CVR: Efficient Vectorization of SpMV on X86 Processors

Biwei Xie
State Key Laboratory of Computer Architecture,
Institute of Computing Technology, Chinese Academy of Sciences,
University of Chinese Academy of Sciences
China
xiebiwei@ict.ac.cn

Jianfeng Zhan
State Key Laboratory of Computer Architecture,
Institute of Computing Technology, Chinese Academy of Sciences,
University of Chinese Academy of Sciences
China
zhanjianfeng@ict.ac.cn

Xu Liu
Department of Computer Science,
College of William and Mary
USA
xl10@cs.wm.edu

Wanling Gao
Institute of Computing Technology, Chinese Academy of Sciences,
University of Chinese Academy of Sciences
China
gaowanling@ict.ac.cn

Zhen Jia
Department of Computer Science,
Princeton University
USA
zhenj@princeton.edu

Xiwen He
Institute of Computing Technology,
Chinese Academy of Sciences
China
hexiwen@ict.ac.cn

Lixin Zhang
Institute of Computing Technology,
Chinese Academy of Sciences
China
zhanglixin@ict.ac.cn

Abstract

Sparse Matrix-vector Multiplication (SpMV) is an important computation kernel widely used in HPC and data centers. The irregularity of SpMV is a well-known challenge that limits SpMV's parallelism with vectorization operations. Existing work achieves limited locality and vectorization efficiency with large preprocessing overheads. To address this issue, we present the Compressed Vectorization-oriented sparse Row (CVR), a novel SpMV representation targeting efficient vectorization. The CVR simultaneously processes multiple rows within the input matrix to increase cache efficiency and separates them into multiple SIMD lanes so as to take the advantage of vector processing units in modern processors. Our method is insensitive to the sparsity and irregularity of SpMV, and thus able to deal with various scale-free and HPC matrices. We implement and evaluate CVR on an Intel Knights Landing processor and compare it with five state-of-the-art approaches through using 58 scale-free and HPC sparse matrices. Experimental results show that CVR can achieve a speedup up to 1.70 × (1.33× on average) and a speedup up to 1.57× (1.10× on average) over the best existing approaches for scale-free and HPC sparse matrices, respectively. Moreover, CVR typically incurs the lowest preprocessing overhead compared with state-of-the-art approaches.

CCS Concepts • **Mathematics of computing → Mathematical software performance**; **Computations on matrices**; • **Computer systems organization → Single instruction, multiple data**;

Keywords SpMV, Xeon Phi, SIMD, Vectorization

ACM Reference Format:
Biwei Xie, Jianfeng Zhan, Xu Liu, Wanling Gao, Zhen Jia, Xiwen He, and Lixin Zhang. 2018. CVR: Efficient Vectorization of SpMV on X86 Processors. In *Proceedings of 2018 IEEE/ACM International Symposium on Code Generation and Optimization (CGO'18).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3168818

CGO'18, February 24–28, 2018, Vienna, Austria

ACM ISBN 978-1-4503-5617-6/18/02…$15.00
https://doi.org/10.1145/3168818

1 Introduction

Sparse matrix-vector multiplication (SpMV) is an essential kernel in both scientific and data center domains [26] [25] [32]. For many iterative applications, SpMV dominates the overall execution time. For instance, the large-size linear systems and eigenvalue problems, which are important scientific application components, heavily rely on SpMV [5]. As the data volume grows exponentially, more large-scale and irregular sparse matrices are emerging in various application domains, such as social networks, electronic commerce, search engines etc [37] [14]. We call the large-scale and irregular sparse matrices the scale–free matrices in this paper.

The irregular data layout of sparse matrices results in irregular memory references, and thus poor data locality. The pattern of irregular memory references is input-dependent and not amenable to compile-time analysis. Moreover, preprocessing in advance is widely adopted for runtime optimization of SpMV, which re-organizes the data layout of sparse matrices for better locality, with extra preproessing overhead. So the overall performance of a SpMV solution is usually determined by two separated parts: perprocessing overhead and each-iteration SpMV performance. The preprocessing overhead involves converting a sparse matrix from its original format to another intermediate format, while the each-iteration SpMV performance indicates the efficiency of SpMV with the intermediate format in a single iteration. In this paper, we use SpMV performance to indicate each-iteration SpMV performance. Even though existing work uses lots of techniques, like blocking (tiling) [2, 21, 30] and segmented sum [4], to improve the data locality and parallelism with vectorization operations, most of the state-of-the-art SpMV implementations still suffer from significant number of cache misses and non-negligible preprocessing overhead for scale-free matrices on many-core processors.

To overcome the weakness of existing work, we propose a new SpMV solution, aiming at both vectorization efficiency and locality as well as low preprocessing overhead. Our solution follows three principles: 1) Each row in the sparse matrices is processed by a single SIMD lane. 2) Multiple rows can be processed in parallel. 3) Once the row in a SIMD lane has been processed, the next non-empty row in the sparse matrix would be processed consecutively. Accordingly, we further present a new format, called compressed vectorization-oriented sparse row (CVR). Based on CVR, our SpMV implementation improves data reuse and simplifies the reduction operation in SpMV.

Our experiments show that our CVR based SpMV implementation delivers typically the best performance and lowest preprocessing overhead on average among all the SpMV solutions. On scale-free sparse matrices, our approach achieves an up to 40× (2.84× on average) speedup over MKL and 1.70× (1.33× on average) over state of the art. On HPC matrices, our approach achieves an up to 17× (1.22× on average) speedup over MKL and 1.57× (1.10× on average) over state of the art.

Specifically, we make the following contributions:

- We propose a new format, CVR, for better vectorization efficiency and locality, and present how to convert a sparse matrix to CVR.
- We present a lightweight and efficient SpMV implementation based on CVR.
- We compare our work with five other state-of-the-art SpMV implementations using 58 scale-free (30) and HPC (28) matrices.

The remainder of the paper is organized as follows. Section 2 describes the background details of Xeon Phi and elaborate the SpMV algorithm. Section 3 gives our motivation. Section 4 presents our CVR format. Section 5 elaborates the CVR–based SpMV implementation. Section 6 gives the experiment setup and methodology. Section 7 reports the experiments result. Section 8 describes related work and Section 9 concludes the paper.

2 Background

2.1 Intel Xeon Phi Processor

The Intel Xeon Phi processor is an x86 compatible processor and also a commercial release of the Intel Many Integrated Core (MIC) architecture [27]. Knights Landing (KNL) is the second and latest generation of Xeon Phi. A KNL processor consists of multiple physical tiles, and each tile consists of two cores. There are two vector processing units (VPUs) per core and a 1MB L2 cache shared between two cores. Each KNL core employs four hardware threads with up to 32 registers per thread context. There are two types of memory on KNL: multi-channel DRAM (MCDRAM) and double data rate (DDR) memory. MCDRAM is the 16GB high bandwidth memory comprising eight channels. MCDRAM can be configured to three modes: cache mode, flat mode, and hybrid mode. KNL introduces AVX-512, which provides 512-bit-wide vector instructions. Moreover, it supports non-continuous memory read/write with gather/scatter instructions.

Algorithm 1 scalar-SPMV with the CSR format.

```
1: for i = 0 to numRows − 1 do
2:     sum ← 0
3:     for j = row_ptr[i] to row_ptr[i + 1] − 1 do
4:         sum ← sum + vals[j] × x[col_idx[j]]
5:     end for
6:     y[i] ← sum
7: end for
```

2.2 Sparse Matrix-vector Multiplication (SpMV)

Note that we use matrix A, vector x, and vector y when describing SpMV ($y = y + A \cdot x$) in the following sections.

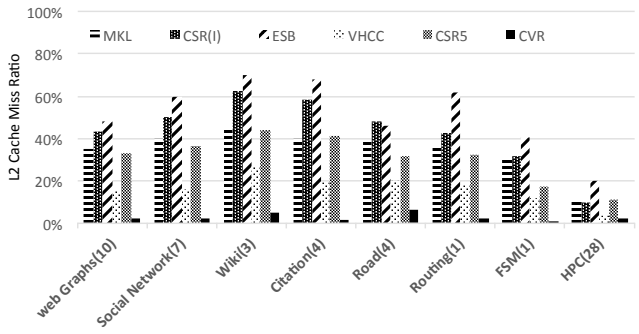


Figure 1. Average L2 Cache Miss ratio of existing work on data sets from various domains.

We first analyze the basic characteristic of SpMV ($y = y + A \cdot x$) with the widely used CSR format. CSR consists of three vectors: *vals* as the value of each nonzero element, *col_idx* as the column index for each nonzero element, and *row_ptr* as the beginning of each row in *vals* and *col_idx*. Algorithm 1 shows the pseudo code of scalar SpMV based on the CSR format.

There are two major obstacles that hinder SpMV from exploiting the potential of long vectorization (SIMD) instructions supported in KNL: 1) *poor data locality* introduced by irregular memory references of vector x and 2) *low computation efficiency* of the reduction in SpMV computation. To utilize SIMD instructions, x[cols[j]] is loaded with gather instructions, but the poor locality issue still exists. To overcome the limitation of this naive vectorization approach, the iteration-based SpMV implementation employs new representation with two computation phases: (1) converting original sparse matrices to a new format and (2) the new format based SpMV implementation. The preprocessing overhead can be amortized with multiple computation iterations. In the next section, we review several state-of-the-art SpMV representations and motivates our approach.

3 Motivation

We perform a comprehensive evaluation of existing SpMV representations using 58 representative sparse matrix data sets. The details of these data sets will be illustrated in Section 6.3.

Locality Cache locality is crucial to SpMV. The proposed techniques, such as blocking and padding in ELLPACK [2] and ESB [21], 2D jagged partition in VHCC [30], and in-tile transposing in CSR5 [19] all aim at improving cache locality. These approaches achieve substantial improvements on cache performance for regular matrices used in the HPC domain, but not for the scale-free matrices in the datacenter domains, which are highly sparse and irregular. To further investigate the locality, we collect L2 (last-level) cache miss ratio of these existing solutions on KNL; the L2 cache miss penalty exceeds 200 cycles on KNL and is crucial to the SpMV performance. We evaluate 30 scale-free sparse matrices from various domains and 28 HPC sparse matrices. We group them by application domains and present their average L2 cache miss ratios in Figure 1 (details are shown in Figure 7). From the figure, we can find that all existing solutions own higher L2 cache miss ratios on the scale-free sparse matrices comparing to the HPC sparse matrices. This poor data locality is due to the high irregularity and sparsity in scale-free sparse matrices. Thus, there raises an urgent requirement to design a new SpMV representation to reduce the cache misses. Figure 1 also shows that our CVR format has much lower cache miss.

Table 1. The number of iterations (Median value) needed to amortize the preprocessing overhead on scale-free matrices.

formats	CSR(I)	ESB	VHCC	CSR5	CVR
overhead	49	285	2653	5.36	2.14

Preprocessing overhead Preprocessing phase is important to SpMV computation. A higher preprocessing overhead requires more iterations to amortize the cost. To investigate the preprocessing performance of existing work, we calculate the number of iterations that a method needs to amortize the preprocessing overhead on all the investigated data sets. A detailed description of how to calculate the preprocessing overhead can be found in Section 7.2. The results in Table 1 indicate that the preprocessing phase of most existing SpMV methods could be time-consuming and requires iterations up to thousands to amortize the overhead (details can be found in Table 4). So the preprocessing overhead may become one of the factors that prevent the SpMV to achieve satisfactory performances, especially for the ones with a small amount of SpMV iterations. For comparison, we also summarize our approach CVR's, preprocessing overhead in Table 1, which is orders of magnitudes lower than most of the existing techniques.

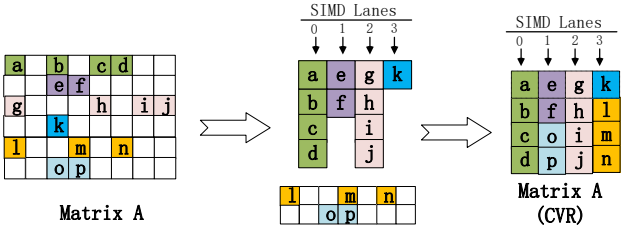
4 CVR Methodology

4.1 CVR Representation

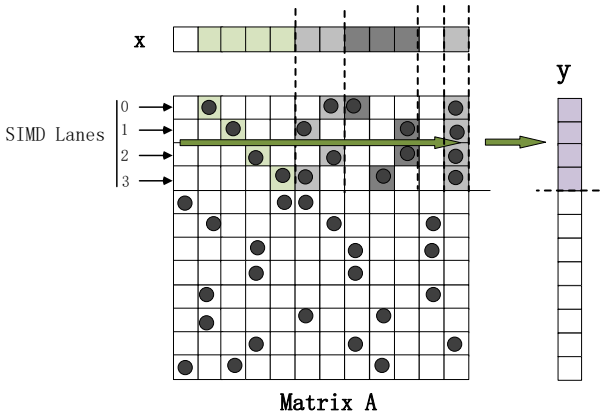
To deal with the bottlenecks mentioned in Section 3, we revisit the data layout of sparse matrices from the perspective of locality and vectorization efficiency.

Our design principles can be summarized as follows: 1) Each row in matrix A is processed by a single SIMD lane. 2) Many rows can be processed in parallel, and the number of them equals to the amount of SIMD lanes. 3) Once the row in a SIMD lane has been finished, the next non-empty row in matrix A would be processed consecutively.

Based on these design principles, we propose Compressed Vectorization-oriented sparse Row (CVR) format for SpMV. Figure 2a presents the preprocessing phase of our method,



(a) Converting matrix A to CVR



(b) SpMV implementation with CVR

Figure 2. Format conversion and SpMV implementation with CVR.

which converts a sparse matrix A to the CVR format. The number of columns in CVR format equals to the number of SIMD lanes. Each row in matrix A will always be inserted into the shortest column in CVR format. This conversion procedure is vectorization friendly and carried out independently by all SIMD lanes. In our SpMV implementation, each column in CVR is processed independently, and each SIMD lane is responsible for a column respectively. From the perspective of a single SIMD lane, the reduction operation in SpMV is performed by consecutive *Add* instructions, which significantly simplifies the SpMV computation.

Figure 2b shows the overview of SpMV execution procedure based on CVR. Elements in the same row of CVR are executed together by different SIMD lanes, accessing adjacent values of vector *x*. As the execution proceeds, CVR keeps the memory access of vector *x* in a short range. This largely improves the reuse of vector *x*.

4.2 General Description of Conversion

Since matrix A is converted into CVR format by row, we introduce a tracker-based mechanism to track the row information. There will be ω trackers in total, where ω is the number of SIMD lanes (8 for double precision and 16 for single precision on KNL). Each tracker trails a row with three variables: *rowID* indicates the row index; *valID* gives the index of the first nonzero element in the row; *count* represents the total number of nonzero elements in this row. The variables in all trackers form three vectors: ($rowID[\omega]$, $valID[\omega]$, and $count[\omega]$). In the CVR format, vector *cvr_vals* is used to store nonzero elements and vector *cvr_colidx* is used to store their corresponding column indices. For parallel execution, we divide the nonzero elements evenly to T parts, where T equals to the number of threads. For a certain thread, *nnz_start* and *nnz_end* indicate the indices of the first and last nonzero element, as well as *first_row* and *last_row* indicate the indices of the first and last row. Each thread conducts the data conversion independently. We describe the conversion of a single thread for simplicity. In Figure 3, we use 4 trackers (assuming *nLanes* = 4) to illustrate the conversion procedure [1].

Algorithm 2 Initializing the tracking vectors.

```
1:  valID[ω]
2:  rowID[ω]
3:  count[ω]
4:  rs ← first_row
5:  for i = 0 to ω - 1 do
6:      valID[i] ← row_ptr[rs]
7:      rowID[i] ← rs
8:      if rs == first_row then
9:          count[i] ← row_ptr[rs + 1] − nnz_start
10:     else if rs < last_row && rs! = first_row then
11:         count[i] ← row_ptr[rs + 1] − row_ptr[rs]
12:     else if rs == last_row then
13:         count[i] ← nnz_end − row_ptr[rs]
14:     end if
15:     Inc(rs)
16: end for
```

Trackers Initialization In Figure 3, the four trackers are initialized by the first four rows of matrix A respectively. For the rs^{th} tracker, its *valID* denotes the location of the first element in the rs^{th} row. The *rowID* gives the index of the rs^{th} nonempty row in matrix A, and *count* is the number of nonzero elements in the rs^{th} row. Algorithm 2 shows the initialization procedure. If there are no enough non-empty rows to initialize all the trackers, the remainder trackers will be initialized to zero.

Conversion The format conversion is carried out by gathering nonzero elements from matrix A and storing them into the CVR format (i.e., vector *cvr_vals* and *cvr_colidx*). $valID[\omega]$ indicates which elements to gather. After ω nonzero elements are gathered and then stored in the CVR format, $valID[\omega]$ will increase by one and $count[\omega]$ will decrease by one. This gather-store and update procedure is shown in Algorithm 3 (line 56–59). The conversion iterates until an empty tracker appears. Once a tracker is empty, feeding or stealing, which we explain later, will be triggered to reset it. Then, this gather-store and update procedure would carry on until the next empty tracker appears.

[1] On KNL, the number of trackers is 8 for double precision and 16 for single precision. We use 4 in the figure only for simplicity.

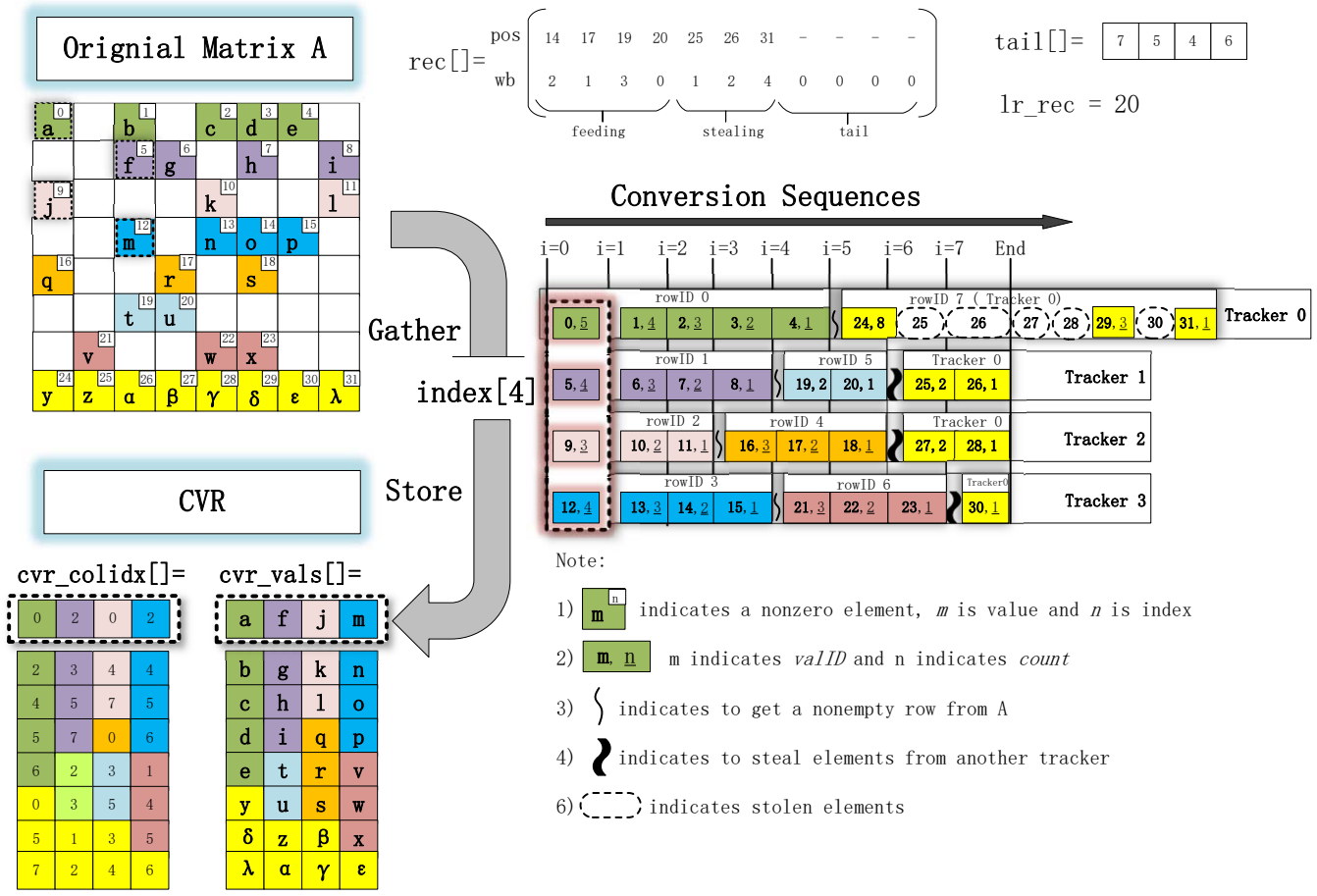


Figure 3. Converting a sparse matrix A to CVR.

Tracker Feeding As aforementioned, when a tracker becomes empty (i.e., its *count* is zero), it needs to be filled by a new row in matrix A. We name this procedure as feeding, which means feed a new row in matrix A to the empty tracker. We choose the first unprocessed row in matrix A to be the candidate row. In Figure 3, tracker 2 is initialized by row 2. After three times conversion, tracker 2 becomes empty. Row 4 is the first processed row and thus in turn fed into tracker 2. Similar to the initialization procedure, *rowID*, *valID*, and *count* are reset accordingly. After all empty trackers are filled, the conversion procedure continues. In Algorithm 3, *rs* denotes the index of the candidate row and line 16-21 describe the feeding procedure.

Tracker Stealing If there is no row left in matrix A to feed the empty tracker but there still have nonzero elements in other trackers, the empty tracker will steal elements from non-empty trackers for balance. This is a rare case.

We first sum up the number of elements left in all the trackers with the *reduce_add* SIMD instruction, and divide the sum by ω to get the *average*, which is the number of elements the empty tracker needs to steal. Then we traverse all the trackers to find out the tracker that has elements more than *average*. We name this tracker the candidate to steal from. We use *candi* to indicate the candidate tracker and use *em* to indicate the empty tracker. Tracker *em* will take the first *average* elements from tracker *candi*. Tracker *em* will set the (*rowID*[*em*], *valID*[*em*], *count*[*em*]) to (*candi*, *valID*[*candi*], average), and tracker *candi* will change the (*rowID*[*candi*], *valID*[*candi*], *count*[*candi*]) to (*candi*, *valID*[*candi*] + average, *count*[*candi*] - average). Note that *rowID*s of both tracker *em* and *candi* has changed to be the index of tracker *candi* (i.e., *candi*). In Figure 3, tracker 1, 2, 3 steal nonzero elements from tracker 0 for balance. Algorithm 3 shows this procedure in line 29–43.

Recording An important part of the conversion is to record where the reduction result should be written back into vector *y*. We introduce a new vector *rec* to store this information. When a tracker becomes empty (i.e., its *count* equals zero), we need to record two information: *pos*, which indicates the current position of this tracker in CVR (i.e., cvr_vals) and *wb*, which records the write-back location. *pos* is computed as (i×ω+*em*), where *em* denotes the index of the empty tracker

Algorithm 3 Format Conversion of CVR.

```
1:  seg_idx ← 0
2:  lr_rec ← −1
3:  tail[] ← −1
4:
5:  for i = 0 to (nnz/ω) − 1 in parallel do
6:      if !(vector_reduceAnd(count)) then
7:          for em = 0 to ω − 1 do
8:              if !count[em] then
9:                  if rs <= last_row then
10:                     if rs == first_row then
11:                         fr_rec ← i × ω + em
12:                     else
13:                         rec[seg_idx].pos ← i × ω + em
14:                         rec[seg_idx].wb ← rowID[em]
15:                     end if
16:                     while (row_ptr[rs + 1] − row_ptr[rs]) == 0 do
17:                         Inc(rs)
18:                     end while
19:                     valID[em] ← row_ptr[rs]
20:                     count[em] ← row_ptr[rs + 1] − row_ptr[rs]
21:                     rowID[em] ← rs
22:                     if rs == last_row then
23:                         tail[ω] ← rowID[ω]
24:                     end if
25:                     Inc(rs)
26:                 else
27:                     if lr_rec == −1 then
28:                         lr_rec ← i ∗ ω + em
29:                     end if
30:                     average ← reduceAdd(count)/ω
31:                     candi ← 0
32:                     for k = 0 to ω − 1 do
33:                         if count[k] > average then
34:                             candi ← k
35:                         end if
36:                     end for
37:                     rec[seg_idx].pos ← i ∗ ω + em
38:                     rec[seg_idx].wb ← rowID
39:                     valID[em] ← valID[candi]
40:                     rowID[em] ← candi
41:                     count[em] ← average
42:                     valID[candi] ← valID[candi] + average
43:                     rowID[candi] ← candi
44:                     count[candi] ← count[candi] − average
45:                     if i == (nnz/ω − 1) then
46:                         for k = 0 to ω − 1 do
47:                             rec[seg_idx].pos ← −1
48:                             rec[seg_idx].wb ← rowID
49:                             seg_idx ← seg_idx + 1
50:                         end for
51:                     end if
52:                 end if
53:                 seg_idx ← seg_idx + 1
54:             end if
55:         end for
56:     else
57:         cvr_vals ← gather(valID, vals)
58:         cvr_colidx ← gather(valID, col_idx)
59:         vector_Inc(valID, 1)
60:         vector_Dec(count, 1)
61:     end if
62: end for
```

and i denotes the number of conversion sequences (shown in Figure 3). *wb* is simply set to be the *rowID* of this empty tracker (i.e., *rowID*[*em*]). All (*pos*, *wb*) pairs are stored into vector *rec*. Algorithm 3 (line 13-14) shows the corresponding pseudo code. As shown in Figure 3, vector *rec* is separated into three parts: feeding part, stealing part and tail part. Note that *wb* indicates a place in vector y in feeding part, and indicates the index of another tracker in stealing part. The *lr_rec* indicates the boundary between feeding part and

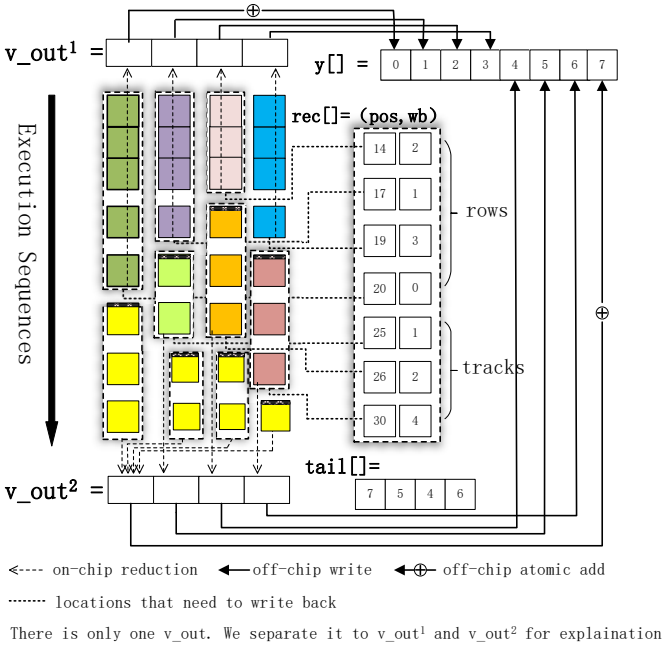


Figure 4. SpMV Implementation with CVR.

stealing part in vector *rec*. The tailing part in vector *rec* records the final values in *rowID*[ω]. We also introduce vector *tail* to record the values in *rowID*[ω] when the last time feeding is triggered (line 22–24 in Algorithm 3). The tailing part in vector *rec* is used to return the results back to the tracker where it steals from. Vector *tail* is involved in writing the final result back to vector y.

5 SpMV with CVR

We show the CVR-based SpMV algorithm in Figure 4 and illustrate it using Algorithm 4. In the figure, *v_out* stores the temporary value of reduction. We split *v_out* into v_out^1 and v_out^2 for better explanation in Figure 4, but there is only one *v_out* in our algorithm. *rec_idx* is used to keep the index of vector *rec*, which contains a two-element tuple (*pos*, *wb*). Each row of CVR format contains ω elements, which are multiplied and added to *v_out* with a single *fmadd* SIMD instruction. We perform this procedure iteratively until it reaches the position indicated by *rec*[*rec_idx*].*pos*. Then, the value in *v_out*[*pos* %ω] needs to be written back to the place indicated by *rec*[*rec_idx*].*wb*. After that, the value in *v_out*[*pos* %ω] is set to zero. We introduce vector *t_result* to store temporary results in the stealing part. The field *rec*[*rec_idx*].*wb* indicates an index in vector y in the feeding part and indicates the index of vector *t_result* in the stealing part. Then the multiply-add procedure continues until it comes across the next position that needs writing a value back. We describe the feeding part and the stealing part separately in Figure 4. In the feeding part, values are reduced to vector *v_out* and written back to vector y directly

(line 6–11 in Algorithm 4). In the stealing part, values are reduced to vector *v_out* and added to the corresponding element of another temporary vector *t_result* (line 14–19 in Algorithm 4). At last, values in vector *t_result* would be written back to vector *y* using the indices provided by vector *tail*.

Algorithm 4 SPMV with the CVR format.

```
1:  vec_out[ω] ← 0
2:  rec_idx ← 0
3:  t_result ← 0
4:  for i = 0; i < nnz − ω; i+ = ω do
5:      if i < lr_rec then
6:          while (rec[rec_idx].pos/ω) == (i/ω) do
7:              offset ← rec[rec_idx].pos%ω
8:              wb ← rec[rec_idx].wb
9:              y[wb] ← v_out.elems[offset]
10:             v_out.elems[offset] ← 0
11:             rec_idx ← rec_idx + 1
12:         end while
13:     else
14:         while (rec[rec_idx].pos/ω) == (i/ω) do
15:             offset ← rec[rec_idx].pos%ω
16:             wb ← rec[rec_idx].wb
17:             t_result[wb] ← t_result[wb] + v_out.elems[offset]
18:             v_out.elems[offset] ← 0
19:             rec_idx ← rec_idx + 1
20:         end while
21:     end if
22:     if i%16 == 0 then
23:         v_cols ← load(cvr_colidx + i)
24:     else
25:         v_cols ← permute(cvr_colidx, _MM_PERM_BADC)
26:     end if
27:     v_x = gather(cvr_colidx, x)
28:     v_vals = load(cvr_vals, i)
29:     v_out.reg = fmadd(cvr_vals, v_x, v_out)
30: end for
31: for k = 0 to ω − 1 do
32:     y[tail[k]] = t_result[k]
33: end for
```

6 Experimental Methodology

6.1 Environment Setup

We conduct experiments on a Intel Xeon Phi (Knights Landing 7250) based platform, which has 68 cores and employs 16GB MCDRAM and 96GB DDR4, running CentOS 7.3 operating system. The MCDRAM on Xeon Phi has three modes: cache mode, hybrid modes, and flat modes. In order to use high bandwidth, we configure the MCDRAM in the flat mode and use *numactl* to place all data in MCDRAM. An Intel C/C++ compiler version 17.04 is used for compiling with the flag '–MIC–AVX512 –O3'.

6.2 Formats for Comparison

We compare our method with the following five existing work on SpMV. Each of them proposes a new format for SpMV. We will use the format name to refer the corresponding work.

- **CSR (Intel MKL)** [35], which has been incorporated into Intel MKL (Math Kernel Library) as default format for SpMV, is widely used in matrix computation and commonly adopted by many works for comparison.

Table 2. The real-world datasets.

Area	Scenario	Dataset	Dimensions	nnz	nnz/row
Scale-free	web graph	web-Google	916K*916K	5.10M	5
		web-Stanford	281K*281K	2.31M	8
		com-youtube	1.15M*1.15M	2.98M	2
		amazon	400K*400K	3.20M	7
		IMDB	428K * 896K	3.78M	8
		NotreDame_actors	392K*127K	1.47M	3
		webbase-1M	1M*1M	3.10M	3
		hollywood2009	1.1M*1.1M	113.9M	99
		connectus	0.5K*395K	1.13M	2202
		digg.com	12K*872K	22.6M	1814
	Social Network	com-orkut	3M*3M	117M	38
		soc-pokec	1.63M*1.63M	30.6M	18
		soc-livejournal	4.85M*4.85M	68.99M	14
		flickr	820K*820K	9.84M	11
		soc-sign-epinions	131K*131K	0.84M	6
		soc-facebook-konect	59.2M*59.2M	92.5M	1
		higgs-twitter	456K*456K	14.8M	32
	Wiki	wikipedia2009	1.86M*1.86M	4.50M	2
		wiki-talk	2.39M*2.39M	5.02M	2
		wiki-topcats	1.79M*1.79M	28.5M	15
	Citation	com-DBLP	317K*317K	1.05M	3
		patents	6M*6M	16.5M	2
		citationCiteseer	268K*268K	1.15M	4
		coPapersCiteseer	434K*434K	16.03M	36
	Road	road_central	14M*14M	16.9M	1
		road_USA	23.9M*23.9M	28.8M	1
		roadNet-CA	1.97M*1.97M	5.53M	2
		rail4284	4.2K*1.1M	11.3M	2633
	Routing	as-skitter	1.69M*1.69M	22.2M	13
	FSM	language	399K*399K	1.22M	3
HPC	Engineer Scientific	dense4k	4K*4K	16.77M	4096
		FEM/Accelerator	121K*121K	2.62M	21
		FEM/Harbor	46K*46K	2.37M	50
		FEM/Ship	140K*140K	3.97M	28
		FEM/Cantilever	62K*62K	4.00M	64
		FEM/Spheres	83K*83K	6.01M	72
		Ga41As41H72	268K*268K	9.38M	34
		Si41Ge41H72	185K*185K	7.59M	40
		dc2	116K*116K	0.76M	6
		ins2	309K*309K	1.53M	4
		Epidemiology	525K*525K	2.10M	3
		Economics	206K*206K	1.27M	6
		rajat31	4.69M*4.69M	20.3M	4
		circuit5M	5.5M*5.5M	59.5M	10
		cage15	5.15M*5.15M	99.2M	19
		mip1	66K*66K	5.21M	78
		Wind Tunnel	217K*217K	5.92M	27
		bone010	986K*986K	36.3M	36
		ASIC_680k	682K*682K	3.87M	5
		Circuit	170K*170K	0.96M	5
		fullchip	2.98M*2.98M	26.6M	8
		Rucci1	1.97M*110K	7.79M	3
		spal_004	10K*322K	46.1M	4524
		ldoor	952K*952K	23.73M	24
		Protein	36K*36K	2.19M	60
		mouse_gene	45K*45K	14.5M	321
		human_gene2	14K*14K	9.04M	630
		crankseg_2	63K*63K	14.14M	221

- **CSR(I)** [35] is a SpMV implementation in Intel MKL SpMV Format Prototype Package. It converts a classic CSR into an internal CSR, and then conducts the SpMV computation in an Inspector-Executor way. For each matrix in our experiments, we run all three schedule policies of CSR(I) and choose the best one.
- **ESB** [21] is an evolved version of ELLPACK, and also included in Intel SpMV Format Prototype Package. For each matrix in our experiments, we run all three schedule policies of ESB and choose the best one.
- **VHCC** [30] converts the classic CSR format through a 2D jagged partition and uses segmented sum to conduct the SpMV computation. We run all possible panel numbers in VHCC for each matrix and choose the one having the best performance.
- **CSR5** [19] converts the classic CSR into tiles and uses segmented sum to conduct the SpMV computation. We run it using the default tile size provided in its code.

We evaluate SpMV with double precision. For all experiments, we run 1000 iterations of SpMV and record both the preprocessing time and the average single-iteration execution time. The impact of SMT threads is considered in our experiment. For all SpMV implementations, we adjust the SMT threads number (from 1 to 4) and report the best one. The locality experiments are conducted with the best configuration (thread number, number of panels in VHCC, schedule policy in CSR(I) and ESB, and tiling size in CSR5).

6.3 Real-world Data Sets

The input data sets we used for evaluation include 30 scale-free and 28 HPC sparse matrices, which are listed in Table 2. These matrices are collected from the Stanford Network Analysis Platform (SNAP)[17] and the University of Florida Sparse Matrix Collection[9]. They cover most sparse matrices used in previous researches in SpMV optimization work [19, 21, 30]. In addition, they own more scale-free matrices[6, 36] covering various domains: web graphs, social networks, wiki networks, citation networks, road networks, routing network, and a finite state machine graph for language processing. In our following experiments, we group these 58 sparse matrices by application domains and discuss the sparse matrices according to their application domains.

7 Experimental Results

In this section, we first evaluate the performance of SpMV implementation with different formats. We then analyze the preprocessing overhead of each format (i.e., format conversion overhead). We consider not only the preprocessing but also SpMV execution to reveal the overall performance under iteration-based scenarios. In general, we group the sparse matrices by application domains and show the average performance results.

Table 3. Summary of Isolated SpMV Performance (GFlop/s).

	MKL	CSR(I)	ESB	VHCC	CSR5	CVR	S-1	S-2
web Graph	2.92	4.59	2.55	5.48	5.78	**7.28**	1.26	2.50
social network	2.40	3.22	1.71	2.40	4.92	**6.59**	1.34	2.74
wiki	1.06	2.64	0.90	2.90	3.99	**5.77**	1.45	5.43
citation	4.53	2.73	2.75	3.25	4.55	**6.26**	1.38	1.38
road	4.24	4.29	5.21	5.60	7.00	**9.57**	1.37	2.26
routing	2.73	7.23	2.97	8.29	12.29	**17.11**	1.39	6.27
FSM	2.01	4.64	0.79	4.30	5.33	**8.09**	1.52	4.02
ES[3]	17.09	17.24	13.46	15.07	19.24	**21.11**	1.10	1.24

[1] S-1 is the speedup of CVR over the second best method.
[2] S-2 is the speedup of CVR over MKL.
[3] ES stands for 'Engineering Scientific'

7.1 SpMV Performance

Table 3 summarizes the average SpMV performance in GFlops (Giga floating-point operations per second) for each application domain. Figure 5 shows the SpMV performance of each data set in detail and is organized according to application domains:

- Figure 5 (a): web graph matrices,
- Figure 5 (b): social network and wiki matrices,
- Figure 5 (c): road, citation, routing and FSM matrices,
- Figure 5 (d, e, f): engineering scientific matrices of HPC.

On average, the CVR-based SpMV implementation delivers the highest throughput on each application domain, especially on scale-free matrices. The throughput of ESB is lower than Intel MKL for many matrices. Blocking and padding in ESB are not effective for sparse matrices, especially for scale-free sparse matrices. SpMV delivers much lower throughput on scale-free matrices than on engineering scientific matrices, mainly due to the high sparsity and irregularity in scale-free matrices.

We show the speedups of CVR over MKL and over the second best work in Table 3. For scale-free matrices, CVR gives performance improvement up to 6.27× over Intel MKL. This is mainly due to the vectorization-oriented data layout of CVR, which can guarantee better locality and thus higher throughput. As to HPC matrices, CVR also shows moderate average speedups: 1.24× over MKL and 1.10× over the second best. *wiki-talk* in Figure 5(b) achieves the best speedup over MKL (up to 39.53×) in scale-free matrices, and *ins2* in Figure 5(e) shows the best speedup over Intel MKL (up to 17.58×) in HPC matrices. The main reason is that adjacent rows in these two matrices are following a similar pattern, which makes CVR much easier to exploit the locality.

In Figure 5, VHCC shows the best SpMV performance on some rectangle matrices, which has much fewer rows than columns, like *connectus*, *12month1* and *rail4284*. These three matrices have much more nonzero elements per row than any other matrices. VHCC achieves good SpMV performance mostly because of its 2D jagged partition mechanism. However, this partition mechanism introduces high preprocessing overhead, which is shown in Section 7.2.

Table 4. Iterations that need to amortize the Format-conversion overhead.

Scenario	Dataset	CSR(I)	ESB	VHCC	CSR5	CVR
web-Graph	web-Google	∞	∞	21364	309	**8.4**
	web-Stanford	∞	∞	∞	98	**25**
	com-youtube	33	∞	728	1.6	**0.9**
	amazon	∞	8596	∞	∞	**33**
	IMDB	769	∞	7912	39	**5**
	NotreDame_actors	863	∞	7350	13	**11**
	webbase-1M	271	∞	3683	**6.05**	6.64
	hollywood2009	11	41	1519	1.15	**0.58**
	connectus	19	∞	2300	**5.13**	5.51
	digg.com	49	∞	2653	2.41	**1.65**
SocialNetwork	com-orkut	22	104	∞	2.12	**0.58**
	soc-pokec	∞	∞	∞	20	**2.9**
	soc-livejournal	18	250	2237	2.02	**0.47**
	flickr	89	∞	2255	5.36	**1.67**
	soc-sign-epinions	209	∞	4060	10	**7.06**
	soc-facebook-konect	773	∞	∞	7.7	**1.93**
	higgs-twitter	219	∞	21396	14	**4.89**
Wiki	wikipedia2009	38	∞	1157	1.59	**0.67**
	wiki-talk	399	4.98	68	2.95	**0.27**
	wiki-topcats	39	∞	3156	2.26	**0.93**
Citation	com-DBLP	∞	∞	33072	58	**15**
	patents	∞	∞	2542	1.94	**0.71**
	citationCiteseer	∞	∞	22953	48	**13**
	coPapersCiteseer	∞	∞	∞	∞	**25**
Road	road_central	∞	285	∞	4.83	**1.64**
	road_USA	∞	360	36970	6.59	**2.14**
	roadNet-CA	∞	2924	∞	39	**18**
	rail4284	16	161	1565	**1.42**	1.9
Routing	as-skitter	26	373	1419	1.76	**0.81**
FSM	language	201	∞	3502	**5.61**	9.1
HPC	dense4k	1538	4896	112464	180	**79**
	FEM/Accelerator	∞	∞	∞	158	**35**
	FEM/Harbor	∞	∞	∞	∞	∞
	FEM/Ship	∞	∞	∞	∞	**509**
	FEM/Cantilever	∞	∞	∞	∞	∞
	FEM/Spheres	∞	∞	∞	∞	∞
	Ga41As41H72	∞	∞	∞	∞	**76**
	Si41Ge41H72	∞	∞	∞	∞	**65**
	dc2	29	∞	1534	**2.67**	3.53
	ins2	15	∞	405	1.28	**1.27**
	Epidemiology	∞	1380	∞	**27**	89
	Economics	∞	∞	∞	**67**	429
	rajat31	∞	2712	∞	**36**	46
	circuit5M	17	∞	1253	1.01	**0.5**
	cage15	∞	∞	∞	173	**35**
	mip1	1311	∞	∞	50	**24**
	Wind Tunnel	∞	∞	∞	∞	**686**
	bone010	∞	∞	∞	∞	∞
	ASIC_680k	114	∞	3009	**4.83**	6.45
	Circuit	37881	∞	∞	**41**	105
	fullchip	21	∞	1082	0.964	**0.66**
	Rucci1	∞	1310	∞	**25**	40
	spal_004	0	0	37324	∞	**15**
	ldoor	∞	∞	∞	668	**58**
	Protein	7025	∞	∞	∞	**408**
	mouse_gene	682	∞	113355	**50**	281
	human_gene2	828	∞	118588	77	**64**
	crankseg_2	2166	∞	∞	1363	**54**

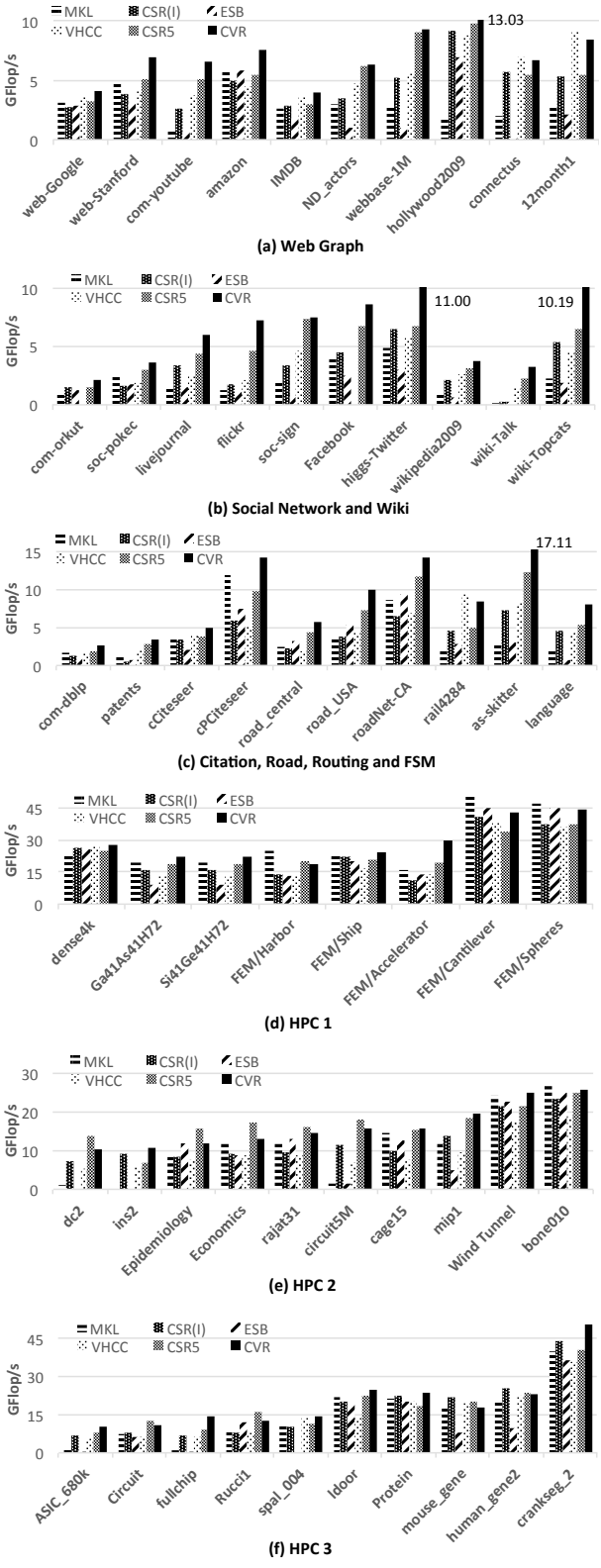


Figure 5. Comparison of the SpMV Performance.

7.2 Preprocessing Overhead

Real world applications consider not only the SpMV performance of each iteration, but also the preprocessing overhead, so methods with lower preprocessing overhead are more practical. We analyze the preprocessing overheads by using metric I_{pre}, which is the number of iterations that can amortize the overhead and be calculated by Equation 1.

$$I_{pre} = \frac{T_{pre}}{(T_{spmv}^{MKL} - T_{spmv}^{new})} \quad (1)$$

Where T_{pre} is the preprocessing time; T_{spmv}^{MKL} is the SpMV time using Intel MKL routine and T_{spmv}^{new} is the SpMV time using other comparing formats. We list the I_{pre} in Table 4. Since we use the execution numbers of Intel MKL SpMV routine as the baseline to quantify the overhead, if one SpMV implementation's performance for certain data set is worse than Intel MKL, the I_{pre} will be ∞, which means its SpMV execution time is longer than that of MKL. Smaller I_{pre} means less preprocessing overhead.

CVR achieves the lowest preprocessing overhead in 26 out of 30 scale-free matrices. For many matrices, CVR needs less than ten iterations to amortize the preprocessing overhead. Methods with blocking, such as CSR(I), ESB, and VHCC have much larger preprocessing overhead that needs hundreds or even thousands of iterations to amortize.

7.3 Overall Performance

We consider both preprocessing overhead and SpMV performance to reveal its overall performance in the iteration-based scenarios. Here we use Intel MKL as the baseline to compute the speedup by Equation 2.

$$Speedup = \frac{nT_{spmv}^{mkl}}{(T_{pre} + nT_{spmv}^{new})} \quad (2)$$

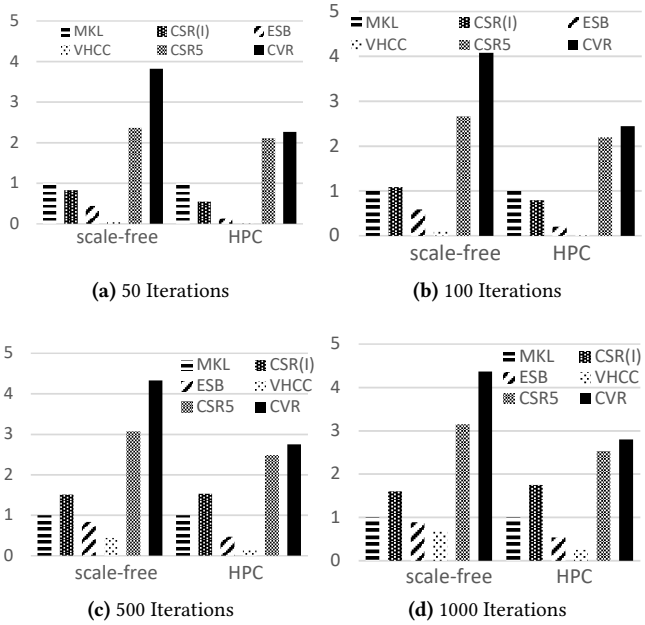
Where n is the number of iterations, nT_{spmv}^{new} is the total SpMV time with new format executed n iterations and nT_{spmv}^{mkl} is the total SpMV time of Intel MKL running n iterations.

Figure 6 shows the average speedup of all matrices over Intel MKL while n equals 50, 100, 500, and 1000, respectively. Our method achieves the best performance over other methods and achieves nearly 3× speedup over Intel MKL. CSR(I) is slower than Intel MKL when n is less than 100, but better than Intel MKL when n is larger. The performance of VHCC is the worst, due to its high preprocessing overhead.

7.4 Locality Analysis

We characterize the cache locality of different SpMV implementations by investigating the miss ratio of L2 cache, which is also the last level cache on our platform. Figure 7 shows the L2 cache miss ratio of each application domain.

SpMV shows much higher L2 cache miss ratio on scale-free matrices than on HPC matrices, mainly due to the irregular sparsity pattern in scale-free matrices. ESB shows the worst locality in all application domains, because the blocking method is not suitable for highly irregular sparse matrices. Our SpMV implementation with CVR format shows an order of magnitude lower L2 cache miss ratios than other formats on all application domains. The main reason of better locality is that CVR can improve the reuse of vector x extensively by processing multiple rows together.



(a) 50 Iterations **(b)** 100 Iterations **(c)** 500 Iterations **(d)** 1000 Iterations

Figure 6. Overall Performance Speedup over MKL.

8 Related Work

SpMV has been studied for decades on various architectures from diverse prospects [8, 10, 11, 13, 15, 20, 22–24, 31, 34, 38, 41]. Many new formats, libraries, and auto-tuning frameworks have been proposed to improve the computational efficiency of SpMV [33].

Targeting at scale-free matrices, VHCC[30] employs 2D jagged partition and the segmented sum for better memory and vectorization efficiency. For engineering scientific sparse matrices, Liu et al. [21] propose ESB to decrease the padding in ELLPACK[2] by sorting and blocking the rows. Designed for multiple platforms, CSR5 [19] can achieve moderate performance on x86-based platforms. Chen et al. [7] present a method for irregular applications, by comparing the nonzero elements one by one to improve vectorization efficiency. These approaches can achieve good performance for HPC matrices, but not for scale-free matrices. Ashari et al. [1] present a blocked row-column (BRC) storage format with a two-dimensional blocking mechanism. Yan et al. [39] present a new format, called blocked compressed common coordinate (BCCOO), which uses bit flags to store

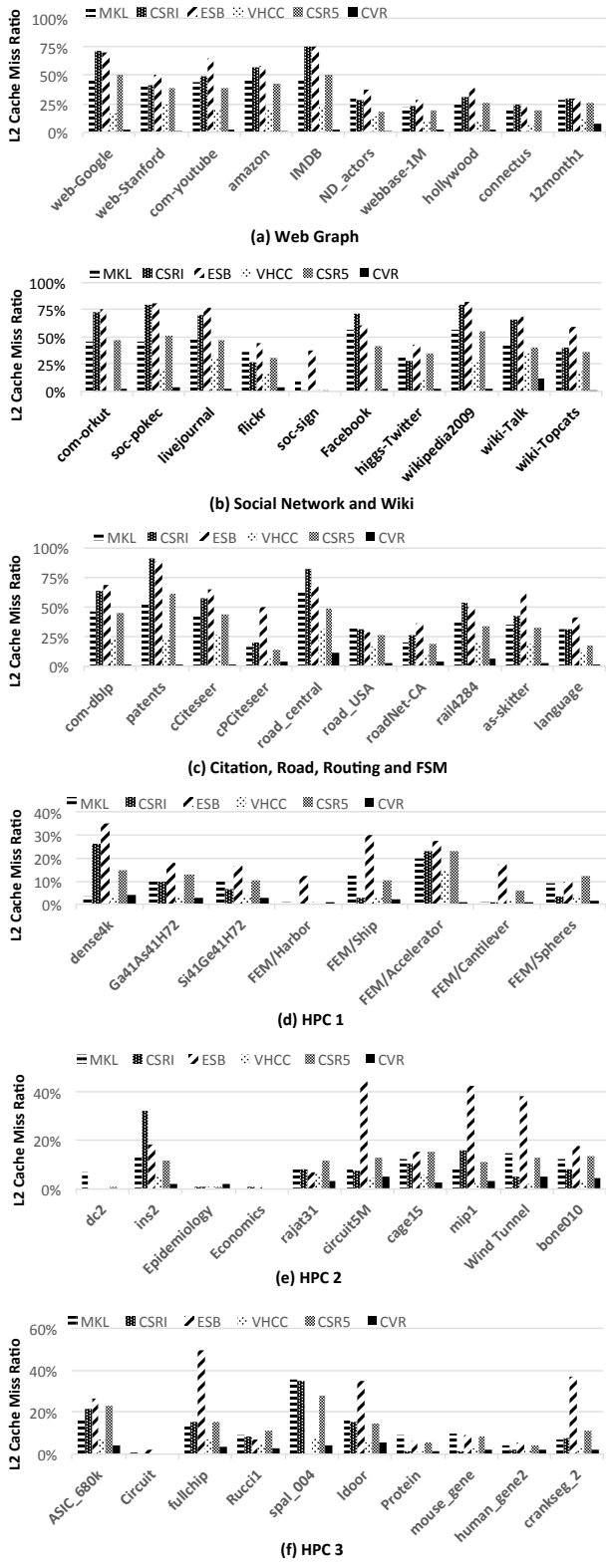


Figure 7. L2 Cache Miss Ratio.

the row indices in a blocked common coordinate (COO) format so as to alleviate the bandwidth congestion. Greathouse et al. [12] present CSR-adaptive, aiming at better load balance and memory reference efficiency. Choi et al. [8] present a model-driven framework for automated performance tuning of SpMV on systems accelerated by graphics processing units (GPU). Kourtis et al. [16] present a new format called Compressed Sparse eXtended (CSX) to exploit the substructures of a sparse matrix. Liu et al. [20] present a method for SpMV on CPU+GPU using speculative segmented sum. Buono et al. [6] find that scale-free sparse matrices show different characteristics from HPC sparse matrices. Based on that, they design a two-phase method to optimize SpMV for scale-free graphs on POWER8. In contrast, our CVR format aims at better memory access and vectorization efficiency for both scale-free and HPC sparse matrices.

As the SpMV formats show various performances on sparse matrices with different sparsity patterns, there are some auto-tuning frameworks have been proposed to select the optimal format for a specific sparse matrix. SMAT [18] can automatically select the best combination of the matrix format and its optimal implementation by learning a large set of sparse matrices. Sedaghati et al. [26] take both the sparsity pattern and the platform features into consideration to select the best sparse matrix representation automatically. Su et al. [28] present Cocktail format to combine multiple formats together and also propose a cross-platform openCL SpMV framework called clSpMV to recommend the best format on different platforms. There are also approaches on SpMV in graph computing [3, 5, 40, 42], which aim at increasing SpMV throughputs on scale-free graphs with large sizes. Many graph frameworks, such as GraphMat [29], GraphLab [5], and GraphTwist [43] abstract graph algorithms to SpMV operations. Our CVR format can serve as a novel format to be implemented in these frameworks.

9 Conclusions

In this paper, we propose the CVR format for SpMV, aiming at better cache locality and better vectorization efficiency. The preprocessing overhead of CVR is pretty low, since our tracker-based conversion method is vectorization friendly and insensitive to the sparsity structure of the input matrix. Our CVR-based SpMV implementation simplifies the SpMV computation by conducting consecutive reductions. Compared with five other commercial and state-of-the-art approaches, experiment results show that our work typically has the best SpMV performance and lowest preprocessing overhead on 58 popular matrices. CVR also improves the memory reference efficiency significantly. Our experiments on locality analysis show that CVR achieves an order of magnitude lower L2 cache miss ratio than other formats on average.

A Artifact Description

A.1 Abstract

The artifact contains the source code of SpMV using CVR after converting a sparse matrix from CSR to CVR. It can support the experiment results in section 8 of our CGO'2018 paper **CVR: Efficient SpMV Vectorization on X86 Processors**. To validate the results, build CVR and run the benchmarks with provided scripts. This artifact provides general instructions to evaluate CVR, while more details are provided on **Github** (https://github.com/puckbee/CVR).

A.2 Description

A.2.1 Check-list (artifact meta information)

- **Algorithm:** SpMV (Sparse Matrix-vector Multiply)
- **Program:** CVR
- **Compilation:** icpc 17.04
- **Data set:** sparse matrices with matrix market format
- **Run-time environment:** CentOS 7.3, numactl, openMP
- **Hardware:** Intel Xeon Phi processors (Knights Landing)
- **Run-time state:** SMT enabled; 'flat mode' of MCDRAM enabled
- **Output:** running time of preprocessing and SpMV; cache related performance results
- **Experiment workflow:** git clone; build cvr; prepare the data set; run benchmarks; observe performance results(execution time and PMU results)
- **Publicly available?:** Yes

A.2.2 How Delivered

The source code, scripts, and instructions are hosted on **Github** at: https://github.com/puckbee/CVR.

A.2.3 Hardware Dependencies

The method of CVR can be applied to any platforms which support vectorization, however it only supports Xeon Phi currently. It can be directly deployed on Xeon Phi(Knights Landing). The MCDRAM of Knights Landing should be configured to be 'flat mode'. The detail description of configuration should refer to the user guide of Intel Xeon Phi (Knights Landing).

A.2.4 Software Dependencies

Our CVR implementation requires standard C/C++ environment with OpenMP. The Intel Compiler icc/icpc that supports avx512, is also necessary. We also need numactl tools to run the program in MCDRAM.

A.2.5 Datasets

Our CVR implementation supports matrix market format, one of the standard formats used for storing sparse matrices. All datasets in our paper can be found in either of these two collections:

1) SuiteSparse Matrix Collection (formerly the University of Florida Sparse Matrix Collection): https://sparse.tamu.edu

2) Stanford Large Network Dataset Collection (SNAP): http://snap.stanford.edu/data

A.3 Installation

After cloning the repository, Makefiles are provided. A simple 'make' would be all to build CVR. No installation is needed. As to other formats/solutions for comparison, we provide **build.sh** for building.

A.4 Experiment Workflow

After building CVR, provided scripts should be used for running all benchmarks. For CVR, we provide **run_sample.sh** for demonstration. For comparison with other formats/solutions, we provide **run_comparison.sh**. To profile the cache performance, we provide **run_locality.sh**. Detailed descriptions of the experiment workflow are provided on the **Github** page.

A.5 Evaluation and Expected Result

The main results of the artifact evaluation is to reproduce the performance comparison given in Section 8 of the paper. The reviewers are invited to investigate the implementation of the CVR and evaluate it against the description given in the paper.

CVR outputs two time values: [pre-processing time] and [SpMV execution time]. 'pre-processing time' is the time of converting a sparse matrix with CSR format to CVR format. 'SpMV execution time' is the average time of running a fixed number of iterations of SpMV with CVR format. We calculate the throughput(Gflops) using the SpMV execution time.

We also elaborate how to compare other formats/solutions(CSR5, VHCC and etc.) with CVR on the **Github** page.

A.6 Experiment Customization

Users can evaluate CVR on more extensive sparse matrices, except for the 58 sparse matrices used in this paper, without changing the source code.

A.7 Notes

If you have any issues or questions related to our proposed idea or the artifact, please contact us. We are planning to implement CVR on GPGPU using CUDA and openCL. Further progresses would be updated on our **Github** page at https://github.com/puckbee/CVR.

Acknowledgments

This work is partially supported by the Major Program of National Natural Science Foundation of China (Grant No. 61432006), the National Key Research and Development Plan of China (Grant No. 2016YFB1000600 and 2016YFB1000601), and the National Science Foundation (NSF) under Grant No. 1618620. The authors are very grateful to anonymous reviewers for their insightful feedback. The authors wish to thank

Beijing San Kuai Yun Technology Co., Ltd for providing the Knights Landing machine and their valuable technical assistance. We also thank Weifeng Liu (KU) for sharing the source code for evaluation. Jianfeng Zhan is the corresponding author.

References

[1] Arash Ashari, Naser Sedaghati, John Eisenlohr, and P. Sadayappan. 2014. An Efficient Two-dimensional Blocking Strategy for Sparse Matrix-vector Multiplication on GPUs. In *Proceedings of the 28th ACM International Conference on Supercomputing (ICS '14)*. ACM, New York, NY, USA, 273–282. https://doi.org/10.1145/2597652.2597678

[2] Nathan Bell and Michael Garland. 2009. Implementing Sparse Matrix-vector Multiplication on Throughput-oriented Processors. In *Proceedings of the ACM/IEEE Conference on High Performance Computing Networking, Storage and Analysis (SC '09)*. ACM, New York, NY, USA, Article 18, 11 pages. https://doi.org/10.1145/1654059.1654078

[3] Maciej Besta, Florian Marending, Edgar Solomonik, and Torsten Hoefler. 2017. SlimSell: A Vectorizable Graph Representation for Breadth-First Search. In *Proceedings of the 31st IEEE International Parallel and Distributed Processing Symposium (IPDPS '17)*. 32–41. https://doi.org/10.1109/IPDPS.2017.93

[4] Guy E. Blelloch, Michael A. Heroux, and Marco Zagha. 1993. *Segmented Operations for Sparse Matrix Computation on Vector Multiprocessors*. Technical Report. Pittsburgh, PA, USA.

[5] Erik G. Boman, Karen D. Devine, and Sivasankaran Rajamanickam. 2013. Scalable Matrix Computations on Large Scale-free Graphs Using 2D Graph Partitioning. In *Proceedings of the ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13)*. ACM, New York, NY, USA, Article 50, 12 pages. https://doi.org/10.1145/2503210.2503293

[6] Daniele Buono, Fabrizio Petrini, Fabio Checconi, Xing Liu, Xinyu Que, Chris Long, and Tai-Ching Tuan. 2016. Optimizing Sparse Matrix-Vector Multiplication for Large-Scale Data Analytics. In *Proceedings of the 30th International Conference on Supercomputing (ICS '16)*. ACM, New York, NY, USA, Article 37, 12 pages. https://doi.org/10.1145/2925426.2926278

[7] Linchuan Chen, Peng Jiang, and Gagan Agrawal. 2016. Exploiting Recent SIMD Architectural Advances for Irregular Applications. In *Proceedings of the 14th International Symposium on Code Generation and Optimization (CGO '16)*. ACM, New York, NY, USA, 47–58. https://doi.org/10.1145/2854038.2854046

[8] Jee W. Choi, Amik Singh, and Richard W. Vuduc. 2010. Model-driven Autotuning of Sparse Matrix-vector Multiply on GPUs. *SIGPLAN Not.* 45, 5 (Jan. 2010), 115–126. https://doi.org/10.1145/1837853.1693471

[9] Timothy A. Davis. 1997. The University of Florida sparse matrix collection. *NA DIGEST* (1997).

[10] Michael Garland. 2008. Sparse Matrix Computations on Manycore GPU's. In *Proceedings of the 45th Annual Design Automation Conference (DAC '08)*. ACM, New York, NY, USA, 2–6. https://doi.org/10.1145/1391469.1391473

[11] Georgios Goumas, Kornilios Kourtis, Nikos Anastopoulos, Vasileios Karakasis, and Nectarios Koziris. 2009. Performance evaluation of the sparse matrix-vector multiplication on modern architectures. *The Journal of Supercomputing* 50, 1 (01 Oct 2009), 36–77. https://doi.org/10.1007/s11227-008-0251-8

[12] Joseph L. Greathouse and Mayank Daga. 2014. Efficient Sparse Matrix-vector Multiplication on GPUs Using the CSR Storage Format. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14)*. IEEE Press, Piscataway, NJ, USA, 769–780. https://doi.org/10.1109/SC.2014.68

[13] Eun-Jin Im, Katherine Yelick, and Richard Vuduc. 2004. Sparsity: Optimization Framework for Sparse Matrix Kernels. *International Journal of High Performance Computing Applications* 18, 1 (Feb. 2004), 135–158. https://doi.org/10.1177/1094342004041296

[14] Zhen Jia, Lei Wang, Jianfeng Zhan, Lixin Zhang, and Chunjie Luo. 2013. Characterizing data analysis workloads in data centers. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC '13)*. 66–76. https://doi.org/10.1109/IISWC.2013.6704671

[15] Kornilios Kourtis, Georgios Goumas, and Nectarios Koziris. 2010. Exploiting Compression Opportunities to Improve SpMxV Performance on Shared Memory Systems. *The ACM Transactions on Architecture and Code Optimization* 7, 3, Article 16 (Dec. 2010), 31 pages. https://doi.org/10.1145/1880037.1880041

[16] Kornilios Kourtis, Vasileios Karakasis, Georgios Goumas, and Nectarios Koziris. 2011. CSX: An Extended Compression Format for Spmv on Shared Memory Systems. *SIGPLAN Not.* 46, 8 (Feb. 2011), 247–256. https://doi.org/10.1145/2038037.1941587

[17] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data. (June 2014).

[18] Jiajia Li, Guangming Tan, Mingyu Chen, and Ninghui Sun. 2013. SMAT: An Input Adaptive Auto-tuner for Sparse Matrix-vector Multiplication. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 117–126. https://doi.org/10.1145/2462156.2462181

[19] Weifeng Liu and Brian Vinter. 2015. CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication. In *Proceedings of the 29th ACM International Conference on Supercomputing (ICS '15)*. ACM, New York, NY, USA, 339–350. https://doi.org/10.1145/2751205.2751209

[20] Weifeng Liu and Brian Vinter. 2015. Speculative Segmented Sum for Sparse Matrix-Vector Multiplication on Heterogeneous Processors. *Parallel Comput.* 49 (2015), 179 – 193. https://doi.org/10.1016/j.parco.2015.04.004

[21] Xing Liu, Mikhail Smelyanskiy, Edmond Chow, and Pradeep Dubey. 2013. Efficient Sparse Matrix-vector Multiplication on x86-based Many-core Processors. In *Proceedings of the 27th ACM International Conference on Supercomputing (ICS '13)*. ACM, New York, NY, USA, 273–282. https://doi.org/10.1145/2464996.2465013

[22] Duane Merrill and Michael Garland. 2016. Merge-based Parallel Sparse Matrix-vector Multiplication. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '16)*. IEEE, Piscataway, NJ, USA, Article 58, 12 pages. https://doi.org/10.1109/SC.2016.57

[23] Nguyen Quang Anh Pham, Rui Fan, and Yonggang Wen. 2015. Reducing Vector I/O for Faster GPU Sparse Matrix-Vector Multiplication. In *Proceedings of the 29th IEEE International Parallel and Distributed Processing Symposium (IPDPS '15)*. 1043–1052. https://doi.org/10.1109/IPDPS.2015.100

[24] Ali Pinar and Michael T. Heath. 1999. Improving Performance of Sparse Matrix-vector Multiplication. In *Proceedings of the 13rd ACM/IEEE Conference on Supercomputing (ICS '99)*. ACM, New York, NY, USA, Article 30. https://doi.org/10.1145/331532.331562

[25] Mahesh Ravishankar, Roshan Dathathri, Venmugil Elango, Louis-Noël Pouchet, J. Ramanujam, Atanas Rountev, and P. Sadayappan. 2015. Distributed Memory Code Generation for Mixed Irregular/Regular Computations. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '15)*. ACM, New York, NY, USA, 65–75. https://doi.org/10.1145/2688500.2688515

[26] Naser Sedaghati, Te Mu, Louis-Noel Pouchet, Srinivasan Parthasarathy, and P. Sadayappan. 2015. Automatic Selection of Sparse Matrix Representation on GPUs. In *Proceedings of the 29th ACM on International Conference on Supercomputing (ICS '15)*. ACM, New York, NY, USA, 99–108. https://doi.org/10.1145/2751205.2751244

[27] Avinash Sodani, Roger Gramunt, Jesus Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu. 2016. Knights Landing: Second-Generation Intel Xeon

Phi Product. *IEEE Micro* 36, 2 (2016), 34–46. https://doi.org/10.1109/MM.2016.25

[28] Bor-Yiing Su and Kurt Keutzer. 2012. clSpMV: A Cross-Platform OpenCL SpMV Framework on GPUs. In *Proceedings of the 26th ACM International Conference on Supercomputing (ICS '12)*. ACM, New York, NY, USA, 353–364. https://doi.org/10.1145/2304576.2304624

[29] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R. Dulloor, Michael J. Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. 2015. GraphMat: High Performance Graph Analytics Made Productive. *Proceedings of the VLDB Endowment* 8, 11 (July 2015), 1214–1225. https://doi.org/10.14778/2809974.2809983

[30] Wai Teng Tang, Ruizhe Zhao, Mian Lu, Yun Liang, Huynh Phung Huynh, Xibai Li, and Rick Siow Mong Goh. 2015. Optimizing and Auto-tuning Scale-free Sparse Matrix-vector Multiplication on Intel Xeon Phi. In *Proceedings of the 13th IEEE/ACM International Symposium on Code Generation and Optimization (CGO '15)*. IEEE Computer Society, Washington, DC, USA, 136–145. https://doi.org/10.1109/CGO.2015.7054194

[31] Yaman Umuroglu and Magnus Jahre. 2014. An energy efficient column-major backend for FPGA SpMV accelerators. In *Proceedings of the 32nd IEEE International Conference on Computer Design (ICCD '14)*. 432–439. https://doi.org/10.1109/ICCD.2014.6974716

[32] Anand Venkat, Mary Hall, and Michelle Strout. 2015. Loop and Data Transformations for Sparse Matrix Code. *SIGPLAN Not.* 50, 6 (June 2015), 521–532. https://doi.org/10.1145/2737924.2738003

[33] Richard Vuduc, James W Demmel, and Katherine A. Yelick. 2005. OSKI: A library of automatically tuned sparse matrix kernels. 16 (Jan. 2005), 521–530. https://doi.org/10.1088/1742-6596/16/1/071

[34] Richard Wilson Vuduc. 2003. *Automatic Performance Tuning of Sparse Matrix Kernels*. Ph.D. Dissertation. AAI3121741.

[35] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. 2014. *Intel Math Kernel Library*. Springer International Publishing, Cham, 167–188. https://doi.org/10.1007/978-3-319-06486-4_7

[36] Lei Wang, Fan Yang, Liangji Zhuang, Huimin Cui, Fang Lv, and Xiaobing Feng. 2016. Articulation Points Guided Redundancy Elimination for Betweenness Centrality. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '16)*. ACM, New York, NY, USA, Article 7, 13 pages. https://doi.org/10.1145/2851141.2851154

[37] Lei Wang, Jianfeng Zhan, Chunjie Luo, Yuqing Zhu, Qiang Yang, Yongqiang He, Wanling Gao, Zhen Jia, Yingjie Shi, Shujie Zhang, Chen Zheng, Gang Lu, Kent Zhan, Xiaona Li, and Bizhu Qiu. 2014. BigDataBench: A big data benchmark suite from internet services. In *Proceedings of the 20th IEEE International Symposium on High Performance Computer Architecture (HPCA '14)*. 488–499. https://doi.org/10.1109/HPCA.2014.6835958

[38] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. 2007. Optimization of Sparse Matrix-vector Multiplication on Emerging Multicore Platforms. In *Proceedings of the 21st ACM/IEEE Conference on Supercomputing (ICS '07)*. ACM, New York, NY, USA, Article 38, 12 pages. https://doi.org/10.1145/1362622.1362674

[39] Shengen Yan, Chao Li, Yunquan Zhang, and Huiyang Zhou. 2014. yaSpMV: Yet Another SpMV Framework on GPUs. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '14)*. ACM, New York, NY, USA, 107–118. https://doi.org/10.1145/2555243.2555255

[40] Xintian Yang, Srinivasan Parthasarathy, and P. Sadayappan. 2011. Fast Sparse Matrix-vector Multiplication on GPUs: Implications for Graph Mining. *Proceedings of the VLDB Endowment* 4, 4 (Jan. 2011), 231–242. https://doi.org/10.14778/1938545.1938548

[41] Leonid Yavits and Ran Ginosar. 2017. Accelerator for Sparse Machine Learning. *IEEE Computer Architecture Letters* PP, 99 (2017), 1–1. https://doi.org/10.1109/LCA.2017.2714667

[42] Andy Yoo, Allison H. Baker, Roger Pearce, and Van Emden Henson. 2011. A Scalable Eigensolver for Large Scale-free Graphs Using 2D Graph Partitioning. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '11)*. ACM, New York, NY, USA, Article 63, 11 pages. https://doi.org/10.1145/2063384.2063469

[43] Yang Zhou, Ling Liu, Kisung Lee, and Qi Zhang. 2015. GraphTwist: Fast Iterative Graph Computation with Two-tier Optimizations. *Proceedings of the VLDB Endowment* 8, 11 (July 2015), 1262–1273. https://doi.org/10.14778/2809974.2809987