector units.

```
input: MAT* z, double v, double e
output: MAT* H
double one_over_pi = 1.0 / PI;
double one_over_e = 1.0 / e;
for (i = 0; i < m; i++) {
    for (j = 0; j < n; j++) {
    double z_val=m_get_val(z, i, j)*v;
    double H_val=one_over_pi * \
           atan(z_val*one_over_e)+0.5;
    m_set_val(H, i, j, H_val);
    }
}
```

**Figure 14: The Heaviside step function.**

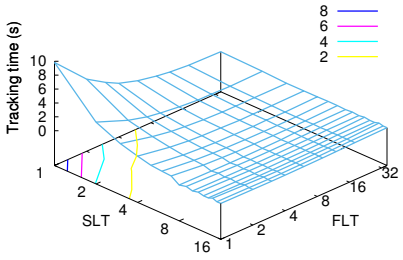



**Figure 15: The selection of $FLT$ and $SLT$.**

We identify three main factors that make code vectorization for leukocyte tracking cumbersome. First, `data alignment`: when vectorizing the code, data accesses should start with an address aligned at 64 bytes (512 bits). This must be insured with specific memory allocation (i.e., dedicated APIs). Second, the `non-unit-strided memory access`: when the 8 elements in a vector are non-contiguous, the offset for each element must be specified. This occurs when calculating the gradient in the tracking kernel. Third, and final, vectorizing a loop requires special care when the number of iterations is not a multiple of the vector length. Thus, we also need to deal with the `remainder` of the inner-loop (i.e., because $n\%8 \neq 0$). Therefore, we use two loops in the tracking kernel: a vector loop (for the bulk of the computation), and a scalar loop (used to deal with the loop remainder).

Fixing all these problems (and thus manually vectorizing this code using intrinsics) takes an expert programmer two days. Moreover, the kernel code doubles in size (from ∼200 lines to ∼400 lines). Correspondingly, the tracking time per frame decreases to 8.5s from 31s (∼4× faster) on Phi. The remaining optimization space is roughly 2×. The limiting factor is that the kernel uses trigonometric operations, which can be further optimized by using EMU (Section 3.1 and [6]).

### 5.2.2 Changing Parallelism

As we have mentioned, the parallelism of leukocyte tracking is limited by the number of leukocytes (36 in the given data set). For the traditional multi-core processors, this number is still larger than that of the hardware threads. But on a Phi with 240 hardware threads, running the tracking kernel with 36 parallel threads can never fully utilize the platform.

We attempt to improve on this situation by increasing parallelism. Thus, we spawn a second-level parallelism over the outer loop of the sub-image in Figure 14. Next, we need to tune the dimensions of the two parallel levels by specifying the number of first-level threads ($FLT$) and the number of second-level threads ($SLT$). We select these two numbers from those that satisfy the following constraints: (1) $FLT \times SLT \leq 240$, (2) $FLT \leq 36$, (3) $SLT \leq 41$. We autotune the kernel using $FLS \in \{1, 2, 3, 4, 6, 9, 12, 18, 36\}$ and $SLT \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16\}$ [9]. Figure 15 shows the tracking time per frame for different combinations. We see that the best performance achieved by Phi is around 0.1s per frame when $FLT = 4$ and $SLT = 8$, indicating that using more threads does not mean a faster tracking ($4 \times 8 < 240$). According to the SCAT model (Figure 12), it is of no use binding multiple threads to the same core due to little data reuse.

### 5.2.3 Overall Performance

We compare the execution time of tracking leukocytes per frame on Phi against the ones achieved by SNB (using a higher clock and better performing cores, but a lot less parallelism), and on an NVIDIA Kepler GPU (K20m, a GPU with a similar peak performance and more massive parallelism, programmed in CUDA implementation [10]). The comparison is illustrated in Table 3. We notice that Xeon Phi is 6× faster than SNB, while it is around 40% slower than K20. Admittedly, optimizing the tracking kernel on SNB (by hand-tuning for enabling vectorization) can lead to a performance increase (a maximum 4×, most likely, with SNB-specific intrinsics). K20 performs better than Phi due to the more efficient reduction implemented in the GPU shared memory [3]. Specifically, at the second level of parallelism, we use multiple threads that are bound to separate cores on Phi, while the CUDA implementation runs the same amount of work on a block (and a multi-processor). Thus, when performing reduction, the shared (reduction-)variable on Phi has to be transferred back and forth at the second-level cache. As we have measured, the remote cache access is as slow as accessing the off-chip memory (Figure 12 and Table 2). With CUDA on K20, this reduction happens in shared memory, with much higher performance. The final code of leukocyte tracking for Phi is publicly available [11].

**Table 3: Tracking time per frame (in seconds). 'VEC' represents 'vectorization'; 'FMT/SMT' is 'to use the first-level/second-level multi-threading', respectively. The optimizations are incrementally added.**

| | 1T | +VEC | +FMT | +SMT | Overall |
|---|---|---|---|---|---|
| Phi | 31 | 8.5 | 0.7 | 0.1 | 0.1 |
| SNB | 6 | – | 0.6 | – | 0.6 |
| K20 | – | – | – | – | 0.06 |

[9] $SLT$ can be as large as 41, but our results show a large $SLT$ is not necessary due to the limited per-thread work.

[10] We change the original Rodinia single-precision version to the double-precision version for a fair comparison.

[11] `https://github.com/haibo031031/mic-apps`

## 5.3 Discussion

One of the important selling points of Phi is the continuity of programming models from the traditional multi-core processors - the OpenMP and MPI models and codes are functionally compatible. Ideally, programmers should obtain high performance without a lot of investment in programming model learning (e.g., OpenCL), tuning and hacking low-level code (e.g., assembly code with pthreads). Effectively, the expectation is that re-compiling the code with the `-mmic` option will do. Our experience leads to a different conclusion: porting legacy code or developing new code still needs a lot of developer interventions.

*Note 1.* Using intrinsics indeed brings us a significant performance gain, but it exposes low-level implementation details to users, conflicting with the principles of encapsulation and high-level programming. It also requires code specialized for Phi, which will further fail to run the on traditional multi-core processors. This deviates from the original design goal of Phi, i.e., to keep using traditional programming models. A possible solution is to provide a high-level vector template/library/model (e.g., ispc [12]). The template can present users with the required operations (e.g., multiply and reduction). When implementing the template, we translate the operations into their equivalent intrinsics specialized to a platform. Thus, we can keep code portable and not hinder performance.

*Note 2.* Xeon Phi truly needs massive parallelism to fully use the hardware threads. This observation makes a significant difference between SNB and Phi. SNB has a dozen of hardware threads, while Phi has over a two hundred. Only those applications with abundant parallelism can fully utilize the machine. When lacking parallelism, applications can either look for finer grain parallelism (atypical for OpenMP, but useful when available), or find a way to load multiple (independent) tasks on the platform. However, note that the number of required threads depends on applications and their run-time contexts.

*Note 3.* On Xeon Phi, using OpenMP can perform global reduction on the globally shared caches, but this proves to be less efficient than expected (apparently due to frequent memory transfers). When using CUDA/OpenCL on GPUs, an efficient reduction can be performed in shared memory (or local memory in OpenCL) at the block (or work-group in OpenCL) level. Further, our experience shows that the leukocyte tracking maps more naturally to the GPU architecture: mapping a leukocyte to a multi-processor. While on Phi, we need to map a leukocyte to multiple processing cores. Thus, we believe that a multiprocessor on GPUs is equivalent to multiple processing cores on Phi, at least in the context of leukocyte tracking.

To summarize, we conclude that (1) although it often destroys portability, manual vectorization is mandatory for exploiting Phi's performance; a high-level library can be used to hide the platform-dependent details, but vectorization *must* be enabled as much as possible, and (2) massive parallelism is needed on Phi to fully use the hardware. In a nutshell, merely relying on compilers with traditional programming models to achieve high performance on Phi is still far from reality.

[12] `http://ispc.github.io/`

# 6. RELATED WORK

In this section, we survey and briefly discuss the work related to our (micro)benchmarking approach. We focus mainly on existing CPU and GPU benchmarking methods, as there are no other comprehensive studies of Xeon Phi - yet.

In [23], the authors develop a high-level program to evaluate the cache and TLB for any machine. Part of our work is based on their approaches (targeting uni-core processors, though). Multiple studies are also performed on multi-core CPUs. In [20], the authors report performance numbers from three multi-core processors , including not only execution time and throughput, but also a detailed analysis on the memory hierarchy performance and on the performance scalability between single and dual cores. Daniel Molka et al. [19] revealed many fundamental details of the Intel Nehalem using benchmarks for latency and bandwidth between different locations in the memory subsystem. We use similar approaches for the access latency of remote caches.

For GPUs, Volkov et al. [28] presented detailed benchmarking of the GPU memory system that reveals sizes and latencies of caches and TLB. Later, Wong et al. [29] presented 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. [26] develop a set of OpenCL benchmarks targeting a large variety of platforms. They include code designed to determine parameters unique to OpenCL, like the dynamic branching penalties prevalent on GPUs. They also demonstrate how their results can be used to guide algorithm design and optimization

Garea et al. [8] 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.

# 7. CONCLUSION AND FUTURE WORK

Given its performance promises, Intel Xeon Phi is very likely to become popular for both low-end high performance computing applications (smaller scale scientific applications like Leukocyte Tracking), and the next generation of supercomputers. In this paper, we presented our hands-on experience with this processor - in both the "lab" and using a real application - and discussed several key insights into the performance of this new many-core processor. By using a set of self-designed microbenchmarks, we characterized the major components of this architecture - cores, memory, and interconnections - summarizing them into four machine-centric observations (potential optimization guidelines). We also made a first attempt to provide a simple machine view (*SCAT*) to facilitate application design and performance tuning on the Xeon Phi.

In general, our benchmarking results are consistent with Xeon Phi's published data. However, the data we have added through this benchmarking effort allowed us to expose more accurately the expected key performance factors

for the Xeon Phi. We have shown that the platform is able to deliver its performance promises in terms of computation, but programmers will need to find the right parallelization strategy to fill 240 hardware threads with compute-intensive tasks, while finding the right balance between data partitioning and coherent memory requests to achieve sufficient memory bandwidth. Thus, we believe the number of applications that can easily use Xeon Phi's potential in their existing, naive form is, for now, very limited. And for high performance, our and other experience show that programmers need to take a lot of efforts on parallelization, analysis, and optimization.

In terms of future work, we are extending our hands-on experience with more application studies. As a long term plan, we are targeting a quantified performance model for Xeon Phi, which could be used in identifying performance bottlenecks and guiding performance optimization. This model would build upon the microbenchmarks and application characteristics for its foundations, but expose different complexity layers depending on the user requirements.

# 8. ACKNOWLEDGMENTS

The authors would like to thank Sabela Ramos Garea from University of A Coruña and Evghenii Gaburov from SURFsara for the numerous on-line discussions. This work is partially funded by CSC (China Scholarship Council), and the National Natural Science Foundation of China under Grant No.61103014 and No.11272352.

# 9. REFERENCES

[1] R. D. e. a. Blumofe. Cilk: an efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8):207–216, Aug. 1995.

[2] O. A. R. Board. OpenMP application program interface (version 4.0). Technical report, July 2013.

[3] M. Boyer et al. Accelerating leukocyte tracking using CUDA: A case study in leveraging manycore coprocessors. In *IPDPS'09*, May 2009.

[4] K. Datta et al. Optimization and performance modeling of stencil computations on modern microprocessors. *SIAM Rev.*, 51(1):129–159, Feb. 2009.

[5] David. *Programming with POSIX Threads*. May 1997.

[6] J. Fang et al. Benchmarking intel xeon phi to guide kernel design. Technical Report PDS-2013-005, Delft University of Technology, Apr. 2013.

[7] A. Fog. Lists of instruction latencies, throughputs and micro-operation reakdowns. Technical report, Copenhagen University, Feb. 2012.

[8] S. R. Garea and T. Hoefler. Modeling Communication in Cache-Coherent SMP Systems - A Case-Study with Xeon Phi. 2013. HPDC'13.

[9] T. Granlund. Instruction latencies and throughput for AMD and intel x86 processors. Technical report, KTH, Feb. 2012.

[10] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann, 5 edition, Sept. 2011.

[11] Intel. *Intel Xeon Phi Coprocessor InstructionSet Architecture Reference Manual*, Sept. 2012.

[12] Intel. *Intel Xeon Phi Coprocessor System Software Development Guide*, Nov. 2012.

[13] Intel. *An Overview of Programming for Intel Xeon Processors and Intel Xeon Phi Coprocessors*, Oct. 2012.

[14] Intel. *Streaming Store Instructions in the Intel Xeon Phi coprocessor*, 2012.

[15] Intel. Intel Xeon Phi Coprocessor. `http://software.intel.com/mic-developer`, April 2013.

[16] John D. McCalpin. STREAM: Sustainable Memory Bandwidth With High Performance Computers, April 2013.

[17] V. W. Lee et al. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. *SIGARCH Comput. Archit. News*, 38(3), June 2010.

[18] L. McVoy et al. lmbench: portable tools for performance analysis. In *USENIX ATEC'96*, 1996.

[19] D. Molka et al. Memory performance and cache coherency effects on an intel nehalem multiprocessor system. In *PACT'09.*, Sept. 2009.

[20] L. Peng et al. Memory hierarchy performance measurement of commercial dual-core desktop processors. *Journal of Systems Architecture*, 54(8):816–828, Aug. 2008.

[21] N. Ray et al. Motion gradient vector flow: an external force for tracking rolling leukocytes with shape and size constrained active contours. *Medical Imaging, IEEE Transactions on*, Dec. 2004.

[22] A. Sclocco et al. Radio astronomy beam forming on Many-Core architectures. In *IPDPS*, 2012.

[23] A. J. Smith et al. Measuring cache and TLB performance and their effect on benchmark runtimes. *IEEE Trans. Comput.*, (10), Oct. 1995.

[24] J. E. Stone et al. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66–72, May 2010.

[25] Technical University of Dresden. BenchIT: Performance Measurement for Scientific Applications, August 2013.

[26] P. Thoman et al. Automatic OpenCL device characterization: Guiding optimized kernel design. In *Euro-Par'11*. 2011.

[27] A. L. Varbanescu, A. S. van Amesfoort, T. Cornwell, G. van Diepen, R. van Nieuwpoort, B. G. Elmegreen, and H. J. Sips. Building high-resolution sky images using the Cell/B.e. *Scientific Programming*, 17(1-2):113–134, 2009.

[28] V. Volkov and J. W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–11. IEEE, Nov. 2008.

[29] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying GPU microarchitecture through microbenchmarking. In *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, pages 235–246. IEEE, Mar. 2010.