WISE: Predicting the Performance of Sparse Matrix Vector Multiplication with Machine Learning

Serif Yesil*
University of Illinois at Urbana-Champaign
syesil2@illinois.edu

Azin Heidarshenas†
University of Illinois at Urbana-Champaign
heidars2@illinois.edu

Adam Morrison
Tel Aviv University
mad@cs.tau.ac.il

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

Abstract

Sparse Matrix-Vector Multiplication (SpMV) is an essential sparse kernel. Numerous methods have been developed to accelerate SpMV. However, no single method consistently gives the highest performance across a wide range of matrices. For this reason, a performance prediction model is needed to predict the best SpMV method for a given sparse matrix. Unfortunately, predicting SpMV's performance is challenging due to the diversity of factors that impact it.

In this work, we develop a machine learning framework called WISE that accurately predicts the magnitude of the speedups of different SpMV methods over a baseline method for a given sparse matrix. WISE relies on a novel feature set that summarizes a matrix's size, skew, and locality traits. WISE can then select the best SpMV method for each specific matrix. With a set of nearly 1,500 matrices, we show that using WISE delivers an average speedup of 2.4× over using Intel's MKL in a 24-core server.

CCS Concepts: • **Computing methodologies** → **Shared memory algorithms**.

Keywords: Sparse matrix, SpMV, Machine learning

1 Introduction

Sparse Matrix-Vector Multiplication (SpMV) is one of the most frequently-used kernels. It is used in processing both sparse linear systems as well as a variety of graphs (e.g., [7, 18, 20, 36]). Examples include graph algorithms such as Pagerank [7] and HITS [20]. Moreover, many applications utilizing the SpMV kernel are iterative, executing SpMV many times with the same sparse input matrix. As a result, SpMV often consumes substantial execution cycles, which makes it an important target to optimize.

Executing SpMV efficiently for a wide range of sparse input matrices from different domains is challenging. The reason is that matrices can have different characteristics, sometimes causing SpMV to issue low-locality memory access patterns. Further, the data-dependent behavior of some accesses makes them hard to predict and optimize for. Hence, many SpMV methods have been invented to speed-up SpMV (e.g., [21, 24, 28, 32, 38, 40]). Each method represents the matrix differently, to overcome irregularity challenges and to benefit from vector (i.e., SIMD) instructions.

Unfortunately, no single method consistently yields the best performance for all sparse matrices due to their different characteristics. Therefore, we need mechanisms to identify the best SpMV method for a given matrix, and then select it.

Traditionally, auto-tuners have been used for this purpose by creating simplified analytical models (e.g., [9, 17]). However, such models, which rely on a small set of matrix characteristics such as the number of rows, columns, and nonzeros, often fail to predict performance in the face of the rich space of existing matrices. The reason is that an SpMV method can behave completely differently depending on the skew and locality characteristics of the nonzeros of the matrix. In addition, auto-tuners can add substantial overhead to the execution.

In this paper, we address this problem. We propose WISE, an SpMV performance predictor. WISE uses machine learning (ML) to estimate the magnitude of the speedup of different SpMV methods over a baseline method for a given matrix. WISE then selects the fastest method. WISE uses a set of carefully designed features to summarize a matrix's locality, skew, and size characteristics. In addition, WISE uses a representative training set of matrices and is easy to extend.

To showcase WISE, we consider several optimizations, such as zero padding minimization for vectorization, column reordering to improve locality, and segmentation for Last-Level

*Now at Nvidia. He can be reached at syesil@nvidia.com.
†Now at Apple.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PPoPP '23, February 25-March 1, 2023, Montreal, QC, Canada
© 2023 Association for Computing Machinery.
ACM ISBN 979-8-4007-0015-6/23/02…$15.00
https://doi.org/10.1145/3572848.3577506

Cache (LLC) locality. These optimizations are used in popular methods such as SELLPACK [28], Sell-c-σ [21], and LAV [40]. We perform a detailed analysis of many sparse matrices and several popular SpMV methods. Our analysis reveals the requirements for a successful ML-based performance predictor. Based on this analysis, we develop an ML model to predict the speedup of SpMV methods. We then use it to select the highest-performing SpMV method.

Our ML model relies on a set of novel matrix features that characterize the sparse matrices' size, skew, and locality traits. To summarize skew properties, WISE uses general statistics such as the mean, standard deviation, Gini index, and p-ratio of nonzero distributions of rows and columns. To identify matrix locality characteristics, WISE uses statistics of the distribution of nonzeros in a 2D tiled version of the matrix. Furthermore, WISE uses additional statistics designed to summarize the structure of the tiles.

We analyze a set of nearly 1,500 matrices of diverse locality and skew behavior. We show that, using WISE, we attain an average speedup of 2.4× over using Intel's Math Kernel Library (MKL) in a 24-core server. To put this number in perspective, an oracle approach that picks the fastest method for each matrix attains an average speedup of 2.5×. Furthermore, WISE achieves a 1.14× average speedup over the more advanced Intel MKL inspector-executor, with less than 50% of its preprocessing overhead. Thanks to WISE's user-transparent approach, WISE can be an effective extension to an existing math library.

The contributions of this paper are:

- An analysis of the challenges of predicting SpMV performance.
- A set of matrix features that characterize a sparse matrix's size, skew, and locality, and can be used by an ML model.
- An ML-based framework called WISE that selects a high-performance SpMV method for a given sparse matrix.
- An evaluation of WISE for a large number of matrices.

2 Background & SpMV Optimization Space

SpMV computes $y = Ax$, where A is a sparse matrix and y and x are dense output and input vectors, respectively. Every element of y is computed as $y_i = \sum_{j=0}^{n-1} A_{i,j} \cdot x_j$, for $0 \leq i \leq m$-1, where m is the dimension of y and n is the dimension of x. In this section, we discuss the baseline implementation of SpMV with the Compressed Sparse Row (CSR) format, and some popular optimized SpMV methods.

2.1 CSR Format and SpMV Implementation with CSR

CSR uses three arrays to store a sparse matrix: *values*, *column ids*, and *row pointers* [14]. The *values* array contiguously stores the nonzero elements of all the matrix rows. Each entry in *column ids* corresponds to an entry in *values* and stores that element's column index. The *row pointers* array stores the index of the first nonzero element of each row.

SpMV with CSR iterates over the matrix rows and calculates the dot product of the row and the input vector. It can be parallelized over the rows. There are different ways to schedule rows to threads in a parallel implementation. We consider three ways: dynamic (Dyn), static (St), and static contiguous (StCont). Dyn and St assign K rows at a time to threads—either dynamically or round-robin statically. StCont divides the rows by the number of threads and assigns the resulting groups of contiguous rows statically to threads.

2.2 Implementing SpMV with Vectorization

We consider vectorized SpMV implementations, where threads use vector instructions to compute over multiple rows simultaneously. Our goal is to evaluate a range of vectorization optimizations. We consider three types of optimizations. The first one is *zero padding minimization* [21], which reduces the unnecessary compute and memory overhead introduced by zeros. Such zeros often appear when multiple rows with different numbers of nonzeros are processed together with a vector instruction. The second optimization is *column reordering* [40], which moves columns with a high number of nonzero elements together. The goal is to place the input vector elements frequently accessed together in the same cache line, increasing locality and reducing the impact of irregular memory accesses. The third optimization is *segmenting* [40], which processes sets of columns at a time. The goal is to fit frequently-accessed portions of the input vector into the LLC, thereby improving LLC locality.

To model different levels of these optimizations, we pick three effective vectorization methods, namely SELLPACK [28], Sell-c-σ [21], and LAV [40]. In addition, we also use specialized versions of Sell-c-σ and LAV. We describe all of them next. In our discussion, we use the example sparse matrix shown in Figure 1a.

Sliced ELLPACK (SELLPACK). SELLPACK groups c consecutive rows of the sparse matrix into *chunks*, packing the nonzeros of c adjacent rows together. All rows in a chunk are processed together with the same vector instructions. As a result, if the rows in a chunk have different numbers of nonzeros, they are padded to the same length. Figure 1b shows the SELLPACK format of the example matrix with c=2. SELLPACK can introduce many zeros due to padding, especially when the input matrix has an unbalanced distribution of nonzeros across rows. The chunk size is typically given by the width of the machine's vector unit and determines the degree of padding. For SELLPACK, we use StCont and Dyn scheduling because they are generally the fastest.

Sell-c-σ. Sell-c-σ reduces the zero padding of SELLPACK. It first considers groups of σ consecutive rows of the matrix. Then, it reorders the rows within each group based on their number of nonzeros in descending order. With this operation, each of the resulting groups of c consecutive rows is likely to have rows with a similar number of nonzeros. Hence, as vectorization is applied, the amount of padding will be

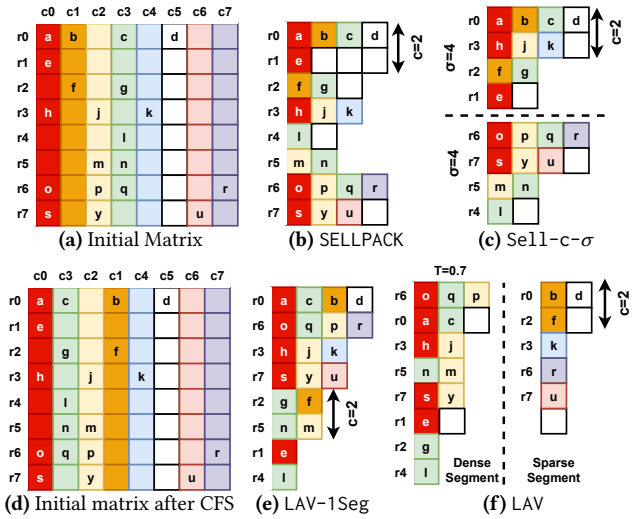


Figure 1. Different formats of an example sparse matrix.

smaller. Figure 1c shows the `Sell-c-`σ format of the example matrix with σ=4 and c=2.

σ needs to be tuned for each matrix. The best value of σ depends on the distribution of nonzeros across the rows of the matrix. If most rows have similar numbers of zeros, σ can be small, and the padding will be tolerable. However, if the number of nonzeros is highly imbalanced, keeping padding tolerable requires large σ values. In this case, many rows get reordered, which typically causes the loss of spatial and temporal locality for the input vector because close-by rows tend to have similar nonzero patterns. For `Sell-c-`σ, we use StCont and Dyn scheduling, as they are generally the fastest.

`Sell-c-R`. `Sell-c-R` sets σ to the number of rows in the matrix. This method is beneficial for matrices with a very imbalanced distribution of the nonzeros to rows. For these matrices, `Sell-c-R` reduces padding at the expense of poor cache locality for the input vector. We refer to such reordering of all rows as *Row Frequency Sorting* (RFS) [40]. `Sell-c-R` uses Dyn scheduling, as the large difference of nonzeros *across* chunks creates load imbalance with static scheduling.

`LAV-1Seg`. In matrices for power-law graphs [22, 27], the number of zeros in both rows and columns is highly imbalanced. As a result, the input vector has poor temporal and spatial locality and suffers frequent LLC misses. To address this problem, `LAV-1Seg` (for `LAV` [40] *with one segment*) orders the columns based on decreasing numbers of nonzeros—a technique called *column frequency sorting* (CFS) [40]. Then, it applies the transformations of `Sell-c-R`. With CFS, columns with similar nonzero distributions are often processed in close temporal succession, reusing input vector elements.

Figure 1d shows the initial matrix after performing CFS. Then, `LAV-1Seg` applies RFS. The result is Figure 1e.

`LAV`. If the matrix is large, input vector elements are evicted from the LLC before being reused. Consequently, the complete `LAV` algorithm [40] takes the matrix after CFS and partitions it into groups of consecutive columns called *Segments*. Then, it applies the transformations of `Sell-c-R` to each segment. A segment should be small enough so that the input vector elements corresponding to the segment fit in the LLC and get reused. In practice, `LAV` typically only needs to partition the matrix into two segments. The first segment includes a fraction T of the nonzeros, where the best T is generally $\geq$0.7. It is called the dense segment, as it has the most populated columns of the matrix. The second segment is called the sparse one.

When a segment is processed, the corresponding input vector elements are loaded into LLC and reused. Since both `LAV-1Seg` and `LAV` use RFS, we only consider Dyn scheduling, like `Sell-c-R`. Figure 1f shows the resulting matrix with `LAV`. Since we pick T=0.7, the dense segment includes columns 0, 3, and 2. We perform RFS in each segment.

Table 1 shows a summary of the `SpMV` methods used.

Table 1. `SpMV` methods used. SCH stands for scheduling.

Method	Params	Description
`CSR`	SCH	Performance of Dyn, St, and StCont depends on the skew characteristics of the matrix.
`SELLPACK`	SCH, c	c affects amount of padding, and depends on the SIMD width and on the matrix structure.
`Sell-c-`σ	SCH, c, σ	σ controls the row-ordering. It is tuned to minimize padding while not hurting locality.
`Sell-c-R`	c	Specialized `Sell-c-`σ version with σ=R.
`LAV-1Seg`	c	`LAV` with a single segment.
`LAV`	c, T	The higher the nonzero skew in the matrix is, the higher the chosen T should be.

Although the matrix formats for these five vectorization methods are different, we design a unified matrix format that we then use to implement `SpMV`. We describe the format in Appendix A.

3 Challenges Predicting `SpMV` Performance

Our goal is to develop a practical approach to pick the method and parameters that deliver the fastest `SpMV` execution for each input matrix. To this end, we characterize the `SpMV` execution with the different methods and parameters of Section 2 on a wide range of matrices. We use 136 large matrices from the SuiteSparse matrix collection [15] and 408 large matrices created with the RMAT graph generator [11]. SuiteSparse contains mostly scientific matrices, although it also has some social and web network graphs. RMAT is widely used (e.g., in the Graph500 benchmark [29] and GAP [5] benchmark suite), and can generate a much wider range of matrix types. We use it to create more matrices like those of social and web network graphs. We discuss the details of the matrices in Section 5.

To ensure that the matrices are large enough to get representative results, we use matrices with 1–67 million rows and columns. However, we limit the maximum number of

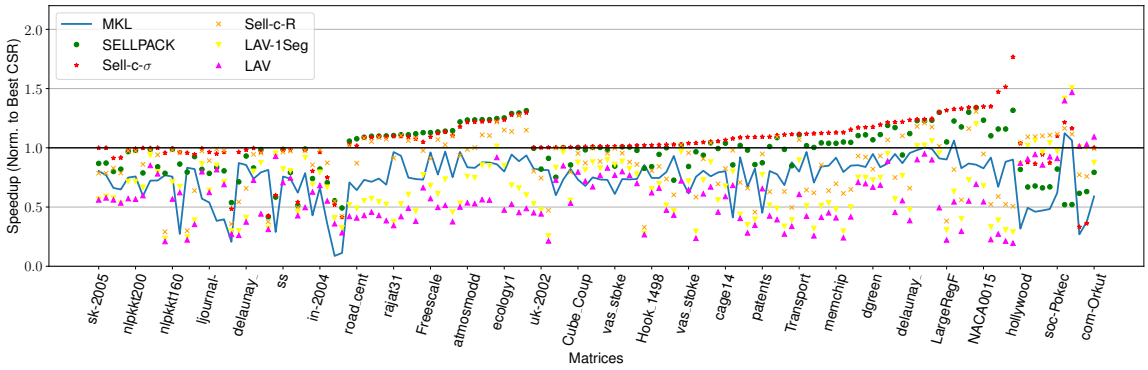


Figure 2. Speedup of different vectorized SpMV methods and MKL over the CSR implementation with the best scheduling policy for each particular SparseSuite matrix. The matrices are grouped by the fastest SpMV method. In the plot, higher is better.

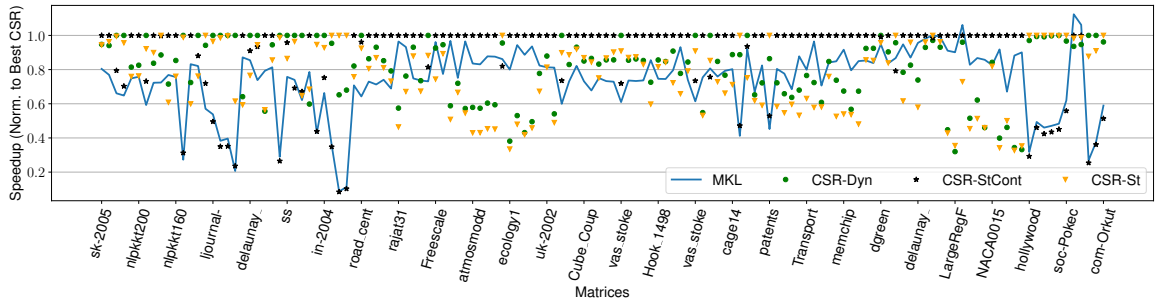


Figure 3. Speedup of CSR with different scheduling algorithms and MKL over the best CSR for the SparseSuite matrices.

nonzeros to 2 billion to ensure the matrices fit in the memory of a single shared-memory server. We discuss our platform and OpenMP-based parallelization strategy in Section 5.

Our analysis provides several insights:

① **The fastest method varies across matrices.** Figure 4 shows the distribution of the best-performing method for SpMV for the 136 SuiteSparse matrices. We see that different methods are faster for different matrices. For example, only 34 of these matrices achieve the best performance with CSR, which is the default format for many BLAS and graph frameworks. On the other hand, Sell-c-σ is the fastest method for 66 matrices. We also run MKL and observe that MKL does not yield the best performance for any of the matrices.

② **Within a method, the magnitude of the speedup over CSR varies.** Even when a method is the fastest for a set of matrices, its actual speedup over CSR varies considerably. Figure 2 shows, for the SuiteSparse matrices, the speedup of each method over the CSR implementation with the best scheduling policy for that matrix. In the figure, the horizontal line at 1.0 corresponds to that CSR implementation. We also report the results for MKL [13], which also uses the CSR representation. The matrices are grouped in the X-axis by their best method. The plot shows the names of a few of the matrices.

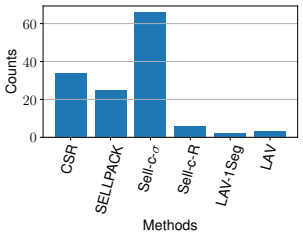


Figure 4. Fastest method for SpMV in SuiteSparse matrices.

Consider SELLPACK, which is the fastest in 25 consecutively placed matrices. In these matrices, its speedup ranges from 1.05 to 1.31×. On the other hand, Sell-c-σ is fastest for 66 matrices, and for these, the speedup ranges from 1.00 to 1.76×. Predicting the magnitude of the speedup can help us pick the best SpMV method when we later consider the preprocessing costs of the methods.

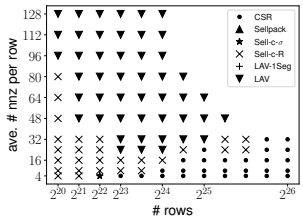
③ **Selecting the correct parameters for a method affects the speedups substantially.** For example, even simple scheduling choices affect the speedups substantially. Figure 3 shows the performance of CSR implementations that use *Dyn*, *St*, and *StCont* scheduling. For each of the 136 SuiteSparse matrices, the figure shows their speedup (always equal or less than 1) over CSR with the best scheduling. We see that a simple parameter like the scheduling choice significantly impacts performance, sometimes causing 10x slowdowns. *Dyn*, *St*, and *StCont* attain the best performance for 28, 16, and 92 matrices, respectively. *Dyn* is the fastest for web and social networks, while *St* and *StCont* perform best with scientific matrices. The figure also shows MKL's performance.

④ **An intricate relationship between matrix size, locality, and skew determines the best method.** To glimpse how the interaction between these three parameters determines the fastest method, we consider two experiments with

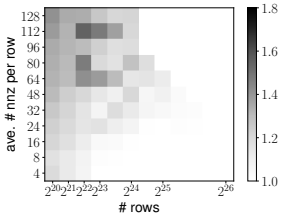
the RMAT-generated matrices. They show the impact of nonzero skew and nonzero locality, respectively.

In the first experiment, we analyze the skew characteristics of sparse matrices. We consider skew in terms of the distribution of nonzeros to rows of the sparse matrix. We use 100 high skew (HighSkew) and 100 low skew (LowSkew) matrices. In the second experiment, we use 100 matrices with high nonzero locality (i.e., most nonzeros are in areas close to the matrix diagonal) (*HighLoc*) and 100 with low locality (i.e., nonzeros are spread all over the matrix) (*LowLoc*). Section 4.5 describes the RMAT parameters used. In each 100-matrix set, we vary the number of rows and average nonzeros per row to model different characteristics.

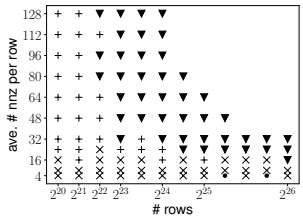
We consider the effect of skew first. We measure the execution time of each SpMV method of Section 2 for the *LowSkew* and *HighSkew* sets. Figures 5a and 5b show, for *LowSkew*, the fastest method and its speedup over the best CSR, respectively. A matrix is characterized by the number of rows (X-axis) and the average number of nonzeros per row (Y-axis). Figures 5c and 5d show the same data for *HighSkew* matrices.



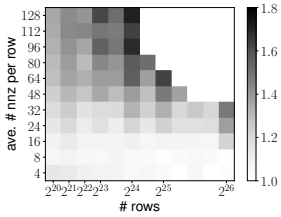
(a) *LowSkew*: Fastest method



(b) *LowSkew*: Speedup over CSR



(c) *HighSkew*: Fastest method

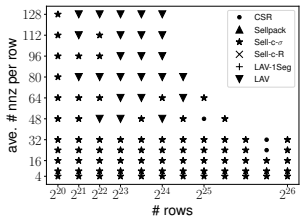


(d) *HighSkew*: Speedup over CSR

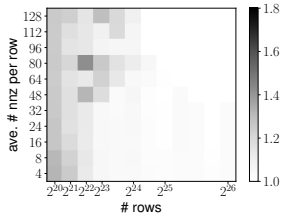
Figure 5. Behavior of matrices with different skew.

From Figures 5a and 5c, we see that LAV, LAV-1Seg, and Sell-c-R deliver the highest speedups most of the time. LAV typically outperforms other methods when the input matrix is larger than the LLC size (i.e., number of rows $\geq 2^{22}$) and the average number of nonzeros per row is high (> 16). LAV's speedup depends on the skew of the matrix; it is highest for the *HighSkew* matrices (Figure 5d). For matrices with few rows, a high average number of nonzeros per row, and high skew, LAV-1Seg often delivers the highest performance. This is because a single segment is enough. When matrices have a low average number of nonzeros per row, few rows and, especially, low skew, Sell-c-R is often the fastest method. This is because the input vector fits in the LLC, and there is no need for the more advanced LAV and LAV-1Seg formats. For large matrices with low average nonzeros per row, the speedups are small.

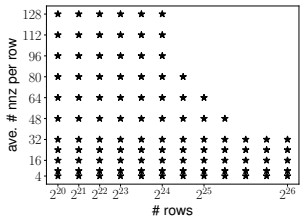
Figure 6 shows the same data for the *LowLoc* set (Figures 6a and 6b) and the *HighLoc* set (Figures 6c and 6d). Sell-c-σ is the fastest method for matrices with high locality (*HighLoc*). For *LowLoc*, Sell-c-σ is often the best, except for matrices with a high number of average nonzeros per row. In this case, LAV is best because it provides segmenting. Although the locality is limited, LAV creates a segment that can fit in the LLC. For *HighLoc*, this is unnecessary because caches work efficiently without segmenting. Finally, the magnitude of the Sell-c-σ speedups is higher for *HighLoc* than for *LowLoc*.



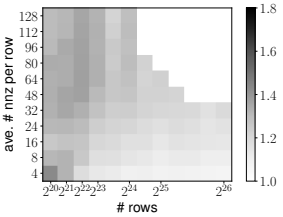
(a) *LowLoc*: Fastest method



(b) *LowLoc*: Speedup over CSR



(c) *HighLoc*: Fastest method



(d) *HighLoc*: Speedup over CSR

Figure 6. Behavior of matrices with different locality.

⑤ SuiteSparse matrices do not have many diverse behaviors. Figure 4 shows that the fastest method in SparseSuite matrices is often Sell-c-σ. However, our experiments with RMAT-generated matrices in Figure 5 show that LAV and LAV-1Seg are often the fastest. This discrepancy occurs because SuiteSparse matrices mostly come from the scientific domain or from graphs like road graphs; not many come from power-law graphs.

We can see SuiteSparse's bias by plotting a histogram of the *p-ratio* of nonzeros per row in SuiteSparse matrices. A p-ratio of p indicates that a p fraction of the rows has a $(1-p)$ fraction of the nonzeros in the matrix.

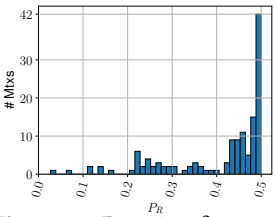


Figure 7. P-ratio of nonzeros per row in SuiteSparse.

Figure 7 shows the distribution of the nonzero p-ratio. We see that most of the SuiteSparse matrices have a p-ratio higher than 0.4. This means that many matrices have a balanced distribution of nonzeros to rows. This fact makes SELLPACK and Sell-c-σ methods effective. Moreover, SuiteSparse matrices generally have a low number of columns.

Most of them have less than 5M columns, allowing the input vector to fit in the LLC and reducing the need for LAV.

Overall, SuiteSparse favors methods such as Sell-c-σ and SELLPACK. If we want to train our prediction model with a representative set of matrices, we need to augment SuiteSparse with a broader range of matrices.

4 WISE: Picking the Best SpMV Method

Given how challenging it is to pick the best SpMV method for a given input matrix, we propose to leverage machine learning (ML). We develop WISE, an ML-based framework that selects a high-performance SpMV method for a given sparse matrix. We envision that WISE can be integrated in GraphBLAS/BLAS frameworks such as Intel's MKL [13]. WISE is designed to be transparent to the programmer, and integrate easily into existing systems.

4.1 Overall WISE Design

WISE consists of a novel sparse matrix feature set, a set of ML-based performance prediction models, and a heuristic to choose the best SpMV method considering the outputs of the performance prediction models.

Figure 8 shows the operation of WISE. First, WISE extracts the value of certain features from the input matrix (①). The feature set includes features to identify the sparse matrix size, and the skew and locality characteristics of the distribution of nonzeros. In the second step, these features are passed to a set of ML performance models that predict the speedup of the methods over the best CSR. (②). WISE has a performance prediction model for each combination of method and parameter values. Next, WISE picks the method and parameter values predicted to deliver the highest speedup while including the preprocessing cost (③). Then, it transforms the matrix layout from CSR to the layout for the selected method and parameter pair (④). If WISE chooses CSR, no transformation is needed. Finally, we run SpMV with the chosen method, parameters, and matrix layout (⑤).

4.2 Sparse Matrix Features

In Section 3, we observed that a sparse matrix's size, nonzero skew, and nonzero locality characteristics substantially impact the performance of an SpMV method. Based on these insights and previous analyses [22], we choose features to build our ML-based performance models. We start by logically breaking the matrix into K^2 tiles with n_R/K rows and n_C/K columns per tile, where n_R and n_C are the number of rows and columns of the matrix, respectively. We pick $K = 2048$, based on the size of sparse matrices and the L2 cache size. We call rows and columns of tiles as row and column blocks, respectively (Figure 9). We then consider the distribution of nonzeros across rows (R), columns (C), tiles (T), row blocks (RB), and column blocks (CB).

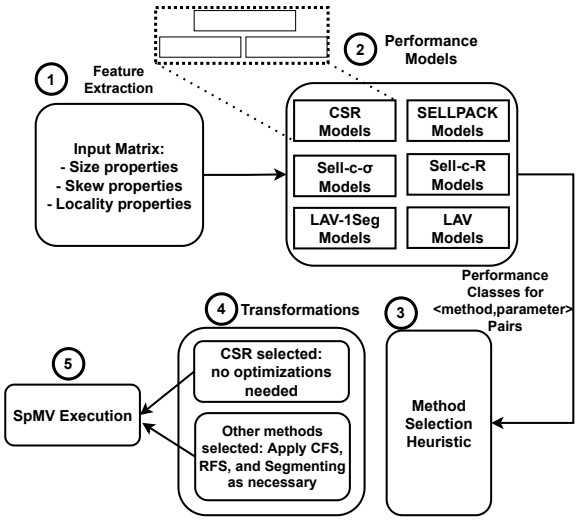


Figure 8. WISE operation.

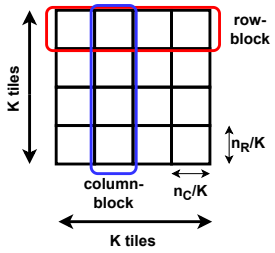


Figure 9. Tiling a matrix.

To characterize each distribution, we use the mean (μ), standard deviation (σ), variance (σ^2), *min*, *max*, *Gini coefficient (G)*, and *p-ratio (P)*. *G* and *P* are statistical measures that calculate the imbalance (or inequality) in a distribution [22]. In a maximally-imbalanced distribution (i.e., all nonzeros are in one bucket), *G* gets close to 1, and *P* gets close to 0. A perfectly-balanced distribution has a *G*=0 and *P*=0.5. Further, we also record the number of buckets that are not zero (*ne*)—i.e., the number of buckets that have some nonzeros. We use $stat_{dist}$ as a naming convention for these statistics, where *stat* is the name of the summary statistic and *dist* is the distribution name (R, C, T, RB, and CB).

Next, we describe how we use these summary statistics and additional statistics to create the sparse matrix features for our ML-based performance predictors. Table 2 shows the features WISE extracts from the input matrix, grouped into matrix size, nonzero skew, and nonzero locality properties.

(1) Matrix Size Properties: WISE measures the number of rows (n_R), columns (n_C), and nonzeros (n_{nnz}). n_C gives information about the size of the input vector, while n_{nnz} gives information about the amount of work to be done.

(2) Nonzero Skew properties: WISE uses features that measure the skewness of the nonzero distribution across rows (*R*) and columns (*C*). The statistics collected are listed in Table 2.

The features of the *R* distribution determine the row scheduling and the padding for vectorization. The features of the *C* distribution determine the irregularity of the memory accesses to the input vector. Therefore, they can indicate the effectiveness of CFS in LAV-1Seg and LAV.

(3) Nonzero Locality Properties: WISE uses *T*, *RB*, and *CB* distributions to capture the nonzero locality characteristics

Table 2. Matrix features used in our WISE performance models.

Property	Distribution	Metrics	What They Determine
Matrix Size		n_R, n_C, n_{nnz}	
Nonzero Skew	*Rows (R)*	$\mu_R, \sigma_R, \sigma_R^2, G_R, P_R, min_R, max_R, ne_R$	Row scheduling and padding for vectorization
	Cols (C)	$\mu_C, \sigma_C, \sigma_C^2, G_C, P_C, min_C, max_C, ne_C$	Memory access patterns to the input vector
Nonzero Locality	*Tiles (T)*	$\mu_T, \sigma_T, \sigma_T^2, G_T, P_T, min_T, max_T, ne_T$	Memory access locality behavior across tiles
	RowBlocks (RB)	$\mu_{RB}, \sigma_{RB}, \sigma_{RB}^2, G_{RB}, P_{RB}, min_{RB}, max_{RB}, ne_{RB}$	Memory access locality in L1 and L2 caches
	ColBlocks (CB)	$\mu_{CB}, \sigma_{CB}, \sigma_{CB}^2, G_{CB}, P_{CB}, min_{CB}, max_{CB}, ne_{CB}$	
		uniqR, uniqC, GrX_uniqR, GrX_uniqC	
		potReuseR, potReuseC, GrX_potReuseR, GrX_potReuseC	Memory access locality in the last level cache (LLC)

of a sparse matrix. The features are shown in Table 2. For example, a low p-ratio in the T distribution tells that the nonzeros are concentrated in a few tiles. Therefore, the memory accesses of SpMV have a relatively high locality.

WISE also collects additional information on the layout of nonzeros inside each tile. Specifically, for tile i, $uniqR_i$ and $uniqC_i$ are the number of unique rows and of unique columns that contain nonzeros, respectively. If many nonzeros are in the same row or column, the accesses are likely to exploit locality. Furthermore, since data is laid out in cache lines, if the nonzeros are nearby, they may share caches lines and further enhance locality. Hence, WISE also measures GrX_uniqR_i and GrX_uniqC_i for tile i, which are unique rows and unique columns with nonzeros *grouped in groups of X adjacent rows (or columns)*. For example, if X=16, 16 adjacent columns (or rows) will count as a single ID. WISE uses $\{4, 8, 16, 32, 64\}$ values for X. To understand their use, consider 64-byte cache lines. In this case, a line may fit four 128-bit elements (and we are interested in X=4) or eight 64-bit elements (and we are interested in X=8). We can get a cache hit if more than one element of the cache line is accessed in close temporal succession.

WISE gets $uniqR_i$, $uniqC_i$, GrX_uniqR_i, and GrX_uniqC_i for each tile. Then, it sums the values across all the tiles and divides the result by the number of nonzeros in the matrix. The resulting values, which we call *uniqR*, *uniqC*, *GrX_uniqR*, and *GrX_uniqC*, are used as matrix features.

Finally, WISE measures, for each row i, the number of tiles where the row has at least one nonzero ($potReuseR_i$). It also measures a similar metric for each column i ($potReuseC_i$). If these metrics are high, there is potential for data reuse in the LLC, as data is reused across tiles. Also, WISE measures, for each group i of X consecutive rows, the number of tiles where the group has at least one nonzero ($GrX\_potReuseR_i$). For each group i of X consecutive columns, it also measures a similar metric, called $GrX\_potReuseC_i$. Finally, WISE takes the average of these measurements across all rows (*potReuseR*), all columns (*potReuseC*), all groups of X consecutive rows (*GrX_potReuseR*), and all groups of X consecutive columns (*GrX_potReuseC*). These averages are used as matrix features (Table 2).

Note that the *R*, *C*, *T*, *RB*, and *CB* distributions are not the features used to train our ML-based performance prediction models. Instead, we use the summary statistics calculated from the distributions.

4.3 Performance Prediction Models

WISE has a performance prediction model for each combination of method and its parameter values. We use *decision trees* for these performance models, which can be seen as creating simple decision rules inferred from the data features. The reason for using decision trees is that each feature uses a different range of values, and thus it is hard to normalize all features to the same range. For example, the number of rows and columns can be in millions while the Gini index is a number between 0 and 1.

Based on the methods and parameters of Table 1, we have the following WISE models. CSR has three models, corresponding to the Dyn, St, and StCont scheduling methods. SELLPACK has as many models as possible combinations of c values and scheduling mechanisms considered (StCont and Dyn). Sell-c-σ has as many models as possible combinations of c values, σ values, and scheduling mechanisms considered (StCont and Dyn). Sell-c-R and LAV-1Seg have as many models as c values, since they only use Dyn scheduling. Finally, LAV has as many models as the combinations of c values and T values, since it uses Dyn scheduling. We pick $c = \{4, 8\}$ because these are the widths of the vector instructions in the machine evaluated. We pick σ=$\{2^9, 2^{12}, 2^{14}\}$ to cover a range of behaviors: 2^9 and 2^{14} roughly correspond to the maximum values for which the input vector fits amply in the L1 cache and the L2 cache, respectively. Finally, we pick T=$\{0.7, 0.8, 0.9\}$ to cover different levels of nonzero skew. In the end, we have a total of 29 decision tree models for all SpMV methods and parameters considered.

It is possible to run into overfitting issues with decision trees. For this reason, we apply pruning techniques. First, we limit the maximum depth of the trees to 15 to avoid creating branches that have only a few samples in them. Second, we enable minimal cost-complexity pruning, with a threshold of 0.005. This maximum depth of the tree and this threshold for pruning are selected experimentally using grid search (Section 6.5). Also, our decision trees use the Gini measure for split criteria.

Speedup Classes: Each of the models predicts the execution time of the method and parameter value combination *relative* to the fastest CSR method. Specifically, a model can output one of seven classes (C0-C6), corresponding to ranges of relative execution time. These ranges are, from higher (i.e., slower execution) to lower (i.e., faster execution): C0 = (∞ - 1.05], C1 = (1.05 - 0.95], C2 = (0.95 - 0.85], C3 = (0.85 - 0.75],

C4 = (0.75 - 0.65], C5 = (0.65 - 0.55], and C6 = (0.55 - 0]. To create classes C1-C5, we created categories with a step size of 0.1 between speedup values ≈1× and ≈2×. C0 includes all the matrices that have a slowdown with the given method, and C6 all the matrices with more than 2× speedup. The classes with values lower than 1 represent speedups.

4.4 Choosing the SpMV Method

WISE chooses the {method, parameter} pair that delivers the highest speedup. If there is a speedup tie among different methods, WISE chooses the method with the smallest preprocessing cost. For this cost, we use the following order (from lower cost to higher): CSR, SELLPACK, Sell-c-σ, Sell-c-R, LAV-1Seg, and LAV. Further, to break the ties among different parameters of a method, we order the parameters of a method from smallest to largest. We empirically observe that smaller parameters give lower preprocessing time. For example, for T in LAV, the order is T = 70%, 80%, and 90%.

4.5 Creating a Representative Training Set

An ML model requires a representative training set to perform predictions. However, our analysis in Section 3 showed that the matrices in SuiteSparse do not have many diverse behaviors. Hence, to obtain a representative training set, we augment SuiteSparse with a set of matrices generated with the RMAT [11] random graph generator. We use carefully selected parameters to model the aspects not well represented in SuiteSparse. The RMAT generator has 3 parameters: (1) number of nodes, (2) average node degree, and (3) probabilities a, b, c, d for an edge to fall into each of the four quadrants (a+b+c+d=1). To place an edge, a quadrant of the matrix is picked according to the probabilities. The selected quadrant is again divided into four quadrants, and the process repeats until ending up at a single matrix cell.

To obtain skewed matrices, we start with the Graph500 [29] parameters, namely a=0.57, b=0.19, c=0.19, and d=0.05 (*HighSkew*). They generate a power-law graph. Then, we decrease the a parameter while increasing b, c, and d to have less skew while still creating a community structure (a>d). We analyze two graph types, with parameters a=0.46, b=0.22, c=0.22, and d=0.10 (*MedSkew*), and a=0.35, b=0.25, c=0.25, and d=0.15 (*LowSkew*). The p-ratio of nonzero distribution of rows for *HighSkew*, *MedSkew*, and *LowSkew* graphs is ≈0.1, ≈0.2, ≈0.3, respectively.

To obtain matrices with varying locality, we start with a=b=c=d=0.25 (Erdos-Renyi) (*LowLoc*), where the nonzeros are uniformly distributed across all columns and rows. We obtain matrices with nonzeros gathered around the diagonal by equally increasing the a and d parameters and decreasing b and c with the same amount. With *MedLoc* (a=d=0.35, b=c=0.15) and *HighLoc* (a=d=0.45, b=c=0.05), locality goes up. For *LowLoc*, *MedLoc*, and *HighLoc* graphs, the p-ratio of the rows' nonzero distribution is 0.4-0.5, showing little skew.

In addition to RMAT graphs, we also include random geometric graphs (RGG) [16]. RGGs are undirected spatial graphs. An RGG is generated by placing n vertices uniformly at random in a 2-dimensional unit grid. An edge connects the vertices if their distance in the 2D grid is below a given radius r. The parameter r is set to be $r = \sqrt{\text{degree}/(\#\text{ rows} * \pi)}$, where *degree* is the desired average degree of the random graph. We include *RGG* graphs to model the behavior of matrices with spatial structure. *RGG* graphs have high locality. Table 3 shows the parameters for the matrices generated.

Table 3. Parameters for the RMAT/RGG matrices.

Property	Matrix	Parameters
Skew	*HighSkew* (HS)	$a = 0.57$, $b = 0.19$, $c = 0.19$, $d = 0.05$
	MedSkew (MS)	$a = 0.46$, $b = 0.22$, $c = 0.22$, $d = 0.10$
	LowSkew (LS)	$a = 0.35$, $b = 0.25$, $c = 0.25$, $d = 0.15$
Locality	*LowLoc* (LL)	$a = b = c = d = 0.25$
	MedLoc (ML)	$a = d = 0.35$, $b = c = 0.15$
	HighLoc (HL)	$a = d = 0.45$, $b = c = 0.05$
	RGG (rgg)	r set based on avg. # of nonzeros per row

5 Experimental Setup

Machine. For our experiments, we use a 2.6 GHz Intel Gold 6126 Skylake shared-memory machine. It has two sockets, each with 12 cores, for a total of 24 cores. Each core has private 32 KB I and D L1 caches and a private 1 MB L2 cache. Each socket has a shared 19 MB LLC. The machine has 192 GB of main memory. It uses 4- and 8-wide vector instructions such as avx512f, avx512dq, avx512cd, avx512bw, and avx512vl.

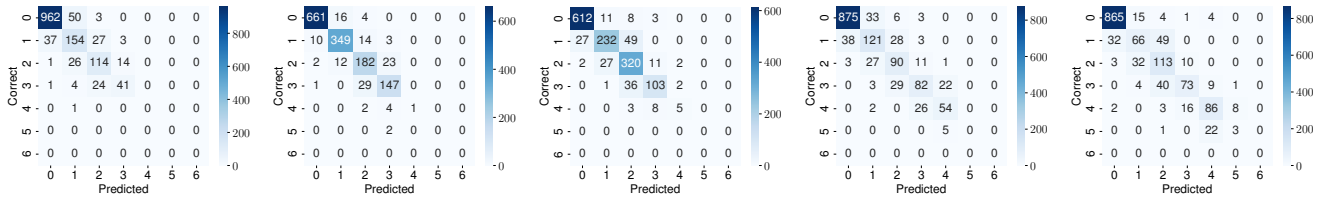
Matrices. To train and test WISE, we use 136 matrices from SuiteSparse [15] and 1326 matrices from graphs generated as described in Section 4.5. For SuiteSparse matrices, we use large matrices with 2^{20}–2^{26} rows, but less than 2 billion nonzeros to fit in a single machine's memory. For the RMAT and RGG matrices, we use matrices with 2^{20}, 2^{21}, 2^{22}, 2^{23}, 2^{24}, $2^{24.58}$, 2^{25}, $2^{25.30}$, $2^{25.58}$, $2^{25.80}$, and 2^{26} rows, with an average number of nonzeros per row equal to 4–128. We do not use more than 2 billion total nonzeros per matrix.

Implementations. We use C++ and OpenMP for parallelism. We rely on the OpenMP simd pragma for vectorization. Our implementation closely follows the one in [40]. We use the Intel compiler v2021 with the *O3* and *vec* flags. During execution, we run 24 threads, which are all pinned to physical cores. We use *numactl -i all* to interleave memory allocation across NUMA nodes.

Model Training & Testing. To test our model, we use k-fold cross-validation with k = 10. In this case, ten separate training and test sets are formed from our initial matrix set of 1462 matrices. Each of these training and test set pairs have disjoint matrices. To assess the model's accuracy, we report the combined confusion matrices of 10-fold cross-validation.

6 Experimental Results

6.1 Characteristics of the Matrix Set



(a) SELLPACK **(b)** Sell-c-σ ($\sigma = 2^{12}$) **(c)** Sell-c-R **(d)** LAV-1Seg **(e)** LAV ($T = 80\%$)

Figure 10. Accuracy of the WISE predictions for some representative models. In the figure, we use StCont scheduling for SELLPACK and Sell-c-σ, Dyn scheduling for the rest of the models, and $c = 8$ for all the models.

The addition of the randomly-generated matrices (RMAT and RGG) to the ones from SuiteSparse creates a more representative matrix set. To see why, we consider two aspects. First, we consider the nonzero skew of the random matrices, as measured by the distribution of the p-ratio of the nonzeros per row (P_R) in the random matrices. This distribution, shown in Figure 11, should be contrasted to the one for the SuiteSparse matrices in Figure 7.

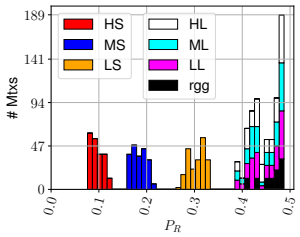


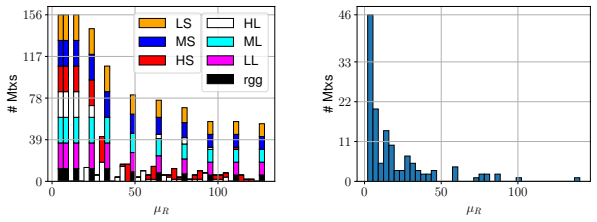
Figure 11. Distribution of P_R for the random matrices.

Figure 11 shows the RMAT matrices with high, medium, and low skew (HS, MS, and LS), the RMAT matrices that model high, medium, and low locality (HL, ML, and LL), and the RGG matrices. From the figure, we see that these randomly-generated matrices cover a wide range of P_R values—in contrast to Figure 7. HS, MS, and LS have P_R values of 0.1, 0.2, and 0.3, respectively. The matrices chosen to model different locality characteristics (rgg, LL, ML, and HL) have a P_R value of ≈0.4-0.5, which allows us to model the effect of locality without interference from skew behavior.

The second aspect we consider is the distribution of the average number of nonzeros per row (μ_R)—i.e., in graphs, the distribution of the average degree of the nodes. Figure 12 shows such distribution for the random matrices (Chart a) and for the SuiteSparse matrices (Chart b). We can see that the random matrices cover a more extensive range of μ_R values. In particular, the random matrix set includes matrices with large μ_R values, which results in higher utilization of vector units. Overall, the combination of both sets of matrices is able to train more effective ML models.

6.2 Classification Accuracy of WISE

Figure 10 shows the accuracy of WISE to classify matrices into speedup classes. The figure shows the confusion matrices for five representative models: SELLPACK, Sell-c-σ with $\sigma = 2^{12}$, Sell-c-R, LAV-1Seg, and LAV with $T = 80\%$. In the figure, we use StCont scheduling for SELLPACK and Sell-c-σ, Dyn scheduling for the rest of the models, and $c = 8$ for all the models. We show correct classes on the vertical axis and predicted ones on the horizontal axis. The



(a) Random matrices. **(b)** SuiteSparse matrices.

Figure 12. Distribution of the average number of nonzeros per row (μ_R).

figure shows the number of matrices falling on each grid point. Darker boxes correspond to higher matrix counts.

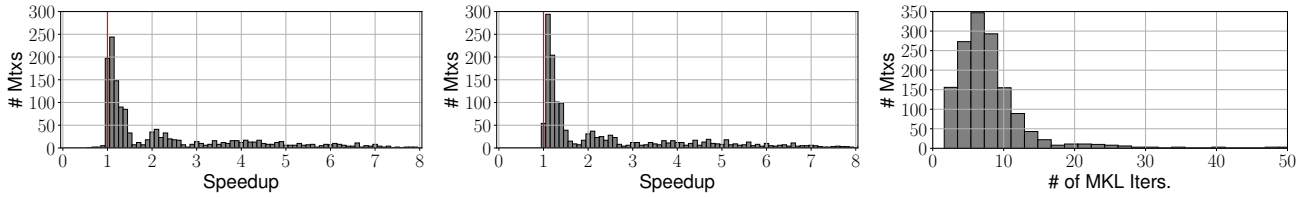
An exact match of the correct and predicted class lies on the diagonal of these matrices. The accuracy of our model can be given as the number of test cases that lie on the diagonal compared to the total number of test cases. However, we also need to consider the distance between the predicted and correct classes. Recall that the classes correspond to 10% intervals of normalized execution time. Therefore, a distance of one between predicted and correct classes means that the WISE estimation is within 10% of the correct execution time. Finally, overestimating the speedup (upper triangular part of the confusion matrix) may be less desirable than underestimating it (lower triangular part of the matrix).

From the confusion matrices, we observe that WISE predicts correctly for the large majority of matrices. Specifically, the accuracy of WISE is 87%, 92%, 87%, 84%, and 83% for SELLPACK, Sell-c-σ, Sell-c-R, LAV-1Seg, and LAV, respectively. Moreover, of the matrices that are classified incorrectly, most are classified at a distance of only one from the correct class. Such matrices are 94%, 89%, 90%, 91%, and 92% of the misclassified matrices for SELLPACK, Sell-c-σ, Sell-c-R, LAV-1Seg, and LAV, respectively. In general, the number of matrices with overestimated and underestimated speedup is comparable.

The results for the other models not shown are comparable to those in Figure 10. Overall, we conclude that WISE predicts the speedups for the different methods with high accuracy.

6.3 WISE: Speedup and Preprocessing Overhead

Figure 13a shows the distribution of speedups obtained by employing WISE over the MKL baseline for all the matrices.



(a) WISE speedup over MKL **(b)** Oracle speedup over MKL **(c)** WISE overhead

Figure 13. Distribution of WISE and oracle speedup over MKL, and WISE preprocessing time overhead.

We observe that WISE delivers speedups over MKL for the very large majority of matrices. The speedups are sometimes substantial, as they reach 4–8×. The average speedup is 2.4×. To put this number in perspective, Figure 13b repeats the experiment for an oracle approach that picks the fastest method for each matrix. We can see that Figures 13a and 13b are very similar. The average speedup of this oracle scheme is 2.5×. Hence, WISE is very close to the best approach that can be designed.

These speedup numbers do not include the preprocessing time overhead. The latter is composed of both format conversion from CSR and feature calculation. Figure 13c shows the distribution of the WISE preprocessing time overhead. We report the overhead in terms of the number of SpMV iterations with the MKL baseline that take the same time. From the figure, it can be computed that the average overhead of WISE is 8.33 MKL SpMV iterations.

6.4 Comparison to MKL Inspector-Executor

We now consider Intel's MKL *inspector-executor* (IE). This approach explores different methods before picking the best one for the matrix being considered. No further details are available about how MKL IE works. MKL IE can reduce the time taken by SpMV. However, it has a high preprocessing time overhead.

We measure that, for our matrix set, MKL IE attains an average speedup of 2.11× over the MKL baseline—again, without including the preprocessing time overhead. Further, the average preprocessing time overhead of MKL IE is 17.43 MKL SpMV iterations. Consequently, WISE is 1.14× faster than MKL IE even without considering the preprocessing time overhead. In addition, WISE has less than 50% of the MKL IE preprocessing time overhead.

6.5 Choosing Decision Tree Parameters

Two important parameters in the decision trees are the trees' maximum depth (D) and the degree of pruning (*ccp_alpha*, which we abbreviate as *ccp*). For D, we test $\{5, 10, 15, 20\}$ values; for *ccp*, we test $\{0, 0.001, 0.005, 0.01, 0.05, 0.1\}$ values. Table 4 shows the average speedups obtained by WISE over the MKL baseline with different D and *ccp* values. We see that *ccp* should be low (i.e., lower than 0.05) and D should be high (i.e., 10 or higher). Overall, we choose D=15 and *ccp*=0.005 for WISE.

Table 4. Average WISE speedup with different maximum depth (D) and pruning (*ccp*) values for the decision trees.

	ccp value					
	0	**0.001**	**0.005**	**0.01**	**0.05**	**0.1**
D=5	2.39	2.38	2.39	2.37	2.21	2.23
D=10	2.40	2.41	2.40	2.38	2.32	2.24
D=15	2.41	2.41	2.40	2.38	2.31	2.24
D=20	2.41	2.41	2.40	2.38	2.31	2.24

7 Related Work

Some previous works have used auto-tuning to speed-up sparse matrix operations (e.g., [9, 17, 34]). Such systems add potentially high auto-tuning overhead every time that they process a matrix. With WISE, after the models are created, the preprocessing overhead for each matrix is small.

Other previous works have utilized ML for method selection, i.e., to select the method that provides the highest performance for a given sparse matrix (e.g., [1, 23, 30, 41, 43–45]). WISE differs from previous works in two aspects. First, the output of the ML model in WISE is not an SpMV method; it is the potential speedup of a given method. The result is an extendable framework where we can add new methods without changing already existing models. Second, WISE does not only consider a given method but it also takes into account the parameters of the given method. For example, WISE picks the LAV method with given values for the c and T parameters.

Zhao et al. [43] use a novel summarization technique that represents a sparse matrix as a small 2D image. Then, they use CNNs on this image to identify the best format for the matrix. WISE also uses a 2D tiled representation of the matrix. However, it uses it to model the matrix's locality characteristics. In addition, WISE uses decision trees as the ML model. Furthermore, WISE uses an order of magnitude larger matrices to train and test the ML models.

Some proposals (e.g., SMAT [23]) use ML features that are driven by the actual SpMV methods used. For example, some of the features are the memory overheads of the particular matrix formats used. In contrast, the features that WISE chooses as inputs to the ML model are oblivious to the SpMV methods used.

There are other works that use ML to predict performance for other BLAS primitives such as matrix-matrix multiplication [39]. Moreover, auto-tuning has also been applied in graph processing—mainly focusing on algorithmic optimizations such as selecting push-pull execution modes based on the active vertex lists during execution [26, 35].

There are many examples of locality optimizations for SpMV and graph processing. For example, Cagra [42] preprocesses a graph, dividing it into smaller LLC-sized subgraphs. Other examples include binning techniques [4, 8]. For all these techniques, WISE can be extended with new performance models. Milk [19] is a set of language extensions to improve the locality of indirect memory accesses, which can also be used for SpMV calculations. There are also compile-time and run-time techniques to accelerate programs with indirect memory accesses [31, 33]. Moreover, partitioning techniques can also be used to improve locality [10].

Previous work targets relabeling vertices of graphs to provide better locality [2, 3, 6, 37]. These techniques achieve high locality but incur large overheads. In this work, we only consider the RFS and CFS reordering mechanisms. However, WISE can be potentially used as a first step to decide whether to apply more advanced reordering techniques, possibly eliminating overhead.

Many different SpMV vectorization methods have been proposed [12, 21, 24, 25, 32, 38]. Their main aim is to maximize the vector unit utilization. Liu et al. [25] propose to use finite window sorting, which is similar to RFS but only considers a small block of rows. VHCC [32] devises a 2D jagged format for efficient vectorization of SpMV. WISE's features can effectively assess the potential of these techniques.

8 Conclusions

To speed-up SpMV, this paper developed an ML framework called WISE that predicts the magnitude of the speedup of different SpMV methods over a baseline method for a given sparse matrix. WISE relies on a novel feature set that summarizes a matrix's size, nonzero skew, and nonzero locality. WISE is able to choose the best SpMV method and their parameters with high accuracy. For a set of nearly 1,500 matrices and several popular methods and optimizations, WISE achieved a 2.4× average speedup over Intel's MKL in a 24-core server. An oracle approach that picked the fastest method for each matrix obtained an average speedup of 2.5×. Further, WISE attained a 1.14× average speedup over Intel's MKL inspector-executor with less than 50% of its preprocessing overhead. Thanks to WISE's user-transparent approach, an extended WISE with more methods and optimizations is an effective addition to GraphBLAS/BLAS libraries like MKL.

Acknowledgments

This work was supported by NSF under grants CCF 2028861 and CNS 1763658.

Appendix A: Unified Matrix Format

We devise a single matrix representation from which the compiler can quickly extract the information needed for vectorization in SELLPACK, Sell-c-σ, Sell-c-R, LAV-1Seg, and LAV. We call the representation *Segmented Reordered Vector Packing* (*SRVPack*).

SRVPack supports partitioning the matrix data into two structures like LAV: one for the dense segment and one for the sparse one. Each structure contains the nonzeros of the rows in the segment with RFS applied like Sell-c-R. Figure 14 shows the representation of the matrix in Figure 1a in the new format. Each segment has a *row_order* array to keep track of the new row order. All the chunks of c rows in the segment are placed in sequence, one after the other, following the order given by the segment's *row_order* array. In each chunk, if a row has fewer columns than the rest of the rows in the chunk, it is padded with zeros. Similarly to CSR, there is a *vals* array that stores the values of the elements and a *col_ids* array that stores the corresponding column indices. These 2D arrays have a Y-dimension equal to c and an X-dimension equal to the sum of the lengths of the chunks of the segment. Finally, an $offset$ array stores the index of the first nonzero of each chunk. The overhead of creating this unified format is part of the pre-processing overhead.

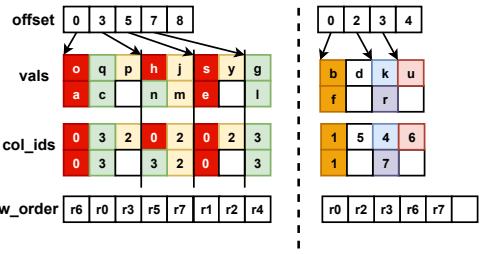


Figure 14. *SRVPack* representation of the matrix in Figure 1a.

Thanks to this unified format, we use a single SpMV implementation for all the matrix representations discussed in Section 2.2.

References

[1] Walid Abu-Sufah and Asma Abdel Karim. 2013. Auto-tuning of Sparse Matrix-Vector Multiplication on Graphics Processors. In *Supercomputing*, Julian Martin Kunkel, Thomas Ludwig, and Hans Werner Meuer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 151–164.

[2] R. C. Agarwal, F. G. Gustavson, and M. Zubair. 1992. A high performance algorithm using pre-processing for the sparse matrix-vector multiplication. In *Supercomputing '92:Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*. 32–41.

[3] J. Arai, H. Shiokawa, T. Yamamuro, M. Onizuka, and S. Iwamura. 2016. Rabbit Order: Just-in-Time Parallel Reordering for Fast Graph Analysis. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 22–31. https://doi.org/10.1109/IPDPS.2016.110

[4] S. Beamer, K. Asanovic, and D. Patterson. 2017. Reducing Pagerank Communication via Propagation Blocking. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 820–831. https://doi.org/10.1109/IPDPS.2017.112

[5] Scott Beamer, Krste Asanovic, and David A. Patterson. 2015. The GAP Benchmark Suite. *CoRR* abs/1508.03619 (2015). arXiv:1508.03619 http://arxiv.org/abs/1508.03619

[6] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered Label Propagation: A Multiresolution Coordinate-free Ordering for Compressing Social Networks. In *Proceedings of the 20th International Conference on World Wide Web* (Hyderabad, India) *(WWW '11)*. ACM, New York, NY, USA, 587–596. https://doi.org/10.1145/1963405.1963488

[7] Sergey Brin and Lawrence Page. 1998. The Anatomy of a Large-scale Hypertextual Web Search Engine. *Comput. Netw. ISDN Syst.* 30, 1-7 (April 1998), 107–117. https://doi.org/10.1016/S0169-7552(98)00110-X

[8] 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 2016 International Conference on Supercomputing* (Istanbul, Turkey) *(ICS '16)*. ACM, New York, NY, USA, Article 37, 12 pages. https://doi.org/10.1145/2925426.2926278

[9] Alfredo Buttari, Victor Eijkhout, Julien Langou, and Salvatore Filippone. 2007. Performance Optimization and Modeling of Blocked Sparse Kernels. *Int. J. High Perform. Comput. Appl.* 21, 4 (Nov. 2007), 467–484. https://doi.org/10.1177/1094342007083801

[10] U. V. Catalyurek and C. Aykanat. 1999. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Transactions on Parallel and Distributed Systems* 10, 7 (July 1999), 673–693. https://doi.org/10.1109/71.780863

[11] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A Recursive Model for Graph Mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining (SDM)*. 442–446. https://doi.org/10.1137/1.9781611972740.43

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

[13] Intel Corp. 2015. 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.

[14] Intel Corp. 2015. Sparse BLAS CSR Matrix Storage Format. https://www.intel.com/content/www/us/en/develop/documentation/onemkl-developer-reference-c/top/appendix-a-linear-solvers-basics/sparse-matrix-storage-formats/sparse-blas-csr-matrix-storage-format.

[15] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (Dec. 2011), 25 pages. https://doi.org/10.1145/2049662.2049663

[16] Daniel Funke, Sebastian Lamm, Ulrich Meyer, Peter Sanders, Manuel Penschuck, Christian Schulz, Darren Strash, and Moritz von Looz. 2019. Communication-free Massively Distributed Graph Generation. *J. Parallel and Distrib. Comput.* 131, C (2019).

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

[18] Z. Jia, L. Wang, J. Zhan, L. Zhang, and C. Luo. 2013. Characterizing data analysis workloads in data centers. In *2013 IEEE International Symposium on Workload Characterization (IISWC)*. 66–76. https://doi.org/10.1109/IISWC.2013.6704671

[19] Vladimir Kiriansky, Yunming Zhang, and Saman Amarasinghe. 2016. Optimizing Indirect Memory References with Milk. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation* (Haifa, Israel) *(PACT '16)*. ACM, New York, NY, USA, 299–312. https://doi.org/10.1145/2967938.2967948

[20] Jon M. Kleinberg. 1999. Authoritative Sources in a Hyperlinked Environment. *J. ACM* 46, 5 (Sept. 1999), 604–632. https://doi.org/10.1145/324133.324140

[21] Moritz Kreutzer, Georg Hager, Gerhard Wellein, Holger Fehske, and Alan R. Bishop. 2014. A Unified Sparse Matrix Data Format for Efficient General Sparse Matrix-Vector Multiplication on Modern Processors with Wide SIMD Units. *SIAM Journal on Scientific Computing* 36, 5 (Jan 2014), C401–C423. https://doi.org/10.1137/130930352

[22] Jérôme Kunegis and Julia Preusse. 2012. Fairness on the Web: Alternatives to the Power Law. In *Proceedings of the 4th Annual ACM Web Science Conference* (Evanston, Illinois) *(WebSci '12)*. Association for Computing Machinery, New York, NY, USA, 175–184. https://doi.org/10.1145/2380718.2380741

[23] 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* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 117–126. https://doi.org/10.1145/2491956.2462181

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

[25] 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 International ACM Conference on International Conference on Supercomputing* (Eugene, Oregon, USA) *(ICS '13)*. ACM, New York, NY, USA, 273–282. https://doi.org/10.1145/2464996.2465013

[26] Ke Meng, Jiajia Li, Guangming Tan, and Ninghui Sun. 2019. A Pattern Based Algorithmic Autotuner for Graph Processing on GPUs. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming* (Washington, District of Columbia) *(PPoPP '19)*. Association for Computing Machinery, New York, NY, USA, 201–213. https://doi.org/10.1145/3293883.3295716

[27] Robert Meusel, Sebastiano Vigna, Oliver Lehmberg, and Christian Bizer. 2014. Graph Structure in the Web — Revisited: A Trick of the Heavy Tail. In *Proceedings of the 23rd International Conference on World Wide Web* (Seoul, Korea) *(WWW '14 Companion)*. ACM, New York, NY, USA, 427–432. https://doi.org/10.1145/2567948.2576928

[28] Alexander Monakov, Anton Lokhmotov, and Arutyun Avetisyan. 2010. Automatically Tuning Sparse Matrix-Vector Multiplication for GPU Architectures. In *High Performance Embedded Architectures and Compilers*, Yale N. Patt, Pierfrancesco Foglia, Evelyn Duesterwald, Paolo Faraboschi, and Xavier Martorell (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 111–125.

[29] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. 2010. Introducing the Graph 500. *Cray Users Group (CUG)* 19 (2010), 45–74.

[30] 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* (Newport Beach, California, USA) *(ICS '15)*. Association for Computing Machinery, New York, NY, USA, 99–108. https://doi.org/10.1145/2751205.2751244

[31] Michelle Mills Strout, Alan LaMielle, Larry Carter, Jeanne Ferrante, Barbara Kreaseck, and Catherine Olschanowsky. 2016. An Approach for Code Generation in the Sparse Polyhedral Framework. *Parallel Comput.* 53, C (April 2016), 32–57. https://doi.org/10.1016/j.parco.2016.02.004

[32] W. T. Tang, R. Zhao, M. Lu, Y. Liang, H. P. Huyng, X. Li, and R. S. M. Goh. 2015. 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). 136–145. https://doi.org/10.1109/CGO.2015.7054194

[33] Anand Venkat, Manu Shantharam, Mary Hall, and Michelle Mills Strout. 2014. Non-Affine Extensions to Polyhedral Code Generation. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Orlando, FL, USA) *(CGO'14)*. Association for Computing Machinery, New York, NY, USA, 185–194. https://doi.org/10.1145/2544137.2544141

[34] R. Vuduc, J. W. Demmel, K. A. Yelick, S. Kamil, R. Nishtala, and B. Lee. 2002. Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply. In *SC '02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*. 26–26. https://doi.org/10.1109/SC.2002.10025

[35] Hao Wang, Liang Geng, Rubao Lee, Kaixi Hou, Yanfeng Zhang, and Xiaodong Zhang. 2019. SEP-Graph: Finding Shortest Execution Paths for Graph Processing under a Hybrid Framework on GPU. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming* (Washington, District of Columbia) *(PPoPP '19)*. Association for Computing Machinery, New York, NY, USA, 38–52. https://doi.org/10.1145/3293883.3295733

[36] 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. 2014. BigDataBench: A big data benchmark suite from internet services. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 488–499. https://doi.org/10.1109/HPCA.2014.6835958

[37] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. 2016. Speedup Graph Processing by Graph Ordering. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. ACM, New York, NY, USA, 1813–1828. https://doi.org/10.1145/2882903.2915220

[38] 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 the 2018 International Symposium on Code Generation and Optimization* (Vienna, Austria) *(CGO 2018)*. ACM, New York, NY, USA, 149–162. https://doi.org/10.1145/3168818

[39] Zhen Xie, Guangming Tan, Weifeng Liu, and Ninghui Sun. 2019. IA-SpGEMM: An Input-Aware Auto-Tuning Framework for Parallel Sparse Matrix-Matrix Multiplication. In *Proceedings of the ACM International Conference on Supercomputing* (Phoenix, Arizona) *(ICS '19)*. Association for Computing Machinery, New York, NY, USA, 94–105. https://doi.org/10.1145/3330345.3330354

[40] Serif Yesil, Azin Heidarshenas, Adam Morrison, and Josep Torrellas. 2020. Speeding Up SpMV for Power-Law Graph Analytics by Enhancing Locality and Vectorization. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. https://doi.org/10.1109/SC41405.2020.00090

[41] Buse Yilmaz, Bariş Aktemur, María J. Garzarán, Sam Kamin, and Furkan Kiraç. 2016. Autotuning Runtime Specialization for Sparse Matrix-Vector Multiplication. *ACM Trans. Archit. Code Optim.* 13, 1, Article 5 (March 2016), 26 pages. https://doi.org/10.1145/2851500

[42] Y. Zhang, V. Kiriansky, C. Mendis, S. Amarasinghe, and M. Zaharia. 2017. Making caches work for graph analytics. In *2017 IEEE International Conference on Big Data (Big Data)*. 293–302. https://doi.org/10.1109/BigData.2017.8257937

[43] Yue Zhao, Jiajia Li, Chunhua Liao, and Xipeng Shen. 2018. Bridging the Gap between Deep Learning and Sparse Matrix Format Selection. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Vienna, Austria) *(PPoPP '18)*. Association for Computing Machinery, New York, NY, USA, 94–108. https://doi.org/10.1145/3178487.3178495

[44] Yue Zhao, Weijie Zhou, Xipeng Shen, and Graham Yiu. 2018. Overhead-Conscious Format Selection for SpMV-Based Applications. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 950–959. https://doi.org/10.1109/IPDPS.2018.00104

[45] Weijie Zhou, Yue Zhao, Xipeng Shen, and Wang Chen. 2020. Enabling Runtime SpMV Format Selection through an Overhead Conscious Method. *IEEE Transactions on Parallel and Distributed Systems* 31, 1 (2020), 80–93. https://doi.org/10.1109/TPDS.2019.2932931