Delft University of Technology
Parallel and Distributed Systems Report Series

Identifying the Key Features of Intel Xeon Phi: A Comparative Approach

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

Completed in May 2013

Report number PDS-2013-006

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/`

© 2013 Parallel and Distributed Systems Group, Department of Software and Computer Technology, Faculty of Electrical Engineering, Mathematics, and Computer Science, Delft University of Technology. All rights reserved. No part of this series may be reproduced in any form or by any means without prior written permission of the publisher.

J. Fang, A.L. Varbanescu, H.J.Sips

Identifying the Key Features of Intel Xeon Phi

With the increasing diversity of many-core processors, it becomes more and more difficult to guarantee *performance portability* with a unified programming model. The main reason lies in the architecture disparities, e.g., CPUs and GPUs have different architectural features from each other, which leads to the differences in performance optimization techniques. Thus, it is of great necessity to abstract performance-wise key features from many-core processors.

In this paper, taking the Intel's Xeon Phi as a case study, we present a two-stage comparative approach to abstract key features. Our approach needs a reference processor and is executed at both the application level and the microbenchmark level. We select multiple benchmarks from the Parboil benchmarks and measure the performance differences to identify performance factors. Further, we perform an in-depth analysis to identify the key features with microbenchmarks. Finally, we briefly discuss a use case in our optimizing framework.

Keywords: Performance Portability, Architectural Features, Many-Core Processors.

Contents

1 Introduction **4**

2 Background **5**
- 2.1 Selected Architectures . . . 5
  - 2.1.1 Intel Many-Integrated Cores: Xeon Phi 5110P . . . 5
  - 2.1.2 Intel Sandy Bridge: Xeon E5-2620 . . . 5
- 2.2 Programming Models on ISB and MIC . . . 6

3 A Two-stage Comparative Approach **6**

4 Performance Evaluation and Analysis **7**
- 4.1 SGEMM . . . 7
  - 4.1.1 Description . . . 7
  - 4.1.2 High-level Analysis . . . 8
  - 4.1.3 Low-level Analysis . . . 8
- 4.2 Histogram . . . 9
  - 4.2.1 Description . . . 9
  - 4.2.2 High-level Analysis . . . 9
  - 4.2.3 Low-level Analysis . . . 10
- 4.3 Cutoff-limited Coulombic Potential (CUTCP) . . . 10
  - 4.3.1 Description . . . 10
  - 4.3.2 High-level Analysis . . . 10
  - 4.3.3 Low-level Analysis . . . 11
- 4.4 Stencil . . . 12
  - 4.4.1 Description . . . 12
  - 4.4.2 High-level Analysis . . . 12
  - 4.4.3 Low-level Analysis . . . 12
- 4.5 Sparse Matrix-Dense Vector Multiplication (SpMV) . . . 13
  - 4.5.1 Description . . . 13
  - 4.5.2 High-level Analysis . . . 13
  - 4.5.3 Low-level Analysis . . . 13

5 Key Features Summary **14**
- 5.1 Cores . . . 14
- 5.2 On-chip Memory . . . 14
- 5.3 Off-chip Memory . . . 15
- 5.4 Miscellaneous . . . 16

6 Discussion and Conclusion **16**

List of Figures

1 The Intel Xeon Phi Architecture. 5
2 The Intel Xeon E5-2620. 6
3 The comparative approach. 7
4 SGEMM 8
5 Memory bandwidth with Stanza Triad. 9
6 The native histogram performance (in seconds). 9
7 Read latencies of core 0 (in seconds) when the cache-line is located at different positions. 10
8 Arithmetic throughput using different threads (60, 120, 240) and issue widths (1 single instruction versus 2 independent instructions) (in GFlops). 11
9 Memory bandwidth of stencil solvers (in GB/s) 12
10 Memory bandwidth of SpMV (in GB/s) 13
11 Memory bandwidth of read and write operations (in GB/s) 14

List of Tables

1 Performance comparison of the Parboil benchmarks before and after code changes. 8
2 Feature List of the Xeon Phi 5110P. 15

1 Introduction

In recent years, more and more many-core processors are superseding sequential ones. Increasing parallelism, rather than increasing clock rate, has become the primary engine of processor performance growth, and this trend is likely to continue [1]. Particularly, today's GPUs (Graphics Processing Units) and Intel's MIC (Many Integrated Cores) [2], greatly outperforming traditional CPUs in arithmetic throughput and memory bandwidth, can use hundreds of parallel processing cores to run tens of thousands of threads.

Programming many-core processors is difficult, as it is a problem with multiple constraints: we want applications to deliver great performance, to be easy to program, and to be portable between architectures [3]. Researchers have been putting many efforts on this problem in both industry and academia. OpenCL [4], managed by the Khronos Group, can give software developers portable and efficient access to diverse processing platforms, while OpenMP-like programming models such as OpenACC [5] and OmpSs [6] have been proposed/extended to support incremental parallelization.

However, the optimizations we apply in one context are not portable in another, i.e., we see poor performance-portability across platforms, data sets, and calling-contexts [7], [8]. To this end, we propose a path towards efficient support for portable performance. Specifically, we propose *Sesame: A User-Transparent Optimizing Framework for Many-Core Processors* [9]. Taking naively parallelized code as input, Sesame performs code transformations and generates specialized kernels for different platforms. The Sesame framework consists of four components: (1) feature identifier, (2) impact predictor, (3) source-to-source translator, and (4) auto-tuner. The *feature identifier* aims to find the architectural features that are significant (in a good or a bad way) to application performance from the state-of-the-art many-core processors. Ultimately, we define these features as *key features*. Thus, the Sesame framework can deal with the architectural disparities and optimizations in a unified way.

In this paper, we focus on the most recent (to date) many-core processors - Intel's Xeon Phi - and we discuss its key features. Specifically, **we focus on three questions: (1) how can we identify key features? (2) what are the key features for Xeon Phi? (3) how can these features be used?** In [10], we microbenchmark an Xeon Phi coprocessor thoroughly. We use these results to generalize a two-stage comparative approach to identify key features. Specifically, we take as reference the native performance of the Parboil benchmarks [11], and, after applying various code and/or data layout changes, we study their performance impact. These performance changes are further analyzed and correlated to architectural features. During the second stage, we give an in-depth analysis of the features using microbenchmarks. We note that the changes that we have applied to isolate the key features can be easily transformed into best-practice and/or optimization techniques. Further, we illustrate how to use the key features in the Sesame framework (Section 6).

On the Xeon Phi, we have identified the following key features: KF1- the number of cores, KF2- SIMD width, KF3- local-to-remote memory latency gap, KF4- ECC support, and KF5- prefetching. Their corresponding optimization techniques are tabulated in Table 2. We believe these features are a subset of the key features on Xeon Phi, and more work is needed to compile a complete list.

Through a comparative study between the Intel Xeon Phi coprocessor and the Intel Xeon processor, we found that Intel Xeon Phi, indeed, inherits a lot of architectural features from traditional multi-core CPUs like Intel Xeon processors. Both of them have SIMD cores and cache-based memory systems. The difference lies in the last-level caches that Xeon Phi uses the 2nd level as the last-level and each cache slice is private to a core. Another difference is that the Xeon Phi adopts graphics memory which can lead to a much higher (6×) bandwidth than that on the Sandy Bridge Xeon processor.

2 Background

2.1 Selected Architectures

We will investigate the key features of Intel Xeon Phi. At the same time, we select the Sandy Bridge as the reference architecture. For this, we select Intel Xeon Phi 5110P (MIC) and Intel Xeon E6-2620 (ISB) in this paper.

2.1.1 Intel Many-Integrated Cores: Xeon Phi 5110P

The Intel Xeon Phi comprises of over 50 cores (the one used in this paper belongs to the 5110P 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 channels (supported by memory controllers) delivering up to 5.0 GT/s (320 GB/s memory bandwidth) [12]. 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.

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 30MB 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.

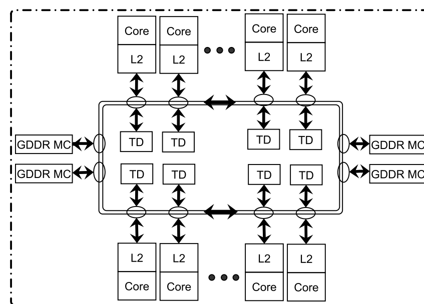


Figure 1: The Intel Xeon Phi Architecture.

2.1.2 Intel Sandy Bridge: Xeon E5-2620

For comparison, we analyze an Intel Sandy Bridge Xeon E5-2620, an six-core processor working at 2.0 GHz (Figure 2). It has AVX support (advanced vector extensions), with the width of the SIMD registers increasing from 128 bits to 256 bits. Thus, it can process 4 double-precision or 8 single-precision data elements at a time. Each core has private L1 (32 KB data and 32 KB instructions) and L2 caches (256 KB unified), and all cores shared a L3 cache (or LLC) of 15 MB. The internal components of the chip, including the LLC slices, are connected via a ring bus composed by a data ring, a request ring, an acknowledge ring and a snoop ring [13]. Any core can use any of the cache slices, thus having access to data stored in any of them. From now on, we use 'MIC' to represent Intel Xeon Phi 5110P and 'ISB' to represent Intel Xeon E5-2620 for brevity.

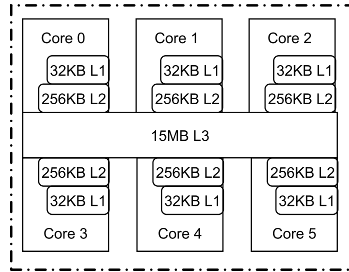


Figure 2: The Intel Xeon E5-2620.

2.2 Programming Models on ISB and MIC

Being an `x86` SMP-on-a-chip running Linux [14], MIC 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 (`V2013.3.163`). 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 [14]. In this work, we measure the performance of all programs with Xeon Phi working in the native execution mode.

3 A Two-stage Comparative Approach

In this section, we illustrate a two-stage comparative approach to abstract key features from Intel Xeon Phi (Figure 3). First of all, we select a *reference architecture*. When running the same kernel on both Intel Xeon Phi and the reference processor, we may observe different performance (or trends). When relating the performance differences with the underlying architectural features, it becomes easier to isolate and/or identify the architectural differences and obtain the key features of Intel Xeon Phi. As we know, Intel Xeon Phi inherits more from traditional multicore CPUs than GPUs. Thus, we take a Sandy Bridge Xeon E5-2620 as a reference.

Overall, our comparative approach is executed at two levels (or in two stages): (1) the application level, and (2) the microbenchmark level. At the application level, we take a couple of benchmarks from the Parboil benchmark suite as input. Based on the *naive* kernel in the Parboil suite, we perform code changes or data layout changes to get a counter-part (a *new* kernel). The changes can be constructive, which leads to a better performance; they can also be destructive, which makes the kernel perform worse. Thereafter, we measure the performance of the two kernels on both Intel Xeon Phi and the reference processor. The performance changes or differences (from using different processors, or different code versions, or different data sets) are analyzed, thus leading to the discovery of performance factors.

A performance factor gives programmers/users intuitive understanding of the performance differences. For example, we know that accessing a cache-line in remote caches is far more expensive than accessing a line in local caches. However, what is the 'exact' performance difference? For this reason, we give an in-depth analysis based on the performance factors. We quantify the 'exact' difference using microbenchmarks, e.g., we use a microbenchmark to measure the latency difference between local accesses and remote access. We found that the access latency is of an order of magnitude differences on Intel Xeon Phi, while the differences depends on the

cache-line state on the reference processor. Hence, we conclude Intel Xeon Phi has a different cache memory from the reference processor.

To summarize, our approach includes two key factors: selecting a reference processor and a two-stage execution. In the next section, we will validate our approach in a step-by-step manner.

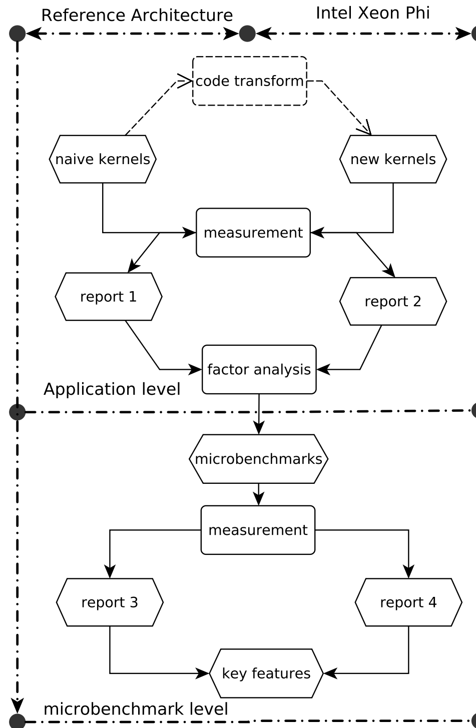


Figure 3: The comparative approach.

4 Performance Evaluation and Analysis

We select five benchmarks from the Parboil suite: SGEMM, Histogram, CUTCP, Stencil and SpMV. We measure their performance on the Xeon E5-2620 (‘ISB’) and the Xeon Phi 5110P (‘MIC’). The overall results before and after code changes are shown in Table 1.

4.1 SGEMM

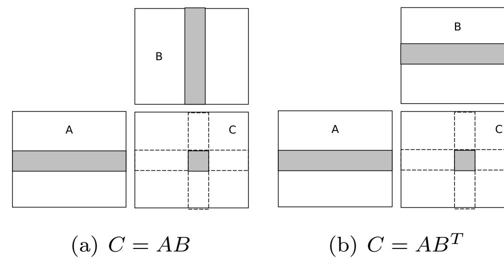
4.1.1 Description

As a building block for many other routines, the SGEMM routine calculates the new value of matrix C based on the matrix-product of matrix A and matrix B, and the old value of matrix C: $C \leftarrow \alpha AB + \beta C$, where α and β values are scalar coefficients. The Parboil benchmark implements the following format: $C \leftarrow \alpha AB^T + \beta C$ (shown in Figure 4). We suppose A, B, C are $M \times K$, $K \times N$, and $M \times N$ in size. Totally, SGEMM performs $2N^3$ operations (suppose $M = N = K$). Without taking caches into account, it accesses $2N^3$ double-precision

Table 1: Performance comparison of the Parboil benchmarks before and after code changes.

		Xeon E5-2620 ('ISB')			Xeon Phi 5110P ('MIC')		
	dataset	t1(s)	t2(s)	sp(x)	t1(s)	t2(s)	sp(x)
SGEMM	medium	1.30	0.04	33	7.73	0.26	30
Histogram	2^{24}	85.77	0.16	537	407.00	0.20	1992
CUTCP	large	15.86	2.19	7	14.41	3.28	4
Stencil	$60 \times 60 \times 15$	26.32	19.75	1.3	21.94	12.46	1.7
SpMV	–	–	–	–	–	–	–

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.



(a) $C = AB$ (b) $C = AB^T$

Figure 4: SGEMM

4.1.2 High-level Analysis

We select the medium data set (M=1024, N=1056, K=992) and use 60 threads. The parboil assume that the matrix (A, B and C) is stored in *column-major* order. In fact, it is stored in *row-major* order. We hold that this code was originally developed in programming languages like Fortran. We change the input data layout and exchange the two dimensions of each matrix. Using the same data set, the kernel runs around 30× faster (Table 1). We conclude that the Xeon Phi prefers accessing data elements contiguously. Similar trends can be found on the Xeon processor (Table 1).

4.1.3 Low-level Analysis

To analyze differences in the presence of 'jump' memory access, we use the `Stanza Triad (STriad)` benchmark to measure the bandwidth 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 time, and we averaged the performance to calculate the memory bandwidth for each stanza length.

Figure 5 shows the results of the STriad experiments on the Xeon Phi and the Xeon processor. Overall, we see an increase in memory bandwidth over the stanza length (until a big enough stanza length). Our experimental results show that the length is around 2048 words for the ISB and 8192 words for MIC. Hence, the non-contiguous access to memory is detrimental to memory bandwidth efficiency and thus the performance of memory-bound kernels. In fact, the frequent 'jump' memory access will disturb the data prefetching from

the off-chip memory to on-chip memory. For these reasons, programmers should try to create a long enough stanzas of contiguous memory accesses for better prefetching effects and larger memory bandwidths.

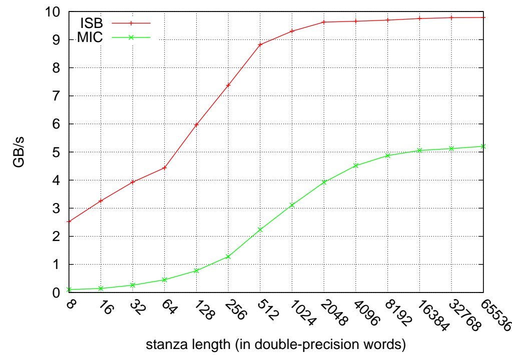


Figure 5: Memory bandwidth with Stanza Triad.

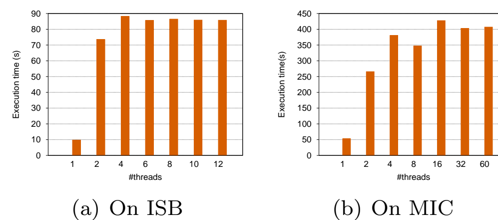
4.2 Histogram

4.2.1 Description

The Histogram benchmark accumulates the number of occurrences of each output value in the input data set. To make the benchmark more adjustable, we take the number of `bins` as a parameter and generate the input image randomly.

4.2.2 High-level Analysis

The naive implementation adds `parallel for` pragma over the outer loop and `critical` pragma over the accumulating statement. We measure the execution time of the naive implementation using different number of threads and show the results in Figure 6. We note that using multiple threads leads to significant performance degradation (up to 9× slower). This is because the all threads updates the same memory space exclusively and thus results in repeated cache-lines invalidation or transfers between the processing cores. Thus, we allocate a local space for each thread and let them accumulate results 'independently'. Thereafter, we summarize the sub-results into a total bin. Table 1 show that we can dramatically decrease the execution time (537× faster on ISB and 1992× faster on MIC).



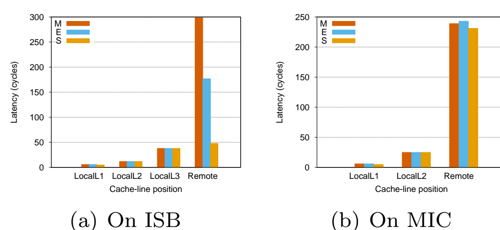
(a) On ISB (b) On MIC

Figure 6: The native histogram performance (in seconds).

4.2.3 Low-level Analysis

We further show the cost of cache-line transfers between processing cores in Figure 7. Compared with accessing the local caches, we name the operation of transferring a cache-line from another core as *remote cache access*. we use the approach (proposed by Daniel Molka [15]) to measure the cache access latency. Prior to the measurement, the to-be-transferred cache-line 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 7 shows the latency results of cache accesses. We see the remote cache access is 10× slower than local cache accesses, on both ISB and MIC. Further, the remote read latency does not depends on the cache-line state on MIC, while it differs on the Xeon processor. Specifically, when the cache-line is in `shared` state, the remote read latency is much smaller and close to the latency of local L3 access (Figure 7(a)). The difference comes from the fact that all cores on ISB share the same L3 cache. On the Xeon Phi, however, the last-level cache is private to a core and designed in a distributed manner. Hence, the target cache-line has to be moved to the local cache (via the ring interconnect) even when it is in `shared` state.



(a) On ISB (b) On MIC

Figure 7: Read latencies of core 0 (in seconds) when the cache-line is located at different positions.

4.3 Cutoff-limited Coulombic Potential (CUTCP)

4.3.1 Description

Some molecular modeling tasks requires a high-resolution map of the electrostatic potential field produced by charged atoms distributed through a volume. In a complete application, this would be added to a long-range component computed with a less computationally demanding algorithm. Cutoff-limited Coulombic Potential (CUTCP) computes a short-range component of this map, in which the potential at a given map point comes only from atoms with in a cutoff radius of $12\dot{A}$. In a simple, sequential implementation, each atom is visited in sequence, and electropotential contributions made by a visited atom are accumulated into all output cells within the cutoff distance before proceeding to the next atom.

4.3.2 High-level Analysis

The Parboil benchmark suite provides us with two OpenMP implementations: (a) naive implementation (Listing 1), and (b) the code that is optimized for vectorization (Listing 2). We measure the performance on both the processors shown in Table 1. We note that the optimized code can run 7× faster on the ISB, but its performance suffered a loss on MIC using the code in Listing 2, compared with the naive implementation. We assume the code vectorization on the Xeon Phi was not successful although the emitted message from the compiler reported “LOOP WAS VECTORIZED”. Thus, we use `intrinsics` to translate each statement into an `assembly` format. Then the further optimized code can run 4× faster shown in Table 1.

Listing 1: naive code

```
for (i=ia; i<=ib; i++,pg++,dx+=gridx)
{
  r2 = dx*dx + dydz2;
  if (r2 >= a2) continue;
  s = (1.f − r2 * inv_a2);
  e = q * (1/sqrtf(r2)) * s * s;
  #pragma omp atomic
    *pg += e;
}
```

Listing 2: optimized code

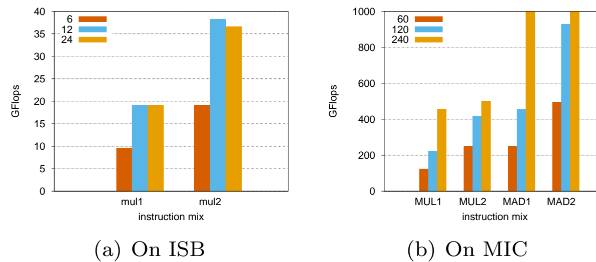
```
for(i=ia; i<=ib; i++,pg++,dx+=gridx)
{
  r2 = dx*dx + dydz2;

  s = (1.f−r2*inv_a2)*(1.f−r2*inv_a2);
  e = q * (1/sqrtf(r2)) * s;

  *pg += (r2 < a2 ? e : 0);
}
```

4.3.3 Low-level Analysis

The optimized CUTCP application is compute-bound, and the *SIMD* usage becomes a key factor to achieve high performance. Here we explore the factors of achieving peak flops apart from the SIMD usage. The Xeon Phi 5110P has 60 cores working at 1.05 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 1008 GFlops. Similarly, we can calculate the peak flops on ISB is 96 GFlops in double-precision.

To measure the achievable instruction throughput, we run 1, 2, 4 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 8. Overall, we can achieve 99% of the peak instruction throughput (using 60 cores and 240 threads) on MIC. Using `mul` instruction can achieve around 40% of the peak flops on ISB [1].

Furthermore, we have the following observations. First, when using one thread per core, the instruction throughput is rather low, compared with the cases when using two or four threads on a core. This is due to the fact that it is not possible to issue instructions from the same threads context in back-to-back cycles on MIC [12]. Thus, programmers need to run at least two threads on each core to fully utilize the hardware resources. 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 written in `intrinsics` and thus we can ensure 100% vector usage.



(a) On ISB (b) On MIC

Figure 8: Arithmetic throughput using different threads (60, 120, 240) and issue widths (1 single instruction versus 2 independent instructions) (in GFlops).

[1] The run-time emitted 'Illegal instruction' when using the `mad` instruction.

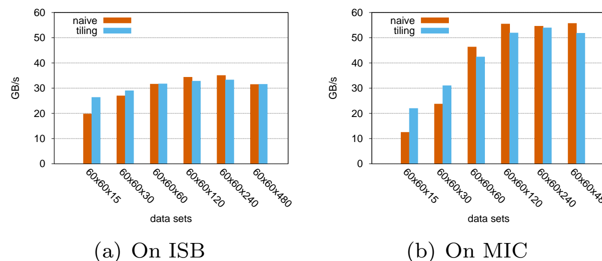
4.4 Stencil

4.4.1 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. The Parboil benchmark includes a stencil code, representing an iterative Jacobi solver of the heat equation on a 3-D structured grid, which can also be used as a building block for more advanced multi-grid PDE solvers. The solver can be expressed as triply nested loops *ijk* over: $B(i,j,k) = \alpha \cdot A(i,j,k) + \beta \cdot (A(i-1,j,k) + A(i+1,j,k) + A(i,j-1,k) + A(i,j+1,k) + A(i,j,k-1) + A(i,j,k+1))$.

4.4.2 High-level Analysis

The Parboil benchmark simply puts an `pragma` over the outer-most loop. We change the code with the *tiling* technique, and we use off-chip memory bandwidth (GB/s) as the performance metric. We measure each data set (6 data sets all together) 10 times and get the median value. Furthermore, we perform the same measurement on the tiling implementation (10 trials for each tile size) and get the maximum bandwidth. The memory bandwidths (in GB/s) for both ISB and MIC are shown in Figure 9. Overall, we see that the achieved bandwidth on MIC is twice as large as that achieved on ISB for the large data sets. On ISB, we see the bandwidth is around 30 GB/s and it changes slightly over data sets and the optimization. On MIC, however, the naive implementation experienced a significant bandwidth change over the data sets (up to 5× faster for large data sets). The tiling implementation brings an increase in memory bandwidth on small data sets ($60 \times 60 \times 15$ and $60 \times 60 \times 30$).



(a) On ISB (b) On MIC

Figure 9: Memory bandwidth of stencil solvers (in GB/s)

4.4.3 Low-level Analysis

Tiling can bring us more parallelism than the naive implementation. As we mentioned, the Parboil simply parallelized the stencil kernel according to the outer loop, and thus the maximum parallelism is K ($K = [15, 480]$). When K is smaller than the number of cores, we can not fully utilize the processing cores on MIC. This gives an explanation why the tiling implementation obtained a larger memory bandwidth on small data sets than the naive implementation.

Furthermore, using the tiling implementation was expected to get better performance than the naive implementation due to better utilization of caches. But in most cases we achieved similar bandwidths on both processors with and without tiling. This is possibly because both the naive and the tiling implementations are implemented in the *tiling* manner but in a different granularity. Another factor is that our tiling implementation in OpenMP has an overhead of distributing data tiles to threads.

4.5 Sparse Matrix-Dense Vector Multiplication (SpMV)

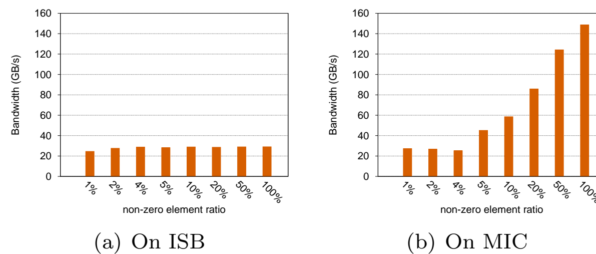
4.5.1 Description

Sparse matrix-vector multiplication is the core of many iterative solvers. We revised the benchmark based on the Parboil OpenMP implementation to make the ratio of non-zero elements adjustable. Given an $M \times N$ sparse matrix A and a dense vector x, we consider the sparse matrix-vector multiply (SpMV) $y \leftarrow Ax$, with y a dense result vector. A typical way of storing a sparse matrix A is the Compressed Row Storage (CRS) format [16], 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 ith row of A. Thus, the amount of memory that needs to be transferred ($bytes$) and the flops ($flops$) are: $bytes = 2 \times M \times 8 + nz(A) \times (8+4) + (M+1) \times 4$, $flops = nz(A) \times 2$ (note that we use `integer` to store the array index and `double` to store the values).

4.5.2 High-level Analysis

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. Parboil parallelizes the SpMV kernel by letting each thread (120 threads in total) process multiple contiguous rows in parallel. The ratio of non-zero data elements can be controlled by a parameter ra. We do not make any code changes, but we change the value of ra for each measurement. For all the experiments, we run the code for 3 times to warm 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. We show the results in terms of off-chip memory bandwidth (in GB/s) in Figure 10. We see that when changing the ra, the bandwidth of SpMV on ISB keeps stable, while it increases over ra on MIC.

when A is sparse (i.e., it has very few non-zero data elements), we obtain poor performance on MIC. This is because of the small bandwidth when accessing vector x. In particular, this occurs when the contiguous 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 though 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. On the other hand, we can achieve high memory bandwidth on the Xeon processor even when ra is small.



(a) On ISB (b) On MIC

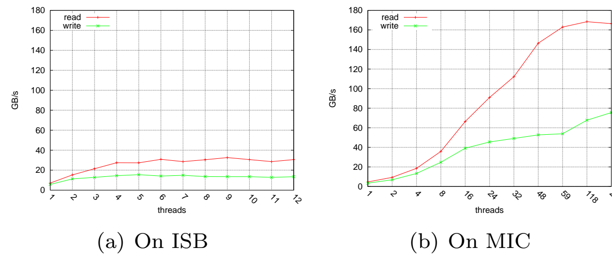
Figure 10: Memory bandwidth of SpMV (in GB/s)

4.5.3 Low-level Analysis

We measure the achieved memory bandwidth for both `read` and `write` (Figure 11). 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 on ISB ($1 \sim 12$) and MIC ($1 \sim 236$). We show the memory

bandwidth in Figure 11. Overall, we see both the maximum bandwidths are far below the theoretical numbers (42.6 GB/s for ISB and 320 GB/s for MIC). On MIC, we note that the `read` bandwidth increases when using more threads, peaking at 168 GB/s (from using 118 threads on). On the other hand, the `write` bandwidth is not as large as the `read` bandwidth. When using 236 threads, we can obtain the maximum `read` bandwidth (76 GB/s). 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.

SpMV is memory-bandwidth bound when the matrix is large. According the read and write ratio in SpMV, we use the `read` bandwidth to approximate the SpMV bandwidth. Hence, the maximum bandwidths on ISB and MIC are 30 GB/s and 168 GB/s, respectively. We note that the bandwidth numbers of SpMV are very close to the maximum ones when $ra = 100\%$ (i.e., dense-matrix dense vector multiplication). However, when A is sparse, we obtain poor performance due to the irregular (unfriendly) memory access pattern of SpMV on MIC. This also relates to the prefetching mechanism mentioned in section 4.1.



(a) On ISB (b) On MIC

Figure 11: Memory bandwidth of read and write operations (in GB/s)

5 Key Features Summary

In Table 2, we categorize the architectural features/properties into five groups: cores, on-chip memory, off-chip memory, network-on-chip (NoC) and miscellaneous. An architectural property includes property name and its value, which is either queried from specifications or measured with microbenchmarks. **We define a property (or a combination of multiple properties) as a key feature when it meets two criteria: (1) it is controllable by programmers in terms of code, and (2) it can make a difference for the overall application performance.** In Table 2, all features with a tick are *key features*. We also show the optimization techniques in the last column of the table.

5.1 Cores

It is indisputable that *the number of cores* is a key feature. Ideally, the performance will increase linearly over the number of cores (shown in section 4.4). When programming kernels, we have control of the number of logical threads and affinity strategies so as to map threads to cores (or hardware threads). Furthermore, within a core, the SIMD usage plays a key role in the overall performance. Each core of MIC has a 512-bit SIMD unit which can work on 8 double-precision or 16 single-precision data elements simultaneously. Thus, programmers are supposed to use a vector-friendly format of code for a high SIMD usage (shown in section 4.3).

5.2 On-chip Memory

We roughly classify on-chip memory into caches (hardware-managed) and scratch-pad memory (programmer-managed). MIC has two levels of hardware-managed caches. We measured that the local access latency (of the

Table 2: Feature List of the Xeon Phi 5110P.

Group	Properties name	value	Measured/Query	Key Features	Optimization Technique
Cores					
	Frequency	1.05 GHz	Q		
	NO. of cores	60	Q	✓	enough parallelism
	SIMD width	512 bits	Q	✓	vectorization
On-chip Memory					
	Levels	2	Q		
	Type	Hardware	Q		
L1 Cache					
	LLC	N	Q		
	latency	6 cycles	M		
	capacity	32KB	M/Q		
	cache-line size	64 bytes	M/Q		
L2 Cache					
	LLC	Y	Q		
	local latency	25 cycles	M	✓	privatization
	remote latency	238 cycles	M	✓	privatization
	capacity	512KB	M/Q		
	cache-line size	64 bytes	M/Q		
Off-chip Memory					
	latency	340 cycles	M		
	capacity	8GB	Q		
	r-bandwidth	168 GB/s	M	✓	friendly MAPs
	w-bandwidth	76 GB/s	M	✓	friendly MAPs
	ECC support	Y	Q	✓	turn off ECC
NoC					
	type	ring	Q		
Miscellaneous					
	prefetching	Y	Q	✓	the long enough stanza

L2 cache) is 25 cycles while the remote access latency is around 238 cycles. Thus, we define a key feature based on a combination of the two properties: we prefer using local caches rather than remote caches. We can ensure the local caches usage by utilizing *privatization*.

5.3 Off-chip Memory

Vendors often report theoretical memory bandwidth for each processor. However, the numbers cannot be achieved in both benchmarks and real-life applications. As we have shown in section 4.5, the maximum read and write memory bandwidths are 168 GB/s and 76 GB/s on MIC, which are far below the theoretical number (320 GB/s). We can achieve these maximum off-chip memory bandwidths in the presence of unit-stride memory access patterns (friendly MAPs). For such cases, the prefetching mechanism can prefetch the data lines nicely, which is further explained in section 5.4. In addition, when turning off ECC support on MIC, we can get a 15% bandwidth improvement [10]. Thus, we see the ECC support as a key feature. Users can choose to switch ECC

ON/OFF based on their requirements.

5.4 Miscellaneous

In addition, MIC provides a prefetching mechanism to fetch data from the off-chip memory into the on-chip memory. Prefetching is based on the heuristics of predicting and moving the next-to-be-used cache-line, and thus a regular memory access pattern can ensure the efficiency of prefetching. Ideally, the prefetcher can achieve its highest efficiency when an application needs contiguous data from the off-chip memory. Thus, programmers need to transform the code or change the data layout to ensure the stanza as long as possible.

6 Discussion and Conclusion

As we have mentioned, *feature identifier* is the component in our optimizing framework (Sesame) that can identify the key features for a given many-core processors. Once we got a list of key features, our source-to-source translator needs two steps to perform optimizations. The first step is *matching*. Each key feature has a corresponding optimization technique(s) and an ideal use scenario(s). The ideal use scenario is the way to fully utilize the key feature. Matching the ideal use scenario with the input kernel gives us an optimization target. Guided by the optimization target, we use one or multiple compiling passes to translate the naive code to an optimized format on the Xeon Phi. This is still the work in progress.

To summarize, we present a two-stage comparative approach to abstract key features for a given many-core processor. In this paper, we took an Intel Xeon Phi coprocessor and demonstrated how to get key features with our approach. The feature abstraction consists of two levels: the high-level (application-level) and the low-level (microbenchmark-level). Using a reference processor makes abstraction straightforward. We summarized the key features of Intel Xeon Phi in Table 2. We found that the Xeon Phi coprocessor works like the traditional multi-core CPUs and thus their key features are alike. But programmers have to put more efforts on the upgraded features (e.g., more cores and wider SIMD) to map a given application to the coprocessor.

References

[1] Michael Garland, Scott Le Grand, John Nickolls, Joshua Anderson, Jim Hardwick, Scott Morton, Everett Phillips, Yao Zhang, and Vasily Volkov. Parallel computing experiences with CUDA. *IEEE Micro*, 28:13–27, July 2008. 4

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

[3] Ana L. Varbanescu, Pieter Hijma, Rob van Nieuwpoort, and Henri Bal. Towards an effective unified programming model for Many-Cores. In *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '11, pages 681–692, Washington, DC, USA, 2011. IEEE Computer Society. 4

[4] The Khronos OpenCL Working Group. OpenCL - The open standard for parallel programming of heterogeneous systems. http://www.khronos.org/opencl/, Februay 2012. 4

[5] OpenACC. Directives for Accelerators. http://www.openacc-standard.org/, January 2013. 4

[6] J. Bueno, J. Planas, A. Duran, R. M. Badia, X. Martorell, E. Ayguade, and J. Labarta. Productive programming of GPU clusters with OmpSs. In *IPDPS'12*, pages 557–568. IEEE, May 2012. 4

[7] Jianbin Fang, Ana L. Varbanescu, and Henk Sips. A comprehensive performance comparison of CUDA and OpenCL. In *2011 International Conference on Parallel Processing (ICPP'11)*, pages 216–225. IEEE, September 2011. 4

[8] Jie Shen, Jianbin Fang, Henk Sips, and Ana L. Varbanescu. Performance gaps between OpenMP and OpenCL for multi-core CPUs. In *Proceedings of the 41st International Conference on Parallel Processing Workshops (ICPPW'12)*, September 2012. 4

[9] Jianbin Fang, Ana L. Varbanescu, and Henk Sips. Sesame: A User-Transparent optimizing framework for Many-Core processors. In *Proceedings of the13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid'13 Doctoral Symposium)*, 2013. 4

[10] Jianbin Fang, Ana L. Varbanescu, Henk Sips, Lilun Zhang, Yonggang Che, and Chuanfu Xu. Benchmarking intel xeon phi to guide kernel design. Technical Report PDS-2013-005, Delft University of Technology, April 2013. 4, 15

[11] John A. Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng D. Liu, and Wen-mei W. Hwu. Parboil: A revised benchmark suite for scientic and commercial throughput computing. Technical Report IMPACT-12-01, University of Illinois at Urbana-Champaign, March 2012. 4

[12] Intel. *Intel Xeon Phi Coprocessor System Software Development Guide*, November 2012. 5, 11

[13] Intel. Sandy Bridge. http://en.wikipedia.org/wiki/Sandy_Bridge, January 2013. 5

[14] Intel. *An Overview of Programming for Intel Xeon Processors and Intel Xeon Phi Coprocessors*, October 2012. 6

[15] 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. 10

[16] Nathan Bell and Michael Garland. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, December 2008. 13