Speeding Up SpMV for Power-Law Graph Analytics by Enhancing Locality & Vectorization

Serif Yesil
Dept. of Computer Science
University of Illinois at Urbana-Champaign
syesil2@illinois.edu

Azin Heidarshenas
Dept. of Computer Science
University of Illinois at Urbana-Champaign
heidars2@illinois.edu

Adam Morrison
Blavatnik School of Computer Science
Tel Aviv University
mad@cs.tau.ac.il

Josep Torrellas
Dept. of Computer Science
University of Illinois at Urbana-Champaign
torrella@illinois.edu

***Abstract*—Graph analytics applications often target large-scale web and social networks, which are typically power-law graphs. Graph algorithms can often be recast as generalized Sparse Matrix-Vector multiplication (SpMV) operations, making SpMV optimization important for graph analytics. However, executing SpMV on large-scale power-law graphs results in highly irregular memory access patterns with poor cache utilization. Worse, we find that existing SpMV locality and vectorization optimizations are largely ineffective on modern out-of-order (OOO) processors—they are not faster (or only marginally so) than the standard Compressed Sparse Row (CSR) SpMV implementation.**

To improve performance for power-law graphs on modern OOO processors, we propose *Locality-Aware Vectorization* (*LAV*). LAV is a new approach that leverages a graph's power-law nature to extract locality and enable effective vectorization for SpMV-like memory access patterns. LAV splits the input matrix into a dense and a sparse portion. The dense portion is stored in a new representation, which is vectorization-friendly and exploits data locality. The sparse portion is processed using the standard CSR algorithm. We evaluate LAV with several graphs on an Intel Skylake-SP processor, and find that it is faster than CSR (and prior approaches) by an average of 1.5x. LAV reduces the number of DRAM accesses by 35% on average, with only a 3.3% memory overhead.

***Index Terms*—Sparse Matrix Vector Products, Graph Algorithms, Vectorization, SIMD, Locality Optimizations**

I. Introduction

Graph analytics algorithms are often used to analyze social, web, and e-commerce networks [1]–[4]. These networks are typically *power-law graphs* — i.e., their degree distribution follows a power law. In these networks, a small fraction of the vertices have a degree that greatly exceeds the average degree.

Graph algorithms can often be recast as generalized Sparse Matrix-Vector multiplication (SpMV) operations [5]–[7]. Examples range from iterative algorithms (e.g., PageRank [3] and HITS [4]) to traversal algorithms (e.g., path/diameter calculations [8]). SpMV-based graph algorithms are faster and have a better multi-core scalability than general graph processing frameworks [8], making SpMV an important kernel to optimize for efficient graph analytics.

Executing SpMV efficiently on real-life power-law graphs is challenging. The reason is that these graphs are large (millions to billions of vertices) and highly irregular, causing the SpMV memory access patterns to have low locality. Moreover, the data-dependent behavior of some accesses makes them hard to predict and optimize for. As a result, SpMV on large power-law graphs becomes memory bound.

To address this challenge, previous work has focused on increasing SpMV's Memory-Level Parallelism (MLP) using vectorization [9], [10] and/or on improving memory access locality by rearranging the order of computation. The main techniques for improving locality are binning [11], [12], which translates indirect memory accesses into efficient sequential accesses, and cache blocking [13], which processes the matrix in blocks sized so that the corresponding vector entries fit in the last-level cache (LLC). However, the efficacy of these approaches on a modern aggressive out-of-order (OOO) processor with wide SIMD operations has not been evaluated.

In this paper, we perform such an evaluation using an Intel Skylake-SP processor. We find that, on large power-law graphs, these state-of-the-art approaches are not faster (or only marginally faster) than the standard Compressed Sparse Row (CSR) SpMV implementation. Moreover, these approaches may cause high memory overheads. For example, binning [11], [12] essentially doubles the amount of memory used.

We then propose *Locality-Aware Vectorization* (*LAV*), a new SpMV approach that successfully speeds-up SpMV of power-law graphs on aggressive OOO processors. LAV leverages the graph's power-law structure to extract locality without increasing memory storage.

LAV splits the input matrix into a dense and a sparse portion. The dense portion contains the most heavily populated columns of the input, which—due to the power-law structure—contain most of the nonzero elements. This dense portion is stored in a new vectorization-friendly representation, which allows the memory accesses to enjoy high locality. The sparse portion is processed using the standard CSR algorithm, leveraging the benefits of OOO execution. Overall, LAV achieves an average speedup of 1.5x over CSR and prior optimized schemes across several graphs, while reducing the number of DRAM accesses by 35% with only a 3.3% storage overhead.

Contributions. We make the following contributions:

- We analyze existing SpMV approaches with power-law graphs and show their shortcomings in modern OOO processors.
- We show that by using a new combination of known graph

SC20, November 9-19, 2020, Is Everywhere We Are
978-1-7281-9998-6/20/$31.00 ©2020 IEEE

TABLE I
VECTOR OPERATIONS USED IN THIS PAPER. INSTRUCTIONS WITH THE *_mask* EXTENSION ALLOW FOR SELECTIVE OPERATIONS ON LANES.

Operation	Details
`load[_mask](addr[, mask])`	loads 16 (8) 32-bit (64-bit) packed values to a vector register from addr.
`gather[_mask](ids, addr[, mask])`	gathers 16 (8) 32-bit (64-bit) values from an array, starting at addr, from the indices provided in the ids vector.
`scatter[_mask](ids, addr, vals[, mask])`	scatters 16 (8) 32-bit (64-bit) values in *vals* to the array starting at addr in the indices provided in the ids vector.
`fp_add / fp_mul`	performs the element-wise addition/multiplication of two vector registers.

pre-processing techniques, we can extract a high-locality dense portion from a sparse power-law matrix.
- We propose LAV, a new SpMV approach that processes the dense portion with vectorization and the sparse portion with the standard CSR algorithm.
- We evaluate LAV with 6 real-world and 9 synthetic graphs which are at least one order of magnitude larger than those used in the majority of the previous works. We show that LAV (1) consistently and significantly outperforms previous approaches including CSR; (2) has minimal storage overhead and small format conversion cost; and (3) significantly decreases data movement for all levels of the memory hierarchy.

II. BACKGROUND

Every graph G can be represented as an adjacency matrix A, in which element $A_{i,j}$ is non-zero if there is an edge from vertex i to vertex j. In this paper, we therefore use "matrix" and "graph" interchangeably. Real-world power-law graphs typically have many vertices (millions to billions) but most vertices only have relatively few neighbors. Therefore, the adjacency matrices of these graphs are sparse.

Many graph algorithms iteratively update vertex state, which is computed from the states of its neighbors. Each iteration can be implemented with *generalized SpMV* [8], where the multiply and add operations are overloaded to produce different graph algorithms.

Consequently, we consider the SpMV problem of computing $y = Ax$, where A is the graph and y and x are dense output and input vectors, respectively, representing the set of updated vertices. The computation of every element of y (for row i of A) is $y_i = \sum_{j=0}^{n-1} A_{i,j} \cdot x_j$, for $0 \leq i \leq$ m-1, where m is the dimension of y and n is the dimension of x, i.e., the number of graph vertices.

The elements of A are read only once, but one can reuse the elements of x and y, whose size typically far exceeds the Last-Level Cache (LLC) capacity. The main challenge is how to reuse elements of x, since the sparseness of A makes the distribution of accesses to x elements irregular.

A. CSR Matrix Representation

Large, sparse matrices require a compact in-memory representation. The compressed sparse row format (CSR) (or one of its variants) is the popular choice for graph processing frameworks [14], [15]. In `CSR`, three arrays are used to represent matrix A: $vals$, col_id, and row_ptr. The $vals$ array stores all of the nonzero elements in matrix A. Within the $vals$ array, all the elements in the same row are stored contiguously. The col_id array stores the column index of the nonzero elements. The row_ptr array stores the starting position in the $vals$ and col_id arrays of the first nonzero element in each row of matrix A. An example CSR representation is shown in Figure 1. In this paper, we use `CSR` as our baseline.

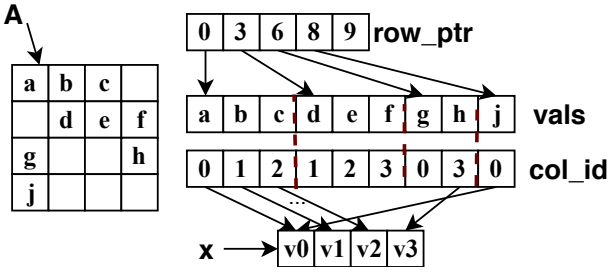


Fig. 1. CSR format.

Algorithm 1 shows a single iteration of `CSR` SpMV. The algorithm iterates over the matrix A row by row, and calculates the dot product of the row in matrix A and the input vector x (Lines 3-5). Parallelization is straightforward. Each row can be executed in parallel without the need for synchronization.

Algorithm 1 Implementation of `CSR` SpMV.

```
1: for i←0 to m-1 in parallel do
2:     sum←0
3:     for e ← row_ptr[i] to row_ptr[i+1]-1 do
4:         sum←sum + vals[e] × x[col_id[e]]
5:     end for
6:     y[i]←sum
7: end for
```

B. Vector Instructions

The Intel Skylake-SP implements the AVX-512 extension, which offers 512 bit vector instructions. This extension allows for 8 double-precision operations (or 16 single-precision operations) to proceed in parallel. Skylake also supports SIMD gather and scatter instructions, which can load/store random elements from/to a given array. Gather/scatter operations are useful for the random accesses performed in SpMV. Masked versions of gather/scatter are also provided, enabling the selective execution of operations on SIMD lanes. Table I describes the vector instructions used in this paper.

C. Previous SpMV Approaches

The main approaches to improve SpMV performance for large-scale power-law graphs on general-purpose processors are: (1) using vectorization to increase memory-level parallelism (MLP), and (2) improving memory locality and thereby cache effectiveness.

Vectorization can decrease the number of instructions executed, and increase the number of in-flight memory accesses

and the floating point (FP) throughput. There are many proposals for vectorization of SpMV [9], [10], [16]–[19].

To improve locality, several graph reordering techniques have been proposed [20]–[22]. However, they require time consuming pre-processing. We instead focus on recent techniques for increasing locality with lightweight pre-processing.

In this work, we focus on the CSR5 [9], binning (BIN) [11], cache blocking (BCOO) [13], and CVR [10] techniques, which are used in the most competitive SpMV algorithms. CSR5 provides efficient vectorization, BIN and BCOO improve locality, and CVR combines both approaches.

CSR5 [9]: CSR5 creates a compact, sparsity-insensitive representation of the input matrix that can be processed efficiently by vector units. CSR5 takes an input matrix in CSR format and partitions the col_id and $vals$ arrays of CSR into equally-sized small 2D tiles. The size of a tile ($w \times \sigma$) is set as follows: w is set to the number of SIMD lanes, and σ is optimized for the specific architecture. The tiles can be processed in parallel.

Figure 2 shows an example of a tile created for a given matrix. A row may end up in multiple tiles (e.g., the 7^{th} row in Figure 2 spans multiple tiles). For this reason, CSR5 uses the segmented sum approach to compute the final value for a row. Overall, this structure creates good load balancing across threads and a high utilization of SIMD units.

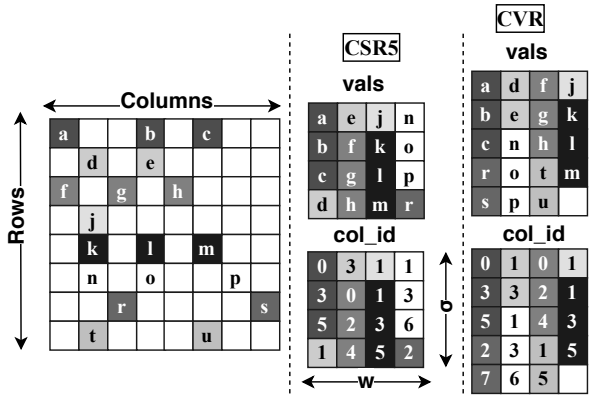


Fig. 2. Tile layout in CSR5/CVR (w = # of SIMD lanes). In CSR5, we omit an incomplete tile and metadata for brevity.

CVR [10]: Compressed Vectorization-oriented sparse Row (CVR) is a compact vectorized representation for SpMV. In CVR, each row of the matrix is processed by a single SIMD lane. Once a SIMD lane finishes processing a row, the next non-empty row from matrix A to be processed is scheduled on the emptied SIMD lane. In addition, CVR implements a sophisticated work stealing method to balance the SIMD lanes. When there are no more rows left to fill the empty SIMD lanes, an empty SIMD lane will steal elements from non-empty SIMD lanes. Figure 2 shows the memory layout for $vals$ and col_id arrays for CVR.

CVR implements a vectorization mechanism that increases MLP by utilizing vector units efficiently. In addition, it improves locality by scheduling multiple rows to be processed in parallel, which may overlap accesses to a given cache line of x by different SIMD lanes.

BIN [11], [12]: Binning (BIN) always performs sequential or high-locality memory accesses. BIN splits the SpMV execution into two phases. In the first phase, it reads the graph edges sequentially to generate every vertex update (i.e., some $A_{i,j} \cdot x_j$ that should be added to y_j). The updates are buffered with sequential writes into *bins*, each of which is associated with a cache-fitting fraction of the vertices. In the second phase, BIN applies the updates in each bin. While applying updates results in irregular memory accesses, they target only a cache-fitting fraction of vertices, and so enjoy high locality. Overall, BIN reduces the cycles-per-instruction (CPI), but executes more instructions, including memory accesses, and needs extra storage for the bins.

Cache Blocking [13]: The idea of cache blocking is to keep r and c elements of the vectors y and x, respectively, cached while an $r \times c$ block of the matrix gets multiplied by this portion of the x vector. We consider BCOO, our hand-optimized implementation of blocking that keeps the x vector portions LLC-resident. BCOO stores the blocks in coordinate format (COO), i.e., as a list of $(row, column, value)$ tuples sorted by row id. To efficiently exploit parallelism, BCOO divides blocks among cores in a row disjoint fashion, which enables writing y vector elements without atomic operations.

III. LAV: Analysis of Prior SpMV Approaches with Power Law Graphs

We find that on aggressive OOO processors, the existing techniques described are not faster (or only marginally so) than a simple CSR implementation. We analyze this behavior on a representative Intel Skylake-SP processor. In our experiments, we use three real-world graphs, *sd1*, *sk*, and *tw*, and a large random graph, *R*-25-64, with 2^{25} nodes and ≈64 average degree. Section V-C describes these graphs.

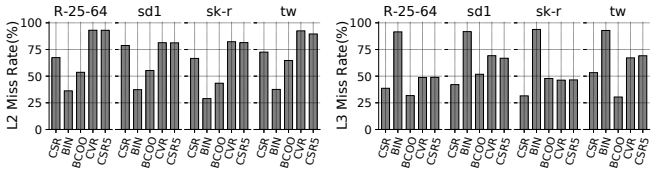
① CSR5 and CVR benefit only from limited amounts of locality, due to relying on the input's layout. These two techniques take the input matrix and build special matrix representations with the goal of keeping SIMD lanes busy. However, the amount of locality exhibited by the memory reads in these representations depends on the structure of the input matrix. Sadly, we find that the locality resulting from the original input matrix is often insufficient.

CSR5 [9] partitions the sequence of all nonzero elements in a CSR matrix (i.e., the *vals* and *col_id* arrays) into 2D tiles of the same size. The tiles are populated based on the row major order of appearance of the nonzero elements within the input matrix. As a result, CSR5 does not guarantee high locality for accesses to the x vector. For example, a row with many nonzeros may be spread over several tiles. Also, a row may occupy multiple SIMD lanes in a tile. In both cases, the accesses are determined by the graph structure, which may not exhibit any spatial or temporal locality in terms of accesses to columns (x vector). Even if there are multiple rows in a single tile, they may access disjoint sets of elements from the x vector and, therefore, fail to exploit spatial locality.

In CVR [10], each row of the matrix is processed to completion by a single SIMD lane. Once a lane finishes a

row, it begins processing the next non-empty row. As a result, CVR can have two lanes processing columns that are far apart (e.g., columns at the start of one row and at the end of another row). The result will be poor locality. However, if the overlap between the rows is high, we can observe good locality.

Figure 3 shows the miss rates of L2 and L3 caches in all of the techniques. We see that the miss rates of CVR, and CSR5 are higher than in CSR. Hence, CVR and CSR5 do not improve locality over CSR.

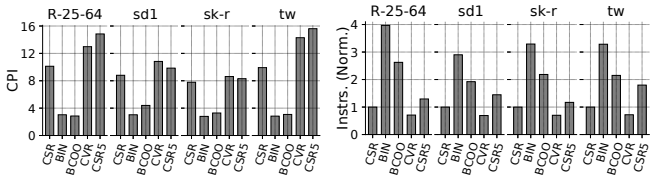


(a) L2 Miss Rate (b) L3 Miss Rate

Fig. 3. L2 and L3 miss rates for different techniques.

② The locality-aware techniques improve cache locality but have limited impact on performance due to increased number of instructions. Binning [11], [12] and cache blocking (BCOO) target large graphs and try to improve locality by either regularizing memory accesses or restricting them to fit in the LLC. We observe, however, that on a modern OOO processor, these techniques do not reduce execution time over CSR significantly, even though they reduce the miss rate of the L2 and, in the case of BCOO, of the L3 (Figure 3), and the average cycles per instruction (CPI). The reason is that they execute more instructions. Furthermore, for BIN, the amount of data moved from memory is similar to the other techniques.

Figure 4 shows the CPI (a) and the instruction count (b) of the different techniques. We see that BIN decreases the CPI substantially compared to most of the other techniques. However, it is the technique with the most instructions executed. BIN was effective in previous generations of OOO processors that did not do a good job at hiding memory latency because their Reorder Buffer (ROB) and Load Queue (LQ) were smaller. Thus, the CPI gains were more significant and BIN was able to tolerate the increase in number of instructions. On the other hand, BCOO improves the CPI like BIN while increasing the number of instructions less. As a result, it is able to improve performance marginally.



(a) CPI (b) Instructions

Fig. 4. CPI and number of instructions executed for different techniques.

③ Vectorization provides limited MLP improvements, since modern OOO cores keep a high number of memory requests in flight due to their deep ROB, LQ, and other buffers. In an SpMV iteration, we observe that most reads of x incur cache misses. The reason is that the access pattern is irregular and the vector size exceeds the capacity of the LLC. Because of this bottleneck, in an in-order or narrow issue processor such as Intel's Xeon Phi, vectorization is effective at increasing MLP: a vector read issues several memory reads concurrently, whereas a scalar narrow-issue core can only issue very few reads at a time. However, modern OOO processors extract significant instruction-level parallelism (ILP) from the SpMV code, and so can sustain over ten in-flight memory accesses at a time. Hence, vectorization of the code does not significantly increase the number of memory accesses in flight.

As an indicator of MLP, we measure the average occupancy of the Line Fill Buffer (LFB) in L1 for CSR and the vectorization techniques. Recall that the LFB is an internal buffer that the CPU uses to track outstanding cache misses; each LFB entry is sometimes called a Miss Status Handling Register or MSHR. We compare CSR (which is a scalar approach) to CVR and CSR5 (which are vector approaches). The result is shown in Figure 5. The maximum LFB occupancy for the machine measured is 12 [23]. The figure shows that the LFB occupancy of CSR is already very similar to the one of the vectorized CVR and CSR5. Consequently, one does not need vectorization to attain high MLP on an aggressive OOO processor.

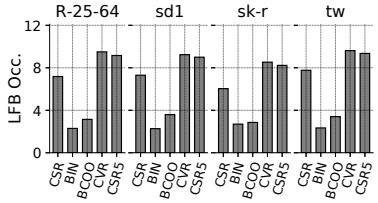


Fig. 5. LFB occupancy for different techniques.

IV. LAV: Locality-Aware Vectorization

The previous observations showed that, on modern OOO processors, the bottleneck of current SpMV techniques is not low MLP. Instead, we observe that current techniques either do not exploit locality or are only able to extract locality by increasing the number of instructions executed, resulting in little or no overall gains.

Based on these observations, we propose *Locality-Aware Vectorization* (*LAV*), a new approach that speeds-up SpMV by harvesting locality from power-law graphs without increasing the number of instructions. LAV relies on a combination of lightweight pre-processing steps that create a compact matrix representation.

In this section, we describe the main idea, the pre-processing steps, the matrix representation, the rationale behind the design choices, and LAV's SpMV algorithm.

A. Main Idea

To improve locality while keeping a compact data representation, LAV takes the input matrix A and divides it into two portions. The first one, called the *Dense* portion, has the majority of the nonzero elements. The second one, called the *Sparse* portion, includes the remaining nonzero elements. Operations on the dense and sparse portions of the matrix access disjoint subsets of elements from the input vector x. LAV uses

different formats and processing mechanisms for the dense and sparse portions. The dense portion is formatted for locality and processed with a vectorized SpMV implementation. The sparse portion is formatted for compactness and processed with a CSR-based SpMV implementation.

Creating the dense portion for locality. The dense portion contains most of the nonzero elements in matrix A. It is obtained by picking the most heavily populated columns of A until, all together, they account for a fraction T of the nonzero elements of A (e.g., 80% of the nonzero elements). For our target real-world power-law graphs, where the degrees of the vertices follow a power law distribution [24], [25], this dense submatrix will only contain a small fraction of the original columns (e.g., 20% of the columns) [26]. In addition, the columns are then divided into *segments*, each of which fits into the LLC. As we will see, the data in the dense portion is stored in a compact, vectorization-friendly format that allows for fast processing.

Creating the sparse portion for compactness. The sparse portion is the rest of the columns. The data is stored in the `CSR` representation, which is compact. The data is processed with the standard `CSR` implementation, since we do not expect to obtain locality benefits during its processing.

B. Splitting the Matrix into Dense and Sparse Portions

To split the matrix into dense and sparse portions, LAV uses the following three simple matrix transformations.

① Column Frequency Sorting (CFS): CFS sorts the columns in descending order of nonzero element count, changing the physical layout of the matrix A and input vector x. Since the degree distribution of the vertices in our target graphs follows a power law, the first few columns of A will include the vast majority of nonzero elements. Thus, CFS's reorganization of A's columns results in *frequently accessed elements of x being stored in the same, or close by, cache lines*. The result is improved cache line and overall cache utilization, which speeds-up computations [27].

To sort the columns, CFS logically moves them by relabeling the column IDs according to the number of nonzeros in descending order. CFS permutes the x vector accordingly.

② Segmenting: Segmenting takes the output of CFS and partitions matrix A into dense and sparse portions. The dense portion is further divided into *segments* of consecutive columns. Each segment contains S columns, where S is chosen so that the corresponding part of the input vector x fits in the LLC. (For example, we use $S = 5\,M$ in our evaluation; see Section VI-C.) The number of segments s in the dense portion is chosen so that the dense portion includes a fraction T (e.g., 80%) of all the nonzero elements in A.

Segmenting can be efficiently implemented if combined with CFS. CFS already generates a sorted list of columns according to their count of nonzero elements. After this list is composed, segmenting creates the segments.

③ Row Frequency Sorting (RFS): RFS sorts the rows in a segment in decreasing count of nonzero elements in the row. The resulting order will determine the order of execution of rows in the segment. With this change, we will be able to execute rows with similar numbers of nonzero elements at the same time in different SIMD lanes. As a result, LAV minimizes load imbalance between different SIMD lanes.

RFS does not actually relabel the rows in a segment. Instead, it only generates a list of row IDs for the execution order. Hence, it is a very lightweight operation.

Parallelizing LAV's transformations All of the steps above can be efficiently parallelized. Specifically, CFS and RFS require counting the number of nonzero elements in each column and row, respectively. These counts can be computed in parallel. Our current implementation uses atomic operations to safely update counts in parallel, but techniques exist [28] that would eliminate the use of atomics. In practice, the number of nonzeros in each column/row is often available without computing, since many graph analytics frameworks [14] store the matrix in both CSR and CSC formats.[1] In this case, the number of nonzeros can be obtained from the offset arrays.

Once the nonzero element counts are known, relabeling the columns for CFS simply requires parallel sorting. Segmenting requires a parallel prefix sum on the relabeled columns, to determine the cut-off point for the number of columns per segment. RFS simply requires sorting the row IDs in decreasing count of non zeros.

C. LAV Walk-Through

Figure 6 shows an example of LAV's transformations. An initial matrix A is shown in Figure 6(a). To highlight the effect of CFS, the x vector is shown on top of matrix A. As Figure 6(a) shows, initially, the nonzero elements of A are scattered over the rows and columns.

The first step performs CFS on A, relabeling all the column IDs with the new order. This is shown in Figure 6(b). Note that the input vector x is permuted according to the new order of the columns. As shown in Figure 6(b), the majority of the nonzero elements now reside in first few columns.

The second step divides A into segments of a fixed number of columns. The number of columns per segment, S, is maximized under the constraint that the corresponding entries of the input vector fit into the LLC. The matrix is then partitioned into dense and sparse portions. The dense portion consists of the first s segments that contains a T fraction of the nonzero elements. The sparse portion contains the remaining columns. Figure 6(c) shows the dense and sparse portions of A with $S = 4$ and $T = 0.7$, where a single segment suffices to obtain the dense portion.

The next step applies RFS to each of the segments of the dense portion. This is shown in Figure 6(d). Note the output vector y is permuted according to the new order of the rows. We will later see that neither the matrix nor the y vector is physically reordered; instead, LAV creates an indirection vector to record the reordering.

[1]The Compressed Sparse Column (CSC) format is similar to CSR, except that the matrix is stored by column rather than by row.

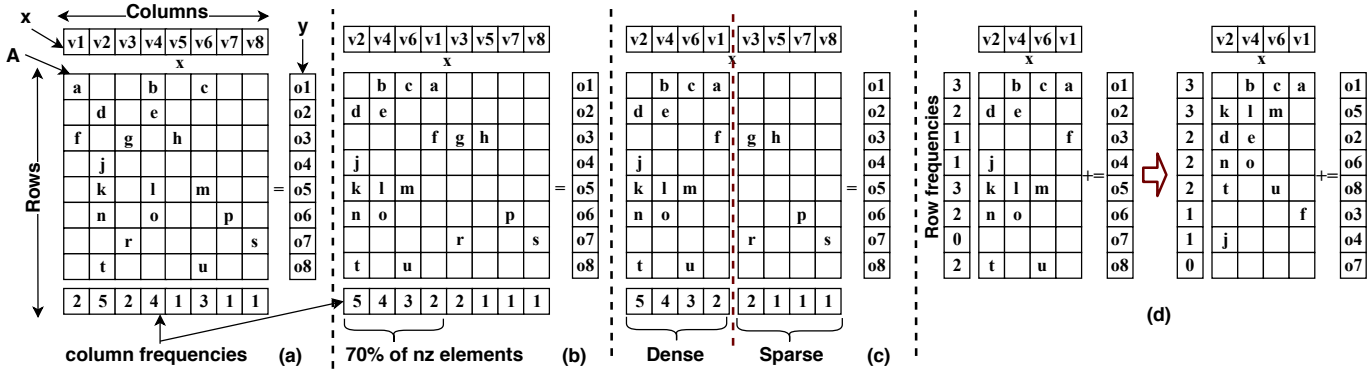


Fig. 6. LAV's transformations on an 8x8 matrix. In the example, the threshold for creating the dense portion is T=0.7 and the segment size is S=4.

D. LAV Matrix Representation

LAV's dense portion is composed of segments of consecutive columns. After RFS, each segment is divided into *Chunks* of rows, where each chunk has as many rows as the number of SIMD lanes in the machine. Figure 7 continues the example of Figure 6 and shows, on its left side, the segment divided into two chunks. We use four rows per chunk because we assume four SIMD lanes. The two chunks have different colors.

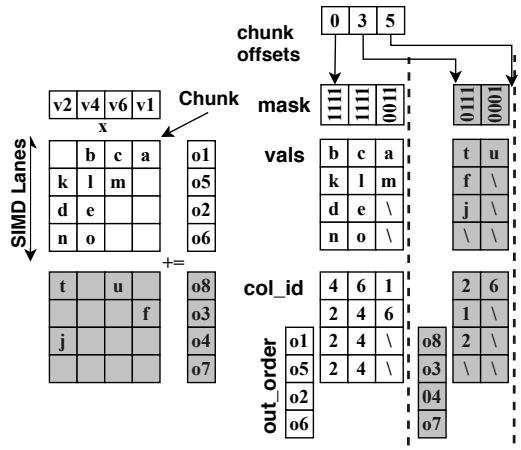


Fig. 7. Vectorization-friendly layout of the dense portion in LAV. The example assumes 4 SIMD lanes.

Each row is next compressed such that zero elements are not stored. Since the different rows of a given chunk may have different numbers of nonzero elements, some SIMD lanes will remain empty towards the end, as they run out of nonzero elements. To handle this case, LAV creates a mask for each SIMD lane, expressing the empty elements in the lane with zeros. To represent a segment, LAV repeats the same operation on all the chunks of the segment, and appends them together.

The right side of Figure 7 shows the memory layout of the segment for the example. A segment in LAV is represented by five arrays:*vals*, *col_id*, *chunk_offsets*, *out_order*, and *mask*.

The *vals* and *col_id* arrays hold the values and column IDs, respectively, of all the chunks in the segment. These are 2D arrays whose one dimension is equal to the number of SIMD lanes, and the other is the sum of the lengths of the chunks of the segment. Recall that the length of a chunk is equal to the maximum number of nonzero elements in any row of the chunk. This layout of *vals* and *col_id* is efficient because it will enable using aligned vector loads.

The *chunk_offsets* array indicates the starting point of every chunk within the *vals* and *col_id* arrays. When a chunk finishes execution, the partial sums generated by the rows are accumulated into the correct position in the output vector y.

Recall that, with RFS, LAV generates a new execution order for each segment. This execution order is stored in the *out_order* array. The *out_order* array stores only the IDs of the rows. Finally, some SIMD lanes in a chunk may be empty. Hence, LAV has an extra array called *mask* to distinguish empty elements in the *vals* and *col_id* arrays.

E. Rationale Behind LAV's Design Choices

We now explain the rationale behind LAV's design choices.
① **Reuse Memory:** LAV uses CFS to group columns with a high number nonzeros together. Combined with segmenting, this transformation limits the memory footprint required to process the dense portion. The dense portion will be processed one segment at a time. Thus, by sizing a segment appropriately, we ensure that the portion of the x vector that is accessed while processing one segment fits in the LLC.
② **Balance SIMD Lanes:** LAV uses RFS in segments to balance the SIMD lanes. Recall that each lane calculates the results for a single row. In a given chunk, consecutive rows have a similar number of nonzero elements. Thus, while processing this chunk, all SIMD lanes have roughly the same amount of work. This approach avoids computation load imbalance.

Combining CFS and RFS increases the possibility of memory access overlap. Indeed, rows that are scheduled together on different lanes often access the same (or close by) elements of the x vector.

We also apply RFS to the sparse portion. This way we find empty rows at the end and do not need to process them.
③ **Handle Sparse Portion Efficiently:** Using CSR for the sparse portion minimizes the number of instructions needed per irregular access and the total storage cost. The fewer bookkeeping instructions enables the processor pipeline to keep more memory accesses in flight simultaneously, thereby maximizing MLP.

F. SpMV Algorithm with LAV

The LAV algorithm first processes the dense portion of the matrix and then the sparse one. Algorithm 2 shows the steps taken to process the dense portion. The sparse portion is processed with the traditional CSR algorithm.

Algorithm 2 SpMV algorithm with LAV.

```
1: procedure SPMV(segments, x, y, m, n, nLanes)
2:     // segments is the list of segments created for vectorization
3:     // x is the input vector, y is the output vector
4:     // m number of rows, n number of cols, nLanes number of SIMD lanes
5:     for r=0 to m-1 do  // initialize y vector to zeros
6:         y[r] ← 0
7:     end for
8:     foreach segment s in segments do  // all segs. in dense portion
9:         // chunks of a segment processed in parallel
10:        for c=0 to s.num_chunks-1 in parallel do
11:            row_sums←(0, ..., 0)
12:            row_mask←0xFF
13:            // last chunk may have all the lines completely idle
14:            if c is the last chunk in the segment then
15:                row_mask ← 0xFF >> (nLanes-(m - c*nLanes))
16:            end if
17:            // Get the rows to be updated by this chunk
18:            row_ids←load_mask(s.out_order[c*nLanes], row_mask)
19:            // Get the prev values from y for rows of this chunk
20:            prev←gather_mask(row_ids, &y, row_mask)
21:            for i← s.chunk_offsets[c] to s.chunk_offsets[c+1]-1  do
22:                ms←s.mask[i] // load the mask for the SIMD lane
23:                cols←load_mask(s.col_ids[i*nLanes], ms)  // get col. ids
24:                vals←load_mask(s.vals[i*nLanes]], ms) // get vals
25:                x_vec←gather_mask(cols, &x, ms) // get vals from x
26:                mul←fp_mul(vals, x_vec)
27:                row_sums←fp_add(row_sums, mul)
28:            end for
29:            // accumulate sum calculated by chunk
30:            row_sums←fp_add(row_sums, prev) // Aggregate over segs.
31:            // update y vector
32:            scatter_mask(row_ids, &y, row_sums, row_mask)
33:        end for
34:    end foreach
35:    // CSR for the sparse portion
36: end procedure
```

Algorithm 2 works by operating on one segment at a time (Line 8), processing all the nonempty chunks in the segment in parallel, with multiple threads (Line 10). While processing a chunk, LAV utilizes wide SIMD units. The rows in the chunk are processed in parallel by SIMD lanes, and the partial sums generated by the rows are accumulated into a vector register (row_sums in Lines 21-28). The *mask* values, which are produced during the data layout generation (Section IV-D), are used to disable computation for lanes with empty elements. After the completion of the chunk, the output values that it has generated are accumulated into the output vector y. This operation involves reading the accumulated values up until this segment in output vector y (Line 20), and adding to them the contributions of this chunk (Lines 30-32).

Finally, there is the case when the number of rows in the last chunk of a segment is less than the number of SIMD lanes. In this case, we calculate a mask (*row_mask*) that is used to omit all the operations in the idle SIMD lanes. This operation is shown in Lines 14-16 of Algorithm 2.

V. Experimental Setup

A. Test Environment

We use a state-of-the-art Intel Skylake-SP shared-memory machine with vector instruction support. Our machine has 2 processors, each with 20 2.4 GHz cores, for a total of 40 cores. Each core is OOO and has a private 32 KB L1 data cache and a private 1 MB L2 cache. Each processor has a shared 28 MB LLC. The machine has 192 GB of 2666 MHz DDR4 main memory, distributed across 12 DIMMs of 16 GB each. Details of our system are summarized in Table II.

TABLE II
System characteristics.

Component	Characteristics
CPU	Intel Xeon Gold 6148 CPU @ 2.40GHz 20 cores per processor, 2 processors
Cache	Private 32 KB instruction and data L1 caches Private 1 MB L2 cache, 28 MB LLC per processor
Memory	192 GB, 12 DIMMs (16 GB each), DDR4 2666 MHz
Vector ISA	avx512f, avx512dq, avx512cd, avx512bw, avx512vl

We use Ubuntu Linux 14.04 with kernel version 4.9.5. All codes are compiled with the Intel compiler version 19.0.3, using the *-O3 -xCORE-AVX512* compiler flags and parallelization is done with OpenMP. When implementing LAV, we use intrinsics provided by Intel compiler for vectorization. `CVR`, `CSR5`, and LAV implementations use 512 bit vector instructions. All SpMV implementations use double precision floating-point arithmetic. We use `numactl` to interleave allocated memory pages across the NUMA nodes. Dynamic voltage and frequency scaling (TurboBoost) is disabled during the experiments.

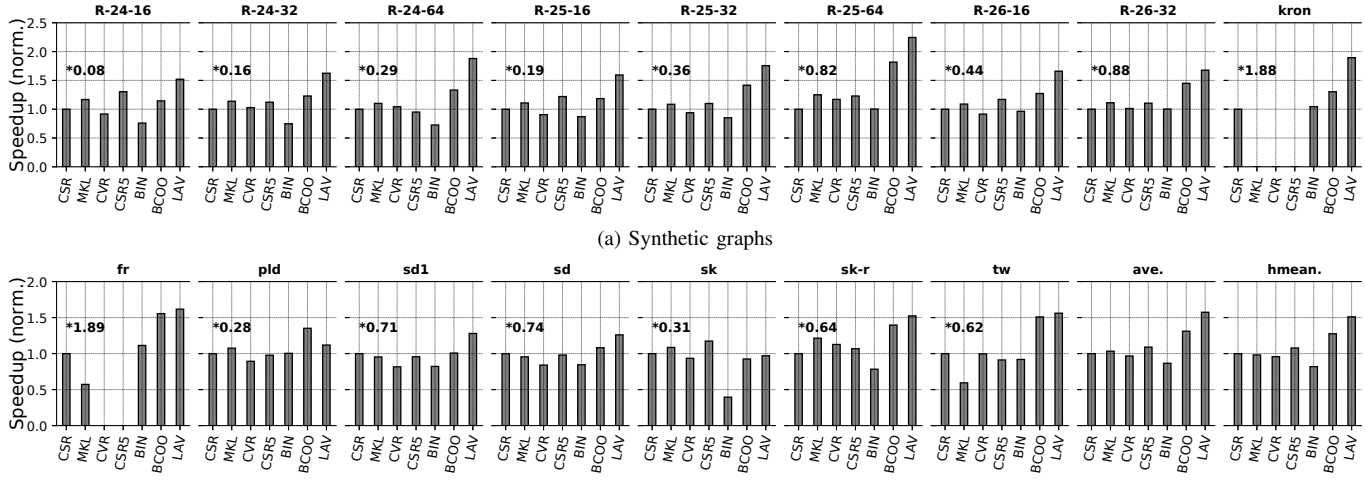
In all the experiments, 40 threads are used and each thread is pinned to a core. In the scalability experiments (Section VI-E), when we have fewer threads than cores, we assign threads to the two sockets in a round robin manner. All experiments run 100 iterations of SpMV. Unless stated otherwise, execution time only includes execution of SpMV. For microarchitectural analysis, we use the LIKWID performance monitoring tool [29].

B. Formats and Techniques for Comparison

We compare LAV to `CSR`, `CSR5` [9], `CVR` [10], `BCOO` [13], and `BIN` [11] (Section II-C). We also compare to the inspector-executor SpMV implementation in Intel's Math Kernel Library (`MKL` [30]), which optimizes the SpMV execution according to the input's sparsity level.

We implement `CSR`, `BCOO`, and `BIN` ourselves; for `BIN`, we closely follow [11]. We use the `CSR5` [31] and `CVR` [32] implementations provided by their authors.

Optimizations: Our compiler automatically vectorizes the baseline scalar CSR code. We manually tune `CSR`'s OpenMP scheduling parameters. For `CSR5`, we use the tile size suggested in [9]. `CVR` does not have any tuning parameters. For `BIN`, we apply the optimizations in [11]. For instance, the target bin and position in the bin of each update are not calculated at run-time, but stored with the matrix. To represent bins, we use the optimization from [12]. Each thread works on



(a) Synthetic graphs

(b) Real-world graphs plus arithmetic and harmonic means

Fig. 8. Comparison of the speedups of the different implementations over CSR for all the input graphs. The numbers marked with * show the execution time in seconds of a single SpMV iteration with CSR.

a sequential portion of each bin. We find that the best bin size is 65 K rows, by sweeping bin sizes from 1 K to 262 K. With this bin size, each portion of the y vector approximately fits into the L2 cache. For BCOO, we try to divide the work equally among threads in a row disjoint manner (see Section II-C). We also manually tune the size of the cache block and find that blocks that include 2.5 M vertices give the best performance. Finally, for LAV, we use 5 M as the segment size (S) and 80% as the fraction of nonzeros in the dense segment (T).

C. Input Graphs

We use 6 real-world graphs and 9 synthetic power-law graphs. Table III summarizes the properties of the graphs.

Real-world graphs. We use the following large graphs with power-law degree distributions [24], [25]: com-Friendster (*fr*), Pay-Level Domain graph (*pld*), 1st Subdomain graph (*sd1*), Subdomain/Host graph (*sd*), sk-2005 (*sk*), and twitter7 (*tw*). We obtain *fr*, *sk*, and *tw* from [33], and *pld*, *sd1*, and *sd* from [34]. The number of vertices (or rows or columns) per graph ranges from 41 M to 101 M. In prior work, only BIN was evaluated with graphs of these sizes; CSR5 and CVR used orders of magnitude smaller graphs. The graphs are also very sparse. The number of nonzero elements (#nnz) ranges from 623 M to 3.6 B. The average number of nonzero elements per row (i.e., degree) ranges from 14 to 55, and the maximum number ranges from 5.2 K to 3.9 M.

We observe that, unlike for all the other inputs, SpMV for *sk* has high L1 hit rates. For this reason, we also evaluate *sk*-r, a version of *sk* that randomized the vertex IDs.

Synthetic graphs. We generate 8 synthetic Kronecker graphs that show power-law behavior, as in the inputs of the Graph500 benchmark [35]. They are named *R*-X-Y, where X is the scale of the graph (i.e., the number of vertices is 2^X) and Y is the approximate average degree. The number of vertices for these graphs varies between 16 M and 67 M, and the average number of edges per vertex is 16–64. These graphs provide insight into the performance of the evaluated techniques for different graph sizes and sparsity levels. We also use *kron*, a synthetically generated graph in the GAP benchmark suite [36].

TABLE III
INPUT GRAPHS.

Input	#rows (M)	#nnz (M)	Avg. #nnz per row	Max. #nnz per row (K)
fr [33], [37]	65.61	3,612.13	55.06	5.21
pld [34]	42.89	623.06	14.53	3,898.56
sd1 [34]	94.95	1,937.49	20.41	1,309.80
sd [34]	101.72	2,043.20	20.09	1,317.32
sk (sk-r) [21], [33]	50.64	1,949.41	38.50	12.87
tw [33], [38]	41.65	1,468.37	35.25	2,997.47
R-*X*-*Y*	2^X	$\approx 2^X \times Y$	$\approx Y$	200-1K
kron [33], [36]	134.22	4,223.26	31.47	1,572.84

VI. Experimental Results

A. Performance Results

Figure 8 compares the speedup of MKL, CVR, CSR5, BIN, BCOO, and LAV over CSR, running a single SpMV iteration. The figure shows bars for each synthetic input graph (a) and for each real-world graph plus the arithmetic and harmonic means across all the graphs (b). The number marked with * above the bars is the execution time of a single SpMV iteration with CSR in seconds. For the *kron* and *fr* inputs, we are unable to evaluate CVR and CSR5, because their implementations use 32-bit indices and these two large graphs cannot be represented. MKL with *kron* runs out of memory.

LAV delivers an average speedup of 1.5x over our optimized CSR. On the other hand, CVR, CSR5, and BIN provide comparable average speedups as CSR. As discussed in Section III, these vectorization and locality-optimized approaches are not effective on our aggressive OOO processor, which has an increased ability to hide memory latency. MKL is also not faster on average than CSR. BCOO is the only technique that is faster than CSR on average, achieving a 1.28x speedup.

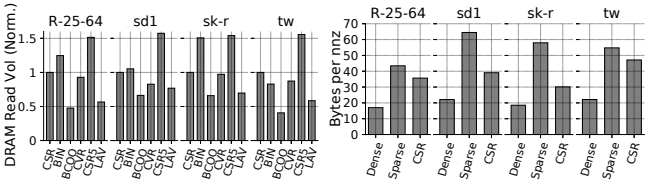
Generally, LAV's speedup over CSR increases as the input's average degree increases, which makes LAV's dense portion contain larger fractions of the nonzero elements. Overall, LAV is the fastest in 14 out of 16 inputs. One of the inputs where CSR outperforms LAV is *sk*. *sk* shows good locality behavior even for the baseline CSR, making the locality optimization techniques not effective. Even so, LAV does not hurt performance compared to CSR on this input. For *pld*, LAV outperforms CSR and other methods except BCOO.

LAV is designed to improve locality, i.e., to minimize data movement from memory and throughout the cache hierarchy. The two causes for data movement are: (1) the accesses to the nonzero elements of matrix A, and (2) the irregular accesses to the x vector. Nonzero element accesses cause data movement proportional to the storage size of the SpMV technique, since each nonzero element is accessed once in a single SpMV iteration. Data movement due to x vector accesses is dictated by the locality of the accesses and, hence, the amount of cache reuse. LAV improves in both reduced storage size and increased locality of x vector accesses. LAV reduces storage size by creating a compact representation using RFS; LAV increases locality of x vector accesses by using CFS and segmenting.

B. Data Movement

To understand the data movement characteristics of the evaluated techniques, we measure the total volume of data read from DRAM for each input. It can be shown that none of the techniques reaches the system's maximal memory bandwidth (reported by the STREAM benchmark [39]) for any of the inputs.

Figure 9a shows the amount of data read from DRAM in four representative inputs, normalized to CSR. We can see that both LAV and BCOO substantially decrease the DRAM transfer volume. The reduction achieved by LAV is 35% on average, and can be as high as 50%.

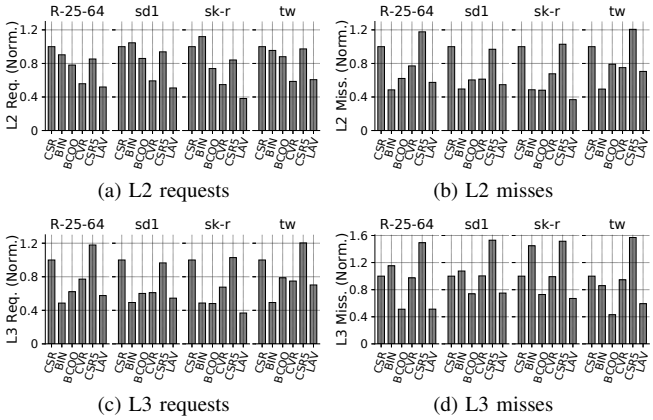


(a) Data read from DRAM (b) Bytes read per nonzero

Fig. 9. Data transfer comparison.

To analyze the benefits of LAV's dense portion, Figure 9b shows the number of bytes read per nonzero element for LAV's dense and sparse portions, and for CSR. LAV's dense portion keeps the bytes transferred per nonzero element very low. It can be shown that, across all inputs, LAV's dense portion transfers an average of 27.3 bytes per nonzero element, compared to CSR's 37.4. In contrast, LAV's sparse portion transfers more bytes per nonzero element than CSR. This is because the sparse portion is sparser than the overall graph. However, this overhead does not impact execution time much because the sparse portion is—by design—only a small portion of the matrix.

Figure 10 analyzes the data movement in the cache hierarchy for selected inputs. The figure shows the number of L2 and L3 requests and misses of the techniques, normalized to CSR. From the figures, we see that LAV minimizes the requests and misses in both levels of the cache hierarchy. In fact, it is the only technique that minimizes data movement across all levels of the memory hierarchy. This is because it does not increase the memory storage for the matrix and it improves the locality of accesses to x. (In Section VI-D, we further discuss LAV's storage overhead and show that it is negligible.)



(a) L2 requests (b) L2 misses

(c) L3 requests (d) L3 misses

Fig. 10. L2 and L3 request and miss counts.

The other techniques do not reduce data movement as much. Specifically, CSR5 and BIN suffer from increased storage. CSR5 increases data movement from memory to L3 and from L2 to L3. This is because it has auxiliary data to facilitate efficient vector execution. BIN improves L2 locality but has a high L3 cache miss rate. It ends up with same number of L3 misses or more than in CSR. CVR has reduced the data movement from L3 to L2. However, it has the same amount of data movement as CSR from memory to L3. BCOO has low data movement between memory and L3, and between L3 and L2. However, it has more L2 requests than LAV.

C. Analysis of LAV

Dense vs. sparse portions. Figure 11 shows the breakdown of execution time and of the number nonzero elements for LAV's dense and sparse portions. By construction, the dense portion contains at least 80% of the nonzero elements. However, its size is also dictated by the size of the segments from which it is composed. The segment size is such that the corresponding portion of the x vector is roughly equal to the L3 size (5 M columns in our case). Therefore, the last segment may push the number of nonzero elements in the dense portion above 80%. In particular, when the number of vertices is small (e.g., the *R*-24-XX and *R*-25-XX graphs), the dense portion contains more than 95% of the vertices.

The execution times of the dense and sparse portions are not proportional to their number of nonzero elements. Specifically, the sparse portion's execution time is often higher than its share of the nonzero elements. Although the percentage of

nonzero elements in the sparse portion is less than 20% in all cases, the sparse portion execution can take up to 38% of the execution time. The reason is that the sparse portion does not enjoy the dense portion's high locality, and consequently requires more data from DRAM per each nonzero element (Figure 9b).

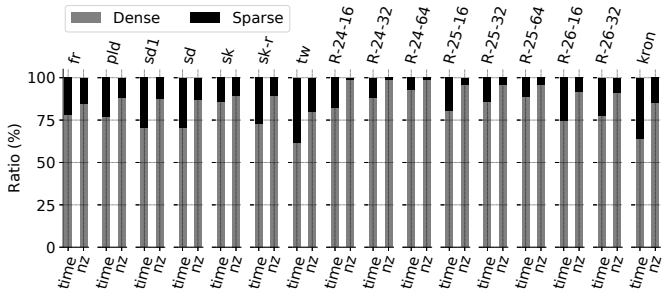


Fig. 11. Percentage of execution time and nonzero elements in LAV's dense and sparse portions.

Finding the best segment size. LAV can use LLC-sized segments without tuning. To show this, we vary the segment size for selected inputs and show that a segment size similar to the LLC size produces good results. Figure 12 shows the execution time of LAV for different segment sizes. We test segment sizes from 2 to 10 million columns. As can be seen from the figure, the best performance is achieved between 2 M and 5 M columns, and the differences in this range are not significant. The 2 M–5 M column range roughly corresponds to 16 MB–40 MB storage, which is approximately our LLC size. In our experiments, we use 5 M as the segment size.

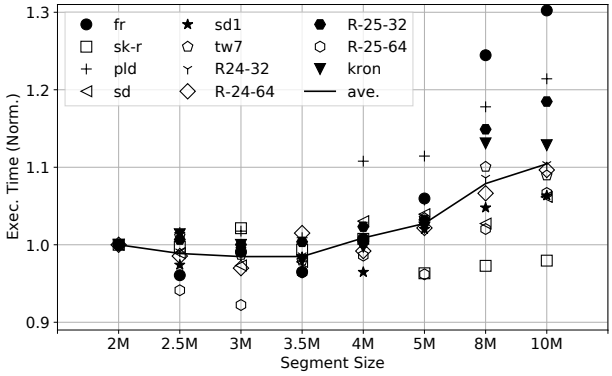


Fig. 12. Effect of the segment size in number of columns on execution time. Execution times are normalized to a 2M-column segment size.

D. LAV Overhead

Table IV reports the overhead of constructing the LAV matrix representation, in terms of time and memory storage, compared to constructing a graph's CSR representation.

Memory Overhead. LAV's storage is compact, thanks to segmenting, which limits the number of columns processed at a time, and prevents rows in the dense portion from having a large number of zero elements. As shown in the table, the memory overhead over CSR is small. On average, it is 3.35%.

TABLE IV
LAV MEMORY OVERHEAD AND NUMBER OF ITERATIONS TO AMORTIZE LAV'S COST.

Input	Number of Iterations			Memory Overhead
	Unordered	CSR	CSR+CSC	
fr	14.10	22.13	11.78	3.12%
pld	81.12	122.64	82.98	8.52%
sd1	40.85	61.84	39.81	4.94%
sd	71.27	70.02	45.70	4.99%
sk	∞	∞	∞	1.98%
sk-r	13.51	31.86	17.85	1.98%
tw	25.63	35.72	21.07	2.25%
R-*24-16*	37.34	49.08	31.66	4.09%
R-*24-32*	46.16	41.71	24.96	2.80%
R-*24-64*	37.88	33.43	18.66	2.09%
R-*25-16*	42.94	37.54	23.29	3.89%
R-*25-32*	42.42	32.75	17.55	2.62%
R-*25-64*	19.98	23.91	14.06	1.93%
R-*26-16*	34.00	29.88	18.40	3.70%
R-*26-32*	28.55	28.82	16.25	2.44%
kron	18.54	21.78	13.15	2.31%

Format construction time. It can be shown that, on average, constructing the LAV's format takes 1.6x the time of building a CSR representation. Both CSR and LAV initially sort the input's edges by column ID and row ID to construct the matrix rows. Sorting dominates the execution time. In addition, LAV requires computing the number of nonzero elements per columns, which has significant overhead.

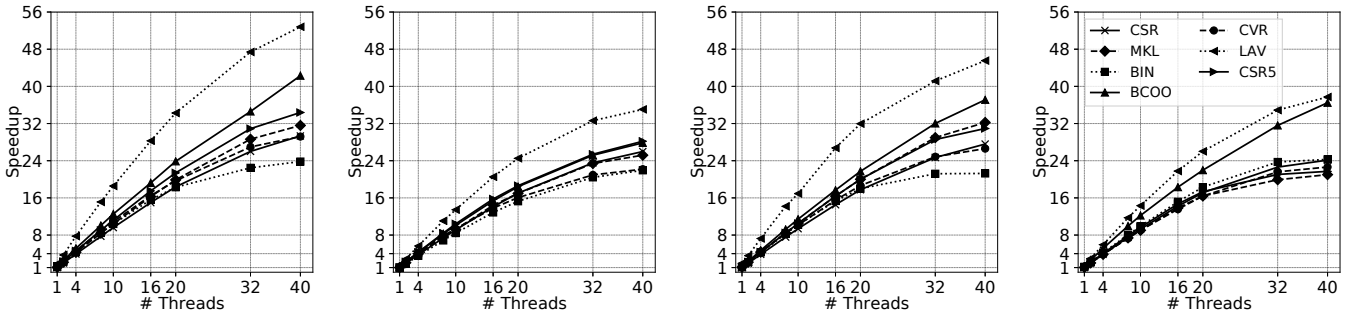
LAV's higher construction time pays for itself by yielding a faster SpMV execution. Table IV evaluates this trade-off by showing the number of LAV SpMV iterations required to make LAV faster than CSR (i.e., we are comparing construction plus SpMV execution times). We consider three possible starting points for constructing LAV. First, from an unordered list of nonzeros elements, which is the common textual representation of edge lists. Second, from a CSR representation. Third, when given both CSR and CSC representations. The latter setup exists in graph frameworks that use pull/push-based graph processing, which require both the original matrix and its transpose (for the pull and push directions, respectively).

For the large majority of the graphs used in this work, we see that 10–70 SpMV iterations are enough for the end-to-end execution time of LAV to be better than CSR, irrespective of how the LAV's format is constructed. We believe this overhead is acceptable since these matrices are generally used as inputs for multiple algorithms, or many iterations of SpMV are executed in each application.

The only exceptions are *sk* and *pld*. In *sk*, a LAV SpMV iteration is slower than a CSR SpMV iteration. Hence, LAV cannot catch up with CSR. In *pld*, the difference in execution time per SpMV iteration is small. Therefore, LAV takes 81–123 iterations to be faster than CSR.

E. Scalability Analysis

Strong scaling is defined as scalability with respect to a fixed problem size. Figure 13 compares the strong scaling of LAV to the other techniques over a representative subset of the input graphs (*R*-25-32, *sd1*, *sk-r*, and *tw*). All curves show



(a) R-25-32 (b) sd1 (c) sk-r (d) tw

Fig. 13. Strong scaling of LAV and other techniques. All points are normalized to the single threaded CSR execution.

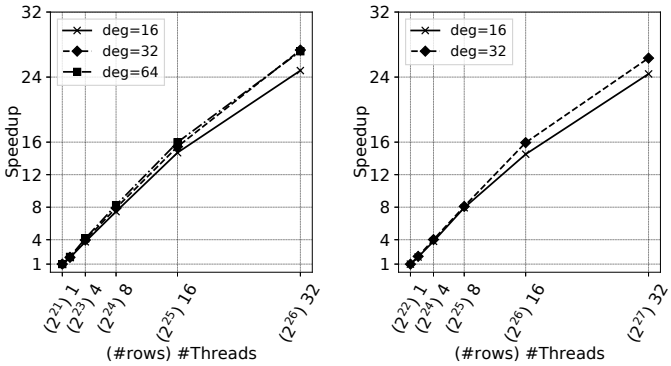
speedup values, for different numbers of threads, normalized to the single-threaded CSR execution time.

Overall, LAV gives the same or better performance compared to the other techniques at every thread count, except on the *pld* and *sk* inputs (not shown in Figure 13). For *pld*, LAV and BCOO perform similarly up to 16 threads, but BCOO outperforms LAV subsequently, achieving a 20% higher speedup for 40 threads. For *sk*, all techniques have comparable speedups (10–12× for 40 threads), with MKL being the best performer for all thread counts.

LAV's speedups vary greatly across the different input graphs. For example, while LAV achieves a 53× speedup at 40 threads on *R*-25-32, it only achieves 35× on *sd1*. This variability is expected, since the computation's memory access pattern depends on the input graph, and so the benefits from CFS and segmenting differ greatly. The maximum speedups achieved by LAV, BCOO, MKL, CSR, CSR5, CVR, and BIN with 40 threads are 53×, 42×, 35×, 31×, 31×, 29×, and 26×, respectively. If single-threaded executions are considered, LAV is still the best performer. On average, LAV achieves 1.6× speedup over CSR, while CVR, CSR5 BCOO, BIN, and MKL only attain 1.17×, 1.09× 1.07×, 1.05×, and 1.01×, respectively.

We now consider weak scaling, i.e., maintaining a fixed problem size per thread as we increase the number of threads. Figure 14 compares the weak scaling of LAV for different amounts of per-thread work. To keep the per-thread work fixed, we generate a random graph (as described in Section V) with a scaled number of rows for each thread count. Figure 14a shows weak scaling behavior when per-thread work is 2^{21} vertices, for graphs with average degrees of $\{16, 32, 64\}$. Figure 14b shows weak scaling when per-thread work is 2^{22} vertices, for graphs with average degrees of $\{16, 32\}$. In both figures, the data points of each thread count are relative to the single-threaded execution time on the corresponding random graph.

LAV has almost linear scaling up to 16 threads in both Figure 14a and 14b. Above 16 threads, the scaling is sub-linear, due to memory hierarchy bandwidth saturation. Additionally, LAV has slightly better scaling behavior on the graphs with larger average degrees (32 and 64). Changing the amount of work per thread does not change LAV's scaling behavior.



(a) Per-thread work: 2^{21} rows (b) Per-thread work: 2^{22} rows

Fig. 14. Weak scaling of LAV with random graphs.

VII. Related Work

Locality optimizations: CSR segmenting in Cagra [40] preprocesses the graph, dividing it into smaller LLC-fitting subgraphs. This approach is similar to our technique, but it does not consider density of the segments and vectorization. Binning techniques [11], [12] are discussed in detail in Section II-C. Milk [41] proposes language extensions to improve the locality of indirect memory accesses that are also observed in SpMV calculations. There are also compile-time and runtime techniques to accelerate programs with indirect memory accesses [42], [43]. Moreover, partitioning techniques can also be used for improving locality [44]. There are also locality optimizations targeting different primitives, such as sparse matrix dense matrix (multi vector) multiplications [45], [46].

The effectiveness of locality optimizations and auto-tuning approaches were previously evaluated on older hardware generations [13], [47]. We provide a similar analysis on a state-of-the-art multi-core system, which shows that modern hardware renders prior techniques significantly less effective, and motivates LAV, which is very effective on modern hardware.

Graph reordering: A large body of previous work targets relabeling vertices of graphs to provide better locality [20]–[22], [48]. These sophisticated techniques achieve high locality but incur large overheads. Rabbit Ordering [22] reduces preprocessing time using parallelization. However, it was recently shown that for many graph algorithms, Rabbit Ordering only

yields limited end-to-end performance benefits [27]. Finally, FEBA [48] tries to find dense subblocks and uses graph reordering to improve locality and computational efficiency.

Graph Processing Platforms: Several platforms reformulate graph algorithms as sparse matrix operations. GraphMat [8] maps a vertex program into generalized SpMV operations, and outperforms state-of-the-art graph processing frameworks. Pegasus [5] builds a large-scale system on Hadoop using generalized matrix-vector multiplication.

Vectorization: Many different SpMV vectorization methods have been proposed [9], [10], [16]–[19]. Their main aim is to maximize the vector unit utilization. Liu et al. [18] propose to use finite window sorting, which is similar to RFS but only considers a small block of rows. VHCC [17] devises a 2D jagged format for efficient vectorization of SpMV. Kreutzer et al. [16] use an RFS-like sorting for a limited number of rows. These works do not exploit the opportunities provided by CFS and segmenting. Furthermore, the matrices evaluated in these works are at least an order of magnitude smaller than our inputs. Finally, significant imbalance between the number of nonzeros per row in power-law graphs limits these prior works' applicability to these graphs. Of these related approaches, we compare to CSR5 [9] and CVR [10].

Vectorization is also studied in the graph applications domain. For instance, SlimSell [49] proposes a vectorizable graph format for accelerating breadth-first search. Grazelle [50] is a graph processing framework providing a new graph representation extended for efficient vectorization.

VIII. Conclusions and Future Work

In modern OOO processors, existing techniques to speed-up SpMV of large power-law graphs through vectorization and locality optimizations are not effective. To address this problem, we propose LAV, a new SpMV approach that leverages the input's power-law structure to extract locality and enable effective vectorization. LAV splits the input matrix into a dense and a sparse portion. The dense portion is stored in a new matrix representation, which is vectorization-friendly and exploits data locality. The sparse portion is processed using the standard CSR algorithm. We evaluate LAV on several real-world and synthetic graphs on a modern aggressive OOO processor, and find that it is faster than CSR (and prior approaches) by an average of 1.5x. LAV reduces the number of DRAM accesses by 35% on average.

We believe that LAV's ideas are applicable beyond CPU architectures. For example, in GPUs, LAV's segmenting idea can be generalized to consider GPU memory as the last level of the memory hierarchy. Moreover, CFS and similar optimizations may be useful in GPUs to improve memory coalescing. Future work could explore these opportunities.

Another promising line of future work is to incorporate LAV into graph frameworks such as Ligra [14] and GraphMat [8]. This work would require supporting the use of frontiers for dense pull-based implementations [14] and enabling masking for the output vector as in GraphBLAS [51].

Finally, users currently need to manually decide whether to employ LAV, based on their domain knowledge about the structure of the input graph (e.g., whether the graph is skewed or not). In future work, we plan to explore how this process can be automated.

Acknowledgements

This work was funded in part by NSF under grants CNS 1956007, CNS 1763658, CCF 1725734, and ISF grant 2005/17.

References

[1] Z. Jia, L. Wang, J. Zhan, L. Zhang, and C. Luo, "Characterizing data analysis workloads in data centers," in *2013 IEEE International Symposium on Workload Characterization (IISWC)*, Sep. 2013, pp. 66–76.

[2] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu, "BigDataBench: A big data benchmark suite from internet services," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2014, pp. 488–499.

[3] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Comput. Netw. ISDN Syst.*, vol. 30, no. 1-7, pp. 107–117, Apr. 1998. [Online]. Available: http://dx.doi.org/10.1016/S0169-7552(98)00110-X

[4] J. M. Kleinberg, "Authoritative sources in a hyperlinked environment," *J. ACM*, vol. 46, no. 5, pp. 604–632, Sep. 1999. [Online]. Available: http://doi.acm.org/10.1145/324133.324140

[5] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "Pegasus: A peta-scale graph mining system implementation and observations," in *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining*, ser. ICDM '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 229–238. [Online]. Available: https://doi.org/10.1109/ICDM.2009.14

[6] A. Buluc and J. R. Gilbert, "The Combinatorial BLAS: Design, implementation, and applications," *Int. J. High Perform. Comput. Appl.*, vol. 25, no. 4, pp. 496–509, Nov. 2011. [Online]. Available: http://dx.doi.org/10.1177/1094342011403516

[7] D. Buono, J. A. Gunnels, X. Que, F. Checconi, F. Petrini, T. Tuan, and C. Long, "Optimizing sparse linear algebra for large-scale graph analytics," *Computer*, vol. 48, no. 8, pp. 26–34, Aug 2015.

[8] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey, "GraphMat: High Performance Graph Analytics Made Productive," *Proc. VLDB Endow.*, vol. 8, no. 11, pp. 1214–1225, Jul. 2015. [Online]. Available: https://doi.org/10.14778/2809974.2809983

[9] W. Liu and B. Vinter, "CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, ser. ICS '15. New York, NY, USA: ACM, 2015, pp. 339–350. [Online]. Available: http://doi.acm.org/10.1145/2751205.2751209

[10] B. Xie, J. Zhan, X. Liu, W. Gao, Z. Jia, X. He, and L. Zhang, "CVR: Efficient Vectorization of SpMV on x86 Processors," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, ser. CGO 2018. New York, NY, USA: ACM, 2018, pp. 149–162. [Online]. Available: http://doi.acm.org/10.1145/3168818

[11] D. Buono, F. Petrini, F. Checconi, X. Liu, X. Que, C. Long, and T.-C. Tuan, "Optimizing sparse matrix-vector multiplication for large-scale data analytics," in *Proceedings of the 2016 International Conference on Supercomputing*, ser. ICS '16. New York, NY, USA: ACM, 2016, pp. 37:1–37:12. [Online]. Available: http://doi.acm.org/10.1145/2925426.2926278

[12] S. Beamer, K. Asanovic, and D. Patterson, "Reducing Pagerank Communication via Propagation Blocking," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2017, pp. 820–831.

[13] E.-J. Im, K. Yelick, and R. Vuduc, "Sparsity: Optimization framework for sparse matrix kernels," *The International Journal of High Performance Computing Applications*, vol. 18, no. 1, pp. 135–158, 2004. [Online]. Available: https://doi.org/10.1177/1094342004041296

[14] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 135146. [Online]. Available: https://doi.org/10.1145/2442516.2442530

[15] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP'13. New York, NY, USA: Association for Computing Machinery, 2013, pp. 456–471. [Online]. Available: https://doi.org/10.1145/2517349.2522739

[16] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop, "A unified sparse matrix data format for modern processors with wide SIMD units," *CoRR*, vol. abs/1307.6209, 2013. [Online]. Available: http://arxiv.org/abs/1307.6209

[17] W. T. Tang, R. Zhao, M. Lu, Y. Liang, H. P. Huyng, X. Li, and R. S. M. Goh, "Optimizing and auto-tuning scale-free sparse matrix-vector multiplication on Intel Xeon Phi," in *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, Feb 2015, pp. 136–145.

[18] 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*, ser. ICS '13. New York, NY, USA: ACM, 2013, pp. 273–282. [Online]. Available: http://doi.acm.org/10.1145/2464996.2465013

[19] L. Chen, P. Jiang, and G. Agrawal, "Exploiting Recent SIMD Architectural Advances for Irregular Applications," in *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, ser. CGO '16. New York, NY, USA: ACM, 2016, pp. 47–58. [Online]. Available: http://doi.acm.org/10.1145/2854038.2854046

[20] H. Wei, J. X. Yu, C. Lu, and X. Lin, "Speedup graph processing by graph ordering," in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD '16. New York, NY, USA: ACM, 2016, pp. 1813–1828. [Online]. Available: http://doi.acm.org/10.1145/2882903.2915220

[21] P. Boldi, M. Rosa, M. Santini, and S. Vigna, "Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks," in *Proceedings of the 20th International Conference on World Wide Web*, ser. WWW '11. New York, NY, USA: ACM, 2011, pp. 587–596. [Online]. Available: http://doi.acm.org/10.1145/1963405.1963488

[22] J. Arai, H. Shiokawa, T. Yamamuro, M. Onizuka, and S. Iwamura, "Rabbit Order: Just-in-time parallel reordering for fast graph analysis," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2016, pp. 22–31.

[23] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "ZombieLoad: Cross-privilege-boundary data sampling," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS 19. New York, NY, USA: Association for Computing Machinery, 2019, p. 753768. [Online]. Available: https://doi.org/10.1145/3319535.3354252

[24] P. Boldi and S. Vigna, "The Webgraph Framework I: Compression techniques," in *Proceedings of the 13th International Conference on World Wide Web*, ser. WWW '04. New York, NY, USA: ACM, 2004, pp. 595–602. [Online]. Available: http://doi.acm.org/10.1145/988672.988752

[25] R. Meusel, S. Vigna, O. Lehmberg, and C. Bizer, "Graph structure in the web — revisited: A trick of the heavy tail," in *Proceedings of the 23rd International Conference on World Wide Web*, ser. WWW '14 Companion. New York, NY, USA: ACM, 2014, pp. 427–432. [Online]. Available: http://doi.acm.org/10.1145/2567948.2576928

[26] A. Addisie, H. Kassa, O. Matthews, and V. Bertacco, "Heterogeneous memory subsystem for natural graph analytics," in *2018 IEEE International Symposium on Workload Characterization (IISWC)*. Los Alamitos, CA, USA: IEEE Computer Society, oct 2018, pp. 134–145. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/IISWC.2018.8573480

[27] V. Balaji and B. Lucia, "When is graph reordering an optimization? studying the effect of lightweight graph reordering across applications and input graphs," *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 203–214, 2018.

[28] H. Wang, W. Liu, K. Hou, and W.-c. Feng, "Parallel transposition of sparse data structures," in *Proceedings of the 2016 International Conference on Supercomputing*, ser. ICS'16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2925426.2926291

[29] "Performance monitoring and benchmarking suite," https://github.com/RRZE-HPC/likwid/, accessed: 2019-04-30.

[30] "Intel Math Kernel Library Inspector-executor Sparse BLAS Routines," https://software.intel.com/en-us/articles/intel-math-kernel-library-inspector-executor-sparse-blas-routines, Mar. 2015.

[31] W. Liu and B. Vinter, "CSR5-based SpMV on CPUs, GPUs and Xeon Phi," https://github.com/bhSPARSE/Benchmark_SpMV_using_CSR5, accessed: 2020-01-30.

[32] B. Xie, J. Zhan, X. Liu, W. Gao, Z. Jia, X. He, and L. Zhang, "Parallelized and vectorized SpMV on Intel Xeon Phi (Knights Landing, AVX512, KNL)," https://github.com/puckbee/CVR, accessed: 2020-01-30.

[33] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, Dec. 2011. [Online]. Available: https://doi.org/10.1145/2049662.2049663

[34] R. Meusel, O. Lehmberg, C. Bizer, and S. Vigna, "Web data commons - hyperlink graphs," http://webdatacommons.org/hyperlinkgraph/, accessed: 2020-01-30.

[35] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the Graph 500," *Cray Users Group (CUG)*, vol. 19, pp. 45–74, 2010.

[36] S. Beamer, K. Asanovic, and D. A. Patterson, "The GAP benchmark suite," *CoRR*, vol. abs/1508.03619, 2015. [Online]. Available: http://arxiv.org/abs/1508.03619

[37] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[38] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a social network or a news media?" in *WWW '10: Proc. the 19th Intl. Conf. on World Wide Web*. New York, NY, USA: ACM, 2010, pp. 591–600.

[39] J. D. McCalpin, "Stream: Sustainable memory bandwidth in high performance computers," University of Virginia, Charlottesville, Virginia, Tech. Rep., 1991-2007, a continually updated technical report. http://www.cs.virginia.edu/stream/. [Online]. Available: http://www.cs.virginia.edu/stream/

[40] Y. Zhang, V. Kiriansky, C. Mendis, S. Amarasinghe, and M. Zaharia, "Making caches work for graph analytics," in *2017 IEEE International Conference on Big Data (Big Data)*, Dec 2017, pp. 293–302.

[41] V. Kiriansky, Y. Zhang, and S. Amarasinghe, "Optimizing Indirect Memory References with Milk," in *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, ser. PACT '16. New York, NY, USA: ACM, 2016, pp. 299–312. [Online]. Available: http://doi.acm.org/10.1145/2967938.2967948

[42] 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*, ser. CGO'14. New York, NY, USA: Association for Computing Machinery, 2014, pp. 185–194. [Online]. Available: https://doi.org/10.1145/2544137.2544141

[43] M. M. Strout, A. LaMielle, L. Carter, J. Ferrante, B. Kreaseck, and C. Olschanowsky, "An approach for code generation in the sparse polyhedral framework," *Parallel Comput.*, vol. 53, no. C, pp. 32–57, Apr. 2016. [Online]. Available: https://doi.org/10.1016/j.parco.2016.02.004

[44] U. V. Catalyurek and C. Aykanat, "Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 7, pp. 673–693, July 1999.

[45] C. Hong, A. Sukumaran-Rajam, I. Nisa, K. Singh, and P. Sadayappan, "Adaptive sparse tiling for sparse matrix multiplication," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP'19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 300–314. [Online]. Available: https://doi.org/10.1145/3293883.3295712

[46] H. M. Aktulga, A. Buluc, S. Williams, and C. Yang, "Optimizing sparse matrix-multiple vectors multiplication for nuclear configuration interaction calculations," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, May 2014, pp. 1213–1222.

[47] R. Vuduc, J. W. Demmel, K. A. Yelick, S. Kamil, R. Nishtala, and B. Lee, "Performance optimizations and bounds for sparse matrix-vector multiply," in *SC '02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, Nov 2002, pp. 26–26.

[48] R. C. Agarwal, F. G. Gustavson, and M. Zubair, "A high performance algorithm using pre-processing for the sparse matrix-vector multipli-

cation," in *Supercomputing '92:Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*, 1992, pp. 32–41.

[49] M. Besta, F. Marending, E. Solomonik, and T. Hoefler, "SlimSell: A vectorizable graph representation for breadth-first search," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2017, pp. 32–41.

[50] S. Grossman, H. Litz, and C. Kozyrakis, "Making pull-based graph processing performant," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 246260. [Online]. Available: https://doi.org/10.1145/3178487.3178506

[51] A. Buluc, T. Mattson, S. McMillan, J. Moreira, and C. Yang, "Design of the GraphBLAS API for C," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2017, pp. 643–652.