Optimizing and Auto-Tuning Scale-Free Sparse Matrix-Vector Multiplication on Intel Xeon Phi

Wai Teng Tang[†], Ruizhe Zhao[§], Mian Lu[†], Yun Liang[§], Huynh Phung Huyng[†], Xibai Li[§], Rick Siow Mong Goh[†]

[†]Institute of High Performance Computing, Agency for Science, Technology and Research, Singapore
[§]Center for Energy-Efficient Computing and Applications, School of EECS, Peking University, China

Abstract

Recently, the Intel Xeon Phi coprocessor has received increasing attention in high performance computing due to its simple programming model and highly parallel architecture. In this paper, we implement sparse matrix vector multiplication (SpMV) for scale-free matrices on the Xeon Phi architecture and optimize its performance. Scale-free sparse matrices are widely used in various application domains, such as in the study of social networks, gene networks and web graphs. We propose a novel SpMV format called vectorized hybrid COO+CSR (VHCC). Our SpMV implementation employs 2D jagged partitioning, tiling and vectorized prefix sum computations to improve hardware resource utilization, and thus overall performance. As the achieved performance depends on the number of vertical panels, we also develop a performance tuning method to guide its selection. Experimental results demonstrate that our SpMV implementation achieves an average 3× speedup over Intel MKL for a wide range of scale-free matrices.

1. Introduction

Sparse matrix-vector multiplication (SpMV) is a critical kernel that finds applications in many high performance computing (HPC) domains including structural mechanics, fluid dynamics, social network analysis and data mining. Many algorithms use SpMV iteratively for their computation. For example, the conjugate gradient method [16] uses SpMV to solve a system of linear equations, whereas the PageRank algorithm [14] uses SpMV to determine the ranks of web pages. SpMV computation is a performance bottleneck for many of these algorithms [10, 20, 21]. 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 the Xeon Phi architecture. Scale-free sparse matrices arise in many practical applications, such as in the study of web links, social networks and transportation networks [2]. Unlike sparse matrices from engineering applications, which are more regular in nature (i.e. the number of non-zeros in each row is similar), a sparse matrix that exhibits scale-free properties is highly irregular. It has many rows with very few non-zeros but has only a few rows with a large number of non-zeros. As such, SpMV computation on such matrices is particularly challenging due to the highly irregular distribution of non-zeros. Many existing implementations such as Intel MKL perform well for regular matrices, but are inefficient for scale-free sparse matrices. Previous works have also studied partitioning algorithms for scale-free SpMV computation on distributed memory computers [6, 13]. However, such partitioning schemes are expensive and do not scale well for applications that require online analysis, e.g. dynamically changing web graphs. More importantly, these implementations are not designed for the Intel Xeon Phi architecture.

In this work, we present an efficient scale-free SpMV implementation named VHCC for Intel Xeon Phi. VHCC makes use of a vector format that is designed for efficient vector processing and load balancing. Furthermore, we employ a 2D jagged partitioning method together with tiling in order to improve the cache locality and reduce the overhead of expensive gather and scatter operations. We also employ efficient prefix sum computations using SIMD and masked operations that are specially supported by the Xeon Phi hardware. The optimal panel number in the 2D jagged partitioning method varies for different matrices due to their differences in non-zero distribution. Therefore, we develop a performance tuning technique to guide its selection. Experiments indicate that our SpMV implementation achieves an average 3× speedup over Intel MKL for scale-free matrices, and the performance tuning method achieves within 10% of the optimal configuration.

The remainder of the paper is organized as follows. In Section 2, we present background details on Intel Xeon Phi and scale-free sparse matrices. We also discuss the bottlenecks that these matrices face during SpMV computation. Section 3 presents our implementation for computing scale-free SpMV on Xeon Phi, as well as our proposed performance tuning method. Experiment results are then presented

1-4799-8161-8/15/$31.00 ©2015 IEEE

in Section 4. Section 5 discusses prior research which is related to ours, and Section 6 concludes the paper.

2. Background and Motivation

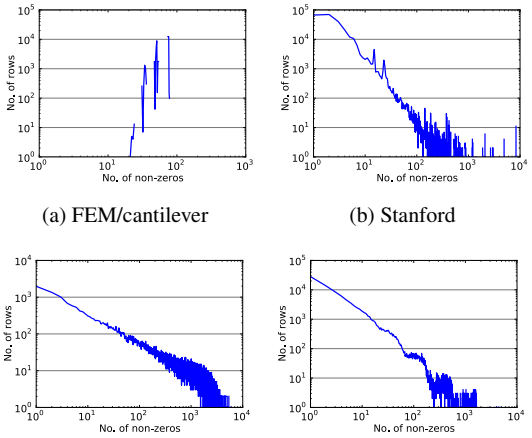
2.1 The Intel Xeon Phi Coprocessor

The Intel Xeon Phi coprocessor is based on the Intel Many Integrated Core (MIC) Architecture. In this paper, we use the Xeon Phi 5110P coprocessor which integrates 60 cores on the same package, each running at 1.05GHz. Up to 4 hardware threads per core are supported, and a maximum limit of 240 threads can be scheduled on the coprocessor. The MIMD (Multiple Instructions, Multiple Data) execution model allows different workloads to be assigned to different threads. The amount of off-chip memory that is available on the coprocessor is 8GB. There are 8 memory controllers which support 16 GDDR5 channels, and altogether they operate at a peak theoretical bandwidth of 320GB/s. Every core in the processor contains a local 64KB L1 cache, equally divided between the instruction and data caches, and a 512KB L2 unified cache. All the 512KB L2 caches on the 60 cores are fully coherent via the tag directories, and are interconnected 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. L2 cache miss penalty is on the order of hundreds of cycles, thus optimizing the data locality for L2 cache is important for Xeon Phi.

One of the major features of Xeon Phi is the wide 512-bit vector processing unit (VPU) present on each of the cores, effectively doubling the 256-bit vector width of the latest Intel Xeon CPUs. In addition, new SIMD (Single Instruction, Multiple Data) instructions including scatter and gather, swizzle, maskable operations, and *fused multiply-add* (FMA) operations are supported. To achieve high performance on Xeon Phi, it is crucial to utilize the VPUs effectively. The Xeon Phi is capable of yielding peak double precision performance of ∼1TFlops. Compared to alternative coprocessor architectures such as that of GPUs, Xeon Phi features low cost atomic operations that can be used for efficient parallel algorithm implementations.

2.2 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 [2]. Figure 1 shows the differences between regular and scale-free matrices in terms of the distribution of the non-zeros per row of a matrix. The plots are logarithmic in both axes; the horizontal axis denotes the number of non-zeros per row and the vertical axis denotes the number of rows having that specified number of non-zeros. Figure 1a demonstrates a matrix from the engineering sciences (e.g. constructed using FEM). Such matrices tend to have multi-modal 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. 1b) and networks (see Fig. 1c) tend to exhibit scale-free properties. These matrices are termed "irregular matrices" or "scale-free matrices". As a comparison, Fig. 1d shows the distribution of the non-zeros per row of a scale-free sparse matrix generated from a Kronecker graph model [7]. 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 non-zeros per row (e) is obtained. One can easily see that Fig. 1b and Fig. 1c look very similar to Fig. 1d, whereas there is no resemblance at all between the regular matrix in Fig. 1a and the R-MAT in Fig. 1d.



(a) FEM/cantilever (b) Stanford

(c) mouse_gene (d) R-MAT(18, 16)

Figure 1: Comparison of regular and irregular (scale-free) matrices. x-axis: non-zeros per row, y-axis: frequency.

2.3 Performance Bottlenecks of Intel MKL's SpMV

The *coordinate* (COO) and *compressed sparse row* (CSR) are two commonly used SpMV formats that are provided by Intel MKL [1]. COO stores both the row and column indices of all the non-zeros. On the other hand, CSR does not store the row indices; instead, it stores a pointer to the start of each row. Because of this, CSR uses less memory bandwidth and thus performs better than COO. In this paper, we will

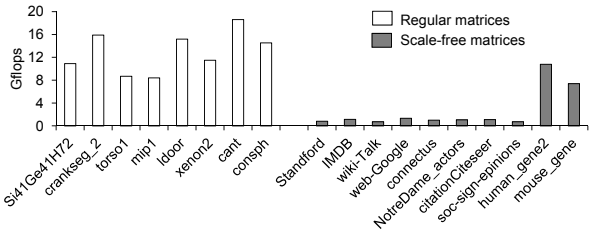


Figure 2: Performance of MKL CSR SpMV for regular and scale-free matrices.

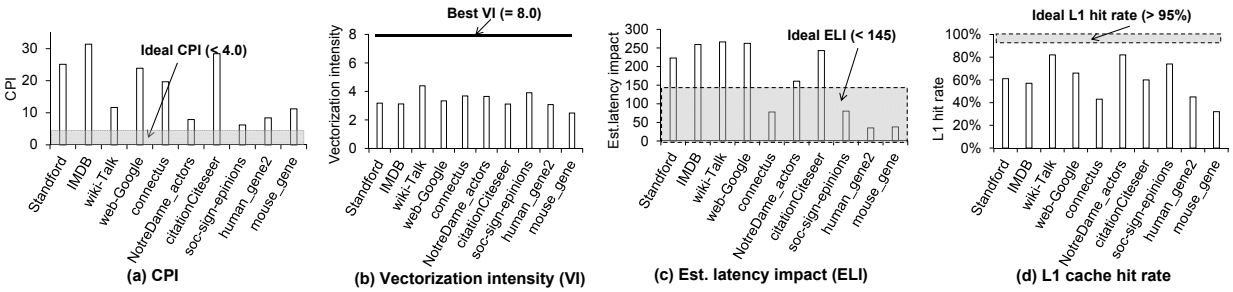


Figure 3: Performance of MKL: (a) CPI (b) vectorization intensity (c) estimated latency impact (d) L1 cache hit rate.

use Intel MKL's CSR format (denoted as MKL) for all our performance comparisons.

Figure 2 compares the performance of MKL for both regular and the scale-free matrices. 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. 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. In spite of this, we will show later that a higher performance can be achieved for these two matrices using our VHCC implementation.

We use the Intel VTune profiler [11] to analyze the performance bottlenecks of the MKL implementation in details. We first collect the average clocks per instruction (CPI) for each of the matrices as shown in Fig. 3a. The ideal CPI is 4 [11]. 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 [11]. We examine the vector utilization efficiency of MKL in Fig. 3b. The vectorization intensity metric is defined as the average number of active elements per VPU instruction executed. The maximum value for vectorization intensity 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.

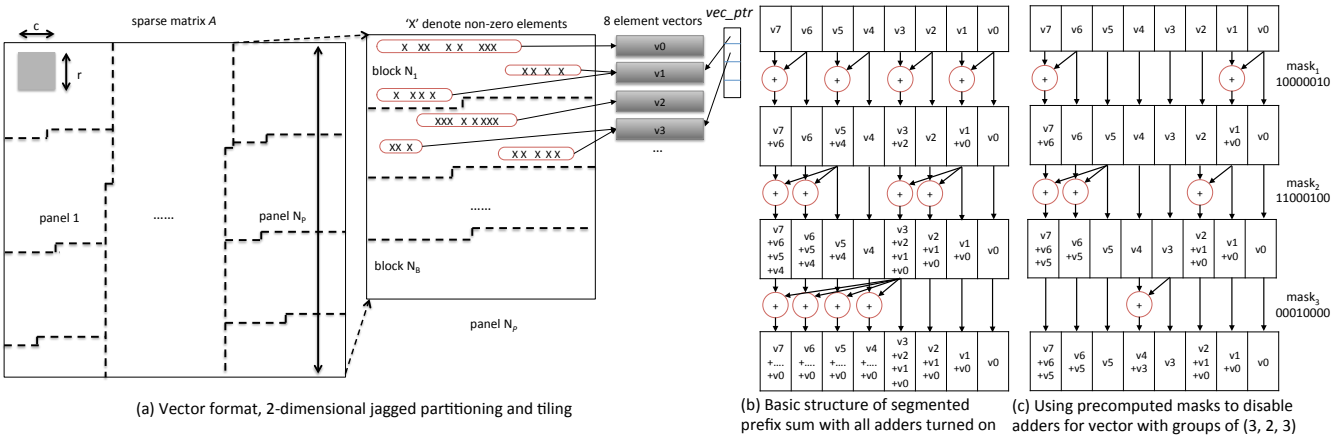
Furthermore, because SpMV is inherently memory bound, its overall performance critically depends on the performance of the memory hierarchy. Therefore, we also investigate the average latency of memory accesses. The profiler does not give the average latency per memory access directly, but provides the estimated latency impact (ELI) metric. ELI provides a rough estimate of the average miss penalty (in cycles) per L1 cache miss [11], and can be used to gauge L2 cache performance. Figure 3c shows that for most matrices, the ELI is much higher than the ideal value (145) recommended by Intel. This indicates that the L2 cache hit ratio is low, resulting in long delays to fetch data from main memory instead of from cache. Similarly, in Fig. 3d, we observe that the L1 cache hit rates are generally lower than the ideal value. In summary, MKL suffers from low vectorization intensity and poor cache performance for scale-free matrices. In the next section, we will describe an implementation that remedies these problems.

3. SpMV Implementation

We provide details of our scale-free SpMV implementation ($y = y + A \cdot x$) on Intel Xeon Phi. We first devise a new vector format by grouping non-zeros in a sparse matrix into vectors of 8 elements. This enables efficient vector processing and load balancing. Then, we use 2D jagged partitioning and tiling to improve cache locality. We also employ an efficient prefix sum operation for computation and implement it using SIMD instructions. Finally, we develop a performance auto-tuning method to guide the selection of panel number.

3.1 Vector Format, 2D Jagged Partitioning and Tiling

The vector processing unit (VPU) is a key architectural feature of the Intel Xeon Phi coprocessor. To effectively utilize this resource, we devise a vector format, VHCC (*Vectorized Hybrid COO+CSR*), which groups non-zeros together so that the VPU can be efficiently used. As we will see in the experiments later, this improves vectorization intensity by reducing the empty slots in a vector operation. As shown in Fig. 4a, VHCC first arranges the non-zeros contiguously and then equally divides them among a number of vertical panels. The non-zeros in each vertical panel are then further partitioned equally into a number of blocks that are layout vertically in that panel. This 2D jagged partitioning scheme effectively partitions a sparse matrix into jagged blocks so that the number of non-zeros in each block is the same. We then map each thread in the coprocessor to one block. The non-zero elements in each block are further grouped into vectors of 8 elements. Each element contains a tuple with two data – the double-precision floating-point value of the non-zero, and the column index of the non-zero. We also use a *vec_ptr* array to keep track of the vectors that contain elements across multiple rows, and a *row_idx* array to store the corresponding row indices in order to identify the correct position in the output array to write to. The vector format of VHCC combines the benefits of both COO and CSR for-



(a) Vector format, 2-dimensional jagged partitioning and tiling

(b) Basic structure of segmented prefix sum with all adders turned on

(c) Using precomputed masks to disable adders for vector with groups of (3, 2, 3)

Figure 4: 2D jagged partitioning and the vector format.

mats. On one hand, being similar to COO allows us to partition the workload equally among the threads. On the other hand, being similar to CSR allows us to save storage space as most row indices need not be stored.

The 2D jagged partitioning scheme used by VHCC serves two purposes. First, it ensures a balanced workload by dividing the non-zeros equally among all the threads. Secondly, it helps to improve L2 cache hit rates. A low L2 cache hit ratio will pose a serious challenge for SpMV, and this is especially true for scale-free sparse matrices. The L2 cache miss penalty on Xeon Phi is high as it involves a sequence of requests to the tag directories and memory controllers. It has been shown that the L2 cache miss penalty on Xeon Phi can be an order of magnitude larger than that of multicore CPUs [12]. Therefore, 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 input vector. This helps to reduce the possibility that elements in x are evicted from the cache by 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.

Apart from 2D jagged partitioning, within each block, we also perform tiling in order to improve the L1 cache locality. Tiling is depicted in Fig. 4a by the gray r by c tile. The optimal tile size corresponds to the L1 data cache size of 32KB. Through empirical evaluation, we picked the optimal sizes for r and c (see Section 4.1.1). Altogether, both 2D jagged partitioning and tiling balance the cost of gathering from x and the cost of scattering to y.

3.2 SIMD Segmented Prefix Sum

Our VHCC entails tightly packing the non-zero values into vectors so that they can be efficiently operated on by the vector processing units. Because of the fact that the packed non-zero values may cross row boundaries (i.e. when groups of values are from different rows), we have implemented a SIMD segmented prefix sum operation using Intel Intrinsics [11] to calculate the values that are to be written out to the y vector. Figure 4b shows the basic structure of the segmented prefix sum that is used in our SpMV kernel. This operation can be implemented with 3 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, 3 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 **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), (v_5, v_6, v_7), and each group corresponds to elements from different rows. Figure 4c 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. 4c. 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.

3.3 Putting It All Together

Algorithm 1 presents the detailed implementation of our SpMV kernel. Each thread on Xeon Phi performs computation for a range of vectors specified from *startvec* to *endvec*. The double-precision values (*val_vec*), column indices (*col_vec*), and input (*x_vec*) are read from the arrays *val_arr*,

Algorithm 1 VHCC kernel ($y = Ax + y$) for Xeon Phi.

```
1:  v ← startvec
2:  (pidx, ridx, vidx) ←load initial values for thread i
3:  last_row ← load value for thread i
4:  overflow_row ← load value for thread i
5:  set tmp_y to array for current panel of thread i
6:  while v < endvec do
7:      col_vec ← load(&col_idx[vidx])
8:      val_vec ← load(&val_arr[vidx])
9:      x_vec ← gather(&x[0], col_vec)
10:     res_vec ← res_vec + val_vec × x_vec
11:     if v = vec_ptr[pidx] then
12:         mask_{1,2,3,4} ← load(&mask_arr[ridx × 4])
13:         res_vec ← prefix_sum(res_vec, masks_{1,2,3})
14:         row_vec ← load(&row_idx[ridx])
15:         scatter(&tmp_y[0], res_vec, row_vec)
16:         pidx ← pidx + 1
17:         ridx ← ridx+ popcnt(mask_4)
18:     else
19:         res_vec[0] ← reduce_add(res_vec)
20:     end if
21:     v ← v + 1
22:     vidx ← vidx + 8
23: end while
24: barrier_wait()
25: atomic{tmp_y[last_row] ← tmp_y[last_row]+
        tmp_y[overflow_row]}
26: sum tmp_y arrays from all panels
```

col_idx and *x* respectively (lines 7-9). These are then used to compute the *res_vec* vector. When a vector contains elements that cross rows, the first branch is taken (lines 12-17) and the segmented prefix sum is computed (line 13). Otherwise, a sum reduction operation (line 19), equivalent to having all adders turned on, is used to sum up all the elements in the vector, and the result is placed in the first element of the result vector for the next iteration of computation. The last mask, $mask_4$, is used to indicate which elements in the vector are the last elements of a row. The operation `popcnt` can then be used to count the number of elements in $mask_4$ and is used to advance the $ridx$ pointer accordingly (line 17). The counters *v* and *vidx* are then updated for the next iteration.

As mentioned earlier, the *row_idx* array stores the row indices of the corresponding result to be written to output array. Recall that each thread processes a horizontal block that is jagged. Hence, there may exist rows that are shared by adjacent threads. The tmp_y array contains additional slots that are used locally by each thread to store the intermediate sum for the last row in each block. This intermediate sum is then atomically added to the corresponding element (*last_row*) in the output array after a barrier operation (lines 24-25).

3.4 Auto-Tuning for the Number of Panels

For a $m \times n$ scale-free matrix, the number of panels in the 2D jagged partitioning technique affects the performance of VHCC. In this section, we develop a performance tuning technique to guide the selection of the panel number. Recall that the matrix is first partitioned into panels and each panel is further divided into blocks. Let N_P be the number of panels, N_B be the number of blocks, and N_T be the number of threads (where $N_T = N_B \times N_P$). N_P affects the L2 cache locality of the input vector (x) accesses. When there are more panels, access to the input vector is segmented across the panels and hence the data locality of accessing the x vector is improved. On the other hand, if there are more panels, the number of temporary arrays required for storing the intermediate output vectors increases, and thus the cost of the final reduction or merge phase will increase.

The main idea behind our auto-tuning procedure is to extract parameters from an idealized model of our scale-free SpMV kernel and use it to estimate N_P for a given irregular matrix. The computation shown in Algorithm 1 can be divided into two phases, the actual computation phase (lines 1-25) and the merge phase (line 26). Thus, the total execution time of each thread can be estimated by

$$T_B = T_{Comp} + T_{Merge}, \tag{1}$$

where T_B is the total time taken by a thread to perform the overall SpMV computation on a given block with nnz_B non-zeros, T_{Comp} is the time used for the computation phase, and T_{Merge} is the amount of time required to perform a final reduction on the intermediate output arrays. Here, we assume that each block takes a similar amount of time because of the workload balancing achieved by VHCC. The time required for the computation phase is given by

$$\begin{aligned} T_{Comp} = \{T^{int}_{C,r} + T^{dbl}_{C,r} + T_{gather} + T_{SpMV}\} \\ \times nnz_B + p \times (T^{int}_{C,r} + T^{dbl}_{C,rw}). \end{aligned} \tag{2}$$

This expression includes the time to read the column index from *col_idx* ($T^{int}_{C,r}$) and a double-precision value from *val_arr* ($T^{dbl}_{C,r}$). T_{gather} is the time required to gather a value from the x vector, the exact form of which will be described later. T_{SpMV} represents the time for the actual arithmetic computation; it can be set to zero if we are only interested in the lower bound of T_{Comp}. For a block with dimensions p by q, the last term in Eq. 2 represents the time required to read a row index and write a double to the *tmp_y* array which has length $p = \alpha m / N_B$. Table 1 provides a summary of the parameters and their values used in our model. These values are obtained using microbenchmarks described in [9] and the values obtained by us are also similar to theirs. The time to modify a value is typically twice that of an equivalent read access because it includes both read and write accesses.

To derive T_{gather}, we make use of the properties of a scale-free matrix. In a scale-free matrix, the distribution of the non-zeros per row follows a power-law distribution [2], and because of its scale-free nature, we can assume that the distribution of non-zeros in a $p \times q$ block also follows the power-law,

$$Pr[K = k] = f(k) = \frac{k^{-\lambda}}{\sum_{j=1}^{q} j^{-\lambda}}, \tag{3}$$

for $1 \leq k \leq q$. For every matrix, λ is obtained by fitting the actual non-zero distribution to the power law. Since the non-zeros nnz_B are distributed according to this distribution for a given k, the average stride between the non-zeros can be computed as $stride = q/(nnz_B \times f(k))$. To determine the gather time for a given stride, we introduce a function $h(stride)$ that models the gather time as a function of the stride,

$$h(stride) = \begin{cases} \beta T^{dbl}_{L,r} + \beta \frac{stride \times (T^{dbl}_{G,r} - T^{dbl}_{L,r})}{\gamma}, & stride \leq \gamma, \\ \beta T^{dbl}_{G,r}, & stride > \gamma, \end{cases} \quad (4)$$

where the time to perform a gather from main memory and from L2 cache are given by $T^{dbl}_{G,r}$ and $T^{dbl}_{L,r}$, respectively. When the stride is less than the threshold γ, the gather time is a linear function of the stride. The threshold depends on the size of the L2 cache, and is set to 65536 (equivalent to 512KB). Parameters α and β are derived using R-MAT matrices, which will be described later. The average gather time can therefore be computed using the expectation operator ($\mathrm{E}_K[\cdot]$) with respect to the random variable K,

$$T_{gather} = \mathrm{E}_K[h(stride)]. \quad (5)$$

The merge phase performs a reduction of all the intermediate *tmp_y* output arrays, and the time required for this phase modeled as

$$T_{Merge} = \frac{m}{N_T}(N_p \times T^{dbl}_{M,r} + T^{dbl}_{M,rw}). \quad (6)$$

$T^{dbl}_{M,r}$ and $T^{dbl}_{M,rw}$ denote the time to read or read-and-write a double-precision element to main memory (see Table 1). We make a distinction between the computation and merge phases for reading a double-precision value because they incur different overheads in the different phases.

In our auto-tuning procedure, we use a lookup table that has been pre-built offline from R-MAT(s, e) by varying the scale (s) and edge factor (e) of the matrix. The lookup table contains (s, e, α, β) tuples, where α and β are derived by fitting to the model described in Eqs. 1–6. This table needs to be built once offline and can be reused subsequently. When given a real-world sparse matrix, we obtain (α, β) from the lookup table using the nearest (s, e) tuple, and use it to predict the best N_P value. In Section 4.2, we will evaluate this auto-tuning method using real-world matrices.

4. Evaluation

All experiments are conducted on an Intel Xeon Phi 5110P coprocessor. We use ten real-world scale-free sparse matrices from the University of Florida sparse matrix collection [8] as listed in Table 2. These scale-free matrices are from different application domains; many of them represent web graphs, gene networks, or citation networks. We compare the performance of VHCC with Intel MKL's CSR implementation (denoted as MKL) found in Intel Math Kernel Library 11.1. 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. We also use the Intel VTune profiler [11] to investigate the execution efficiency.

Table 1: Description of parameters and their values used in the performance model.

Parameter	Description	Value
T_B	Time taken by a block to complete both SpMV phases	n.a.
T_{Comp}	Total time for SpMV computation phase	n.a.
T_{Merge}	Total time for SpMV merge phase	n.a.
$T^{int}_{C,r}$	Time to read a column or row index (integer) from main memory in the computation phase	6ns
$T^{dbl}_{C,r}$	Time to read a double-precision value from main memory in the computation phase	12ns
$T^{dbl}_{C,rw}$	Time to modify a double-precision value in main memory in the computation phase	24ns
T_{gather}	Time to gather an element from the x vector	n.a.
T_{SpMV}	Time to perform SpMV arithmetic operations	n.a.
$T^{dbl}_{G,r}$	Time to gather a double-precision value from main memory	320ns
$T^{dbl}_{L,r}$	Time to gather a double-precision value from L2 cache	12ns
$T^{dbl}_{M,r}$	Time to read a double-precision value from main memory in the merge phase	21ns
$T^{dbl}_{M,rw}$	Time to modify a double-precision value in main memory in the merge phase	42ns
α, β	Parameters to be fit using R-MAT matrices	n.a.

Table 2: List of sparse matrices used for evaluation. Columns are: the dimensions, total no. of non-zeros (*nnz*), average (avg) and maximum (max) non-zeros 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

4.1 Performance Study

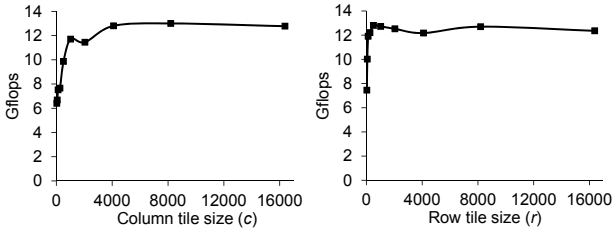
We perform three sets of experiments to evaluate our VHCC SpMV implementation: (a) the effects of 2D jagged partitioning and tiling on cache efficiency, (b) improvement in SIMD vector utilization due to SIMD segmented prefix sum, and (c) workload balance and cycles per instruction (CPI) comparison. In this section, we first report perfor-

mance numbers and profiling results using the optimal value of the panels. The effectiveness of our auto-tuning technique will be studied in Section 4.2.

4.1.1 Tiling and 2D Jagged Partitioning

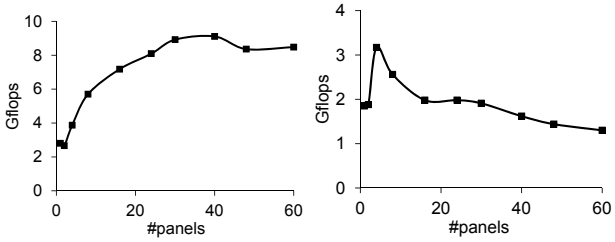
Results for tiling are first presented in Fig. 5. Only performance numbers for the *mouse_gene* matrix are shown as all the other matrices have similar results. First, we fix the row tile size r to 512, and vary the column tile size. Figure 5a shows that in general, performance increases when the column tile size becomes larger, and does not differ significantly after $c = 8192$. Next, we fix the column tile size c to 8192 and vary the row tile size as shown in Fig. 5b. This figure shows that the best performance is achieved when $r = 512$. Hence, we set $r = 512$ and $c = 8192$ as the default tile size for all subsequent experiments unless stated otherwise. The tile size of $r = 512$ and $c = 8192$ takes up 32 KB if fully utilized, which is the same size as the L1 data cache size on Xeon Phi.

Whereas tiling is used to improve L1 cache efficiency, 2D jagged partitioning serves to improve L2 cache efficiency. Figure 6 shows the result of varying the number of vertical panels (N_P) for two representative matrices. Unlike tiling, we observe that the effects of 2D jagged partitioning is more pronounced. Furthermore, the optimal number of panels depends on a given matrix. For *connectus*, the best performance is achieved when there are 40 panels, whereas for *Stanford*, only 4 panels are required. As we can see, the optimal value for N_P can vary substantially, and we attribute this to two reasons. The first reason is that L2 cache efficiency is sensitive to N_P because cache miss penalties are high on Xeon Phi and a cache miss would involve a series of requests to the tag directories and memory controllers. Secondly, there exists a tradeoff between improving SpMV computation time due to better cache efficiency, and incurring overhead for merging additional intermediate arrays.



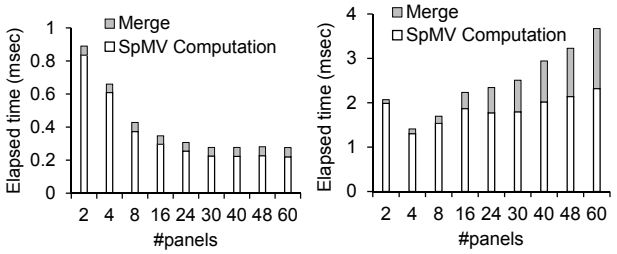
(a) r=512, with c varied (b) c=8192, with r varied

Figure 5: Performance tuning for the tile size (*mouse_gene*)



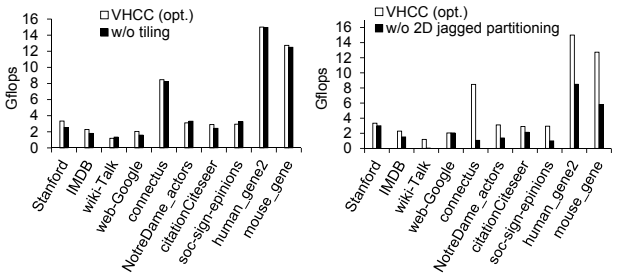
(a) connectus (b) Stanford

Figure 6: Performance tuning for the number of vertical panels (N_P) in 2D jagged partitioning



(a) connectus (b) Stanford

Figure 7: Time breakdown when number of panels is varied

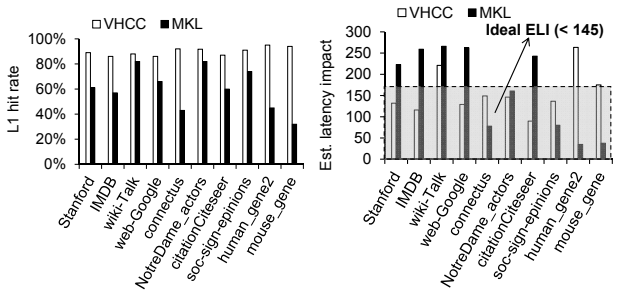


(a) Tiling (b) 2D jagged partitioning

Figure 8: Impact of tiling and 2D jagged partitioning

The tradeoff can be seen in Fig. 7, which shows the breakdown of the time between the two phases of SpMV computation – the computation phase and the merge phase. The figure shows that for *connectus*, SpMV computation time is considerably reduced when the number of panels is increased. On the other hand, having more panels does not improve the performance for *Stanford*. This is because having more panels will require additional intermediate arrays and as a result, the merge phase will become more expensive. To determine the best tradeoff, our auto-tuning procedure can be used to select a good panel number for a given scale-free matrix (see Section 4.2).

Figure 8 shows the performance impact of VHCC when tiling and 2D jagged partitioning are disabled. 'VHCC (opt.)' indicates that both techniques are used, whereas 'w/o tiling' indicates that the tiling technique is not used. 'w/o 2D jagged partitioning' indicates that 2D jagged partitioning is not used, and instead, the matrix is divided into equi-sized 2D blocks. The figure shows that, on aver-



(a) L1 hit rate (b) Estimated latency impact

Figure 9: Profiling of cache efficiency

age, tiling improves performance by 1.1×. For 2D jagged partitioning, performance is improved by 4.4× on average. This result is not surprising considering that tiling mainly improves L1 data locality, whereas 2D jagged partitioning improves L2 cache locality and hence has a greater impact on the overall performance.

Figure 9 shows profiling results on the memory hierarchies gathered using hardware counters. Figure 9a shows the L1 cache hit rates and the estimated latency impact of VHCC compared to MKL. The L1 cache hit rates are improved by 7%-194% relative to MKL, and they typically reach around 90%. Figure 9b shows that the estimated latency impact is reduced as well for most matrices. More importantly, the estimated latency impact is now in the ideal range (< 145) for most matrices. These results show that the 2D jagged partitioning and tiling methods used in VHCC are in general cache effective for scale-free matrices.

4.1.2 SIMD Segmented Prefix Sum

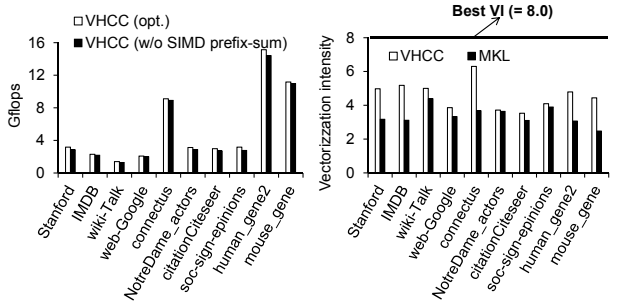
Our SIMD segmented prefix sum method aims to improve vectorization efficiency. To study the impact of this method, we replace the SIMD segmented prefix sum with normal C code (denoted as *w/o SIMD prefix-sum*), and compare its performance with the optimized VHCC with SIMD segmented prefix sum. Figure 10a shows the comparison with and without SIMD segmented prefix sum. We find that overall performance was increased by up to 14.3%. Figure 10b shows a comparison of the vectorization intensity between VHCC and MKL. On average, VHCC achieves a vectorization intensity that is 1.4 times higher than that of MKL. This indicates that the SIMD segmented prefix sum uses the VPU more effectively. The ideal vectorization intensity is often difficult to achieve because of the loop and counter overheads.

4.1.3 Workload Balance and CPI Comparison

Next, we look at the workload balancing and performance scalability characteristics of VHCC. In general, because VHCC inherits the benefits of both COO and CSR and employs equal division of non-zeros, we expect the workload to be balanced across threads. Figure 11a illustrates the relative standard deviation of execution time of all threads on Xeon Phi for the different matrices. Except for *connectus* which is a highly irregular matrix, all other matrices have very low workload difference among threads. The average relative standard deviation is smaller than 2.0%. This shows that VHCC is effective in workload balancing for scale-free matrices. Figure 11b shows a comparison of the cycles per instruction (CPI) metric between VHCC and MKL. As shown, VHCC significantly reduces the cycles per instruction (CPI) by 42.5% on average relative to MKL. In addition, for most matrices, the CPI is within or close to the ideal CPI range. All of the techniques we have described so far contributed to the reduced CPI values.

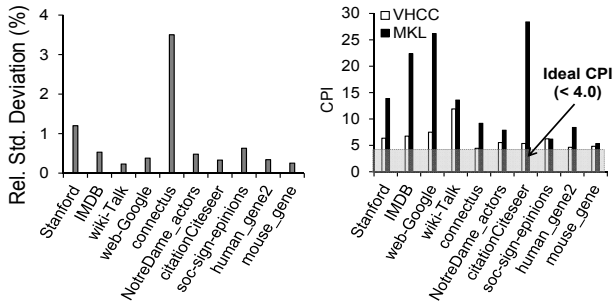
4.2 Overall Performance and Effectiveness of the Auto-Tuning Procedure

Figure 12 compares the performance of auto-tuned VHCC and MKL using R-MAT matrices, whereas Fig. 13 shows the performance of auto-tuned VHCC and MKL using the



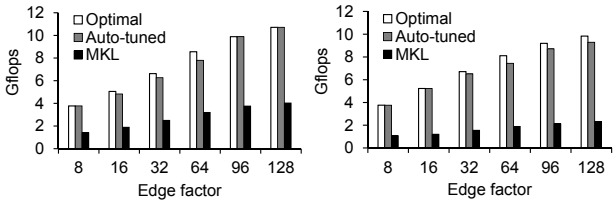
(a) Gflops (b) Vectorization intensity

Figure 10: SIMD vector utilization: (a) performance impact of SIMD segmented prefix sum, (b) vectorization intensity comparison between VHCC and MKL



(a) Workload balance (b) CPI comparison

Figure 11: (a) Relative standard deviation of execution time of each thread, (b) comparison of cycles-per-instruction (CPI).



(a) Scale = 16 (b) Scale = 18

Figure 12: Performance comparison between VHCC and MKL using R-MAT scale-free matrices

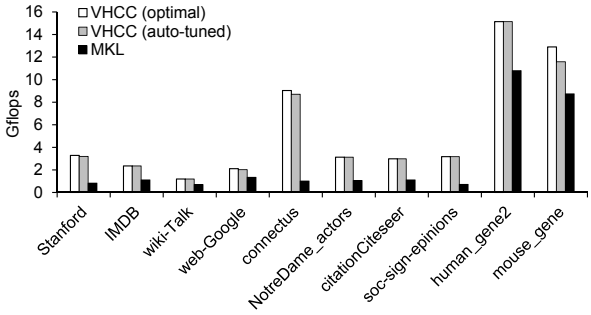


Figure 13: Performance comparison between VHCC and MKL using real-world scale-free matrices.

real-world matrices. The optimal performance denoted by "VHCC (optimal)" is derived using exhaustive tuning. Table 3 summarizes the speedups for real-world matrices. Results in the table show that both the auto-tuned VHCC and optimal VHCC achieve an average speedup of 3× over MKL. Figure 14 depicts the effectiveness of the auto-tuning procedure on the real-world matrices. On average, it achieves 98% of the optimal performance.

Finally, even though VHCC is designed specifically for scale-free matrices, we also test the performance of VHCC on regular matrices. Figure 15 shows the performance comparison between VHCC and MKL for regular matrices. For some matrices, VHCC performs better than MKL, while for others, MKL is better. On average, VHCC achieves similar performance to MKL (∼92% of MKL's performance) for regular matrices.

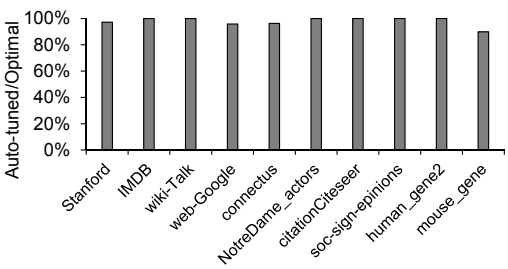


Figure 14: Auto-tuned performance as a percentage of the optimal performance for real-world matrices.

Table 3: Speedups achieved by exhaustive-tuned (Opt.) and auto-tuned VHCC over MKL.

Matrix	Opt. vs MKL	Auto-tuned vs. MKL
Stanford	4.0	3.8
IMDB	2.1	2.1
wiki-Talk	1.7	1.7
web-Google	1.6	1.5
connectus	8.9	8.6
NotreDame_actors	3.0	3.0
citationCiteseer	2.7	2.7
soc-sign-epinions	4.4	4.4
human_gene2	1.4	1.4
mouse_gene	1.5	1.3

5. Related Work

A large body of work has been published on SpMV computation on modern processors. Im and Yelick studied the effects of register blocking and cache blocking to improve SpMV performance on processors with memory hierarchies [10]. The OSKI (Optimized Sparse Matrix Kernel Interface) library was developed to automatically tune SpMV kernels on processors with cache-based memory hierarchies [19]. Matrix reordering methods such as those in [15] and [13] were investigated to improve the locality of memory accesses, thereby increasing cache hit rates. Williams *et al.* [21] studied thread blocking for multicore processors and evaluated its performance on modern multicore architectures. Graph partitioning methods were also proposed for SpMV on distributed-memory computers [6, 13]. Apart from blocking, other techniques have been proposed to improve performance by reducing memory traffic [4, 20]. There have also been studies on optimizing SpMV for coprocessors such as GPUs [3, 5, 18] and Intel Xeon Phi [12, 17]. Saule showed that register blocking was not a viable technique on Xeon Phi, whereas Liu and coworkers developed a format called ESB based on the ELLPACK format [12].

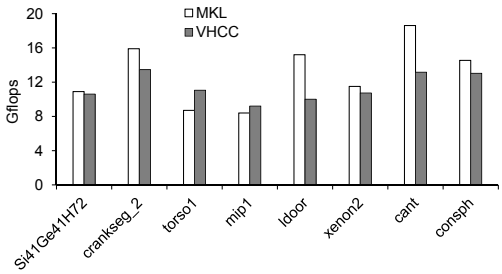


Figure 15: Performance comparison between VHCC and MKL using regular matrices.

In comparison, our proposed VHCC is optimized for Xeon Phi. It takes into consideration Xeon Phi's memory hierarchy and makes use of the wide-vector VPUs. Furthermore, many prior methods are not optimized for scale-free matrices, and they used exhaustive tuning to select the best runtime configuration, whereas we have developed an auto-tuning procedure that works well in practice.

6. Conclusion

In this paper, we develop VHCC for computing SpMV for scale-free sparse matrices on Intel Xeon Phi. It employs 2D jagged partitioning and tiling to achieve good cache efficiencies and work balancing. We also develop an efficient SIMD segmented prefix sum implementation that is made possible by Xeon Phi's enhanced SIMD capabilities. A performance tuning procedure for selecting the number of panels is described. Experimental results demonstrate that our implementation is able to achieve average speedups of 3× compared to Intel MKL's CSR kernel. The auto-tuning procedure is able to achieve performances that is within 10% of the optimal performance.

Acknowledgment

This work was partially supported by the National Natural Science Foundation of China (No. 61300005). We are also grateful to Intel for providing us with a Xeon Phi card. The first and second authors contributed equally to this work.

References

[1] Intel Math Kernel Library: https://software.intel.com/en-us/intel-mkl.

[2] A.-L. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, Oct. 1999.

[3] M. M. Baskaran and R. Bordawekar. Optimizing sparse matrix-vector multiplication on GPUs. Technical report, RC24704, IBM T. J. Watson, 2009.

[4] M. Belgin, G. Back, and C. J. Ribbens. Pattern-based sparse matrix representation for memory-efficient SMVM kernels. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pages 100–109, 2009.

[5] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 18:1–18:11, 2009.

[6] E. G. Boman, K. D. Devine, and S. Rajamanickam. Scalable matrix computations on large scale-free graphs using 2D graph partitioning. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 50:1–50:12. ACM, 2013.

[7] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A recursive model for graph mining. In *SIAM International Conference on Data Mining*, 2004.

[8] T. A. Davis and Y. Hu. The University of Florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, Dec. 2011. http://www.cise.ufl.edu/research/sparse/matrices/.

[9] J. Fang, A. L. Varbanescu, H. J. Sips, L. Zhang, Y. Che, and C. Xu. An empirical study of Intel Xeon Phi. *CoRR*, abs/1310.5842, 2013.

[10] E.-J. Im. *Optimizing the performance of sparse matrix-vector multiplication*. PhD thesis, University of California Berkeley, 2000.

[11] J. Jeffers and J. Reinders. *Intel Xeon Phi Coprocessor High-Performance Programming*. Morgan Kaufmann, 2013.

[12] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey. Efficient sparse matrix-vector multiplication on x86-based many-core processors. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 273–282. ACM, 2013.

[13] L. Oliker, X. Li, P. Husbands, and R. Biswas. Effects of ordering strategies and programming paradigms on sparse matrix computations. *SIAM Rev.*, 44(3):373–393, Mar. 2002.

[14] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the web. Tech. report, Stanford Digital Library, 1999.

[15] A. Pinar and M. T. Heath. Improving performance of sparse matrix-vector multiplication. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing*, SC '99, 1999.

[16] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition, 2003.

[17] 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, pages 559–570. Springer Berlin Heidelberg, 2014.

[18] A. Venkat, M. Shantharam, M. Hall, and M. M. Strout. Non-affine extensions to polyhedral code generation. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 185:185–185:194, 2014.

[19] R. Vuduc, J. W. Demmel, and K. A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Proc. SciDAC, J. Physics: Conf. Ser.*, volume 16, pages 521–530, 2005.

[20] J. Willcock and A. Lumsdaine. Accelerating sparse matrix computations via data compression. In *Proceedings of the 20th annual international conference on Supercomputing*, ICS '06, pages 307–316, 2006.

[21] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC '07, pages 38:1–38:12, 2007.