Scale-Free Sparse Matrix-Vector Multiplication on Many-Core Architectures

Yun Liang, *Member, IEEE*, Wai Teng Tang, Ruizhe Zhao, Mian Lu, Huynh Phung Huynh, and Rick Siow Mong Goh

***Abstract*—Sparse matrix-vector multiplication (SpMV) is one of the most important kernels for many applications. In this paper, we study the implementation of SpMV for scale-free matrices on many-core architectures including graphic processing units and Xeon Phi coprocessors. We first propose a hardware oblivious implementation for heterogeneous many-core processors using OpenCL. Our OpenCL implementation uses a novel SpMV format called hybrid COO+CSR (HCC), which employs 2-D jagged partitioning to balance the workload among a large number of cores and improve the data locality. Moreover, the OpenCL implementation is designed to be parametric, which allows systematic performance tuning. We conduct experiments to evaluate the efficiency of our hardware oblivious implementation. Experiments show that it achieves comparable performance to the Intel MKL and state-of-the-art OpenCL-based ViennaCL library implementation. Although the OpenCL implementation provides functional portability for heterogeneous systems, it fails to take advantage of the low-level architectural features. To further improve the performance, we propose a hardware conscious implementation using the native parallel programming language. We use the Xeon Phi platform as a case study. In our hardware conscious implementation, we ensure that the HCC format efficiently utilizes the vector process units on Xeon Phi by employing low-level intrinsics, and improve the overall performance through locality-aware block mapping, and intrablock tiling. Experiments using a wide range of representative scale-free matrices demonstrate that compared with the OpenCL-based hardware oblivious implementation, the hardware conscious implementation achieves 2.2× speedup on average. Compared with MKL, the hardware conscious implementation achieves 3.1× speedup on Xeon Phi.**

***Index Terms*—Graphic processing unit (GPU), performance optimization, sparse matrix-vector multiplication (SpMV), Xeon Phi.**

Manuscript received June 20, 2016; revised December 4, 2016; accepted February 23, 2017. Date of publication March 10, 2017; date of current version November 20, 2017. This work was supported in part by the National Natural Science Foundation of China under Grant 61672048, and in part by the Project CARCH201502 from the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences. This paper was recommended by Associate Editor M. T. Kandemir. *(Corresponding author: Yun Liang.)*

Y. Liang is with the Center for Energy-Efficient Computing and Applications, School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China, and also with the Collaborative Innovation Center of High Performance Computing, National University of Defense Technology, Changsha 410073, China (e-mail: ericlyun@pku.edu.cn).

R. Zhao is with the Center for Energy-Efficient Computing and Applications, School of EECS, Peking University, Beijing 100871, China.

W. T. Tang, M. Lu, H. P. Huynh, and R. S. M. Goh are with the A*STAR Institute of High Performance Computing, Singapore 138632.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2017.2681072

I. Introduction

SPARSE matrix-vector multiplication (SpMV) is a critical kernel that finds applications in many high performance computing and embedded systems domains including structural mechanics, machine learning, embedded vision, and data mining. Many algorithms use SpMV iteratively for their computation. For example, the conjugate gradient method [1] uses SpMV to solve a symmetric system of linear equations, whereas the PageRank algorithm [2] uses SpMV to determine the ranks of Web pages. SpMV computation is a performance bottleneck for many of these algorithms [3]–[5]. However, efficient implementation of the SpMV kernel remains a challenging task due to its irregular memory access behavior.

In this paper, we focus on optimizing SpMV for scale-free sparse matrices for many-core architectures. Scale-free sparse matrices arise in many practical applications, such as in the study of Web links, social networks, and transportation networks [6]. Unlike sparse matrices from engineering applications, which are more regular in nature (i.e., the number of nonzeros in each row is similar), a sparse matrix that exhibits scale-free properties is highly irregular. It has many rows with very few nonzeros but has only a few rows with a large number of nonzeros. As such, SpMV computation on such matrices is particularly challenging due to the highly irregular distribution of nonzeros. Many existing implementations such as Intel MKL perform well for regular matrices, but are inefficient for scale-free sparse matrices due to the imbalanced workloads and poor data locality.

In the recent years, graphic processing units (GPUs) and Intel Xeon Phi are becoming popular hardware platforms for performance acceleration. In general, these platforms feature a large number of cores. For example, Intel Xeon Phi 5110P contains 60 cores and AMD FirePro S9150 GPU contains more than one thousand cores. At the same time, emergence of the OpenCL programming model has lowered the entry barrier for heterogeneous system programming and provides portability across architectures, where the same OpenCL code can be executed across a variety of processors including CPUs and GPUs. Nevertheless, even though different architectures can be programmed using the same OpenCL programming model, they still have distinct architectural features. For example, Xeon Phi features 512-bit Single Instruction Multiple Data (SIMD) vector processing units (VPUs), which do not exist on GPUs.

The goals of this paper are two folds. First, we design an efficient and portable SpMV implementation for scale-free matrices using OpenCL. This hardware oblivious

implementation is portable to different heterogeneous systems. We demonstrate its efficiency and portability on Xeon Phi and GPU. Although the OpenCL implementation provides functional portability, it fails to utilize the distinct low-level architecture features on different platforms. Hence, second, we study the tradeoff between portability and efficiency. We will use the Xeon Phi platform as a case study. We then propose a hardware conscious SpMV implementation for scale-free matrices using the native parallel programming language on Xeon Phi. We optimize its performance by utilizing the underlying low-level programming interfaces and architectural features, which cannot be exploited at the OpenCL level.

To achieve the first goal, we design an efficient SpMV format called hybrid COO and CSR format (HCC), which is applicable to different many-core architectures for scale-free matrices. HCC is designed for proper load balancing and cache optimization. It achieves this by employing a 2-D jagged partitioning to equally partition the nonzeros into blocks. The hardware oblivious version is based on OpenCL for portability. We understand that execution of the same OpenCL program varies on heterogeneous architectures. Hence, the hardware oblivious implementation is designed with a set of parameters that can be tuned for different architectures. For the second goal, we use Xeon Phi as a case study and implement a hardware conscious version that exploits low-level architecture features, such as the SIMD intrinsics available on Xeon Phi. In particular, we design a prefix-sum computation using SIMD intrinsics, and also employ optimization techniques such as locality-aware block mapping, and intrablock tiling. Note that all these optimizations are not supported in the high-level OpenCL model. We find that with these low-level optimizations, the performance of the hardware conscious implementation achieves substantial improvement compared to the hardware oblivious implementation. Hence, we argue that OpenCL model provides portability for heterogeneous systems and is easier to maintain, but its performance can be limited due to the lack of low-level architectural feature support. For high performance, hardware conscious optimization is still required. This paper has the following contributions.

1) We design an efficient and portable SpMV format called HCC for scale-free matrices on many-core architectures including GPUs and Xeon Phi coprocessors. The hardware oblivious implementation is designed with a set of parameters for performance portability on heterogenous architectures.
2) We study the tradeoff between portability and efficiency. We use Xeon Phi as a case study and design a hardware conscious implementation. The hardware conscious implementation employs a group of low-level architecture-aware optimizations.

Experiments using a wide range of representative scale-free matrices demonstrate that our OpenCL-based hardware oblivious implementation is comparable to the Intel MKL implementation on Xeon Phi and gives better portability on Xeon Phi and GPU compared to the state-of-the-art OpenCL-based ViennaCL library implementation. The hardware conscious implementation further improves the performance of the hardware oblivious implementation by 2.2X. A preliminary version of this paper appears in [7]. In this paper, we extend the implementation for Xeon Phi to a hardware oblivious implementation using OpenCL for both GPU and Xeon Phi.

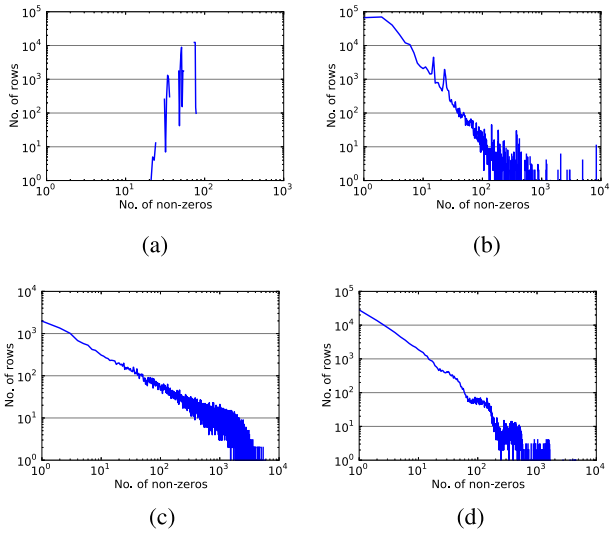


Fig. 1. Comparison of regular and scale-free matrices. *x*-axis: nonzeros per row, *y*-axis: frequency. (a) FEM/cantilever. (b) Stanford. (c) Mouse-gene. (d) R-MAT (18, 16).

The remainder of this paper is organized as follows. We introduce background details on scale-free SpMV, many-core architectures, the OpenCL programming model, as well as the motivation for a high performance SpMV implementation for scale-free matrices in Section II. Details for the hardware oblivious implementation are presented in Section III, and in Section IV for the hardware conscious implementation on Intel Xeon Phi. We conduct experiments in Section V. Finally, Section VI discusses the related work and Section VII concludes this paper.

II. Background

A. Scale-Free Sparse Matrices

The occurrence of sparse matrices or equivalently, networks exhibiting scale-free nature in practical settings, is attributed to a self-organizing behavior called preferential attachment which has attracted a lot of studies [6]. In a scale-free network, the distribution of the number of connections to each node (i.e., the degree of the graph) follows that of a power law. Equivalently, this means that for a scale-free matrix, the distribution of the nonzeros per row of the matrix follows the power law. Fig. 1 shows the differences between regular and scale-free matrices in terms of the distribution of the nonzeros per row of a matrix. The plots are logarithmic in both axes; the horizontal axis denotes the number of nonzeros per row and the vertical axis denotes the number of rows having that specified number of nonzeros. Fig. 1(a) demonstrates a matrix from the engineering sciences (e.g., constructed using Finite Element Method (FEM)). Such matrices tend to have multimodal distributions and their structures are more regular in nature. We will call these matrices "regular matrices." In contrast, matrices such as those derived from Web graphs [see Fig. 1(b)] and networks [see Fig. 1(c)] tend to exhibit scale-free properties,

i.e., their distribution of nonzeros per row follow the power law. These matrices are termed "scale-free matrices." As a comparison, Fig. 1(d) shows the distribution of the nonzeros per row of a scale-free sparse matrix generated from a Kronecker graph model [8]. In this recursively defined model, two key parameters, s and e, define a scale-free R-MAT(s, e) matrix. For a given s and e, a square matrix with dimensions $2^s \times 2^s$ and an average number of nonzeros per row (e) is obtained. One can see that Fig. 1(b) and (c) look very similar to Fig. 1(d), whereas there is no resemblance at all between the regular matrix in Fig. 1(a) and the R-MAT in Fig. 1(d).

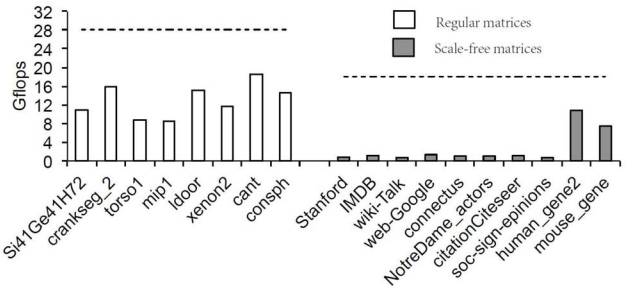


Fig. 2. Performance of MKL CSR SpMV for regular and scale-free matrices.

B. Many-Core Architecture

Recently, many-core architectures have become popular for accelerating scientific applications. Such architectures usually consist of tens to thousands of cores on a chip. Two representative product families are GPUs and Intel's Xeon Phi coprocessors. However, apart from having many cores, these two architectures are vastly different in their design and have distinct hardware features.

The Intel Xeon Phi coprocessor is based on the Intel many integrated core architecture. In this paper, we use the Xeon Phi 5110P coprocessor, which integrates 60 cores on the same package, each running at 1.05 GHz. Up to four hardware threads per core are supported, and a maximum limit of 240 threads can be scheduled on the coprocessor. Every core in the processor contains a local 64 KB L1 cache, and a 512 KB L2 unified cache. All the 512KB L2 caches on the 60 cores are fully coherent via the tag directories, and are connected by a 512-bit wide bidirectional ring bus. When an L2 cache miss occurs on a core, requests will be forwarded to other cores via the ring network. One of the major features of Xeon Phi is the wide 512-bit VPU present on each of the cores, effectively doubling the 256-bit vector width of the latest Intel Xeon CPUs. To achieve high performance on Xeon Phi, it is crucial to utilize the VPUs effectively. Additionally, Xeon Phi supports low cost atomic operations that can be used for efficient parallel algorithm implementations.

Another representative many-core architecture is the GPU. One GPU usually consists of tens of *multiprocessors*, each of which contains hundreds of cores. In total, there are thousands of cores for a typical modern GPU. Each multiprocessor has a local L1 cache, and an L2 cache that is shared among all the multiprocessors. GPU threads have very low context switching overheads. Thus, GPUs are designed for massive threading in order to maximize throughput. An algorithm with massive data parallelism can generally fit very well into GPU's architecture. The smallest thread scheduling and management unit on the GPU is a group of threads, called a *warp* (32 threads) on NVIDIA GPUs and a *wavefront* (64 threads) on AMD GPUs. *Coalesced* memory access within a thread group is essential for achieving high memory bandwidth utilization on the GPU.

C. Programming Model

To program the various many-core architectures, different vendors have designed native programming frameworks or languages to support their own products. For example, Intel's Cilk Plus compiler can be used for development on Xeon Phi, whereas CUDA is developed by NVIDIA to support GPU computing on NVIDIA GPUs.

On the other hand, there are also programming models or standards that are portable to different many-core architecture, such as the OpenCL programming model. Today, OpenCL is supported on most of the heterogeneous many-core architectures. OpenCL defines a smallest processing unit as a *work-item*. Work-items are then grouped into *work-groups*. Multiple work-groups can be executed concurrently by the underlying scheduling system. However, the way that the OpenCL model is mapped to the underlying hardware varies across different architectures. On the GPU, each work-group is scheduled on one multiprocessor. Work-items within a work-group are organized based on *wave-fronts* (similar to the CUDA warps). Wavefronts are executed by the GPU using multiple cores within the same multiprocessor in an SIMD-like style. On the other hand, on Xeon Phi, each work-group is handled by one thread. The parallelism of multiple work-items within a work-group is exploited using SIMD vectorization. These differences between platforms affect the implementation and performance tuning on heterogeneous architectures.

D. Motivation

We first evaluate the state-of-the-art SpMV implementation for scale-free matrices on many-core architecture. In particular, we evaluate Intel's MKL (denoted as MKL) [9] on Intel Xeon Phi. But similar conclusions can be drawn on other many-core architectures, such as GPUs. The MKL implementation uses CSR format.

Fig. 2 compares the performance of MKL for both regular and scale-free matrices. The dotted lines denote the theoretical performance. As we can see, MKL works well for regular matrices, but not for most of the scale-free matrices. Specifically, MKL achieved 11.3 Gflops on average for regular matrices but only 2.6 Gflops for scale-free matrices. Hence, there is a necessity to develop high-performance SpMV algorithm for scale-free matrices on many-core architectures. MKL performed better for the last two scale-free matrices compared with the other scale-free matrices. The reason is because these two matrices are denser than the other scale-free sparse matrices. However, with a better implementation, the performance for these two matrices can be improved further.

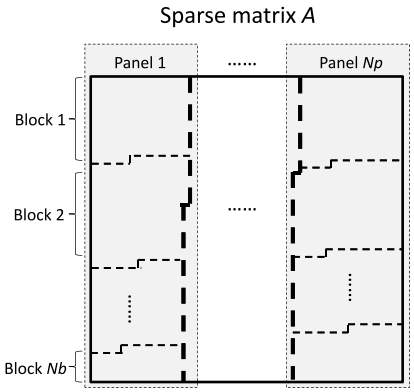


Fig. 3. HCC format with 2-D jagged partitioning. Nonzeros are first equally divided into N_p vertical panels, and then within a vertical panel, further partitioning into N_b blocks. There are $N_p \times N_b$ blocks in total.

III. Hardware Oblivious Implementation Using OpenCL

In this section, we provide the details of our hardware oblivious implementation of SpMV ($y = y + A \cdot x$) for scale-free matrices on many-core processors. A new format called HCC is devised and the details of the portable OpenCL implementation and performance tuning are presented.

A. HCC Format and OpenCL Implementation

We first develop an HCC format for scale-free matrices on many-core architectures. This format has two important features. First, it achieves workload balance by grouping equal number of nonzeros into blocks. Second, it improves data locality by 2-D partitioning. We call such a partitioning approach as *2-D jagged partitioning*.

Fig. 3 shows the layout of the 2-D jagged partitioning. Nonzero elements are equally partitioned into a number of jagged 2-D blocks. The 2-D blocks are organized hierarchically. It first divides the nonzeros among a number of panels. The nonzeros in each panel are then further partitioned equally into a number of blocks that are layout vertically in that panel. A panel is essentially a partitioning of the matrix such that the number of nonzeros in each vertical panel is balanced. Likewise, a block is a vertical partitioning of each panel so that the number of nonzeros in each block within the panel is balanced. Within each block, the nonzero elements are further logically grouped into segments. We use the term *segment* to refer to each logical sub-unit within a block. The same work-items in a work-group can compute for several segments in a block. Overall, the organization of the HCC format can be described using three levels—panel, block, and segment. For ease of reference, Table I provides the definitions of the terminology used by our HCC format. We will describe how these are mapped onto the OpenCL abstractions of work-groups and work-items in the following.

In essense, the 2-D jagged partitioning scheme used by HCC ensures a balanced workload by dividing the nonzeros equally among all the blocks, which can be processed by different cores in many-core processors in parallel. It also helps to improve the locality and thus the cache hit ratio. A low cache hit ratio will pose a serious challenge for SpMV, and this is especially true for scale-free sparse matrices (as shown in Section V). For SpMV ($y = y + A \cdot x$) computation, there exists data reuse opportunity for the x vector. In our HCC format, the vertical panels are designed to improve the temporal locality for the x vector as each panel requires an adjacent block of rows in the x vector. This helps to reduce the possibility that elements in x are evicted from the cache by the conflicting accesses when they are visited again. The blocks within each panel, on the other hand, help to improve the cache locality of the output y vector.

The HCC format requires a few data structures to assist in the SpMV computation. First, each nonzero element contains a tuple with two data—the double-precision floating-point value of the nonzero, and the column index of the nonzero. These are stored contiguously in the *val* array and *col_idx* array, respectively, and padded to the size of a work-group for OpenCL implementation. We also use the *seg_ptr* array to keep track of the segments containing elements that span different rows. For such segments, the end-of-row positions are indicated using a bit field, which is stored in the *eor_arr* array, and their corresponding row indices are stored in the *row_idx* array. Other auxiliary book-keeping data structures include the start segments *start_seg* and the end segments *end_seg* processed by each work-group. The HCC format combines the benefits of both COO and CSR formats. On one hand, being similar to COO, HCC format can partition the workload equally among the threads. On the other hand, being similar to CSR, HCC format saves the storage space as most of the row indices need not be stored.

Next, we describe how the HCC format is mapped to the OpenCL implementation. For ease of reference, Table II defines the different parameters used in our OpenCL implementation. Suppose there are N_p panels and each panel contains N_b blocks. Then, there are $N_p \times N_b$ blocks in total.

TABLE I
Description of Terms Used

Term	Description
Panel	A horizontal partitioning of the matrix such that the non-zero elements are balanced in each panel.
Block	A vertical partitioning of each panel such that the non-zero elements are balanced across the blocks.
Segment	A logical grouping of the non-zero elements within each block.
Work-item	An OpenCL abstraction of a basic unit of work.
Work-group	An OpenCL abstraction that refers to a group of work-items.

TABLE II
Tunable Performance Parameters

Parameter	Description
N_p	The number of vertical panels.
N_b	The number of blocks per panel.
N_g	The number of work-groups to process a block.
N_i	The number of work-items in each work-group.

Algorithm 1 SpMV Computation of $y = Ax + y$ Using HCC Format

```
    Computation phase:
 1: v ← start_seg
 2: (vidx, sidx, ridx) ←load initial values for workgroup g_i
 3: set tmp_y to temporary array for current panel
 4: while v < end_seg do
 5:     col ← load(&col_idx[vidx])
 6:     val ← load(&val_arr[vidx])
 7:     inp ← gather(&x[0], col)
 8:     if v = seg_ptr[sidx] then
 9:         acc ← reduce_add(res)
10:         res ← val × inp
11:         res[0] ← res[0] + acc
12:         eor ← load(&eor_arr[sidx])
13:         res ← prefix_sum(res, eor)
14:         row ← load(&row_idx[ridx])
15:         scatter(&tmp_y[row], res)
16:         sidx ← sidx + workgroup_size
17:         ridx ← ridx+ count(eor)
18:     else
19:         res ← res + val × inp
20:     end if
21:     vidx ← vidx + workgroup_size
22:     v ← v + 1
23: end while
24: wg_row[g_i] ← load(&row_idx[ridx])
25: wg_val[g_i] ← res[workgroup_size-1]

    Update phase:
26: tmp_y[wg_row[g_i]] ← tmp_y[wg_row[g_i]] + wg_val[g_i]

    Merge phase:
27: y ← sum tmp_y arrays from all panels
```

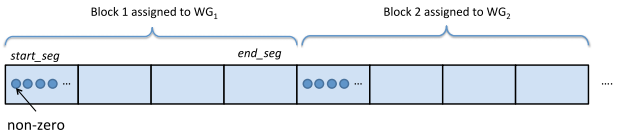


Fig. 4. Illustration where computation for two blocks are assigned to two work-groups. WG means work-group.

To process a block, we use N_g work-groups where each work-group contains N_i work-items. Each work-group will be responsible for a given sequence of data determined by an associated start and end segment pointer (denoted by *start_seg* and *end_seg*). Hence, the work-group and its work-items will start computation from a given segment and continue until the end of segment is reached. Note that the segment size is equal to N_i. N_p, N_b, N_g, and N_i are tunable performance parameters. Fig. 4 illustrates an example where two blocks are assigned to two work-groups.

Algorithm 1 shows the details of how SpMV is computed using the data structures just described. The algorithm consists of three phases: computation phase, update phase, and merge phase. Lines 1–25 describes the computation phase for each block shown in Fig. 3.

Lines 5–7 load the column index, matrix value and input value. The branch condition in line 8 checks if the group contains elements spanning different rows. If all the elements belong to the same row, then the result is computed and accumulated in *res*. Otherwise, it is necessary to perform a reduction followed by a prefix sum in lines 9–13. The result is written to the *tmp_y* array in line 15 because each panel has an associated temporary array. At the end of the loop (line 25), each work-group will write the last element of the prefix sum and its associated row to two small *wg_row* and *wg_val* arrays. These temporary arrays are used to avoid the potential data conflicts when different work-groups write their last prefix sum value to the same row. These temporary values are then written to the *tmp_y* array in the update phase. This is implemented with a separate kernel as OpenCL enforces a synchronization point between the kernel launches. Finally, in the merge phase, the temporary arrays from different panels are merged into the final y array.

B. Performance Tuning

Algorithm 1 presents the backbone of the OpenCL implementation. However, the final achieved performance depends on various parameters such as the number of panels (N_p), and the number of workgroups (N_g). These parameters impact the performance due to two reasons. First, depending on the distribution of the nonzeros, different matrices prefer different settings of these parameters. Second, the vendor implementation of the OpenCL run-time on different architectures varies. This will affect how the OpenCL abstraction of work-groups get assigned to the actual underlying hardware resources. For example on Xeon Phi, one work-group is mapped to one hardware thread by the compiler, whereas on GPUs one work-group will be assigned and executed by one multiprocessor.

Consequently, the optimal parameter setting varies for different matrices and architectures. Hence, we design a parametric OpenCL implementation, which not only enables performance tuning for different matrices on a specific architecture but also better portability across heterogeneous architectures. We categorize the performance parameters into two categories.

1) Parallelization Granularity: Table II lists the four tunable performance parameters related to the parallelization granularity in our OpenCL implementation. Parameters N_p and N_b affect the organization of the HCC format; parameters N_g and N_i determine how the nonzeros in the HCC format are mapped to the OpenCL model. In addition, these parameters are interdependent. In practice, we find that N_g and N_i are sensitive to different hardware architectures, but have small variations across different matrices for the same hardware architecture. On the other hand, the optimal values of N_p and N_b depend on both the architecture and matrix properties.

2) Mapping From Nonzeros to Work-Items: Parameters N_g and N_i determine the number of work-groups for a block and the number of work-items in a work-group. Based on the OpenCL APIs and hardware features, we consider the following two mappings.

1) *Coalesced Mapping:* This is similar to the coalesced memory access in CUDA programming. All the work-items in a work-group access consecutive nonzeroes. Therefore, one work-item processes one nonzero element at a time.
2) *Vectorized Mapping:* OpenCL features a vector load ($vload_n$) function that is used to read vectors from

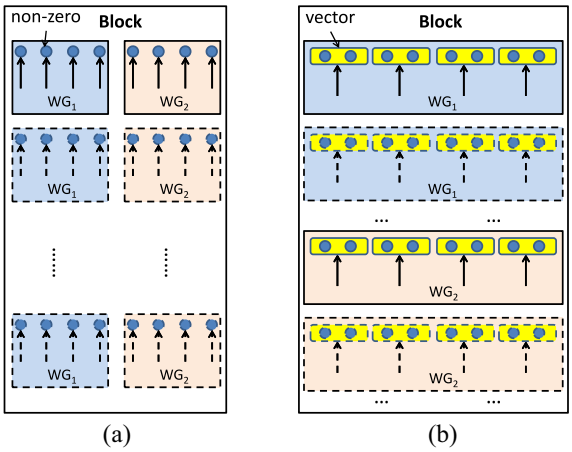


Fig. 5. (a) Coalesced and (b) vectorized mapping from nonzeros to work-items.

memory for generic data types, where n represents the number of elements in the built-in vector type ($n = 2$, 4, 8, or 16). We use the $vload_n$ function to implement our vectorized mapping, where each work-item accesses multiple nonzeros. In our implementation, we let one work-item access eight nonzeros at a time.

Fig. 5 illustrates the coalesced and vectorized mapping together with their work-group and work-item organizations using an example. In this example, we assume there are two work-groups per block ($N_g = 2$) and four work-items per work-group ($N_i = 4$). We also assume there are eight nonzeros per row. Fig. 5(a) shows that with the coalesced mapping, consecutive work-items process the consecutive nonzeros in multiple rounds. Instead, in Fig. 5(b), nonzeros are first organized as vectors (two nonzeros form a vector in this example), and each work-item loads and processes one vector at once.

The performance of the two mapping mechanisms depend on the OpenCL implementations on different hardware architectures. We expect that the coalesced mapping will be more effective for GPU as it can benefit from the high memory bandwidth on GPU. On the other hand, the vectorized mapping is more efficient on Xeon Phi because it can utilize the VPUs available on the cores.

IV. Hardware Conscious Implementation on Intel Xeon Phi

Although the hardware oblivious implementation based on OpenCL provides functional portability for heterogeneous systems, it fails to take advantage of the low-level architecture features. In order to study the tradeoff between portability and performance acceleration capability, we propose a hardware conscious implementation using the native parallel programming model. We use the Xeon Phi platform as a case study. Our native implementation is developed using C++ with OpenMP and intrinsics. Specifically, we particularly focus on the low-level architecture optimization techniques that are unavailable in OpenCL, but are supported by native implementation.

Note that the native implementation has similar tunable parameters as the OpenCL based implementation. However, in the OpenMP programming model, there are only threads rather than work-groups and work-items. We use N_t to denote the number of threads used for processing one block. Therefore, there are three tunable parameters in the native implementation, which are N_p, N_b, and N_t.

A. Performance Bottleneck Analysis

In order to develop an efficient hardware conscious implementation, we first need to understand the performance bottlenecks and resource utilization of the state-of-the-art native SpMV implementation. The Intel VTune profiler [10] is used to analyze the performance bottlenecks of the MKL implementation in detail.

We first collect the average clocks per instruction (CPIs) for each of the matrices as shown in Fig. 6(a). The ideal CPI is 4 [10] on our Xeon Phi platform. However, the achieved average CPI of MKL for scale-free matrices is about 13, indicating that MKL is inefficient for scale-free matrices, and that there exists a large room for improvement. To achieve high performance on Xeon Phi, it is also crucial to use the VPU effectively [10]. We examine the vector utilization efficiency of MKL in Fig. 6(b). The vectorization intensity (VI) metric is defined as the average number of active elements per VPU instruction executed. The maximum value for VI is 8 for double-precision elements. However, the average vectorization intensity achieved by MKL for scale-free matrices is only about 3.4, which is less than half of the ideal value.

Furthermore, because SpMV is inherently memory bound, its overall performance critically depends on the performance of the memory hierarchy. Therefore, we also investigate the average latency of memory accesses. In Fig. 6(c), we observe that the L1 cache hit rates are generally much lower than 95%. A hit in the L1 cache takes only one cycle, whereas an L1 miss results in additional 20 cycles to fetch data from the L2 cache. Therefore, it is important to improve data locality to increase L1 cache hit rate.

Similarly, when an L2 cache miss occurs, a penalty of more than 250 cycles is incurred to fetch data from a remote cache or from main memory. Unfortunately, although Xeon Phi has L1 hit metrics, it does not provide any metric for measuring L2 hit rate. As a workaround, the profiler provides a derived metric called the estimated latency impact (ELI) to give an approximate gauge of L2 cache performance. ELI estimates the average penalty in cycles for each L1 cache miss (estimated using the number of cycles needed to fetch data from memory divided by the number of L1 misses) and a threshold of 145 cycles is prescribed by Intel [10]. This threshold is obtained by averaging the cycles needed to fetch data from L2 cache and from memory.

Fig. 6(c) shows that for most matrices, the ELI is much higher than the prescribed threshold. This is an indication that the L2 cache hit ratio is likely to be low for these matrices, resulting in long delays to fetch data from main memory instead of from cache. The ELI of matrices connectus, soc-sign-epinions, human_gene2, and mouse_gene, on the other

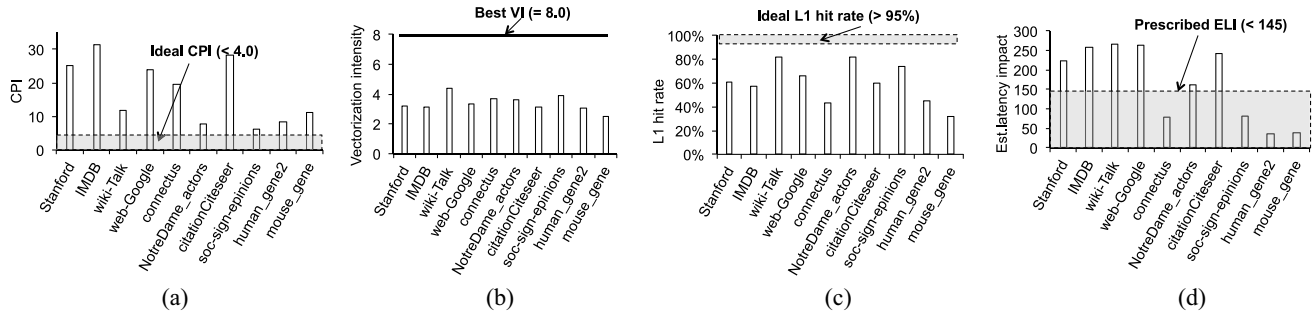


Fig. 6. Performance of MKL. (a) CPI. (b) VI. (c) L1 cache hit rate. (d) ELI.

hand, are observed to be below the prescribed threshold. This is because the ELI is only a derived metric and is not fully accurate for L2 cache performance (see [10] for more details). Other effects such as a high L1 miss rate can distort the metric (since the denominator of ELI is the number of L1 misses) and result in an artificially low ELI value as in the case of these matrices. Thus, the ELI metric only provides an indication of potential problems in L2 performance, and should be interpreted together with the CPI and L1 hit rate metrics.

In summary, MKL suffers from low VI and poor cache performance for scale-free matrices. In the following, we will describe a hardware conscious implementation that remedies these problems by utilizing the low-level architectural features of Xeon Phi.

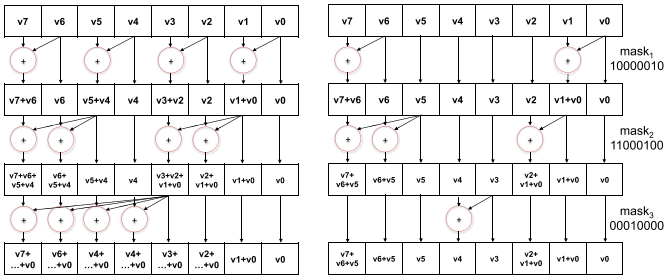


Fig. 7. SIMD segmented prefix-sum. (a) Basic structure of segmented prefix sum operation with all adders turned on. (b) Using precomputed masks to turn off adders for vector with group elements of (3, 2, 3).

B. Native Implementation and Optimization

Our hardware conscious implementation performs several optimizations including vectorization, prefix sum computation using SIMD intrinsics, locality-aware block mapping to hardware threads, and intrablock tiling.

1) Vectorizing the HCC Format: The VPU is one of the key architectural features on the Intel Xeon Phi coprocessor. The 512-bit wide VPU allows us to process eight nonzeros in parallel. In order to efficiently utilize the VPUs, we group nonzeros together to the vector length supported by the coprocessor. This is similar to the vectorized mapping of the OpenCL implementation (see Section III-B). Therefore, in addition to the 2-D jagged partitioning, the nonzero elements in each block are grouped into vectors of eight elements. As we will show in the experiments later, this improves the VI by reducing the empty slots in a vector operation.

2) SIMD Segmented Prefix Sum: The HCC format entails tightly packing the nonzero values into vectors so that they can be efficiently processed by the VPUs. However, the packed nonzero values may cross row boundaries (i.e., when groups of values are from different rows). Hence, we have implemented an SIMD segmented prefix sum operation using Intel Intrinsics [10] to calculate the values that are to be written out to the y vector. Fig. 7(a) shows the basic structure of the segmented prefix sum that is used in our implementation. This operation can be implemented with three vector additions. The efficiency of this implementation is made possible by the enhanced SIMD capabilities supported on Xeon Phi, such as the swizzle (*_mm512_swizzle_pd*) and masked add (*_mm512_mask_add_pd*) operations.

To allow this operation to support computing prefix sums for groups of elements, three masks have to be precomputed to disable adders so that elements belonging to different rows are not summed together. As an illustration, consider a vector $\mathbf{v}$ where each of its element v_0 to v_7 contains the product of the values from the matrix and the input vector. Furthermore, assume that the vector contains elements that are from different rows and are grouped into three groups (v_0, v_1, v_2), (v_3, v_4), and (v_5, v_6, v_7), and each group corresponds to elements from different rows. Fig. 7(b) shows the final structure that is used to calculate the prefix sum in this example. The purpose of the three precomputed masks, $mask_1$, $mask_2$, and $mask_3$ is to turn off the appropriate adders associated with each mask as shown in Fig. 7(b). Note that four adders are associated with each mask. $mask_1$ would be set to the binary value 10101010 if all adders were turned on. Given the grouping in the example above, the three precomputed masks would have the binary values 10000010, 11000100, and 00010000, respectively. The result of the segmented prefix sum is $[(v_0, v_0+v_1, v_0+v_1+v_2), (v_3, v_3+v_4), (v_5, v_5+v_6, v_5+v_6+v_7)]$. End-of-row values such as the third element $v_0+v_1+v_2$ are then written out to the output vector.

3) Locality-Aware Block Mapping: On Xeon Phi, hardware threads in the same core share the same L2 cache. Hence, how work-groups are mapped to the hardware threads affects data locality and thus the performance. For the OpenCL-based runtime implementation by Intel, each work-group is assigned to one hardware thread on Xeon Phi. Furthermore, the exact

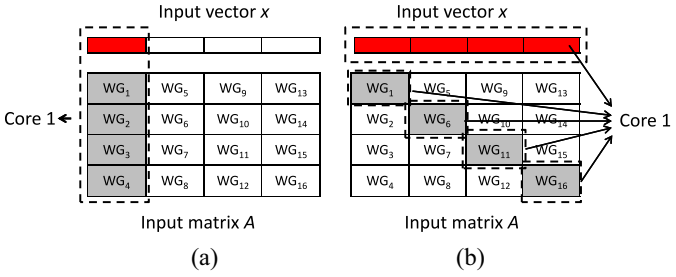


Fig. 8. Examples of mapping work-groups to hardware threads on Xeon Phi, assuming one block is handled by one work-group ($N_g = 1$).

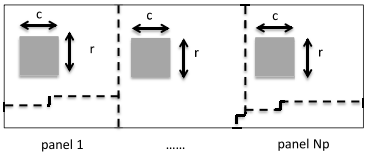


Fig. 9. Examples of intrablock tiling.

way in which work-groups are mapped to hardware threads is vendor-specific (e.g., round-robin or random) and is not disclosed in the case of Xeon Phi. As an illustration, Fig. 8 compares two different mapping schemes. In this example, work-groups are logically arranged in column-major order. Fig. 8(a) maps consecutive four work-groups to the same core while Fig. 8(b) maps the work-groups in the same diagonal to the same core. The mapping scheme shown in Fig. 8(a) is clearly better because better locality can be achieved since the different workgroups on the same core can share references to the input vector.

The OpenCL model does not provide an interface for programmers to directly control the mapping between work-groups and hardware threads. On the other hand, for the native implementation on Xeon Phi, we can manually assign the work-groups to specific threads and cores. In our implementation, we assign consecutive blocks along the column to threads that belong to the same core as shown in Fig. 8(a). We will show in the experiments that this mapping scheme results in better data locality.

4) Intrablock Tiling: Apart from the 2-D jagged partitioning, we also perform intrablock tiling as shown in Fig. 9 for better L1 cache locality. In intrablock tiling, each block is further divided into smaller nonoverlapping sub-blocks called tiles and the nonzeros are then organized and grouped according to these tiles so that the computation will move from one tile to the next.

The optimal tile size ($r \times c$) depends on the L1 data cache size (32 KB on Xeon Phi). We determine r and c through empirical evaluation. 2-D jagged partitioning together with intrablock tiling reduce the cost of gathering from the input vector x and the cost of scattering to the output vector y. The tiling is implemented with the help of low cost atomic operations on Xeon Phi. Note that for the OpenCL implementation, intrablock tiling is not feasible due to the absence of support for atomic floating point operations in the OpenCL model.

TABLE III
LIST OF SPARSE MATRICES USED FOR EVALUATION. COLUMNS ARE DIMENSIONS, TOTAL NO. OF NONZEROS (*nnz*), AVERAGE (AVG), AND MAXIMUM (MAX) NONZEROS PER ROW

Matrix	row × col	*nnz*	avg	max
Stanford	282K×282K	2.3M	8.2	38606
IMDB	428K×896K	3.8M	8.8	1334
wiki-Talk	2.4M×2.4M	5.0M	2.1	100022
web-Google	916K×916K	5.1M	5.6	456
connectus	5K×395K	1.1M	2202	120065
NotreDame_actors	392K×128K	1.5M	3.7	646
citationCiteseer	268K×268K	2.3M	8.6	1318
soc-sign-epinions	132K×132K	841K	6.4	2070
human_gene2	14K×14K	18M	1260	7229
mouse_gene	45K×45K	29M	642	8032

V. EVALUATION

In this section, we first evaluate the performance of our hardware oblivious implementation that is implemented using OpenCL on two different many-core architectures: 1) GPU and 2) Xeon Phi. Then, we evaluate the performance of our hardware conscious implementation that is implemented using the native programming model on Xeon Phi.

A. Experimental Setup

Experiments are conducted on two hardware platforms equipped with the Intel Xeon Phi 5110P coprocessor (referred to as Xeon Phi) and AMD FirePro S9150 GPU (referred to as GPU). We use ten real-world scale-free sparse matrices from the University of Florida sparse matrix collection [11] listed in Table III. These scale-free matrices are from different application domains such as Web graphs, gene networks, or citation networks.

As presented previously in Section III, the hardware oblivious implementation is implemented using OpenCL. We will refer to the execution of this implementation on Xeon Phi and GPU as *Phi_CL* and *GPU_CL*, respectively. For the hardware conscious implementation that uses the native programming model for Xeon Phi, we refer to it as *Phi_native*. As there are different design choices and tunable parameters shown in Sections III and IV, we will report performance numbers with the best configuration, unless specified otherwise. We also compare the performance of our implementation with the state-of-the-art SpMV implementations. On Xeon Phi, we compare with the Intel MKL's CSR implementation (denoted as MKL) [9]. We also compare with the state-of-the-art OpenCL-based SpMV library, ViennaCL [12], which is able to run on both GPU and Xeon Phi.

The metric *Gflops* is used to measure performance and is calculated using $2nnz/t$ where t is the execution time of the SpMV kernel in seconds. Higher *Gflops* indicates better performance. In addition, we use the Intel VTune profiler [10] to investigate the execution efficiency of *Phi_native*.

B. Results of OpenCL-Based Hardware Oblivious Implementations

We first evaluate the performance of *GPU_CL* and *Phi_CL* in detail. We also systematically explore the design

TABLE IV
OPTIMAL PARAMETERS UNDER DIFFERENT ARCHITECTURES

Matrix	GPU_CL $N_i = 256$			Phi_CL $N_b = 236$ $N_g = 1$, $N_i = 1$
	N_p	N_b	N_g	N_p
Stanford	1	8	16	1
IMDB	1	8	64	1
wiki-Talk	1	44	16	1
web-Google	1	32	64	1
connectus	4	8	64	32
NotreDame_actors	1	2	128	1
citationCiteseer	1	4	32	1
soc-sign-epinions	1	8	32	1
human_gene2	1	16	32	1
mouse_gene	1	16	64	4

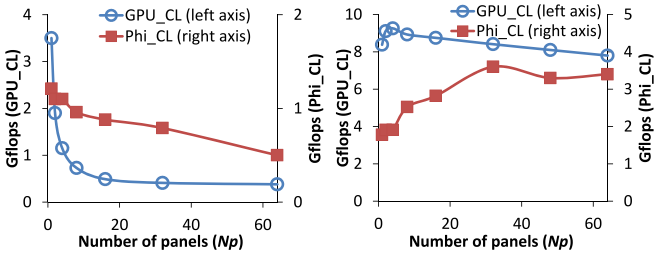


Fig. 10. Performance of GPU_CL and Phi_CL by varying the number of panels (N_p) for (a) stanford and (b) connectus.

space related to performance tuning that was described in Section III-B. For reference, Table IV shows the optimal parameters for the different tunable parameters in Table II and for the matrices in Table III.

1) Parallelization Granularity: From Table IV, we observe that the behavior of the matrices can be roughly divided into two groups, one where the average number of nonzeros in each row are small, and the other where the average and maximum nonzeros are large. In the following, we will use the Stanford and connectus matrices as representative examples to illustrate the behavior of parameter tuning on these two groups of matrices.

Figs. 10–13 show the performance results for Stanford and connectus when varying the tunable parameters for different architectures. Fig. 10(a) shows that for the Stanford matrix, both *GPU_CL* and *Phi_CL* achieve the best performance when $N_p = 1$. Instead, the optimal N_p values for the connectus matrix are 4 and 32, respectively, on GPU and Xeon Phi [Fig. 10(b)]. The reason that the optimal N_p value for connectus is larger than that for Stanford is because the average number of nonzeros per row for connectus is much greater than that of Stanford. Therefore, it is more advantageous to have more panels for connectus since this will improve data locality when accessing the input vector. Fig. 11(a) and (b) shows that for *Phi_CL*, the optimal N_b values are 236 for both matrices.

Based on the results shown in Figs. 10–13 and Table IV, three conclusions can be drawn. First, by comparing the two different curves within each figure, we notice that the

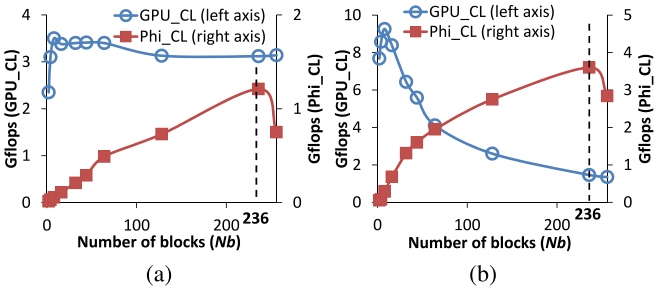


Fig. 11. Performance of GPU_CL and Phi_CL by varying the number of blocks (N_b) for (a) stanford and (b) connectus.

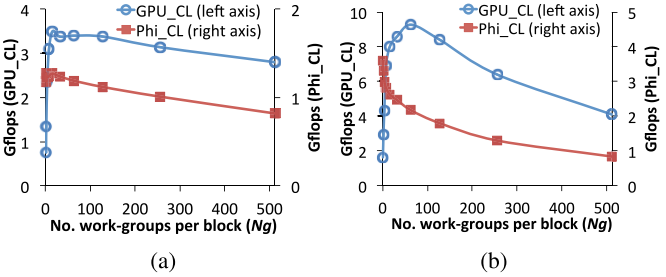


Fig. 12. Performance of GPU_CL and Phi_CL by varying the number of work-groups per block (N_g) for (a) stanford and (b) connectus.

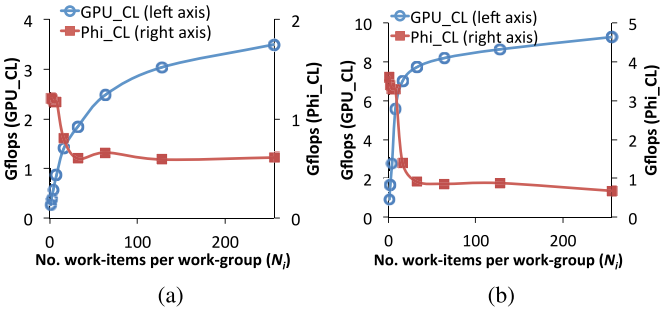


Fig. 13. Performance of GPU_CL and Phi_CL by varying the number of work-items per work-group (N_i) for (a) stanford and (b) connectus.

performance trends on both the GPU and Xeon Phi vary significantly. For instance, as shown in Fig. 11, the optimal N_b values are 8 for both matrices for the *GPU_CL* case. However, the optimal values are 236 for both matrices when executing on the Xeon Phi. Note that the optimal value of N_g for *Phi_CL* are 1 for both matrices (see Fig. 12), which implies that each block is handled by one work-group (essentially by one hardware thread on the Xeon Phi). On the GPU, work-groups are further divided into wavefronts. A large number of wavefronts can help to overlap the computation and memory operations and reduce memory stall. Thus, more work-groups are required for the GPU compared to Xeon Phi.

Second, we notice that the optimal values for these parameters vary for different matrices on the same platform. For instance, Fig. 10 shows that the optimal value of N_p for Stanford and connectus are 1 and 4 on the Xeon Phi. Different values of N_p can result in different performance as the number of panels affects the L2 cache locality, especially when the average nonzeros per row are large.

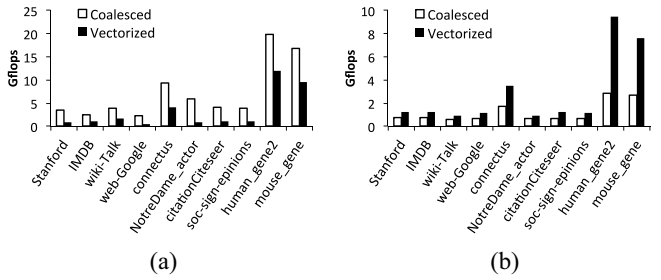


Fig. 14. Performance of (a) GPU_CL and (b) Phi_CL with coalesced and vectorized mapping.

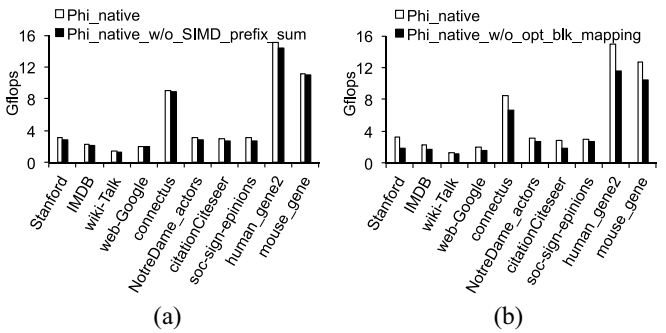


Fig. 15. Performance impact of (a) SIMD segmented prefix and (b) optimized block mapping.

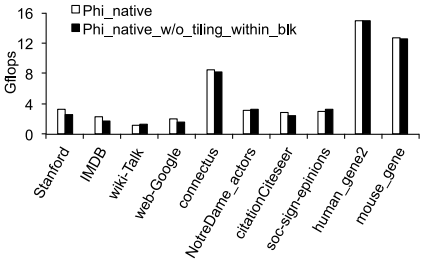


Fig. 16. Performance impact of intrablock tiling.

Third, we can also observe that certain parameters are dependent on the hardware architecture. For instance, for *GPU_CL*, the optimal work-item value (N_i) is 256 which is the maximum number of work-items supported by the GPU. This is expected as the GPU hides memory latency using a large number of work-items. Therefore, on the GPU, as the number of work-items increases, the increased computation workload can better overlap memory accesses, resulting in higher performance. On the other hand, for *Phi_CL*, the optimal N_b and N_i values are always 236 and 1, respectively. This is because first, the OpenCL runtime does map work-groups and work-items directly to hardware threads on Xeon Phi. Second, the maximum number of hardware threads allowable on Xeon Phi is 236.

2) Coalesced Versus Vectorized: As discussed in Section III-B, we designed two ways to map the nonzeros to work-items: coalesced and vectorized mapping. Fig. 14(a) shows that on average the coalesced mapping are 3.3× faster than the vectorized mapping on the GPU. This is because the GPU architecture favors a coalesced memory access pattern since adjacent multiple memory requests can be merged into one memory request and take advantage of the GPU's high memory bandwidth. On the other hand, on the Xeon Phi, Fig. 14(b) shows that the vectorized mapping is on average 2× faster than the coalesced mapping. This is because that Xeon Phi can deliver high memory bandwidth when using the SIMD instructions for memory loads and stores used in the vectorized mapping.

Overall, the above results imply that the same OpenCL implementation exhibits variations for different matrices and architectures. To achieve high performance, performance tuning is essential for different matrices on a specific platform. More importantly, an OpenCL implementation with tunable performance parameters enables both a portable and an efficient implementation on heterogeneous architectures.

C. Results of Hardware Conscious Implementation on Xeon Phi

We next evaluate the performance of *Phi_native* in detail. For each optimization employed by *Phi_native*, we conduct experiments to evaluate its effectiveness. In particular, for each optimization *X*, we compare two implementations—the optimized version with all the optimizations enabled (*Phi_native*), and the implementation with *X* disabled but with all other optimizations enabled (denoted as *Phi_native_w/o_X*). For all the experiments in this section, we use the optimal setting for the tunable parameters (N_p, N_b, and N_t).

1) SIMD Segmented Prefix Sum: To study the impact of this optimization, we replace the SIMD segmented prefix sum computation with standard C implementation for the same functionality (denoted as w/o SIMD prefix-sum), and compare its performance with *Phi_native*. Fig. 15(a) shows the difference with and without this optimization. On average, *Phi_native* achieves 1.2X speedup. The SIMD segmented prefix sum optimization accelerates performance by improving the vectorization efficiency.

2) Locality-Aware Block Mapping: To study the impact of this optimization, we conducted an experiment to simulate random block mapping (e.g., in a vendor-specific OpenCL implementation). Fig. 15(b) shows that locality-aware block mapping is able to give up to 1.76X speedup compared to random block mapping. The optimized block mapping arranges the blocks in column-major order. By doing so, consecutive blocks are mapped to the same core, and this allows exploitation of the locality of input vector *x*. On the other hand, a random block mapping will fail to exploit this locality.

3) Intrablock Tiling: Fig. 16 compares the performance when intrablock tiling is disabled. On average, intrablock tiling achieves 1.1X performance speedup.

D. End-to-End Performance Comparison

1) Hardware Oblivious Implementation: We first compare our OpenCL-based hardware oblivious version with ViennaCL [12], which is the state-of-the-art OpenCL-based

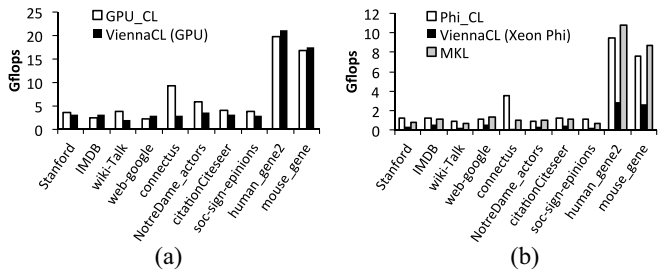


Fig. 17. Performance comparison between OpenCL-based implementations and ViennaCL on the GPU and Xeon Phi. (a) GPU_CL versus ViennaCL. (b) Phi_CL versus ViennaCL versus MKL.

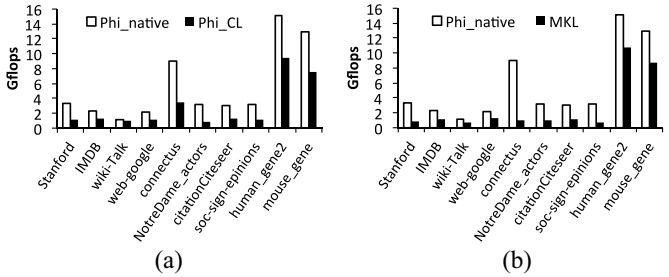


Fig. 18. (a) Performance comparison for Phi_native, Phi_CL, and MKL. (a) Phi_native versus Phi_CL. (b) Phi_native versus MKL.

SpMV implementation. ViennaCL uses an adaptive CSR format for SpMV [13]. Fig. 17(a) shows the comparison of *GPU_CL* and *ViennaCL* on the GPU. For most of the matrices, *GPU_CL* and *ViennaCL* achieve similar performances. However, due to the additional book-keeping data structures such as the bit field array *eor*, *ViennaCL* performs slightly better for a few matrices. On the other hand, *GPU_CL* outperforms ViennaCL for matrices such as *wiki-Talk* and *connectus*. Fig. 17(b) shows the performance comparison of *Phi_CL* and *ViennaCL* on the Xeon Phi. On average, *Phi_CL* achieves 4.9X speedup over *ViennaCL*. The speedup arises from the fact that *Phi_CL* adopts the vectorized mapping on Xeon Phi but *ViennaCL* uses a mapping that is similar to coalesced mapping. Vectorized mapping can better utilize the VPUs on Xeon Phi. Overall, *GPU_CL* has better portability compared to *ViennaCL*. This is because *GPU_CL* is designed with tunable parameters including parallelization granularity and nonzeros mapping schemes, which allows the OpenCL model to adjust to different architectures.

We also compare *Phi_CL* with MKL on Xeon Phi. The results are shown in Fig. 17(b). As shown, *Phi_CL* achieves a performance that is comparable to MKL.

2) Hardware Conscious Implementation: Fig. 18(a) shows a comparison of the performance between *Phi_native* and *Phi_CL*. As of the low-level optimizations, *Phi_native* achieves 2.2X speedup on average relative to *Phi_CL*. When compared against MKL, *Phi_native* achieves on average 3.1X speedup as shown in Fig. 18(b). The speedup numbers for the different matrices are shown in Table V.

The good performance of *Phi_native* comes from the improved resource utilization on Xeon Phi. Fig. 19(a) shows

TABLE V
SPEEDUPS ACHIEVED BY PHI_NATIVE RELATIVE TO PHI_CL AND MKL

Matrix	Phi_native vs Phi_CL	Phi_native vs MKL
Stanford	2.7	4.0
IMDB	1.9	2.1
wiki-Talk	1.3	1.7
web-Google	1.8	1.6
connectus	2.6	8.9
NotreDame_actors	3.4	3.0
citationCiteseer	2.4	2.7
soc-sign-epinions	2.8	4.4
human_gene2	1.6	1.4
mouse_gene	1.7	1.5

that the CPI of *Phi_native* has significantly improved compared with MKL. For most of the matrices, they are close to the ideal CPI value on Xeon Phi. Fig. 19(b) shows that the VI has been improved by about 38%. This improvement is attributed to the vectorized mapping as well as the SIMD segmented prefix sum.

Fig. 19(c) and (d) shows that the cache efficiency is improved as well. The L1 hit rate of *Phi_native* has significantly increased from 60% (MKL) to 90% on average. Similarly, the ELI values of *Phi_native* are generally lower than the prescribed threshold of 145, which likely indicates better L2 cache performance. As discussed earlier, because ELI is a derived metric with the number of L1 misses as its denominator, the value could be artificially inflated such as in the case of human_gene2 and mouse_gene where the L1 miss rate is very low. Nevertheless, we observe in Fig. 18(b) that the performance of these matrices under *Phi_native* has improved compared to MKL. This observation is supported by the improvement in the CPI, VI, and L1 hit rate metrics as shown in Fig. 19. The L2 cache performance is also likely to be improved for these matrices although this is not reflected by the ELI metric.

Overall, improvement in cache efficiency is attributed to our 2-D jagged partitioning scheme, optimized mapping scheme from blocks to hardware threads, as well as tiling.

E. Discussion

Our hardware oblivious implementation based on OpenCL achieves both efficiency and portability. On each tested platform, it delivers high performance. Specifically, on the GPU it achieves comparable performance to *ViennaCL*; on Xeon Phi it achieves comparable performance to MKL. Moreover, our OpenCL implementation maintains the high performance when porting to different architectures. This is attributed to the tunable performance parameters, which allow the OpenCL implementation to adapt to different architectures. In contrast, *ViennaCL* which is portable to different architectures gives high performance on the GPU, but performs badly on the Xeon Phi.

While the OpenCL implementation provides portability, it prevents us from utilizing the low-level architectural features. Thus, we further use Xeon Phi as a case study and design

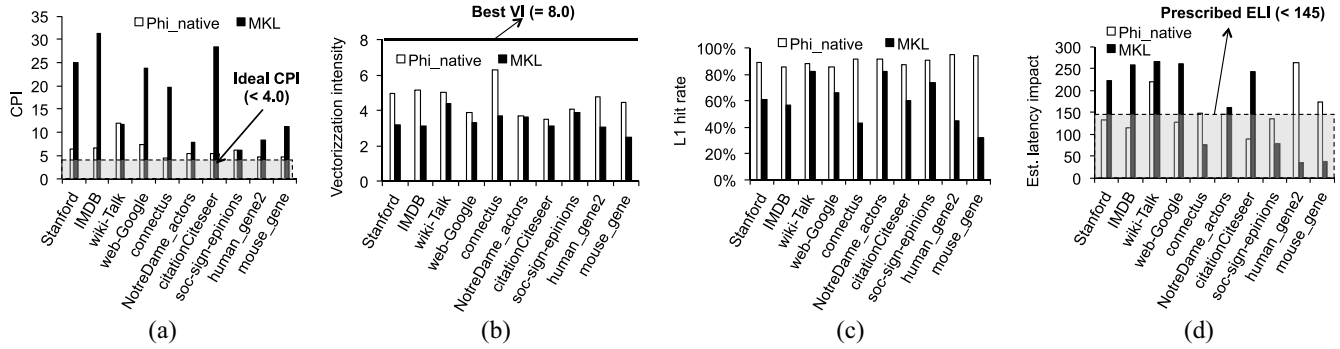


Fig. 19. Performance profiling of Phi_native and MKL. (a) CPI. (b) VI. (c) L1 cache hit rate. (d) ELI.

TABLE VI
STORAGE OVERHEADS OF HCC COMPARED WITH COO AND CSR

Matrix	HCC vs COO (in %)	HCC vs CSR (in %)
Stanford	78.9	101.1
IMDB	78.3	100.6
wiki-Talk	88.9	102.3
web-Google	81.7	102.7
connectus	78.7	104.9
NotreDame_actors	84.4	103.3
citationCiteseer	79.2	101.7
soc-sign-epinions	79.7	100.9
human_gene2	75.3	100.3
mouse_gene	75.6	100.7

a hardware conscious implementation using the native programming model. Experiments show that hardware conscious implementation *Phi_native* achieves 2.2X speedup on average compared to the hardware oblivious implementation *Phi_CL*. Hence, there exists a tradeoff between portability and efficiency for the SpMV kernel. OpenCL-based implementation offers portability and good performance. However, to target higher performance, hardware conscious implementation for the specific platform is still necessary.

Finally, we take a look at the storage overheads required by our new format. Table VI shows the amount of storage space required by HCC compared to the standard COO and CSR formats, respectively. On average, we find that HCC takes up about 80% of the space required by COO due to savings in not having to store all the row indices. Compared to CSR, HCC takes up on average about 102% of the storage space required by CSR, i.e., an extra 2% overhead which is not significant compared to the performance advantage discussed earlier.

VI. RELATED WORK

A. Performance Optimization for Many Core Architecture

In the recent years, we have witnessed the success of many core architecture including GPUs and Intel Xeon Phi. However, tuning many core architecture for high performance was not a trivial task. Analytical performance models have been proposed to predict the performance improvement or identify the performance bottlenecks [14], [15]. Thread structures (e.g., the number of workitems and workgroups) are the first-order parameters that affect the performance. Yang *et al.* [16] proposed to merge workitems and workgroups based on memory reuse. Kayiran *et al.* [17] observed that running with the maximum number of workgroups does not always guarantee the optimal performance due to resource contention such as caches. Therefore, thread throttling techniques has been combined with cache bypassing to reduce the cache contention [18]. The state-of-the-art performance optimization techniques also focused on thread and warp scheduling, cache optimization, register allocation optimization, multitasking, control divergence, etc. [19]–[27].

B. SpMV Optimization on Many Core Architecture

As accelerators such as the GPU and Xeon Phi have also become important in high performance computing, there has recently been much work on optimizing SpMV on these newer platforms. Bell and Garland [28] developed efficient implementations of SpMV computation for the COO, CSR, and ELLPACK formats. Further developments sought to improve SpMV performance based on the ELLPACK format, for instance ELLR-T assigns multiple threads for each row of a matrix [29], sliced-ELLPACK reorders a matrix and paritions the rows into slices of similar lengths [30], and blocked-ELLPACK which adds block optimizations to ELLPACK [31]. Other implementations such as yaSpMV [32] extends COO and uses blocked compression to improve performance, whereas Greathouse and Daga [13], [33] proposed a novel CSR scheme that is adaptive. Ashari *et al.* [34] proposed an adaptive SpMV algorithm based on the standard CSR format but reduced thread divergence by combining similar rows into groups. Su and Keutzer [35] developed a *Cocktail* representation that partitions a matrix into submatrices and uses a different format for each submatrix. For the Xeon Phi many-core coprocessor, an early study by Saule *et al.* [36] showed that register blocking did not work as well on Xeon Phi, whereas Liu *et al.* [37] developed a format called ESB which is based on the ELLPACK format. Kreutzer *et al.* [38] proposed to use a variant of sliced ELLPACK as an SIMD-friendly data format. The format combines long-standing ideas from general-purpose graphics processing units and vector computer programming.

C. Portability Optimization for Many Core Architecture

Cross-platform portability is becoming an important issue due to the presence of different many-core architectures.

For instance, Calore *et al.* [39] implemented a portable OpenCL Lattice-Boltzmann application for GPU and Xeon Phi many-core platforms. Banaś and Krużel [40] studied the performance portability of finite-element numerical integration codes on the GPU and Xeon Phi. O'Boyle *et al.* [41] presented a compiler framework to automatically generate optimized OpenCL code from data-parallel OpenMP programs for GPUs. On SpMV, Liu and Vinter [42] proposed the CSR5 format and provided native implementations on each of the platforms. Like these work, ours is also focused on cross-platform portability. However, unlike their work, we looked in detail into SpMV computation for scale-free matrices, as well as studied the tradeoffs between performance and portability.

Many of the prior studies did not focus on SpMV for scale-free matrices, unlike this paper which is concerned with optimizing SpMV for scale-free matrices that arise from and are important in many studies on networks [7]. For these matrices, we developed a hardware oblivious implementation using OpenCL and a hardware conscious implementation on Intel Xeon Phi.

VII. Conclusion

The performance and portability of SpMV plays an important role for many different applications. In this paper, we focus on SpMV implementation for scale-free matrices on heterogeneous many-core architectures. We propose two implementations with tradeoffs in efficiency and portability. One is a hardware oblivious implementation based on OpenCL. This implementation is designed with a novel HCC format and performance tuning parameters. Another one is a hardware conscious implementation using the native programming language on Xeon Phi. This implementation employs low-level architecture-aware optimizations. Experiments using a wide range of representative scale-free matrices demonstrate that our hardware oblivious implementation based on OpenCL achieves comparable performance to the Intel MKL on Xeon Phi and state-of-the-art OpenCL-based ViennaCL library on GPU. Our OpenCL implementation maintains good performance across GPU and Xeon Phi platforms. In comparison, our hardware conscious implementation further improves the performance by 2.2X on Xeon Phi. Compared with MKL, the hardware conscious implementation achieves 3.3X speedup on Xeon Phi. Hence, we conclude that OpenCL model offers good performance and portability for heterogeneous systems, but hardware conscious optimization on specific platform is still necessary for high performance.

References

[1] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed. Philadelphia, PA, USA: Soc. Ind. Appl. Math., 2003.

[2] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank citation ranking: Bringing order to the Web," Stanford Digit. Library, Tech. Rep. SIDL-WP-1999-0120, 1999.

[3] E.-J. Im, "Optimizing the performance of sparse matrix-vector multiplication," Ph.D. dissertation, Dept. Comput. Sci., Univ. California at Berkeley, Berkeley, CA, USA, 2000.

[4] J. Willcock and A. Lumsdaine, "Accelerating sparse matrix computations via data compression," in *Proc. 20th Annu. Int. Conf. Supercomput. (ICS)*, Cairns, QLD, Australia, 2006, pp. 307–316.

[5] S. Williams *et al.*, "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," in *Proc. ACM/IEEE Conf. Supercomput. (SC)*, Reno, NV, USA, 2007, pp. 1–12.

[6] A.-L. Barabási and R. Albert, "Emergence of scaling in random networks," *Science*, vol. 286, no. 5439, pp. 509–512, Oct. 1999.

[7] W. T. Tang *et al.*, "Optimizing and auto-tuning scale-free sparse matrix-vector multiplication on Intel Xeon Phi," in *Proc. 13th Annu. IEEE/ACM Int. Symp. Code Gener. Optim. (CGO)*, San Francisco, CA, USA, 2015, pp. 136–145.

[8] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in *Proc. SIAM Int. Conf. Data Min.*, Lake Buena Vista, FL, USA, 2004, pp. 442–446.

[9] *Intel Math Kernel Library*. [Online]. Available: https://software.intel.com/en-us/intel-mkl

[10] J. Jeffers and J. Reinders, *Intel Xeon Phi Coprocessor High-Performance Programming*. Amsterdam, The Netherlands: Morgan Kaufmann, 2013.

[11] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1–25, Nov. 2011. [Online]. Available: http://www.cise.ufl.edu/research/sparse/matrices/

[12] *ViennaCL—Linear Algebra Library Using CUDA OpenCL and OpenMP*. [Online]. Available: http://viennacl.sourceforge.net/

[13] J. L. Greathouse and M. Daga, "Efficient sparse matrix-vector multiplication on GPUs using the CSR storage format," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal. (SC)*, New Orleans, LA, USA, 2014, pp. 769–780.

[14] S. Hong and H. Kim, "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness," in *Proc. 36th Annu. Int. Symp. Comput. Archit. (ISCA)*, Austin, TX, USA, 2009, pp. 152–163.

[15] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-M. W. Hwu, "An adaptive performance modeling tool for GPU architectures," in *Proc. 15th ACM SIGPLAN Symp. Principles Pract. Parallel Program. (PPOPP)*, Bengaluru, India, 2010, pp. 105–114.

[16] Y. Yang, P. Xiang, J. Kong, and H. Zhou, "A GPGPU compiler for memory optimization and parallelism management," in *Proc. 31st ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, Toronto, ON, Canada, 2010, pp. 86–97.

[17] O. Kayiran, A. Jog, M. T. Kandemir, and C. R. Das, "Neither more nor less: Optimizing thread-level parallelism for GPGPUs," in *Proc. 22nd Int. Conf. Parallel Archit. Compilation Tech. (PACT)*, Edinburgh, U.K., 2013, pp. 157–166.

[18] X. Chen *et al.*, "Adaptive cache management for energy-efficient GPU computing," in *Proc. 47th Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, Cambridge, U.K., 2014, pp. 343–355.

[19] V. Narasiman *et al.*, "Improving GPU performance via large warps and two-level warp scheduling," in *Proc. 44th Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, Porto Alegre, Brazil, 2011, pp. 308–317.

[20] X. Xie, Y. Liang, G. Sun, and D. Chen, "An efficient compiler framework for cache bypassing on GPUs," in *Proc. Int. Conf. Comput.-Aided Design (ICCAD)*, San Jose, CA, USA, 2013, pp. 516–523.

[21] X. Xie, Y. Liang, Y. Wang, G. Sun, and T. Wang, "Coordinated static and dynamic cache bypassing for GPUs," in *Proc. IEEE 21st Int. Symp. High Perform. Comput. Archit. (HPCA)*, Burlingame, CA, USA, 2015, pp. 76–88.

[22] Y. Liang, X. Xie, G. Sun, and D. Chen, "An efficient compiler framework for cache bypassing on GPUs," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 34, no. 10, pp. 1677–1690, Oct. 2015.

[23] A. Jog *et al.*, "Orchestrated scheduling and prefetching for GPGPUs," in *Proc. 40th Annu. Int. Symp. Comput. Archit. (ISCA)*, Tel Aviv-Yafo, Israel, 2013, pp. 332–343.

[24] X. Xie *et al.*, "Enabling coordinated register allocation and thread-level parallelism optimization for GPUs," in *Proc. 48th Int. Symp. Microarchit. (MICRO)*, Waikiki, HI, USA, 2015, pp. 395–406.

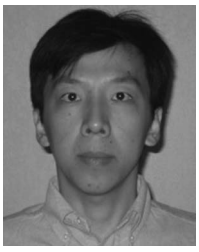
[25] M. Gebhart *et al.*, "Energy-efficient mechanisms for managing thread context in throughput processors," in *Proc. 38th Annu. Int. Symp. Comput. Archit. (ISCA)*, San Jose, CA, USA, 2011, pp. 235–246.

[26] Y. Liang, H. P. Huynh, K. Rupnow, R. S. M. Goh, and D. Chen, "Efficient GPU spatial-temporal multitasking," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 3, pp. 748–760, Mar. 2015.

[27] Y. Liang, M. T. Satria, K. Rupnow, and D. Chen, "An accurate GPU performance model for effective control flow divergence optimization," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 35, no. 7, pp. 1165–1178, Jul. 2016.

[28] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proc. Conf. High Perform. Comput. Netw. Storage Anal. (SC)*, Portland, OR, USA, 2009, pp. 1–11.

[29] F. Vázquez, J. J. Fernández, and E. M. Garzón, "Automatic tuning of the sparse matrix vector product on GPUs based on the ELLR-T approach," *Parallel Comput.*, vol. 38, no. 8, pp. 408–420, Aug. 2012.
[30] A. Monakov, A. Lokhmotov, and A. Avetisyan, "Automatically tuning sparse matrix-vector multiplication for GPU architectures," in *Proc. 5th Int. Conf. High Perform. Embedded Archit. Compilers (HiPEAC)*, Pisa, Italy, 2010, pp. 111–125.
[31] J. W. Choi, A. Singh, and R. W. Vuduc, "Model-driven autotuning of sparse matrix-vector multiply on GPUs," in *Proc. 15th ACM SIGPLAN Symp. Principles Pract. Parallel Program. (PPOPP)*, Bengaluru, India, 2010, pp. 115–126.
[32] S. Yan, C. Li, Y. Zhang, and H. Zhou, "yaSpMV: Yet another SpMV framework on GPUs," *ACM SIGPLAN Notices*, vol. 49, no. 8, pp. 107–118, Aug. 2014.
[33] M. Daga and J. L. Greathouse, "Structural agnostic SpMV: Adapting CSR-adaptive for irregular matrices," in *Proc. IEEE 22nd Int. Conf. High Perform. Comput. (HiPC)*, Bengaluru, India, 2015, pp. 64–74.
[34] A. Ashari, N. Sedaghati, J. Eisenlohr, S. Parthasarathy, and P. Sadayappan, "Fast sparse matrix-vector multiplication on GPUs for graph applications," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal. (SC)*, New Orleans, LA, USA, 2014, pp. 781–792.
[35] B.-Y. Su and K. Keutzer, "clSpMV: A cross-platform OpenCL SpMV framework on GPUs," in *Proc. 26th ACM Int. Conf. Supercomput. (ICS)*, Venice, Italy, 2012, pp. 353–364.
[36] E. Saule, K. Kaya, and Ü. V. Çatalyürek, "Performance evaluation of sparse matrix multiplication kernels on Intel Xeon Phi," in *Parallel Processing and Applied Mathematics* (Lecture Notes in Computer Science). Heidelberg, Germany: Springer, 2014, pp. 559–570.
[37] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey, "Efficient sparse matrix-vector multiplication on x86-based many-core processors," in *Proc. 27th Int. Conf. Supercomput. (ICS)*, Eugene, OR, USA, 2013, pp. 273–282.
[38] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop, "A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide SIMD units," *SIAM J. Sci. Comput.*, vol. 36, no. 5, pp. C401–C423, 2014.
[39] E. Calore, S. F. Schifano, and R. Tripiccione, "A portable OpenCL lattice Boltzmann code for multi-and many-core processor architectures," *Proc. Comput. Sci.*, vol. 29, pp. 40–49, Dec. 2014.
[40] K. Banaś and F. Krużel, "OpenCL performance portability for Xeon Phi coprocessor and NVIDIA GPUs: A case study of finite element numerical integration," in *Proc. Euro Par Parallel Process. Workshops*, Porto, Portugal, 2014, pp. 158–169.
[41] M. F. P. O'Boyle, Z. Wang, and D. Grewe, "Portable mapping of data parallel programs to OpenCL for heterogeneous systems," in *Proc. IEEE/ACM Int. Symp. Code Gener. Optim. (CGO)*, Shenzhen, China, 2013, pp. 1–10.
[42] W. Liu and B. Vinter, "CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication," in *Proc. 29th ACM Int. Conf. Supercomput. (ICS)*, Irvine, CA, USA, 2015, pp. 339–350.



Yun Liang (M'10) received the B.S. degree in software engineering from Tongji University, Shanghai, China, and the Ph.D. degree in computer science from the National University of Singapore, Singapore, in 2004 and 2010, respectively.

He was a Research Scientist with the University of Illinois Urbana-Champaign, Urbana, IL, USA, from 2010 to 2012. He has been an Assistant Professor with the School of Electronics Engineering and Computer Science, Peking University, Beijing, China, since 2012. His current research interests include heterogeneous computing, embedded system, and high level synthesis.

Dr. Liang was a recipient of the Best Paper Award in International Symposium on Field-Programmable Custom Computing Machines (FCCM')11 and the Best Paper Award Nominations in CODES+ISSS'08, FPT'11, DAC'12, and ASPDAC'16. He serves as a Technical Committee Member for Asia South Pacific Design Automation Conference, Design Automation and Test in Europe, International Conference on Compilers Architecture and Synthesis for Embedded System, International Conference on Computer Aided Design, and International Conference on Parallel Architectures and Compilation Techniques.

Wai Teng Tang received the bachelor's (with Hons.) degree in electrical engineering, and the Ph.D. degree in bioimaging from the National University of Singapore, Singapore.

He is currently a Research Scientist with the Institute of High Performance Computing, Agency for Science, Technology and Research, Singapore. His current research interests include modeling and simulation and big data analytics.

Ruizhe Zhao is currently pursuing the undergraduation degree with the School of EECS, Peking University, Beijing, China.

His current research interests include computer architecture and FPGA technology.

Mian Lu received the bachelor's degree in software engineering from the Huazhong University of Science and Technology, Wuhan, China, in 2007, and the Ph.D. degree in computer science from the Hong Kong University of Science and Technology, Hong Kong, in 2012.

He is currently a Scientist with the Institute of High Performance Computing, A*STAR, Singapore. His current research interests include embedded system, high performance computing, and big data analytics.

Huynh Phung Huynh received the Ph.D. degree in computer science from the National University of Singapore, Singapore, in 2010.

His current research interests include embedded system and high performance computing research such as developing productivity tools for GPU/many-core computing and big data analytics.

Rick Siow Mong Goh received the Ph.D. degree in electrical and computer engineering from the National University of Singapore, Singapore.

He is the Director of the Computing Science Department, A*STAR Institute of High Performance Computing, Singapore, where he leads a team of over 70 scientists in performing world-leading scientific research, developing technologies to commercialization, and engaging and collaborating with industry. His current research interests include high performance computing, distributed computing, data analytics, interactive interaction technologies, computational social cognition, discrete event simulation, parallel and distributed computing, and performance optimization and tuning of applications on large-scale computing platforms.