Efficient Tiled Sparse Matrix Multiplication through Matrix Signatures
Süreyya Emre, Aravind Sukumaran-Rajam, Fabrice Rastello, Ponnuswamy Sadayyapan

To cite this version:

HAL Id: hal-03117491
https://inria.hal.science/hal-03117491
Submitted on 21 Jan 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Efficient Tiled Sparse Matrix Multiplication through Matrix Signatures

Süreyya Emre Kurt  
University of Utah  
Salt Lake City, Utah  
semre@cs.utah.edu

Aravind Sukumaran-Rajam  
Washington State University  
Pullman, Washington  
a.sukumaranrajam@wsu.edu

Fabrice Rastello  
Univ. Grenoble Alpes, Inria, CNRS,  
Grenoble INP, LIG,  
38000 Grenoble, France  
fabrice.Rastello@inria.fr

P. Sadayyapan  
University of Utah  
Salt Lake City, Utah  
saday@cs.utah.edu

Abstract—Tiling is a key technique to reduce data movement in matrix computations. While tiling is well understood and widely used for dense matrix/tensor computations, effective tiling of sparse matrix computations remains a challenging problem. This paper proposes a novel method to efficiently summarize the impact of the sparsity structure of a matrix on achievable data reuse as a one-dimensional signature, which is then used to build an analytical cost model for tile size optimization for sparse matrix computations. The proposed model-driven approach to sparse tiling is evaluated on two key sparse matrix kernels: Sparse Matrix - Dense Matrix Multiplication (SpMM) and Sampled Dense-Dense Matrix Multiplication (SDDMM). Experimental results demonstrate that model-based tiled SpMM and SDDMM achieve high performance relative to the current state-of-the-art.

Index Terms—sparse matrix signature, sparse tiling, SpMM, SpMDM, Sparse Dense Matrix Multiplication, Multi-core

I. INTRODUCTION

Sparse Matrix Multi-vector multiplication (SpMM, also sometimes called Sparse-Matrix Dense-Matrix Multiplication, or SpMDM) and Sampled Dense-Dense Matrix Multiplication (SDDMM) are important kernels used in many domains like Fluid Dynamics, Data Analytics, Economic Modelling, and Machine Learning [15], [16]. In areas like Machine Learning and Artificial Neural Networks, these kernels are used iteratively over and over again, therefore optimized implementations are important for many software frameworks like Tensorflow [2] and PyTorch [29]. Several recent efforts have sought to exploit sparsity in deep learning, using an SpMM formulation [11], [17], [22]. Examples of the use of SpMM from numerical simulation include the Locally Optimal Block Preconditioned Conjugate Gradient (LOBPCG) method for finding eigenvalues of a matrix [4], [21], and iterative solvers with multiple right-hand sides, like the Krylov subspace iterative solvers that use SpMV at their core. Sampled Dense-Dense Matrix Multiplication (SDDMM) is a kernel that can be used as a core operation in an efficient formulation of factorization algorithms in machine learning, such as Alternating Least Squares (ALS) [19], Sparse Factor Analysis (SFA) [20] and topic modeling algorithms like Latent Dirichlet Allocation (LDA) [5] and Gamma Poisson (GaP) [37], as describe in detail by Zhao and Canny [8].

A. The challenge of tiling sparse matrix multiplication

The cost of performing arithmetic/logic operations on current processors is significantly lower than the cost of moving data from memory to the ALU (arithmetic/logic units), whether measured in terms of latency or throughput. The reduction of data movement between nodes of a parallel computing system, as well as within the memory hierarchy of each node, is critical for achieving high performance. Therefore data locality optimization is of fundamental importance. Tiling is a key technique for optimizing data movement in dense matrix computations like matrix-matrix multiplication, LU decomposition, and Cholesky factorization. Tiling can enable significant data reuse in levels of cache, so that the limited main-memory bandwidth does not constrain performance. While techniques for tiling of regular computations using dense arrays are well understood and have been incorporated into compilers [6], [26], [27], [39], data locality optimization for parallel irregular sparse computations remains a significant challenge. A number of research efforts have sought to develop compile-time analysis and transformation techniques for sparse computations [24], [30], [31], [36], [38] but the current state of knowledge and tools are quite far from being able to make a practical impact on optimizing SpMM to exceed performance of available implementations in libraries like Intel’s MKL.

A key issue with tiled parallel matrix computations is that of tile size selection, since the choice of tile sizes has a significant impact on performance. If the chosen tile sizes are too small, the cache is under-utilized and sub-optimal data reuse in cache results in high volume of expensive data movement from memory. If the tile sizes are too big, the cache capacity is insufficient to retain the data for reuse within the tile, resulting in excessive cache misses and high volume of data movement from memory. The issue of analytical modeling for effective tile-size selection for dense matrix computations has been the subject of significant research [9], [32], [35], [42]. However, to the best of our knowledge, no prior work has developed a model-driven approach for selection of tile sizes for any sparse matrix computation. While some recent work [12], [13] has focused on tiled sparse matrix computations, the choice of tile sizes was done using empirical heuristics and auto-tuning over a space of choices.
B. Sparse-matrix signatures for model-guided tiling

A significant challenge in modeling the impact of tile size on data movement with sparse matrix computations is that the amount of data movement depends not only on the number of non-zero elements but on pattern of non-zeros. We use a simple example to illustrate this. We first measured the volume of data movement and achieved performance using the SpMM implementation in Intel’s Math Kernel Library (MKL) for forming the product of a 100K × 100K banded sparse matrix (band-size of 97) and a 100K × 128 dense matrix. The data volume to/from main memory was measured to be 4 × 10^8 bytes and achieved performance was 20 GFLOPS. The non-zero elements of the banded sparse matrix were then randomly reordered by performing a randomized row/column permutation. So the total number of arithmetic operations remains unchanged, but the measured data volume from memory shot up to 10^{11} bytes and performance dropped to 0.5 GFLOPS. Thus, the 2D distribution of non-zeros of a sparse matrix can have a significant impact on performance. The question we address in this paper is: Can the impact of the 2D pattern of non-zero elements on performance of sparse matrix computations be captured in some form that can be effectively used to perform model-driven data-locality optimization for such computations?

In this paper, we address this problem and develop a novel approach to enable effective model-driven tiled execution of two important sparse-matrix primitivies – SpMM and SDDMM. The key to the modeling approach is to express data movement volume in terms of a compact one dimensional function signature for a sparse matrix that succinctly captures the impact of the non-zero structure of the matrix on data movement for the class of computations. The compact signature is then used to perform model-driven tile-size optimization. We also present a novel algorithm to efficiently generate the sparse-matrix signature. We demonstrate the utility of the new abstractions by developing new implementations of tiled SpMM and SDDMM for multicore/manycore processors.

The key contributions are as follows:

- We relate data movement requirements for sparse matrix computations to a sparse-matrix locality-signature that can be pre-computed once and reused for optimizing SpMM and SDDMM for execution on different target platforms.
- We develop a novel algorithm for efficient computation of the sparse-matrix locality-signature.
- We develop efficient model-guided tiled SpMM and SDDMM implementations based on use of the summarized sparse-matrix signatures.

II. BACKGROUND AND OVERVIEW OF APPROACH

In this section, we first provide some background on two sparse matrix computations of focus in this work, SpMM and SDDMM, and then present a high-level overview of key ideas behind the approach we develop for model-driven tiling of these sparse matrix computations.

### Algorithm 1: Sequential SpMM

```plaintext
input: CSR S[M][N], dense A[M][K], dense B[N][K]
output: float O[M][K]
for i = 0 to M-1 do
  for j = S.rowptr[i] to S.rowptr[i+1]-1 do
    for k = 0 to K-1 do
      O[i][k] += A[i][k] * B[colidx[j]][k]
```

### Algorithm 2: Sequential SDDMM

```plaintext
input: CSR P[M][N]
output: CSR P[M][N]
for i = 0 to M-1 do
  for j = S.rowptr[i] to S.rowptr[i+1]-1 do
    for k = 0 to K-1 do
      P.values[j] += A[i][k] * B[colidx[j]][k]
```

The most commonly used sparse matrix representation is the Compressed Sparse Row (CSR) representation [33]. In CSR, three 1D arrays are maintained: rowptr, colidx and values. The i-th entry of the rowptr array represents an offset to the start of a compacted set of entries in the values and colidx arrays that hold the numerical values and column-indices, respectively, of the non-zero elements in row i.

**SpMM:** Sparse Matrix-Matrix product (also called Sparse Matrix Multivector product) multiplies an input M × N sparse matrix S and an input N × K dense matrix I to produce a dense M × K matrix O, i.e., O = SI. Algorithm 1 shows the corresponding pseudocode. The outer i loop traverses all rows of S, and for each row, the j loop accesses the non-zero elements from the CSR representation of S and multiplies them with all elements (inner k loop) from the appropriate row of I (corresponding to the column index of the nonzero element of S) to accumulate to the elements of row i of O.

**SDDMM:** SDDMM computes the product of two dense matrices (A and B) and the result matrix is then subjected to Hadamard product (pointwise multiplication) with a sparse matrix S, i.e., P = S ⊙ AB. Algorithm 2 presents the SDDMM pseudocode. The outer most loop i in the first loop nest, iterates over all the rows of P and the j loop identifies the non-zero elements from the CSR representation of P. For each non-zero P element, a K way dot product of the corresponding row of A and a column of B are accumulated to P. The second loop nest performs an element-wise multiplication of the sparse matrix P by S and the result is stored in P.

These two sparse matrix computations have been the subject of several prior optimization efforts [1], [4], [7], [18], [28], [40], [41], but these efforts have not highlighted any relationship between these two computations that could enable the application of some common optimization strategies across the two codes. A key insight driving our work in this paper is that both computations can be viewed as instances of a common pattern with respect to data locality considerations. We present the SpMM and SDDMM computations below in
Data Reuse along Iteration Space Axes: In the abstract view of SpMM and SDDMM, we can observe a common property of the two computations with respect to data reuse. The SpMM computation as well as the computationally dominant 3D loop for SDDMM both feature a sparse 3D iteration space with two sparse dimensions \((i \text{ and } j)\) and one dense dimension \(k\). A dimension of the iteration space is considered dense if for any fixed index in all other dimensions, all iteration-space points along the considered dimension are fully instantiated if any iteration-space point is instantiated. Thus, for a particular pair of values for \(i \text{ and } j\), either all iteration-space points \((i, j, *)\) are instantiated or none are instantiated. In contrast, if we consider a particular pair of values for \(i \text{ and } k\), we may have only a subset of iteration-space points \((i, j, k)\) being instantiated. Therefore \(j\) is a sparse dimension in the iteration space. Similarly, \(i\) is a sparse dimension in the iteration space.

For both SpMM and SDDMM, all data dependencies, including input (read-read) dependences are along the iteration space axes, i.e., there is only one non-zero component in the multi-dimensional data dependence vector, with the reuse direction being different for each array.

The accessed data element from each array for each executed statement instance only depends on two out of the three loop indices; therefore the data element is reused at all active iteration space points as the “missing” loop index is varied. Further, each loop index is a reuse direction for one of the arrays. This property enables the development of efficient tiling strategies for sparse matrix computations (discussed in Sec. IV), as well as model-driven tile-size optimization by use of 1D distributions that serve as compact signatures of the 2D distribution of non-zero elements of a sparse matrix, as discussed next.

Consider a tiled execution of SpMM or SDDMM. The data footprint of the computational tile for each matrix will be the elements in a 2D slice of the index space of the matrix. Since reuse directions for all matrices are along iteration-space axes, and all iterators are reuse directions for some array(s), the extent of feasible reuse in a tile for any array element will correspond exactly to the number of data elements within the appropriate 1D sub-slice of one of the other arrays. With sparse matrices, the 2D data footprint of a 3D computational tile will be sparse. The extent of possible reuse of the two dense arrays in SpMM or SDDMM depends on the number of nonzero elements in row-segments and column-segments in the 2D index space of the sparse matrix data footprint.

In the converse view, only row-segments/column-segments of the sparse matrix that have non-zero elements will induce data movement for the dense matrices during execution of a tile. Thus the total number of row/column segments of the sparse-matrix that are active (have at least one non-zero element) in the 2D data-footprint of the computational tile will determine the volume of data movement of the dense matrices in SpMM/SDDMM. As we explain in detail in Sec. IV, the total data movement of the dense matrices can be modeled in terms of the total number of active row/column segments in the sparse matrix. The total number of active row segments in any 2D tiling of the sparse matrix is only a function of the tile-size along the column dimension and vice-versa. Thus, instead of having to contend with an arbitrary 2D distribution pattern of non-zeros in a sparse matrix, analytical modeling of data movement for SpMM/SDDMM is feasible using two 1D summarizations or signatures for a sparse matrix: the total number of active row-segments as a function of column-tile-size and the number of active column-segments as a function of row-tile-size.

III. RELATED WORK

In this section, we summarize prior research on optimizing SpMM and SDDMM.

a) Taco: Taco [18] is a C++ library which uses compiler techniques to generate kernels for tensor algebra operation. These operations can be for sparse or dense tensors having any possible dimensions. The kernels generated are already optimized and use OpenMP parallel pragma to parallelize. This library and its online code generation tool can be used to generate SpMM kernel, where all the tensors are 2D.

b) Intel Math Kernel Library: Intel MKL is one of the most commonly used BLAS and Sparse BLAS libraries for CPU’s. This library has highly optimized kernels for many sparse BLAS operations like SpMM, SpMV and SpGEMM. MKL supports various matrix representation like CSR, CSC, COO, etc. MKL library also supports AVX512 instructions and has kernels optimized especially for Xeon Phi architecture which results in significant performance gains [14].

c) Compressed Sparse Blocks based SpMM: Compressed Sparse Blocks (CSB) is a sparse matrix storage format which partitions and stores the matrices in smaller square blocks. This representation does not require any extra space than the commonly used CSR or CSC representations. Using CSB format for SpMM kernels shows significant improvement in SpMM as well as for SpMM transpose [3].
d) Data Locality Optimization for SPMM: Some recent efforts have addressed data-locality optimization for SpMM and SDDMM [12, 13, 25]. However, a significant difference between the developments we present in this paper and previous efforts is that of analytical modeling and compact characterization of sparse matrix signatures for such analytical modeling and tile size selection. These previous efforts have used empirical means to select tile sizes and the main focus has been to reorder the sparse matrix elements into highly clustered regions and use two different kernels to process non-zeros in heavily populated blocks versus sparsely populated blocks. We do not consider any sparse-matrix reordering or the use of different kernel execution strategies based on local non-zero density as done by these efforts. In contrast, our focus is on a new direction that can facilitate analytical modeling and optimization for sparse matrix computations like SpMM. We believe there are opportunities to combine ideas from these previous efforts with the matrix-signature based analytical modeling approach we develop in this paper.

e) Inspector/Executor Compiler Optimization: Inspector-executor strategies represent a promising direction, where a one-time execution of an inspector code that analyzes the specific non-zero structure of the sparse matrix can suitably enable efficient execution of the executor code that performs the intended sparse-matrix computation. But the development of effective inspector-executor strategies for arbitrary sparse-matrix computations remains a significant open challenge. While the optimization strategy for sparse-matrix computations presented in this paper does not directly seek to build an optimizing compiler, we believe that the sparse-matrix signatures we develop here can be used in developing an inspector/exector based optimizing compiler for a class of sparse-matrix computations exhibiting the axis-aligned data reuse property like SpMM and SDDMM.

Algorithm 5: Tiled SpMM

```
input : CSR S[M][N], dense I[N][K]
output: dense O[M][K]
1 for kk = 0 to ((K - 1)/Tk) - 1 do
2   kbound = min((kk+1)*Tk, K-1)
3 for jj = 0 to (N/Tj) - 1/j do
4   for ii = 0 to ((M - 1)/Ti) do
5     ibound = min((ii+1)*Ti, M)-1
6     for i = ii*Ti to ibound do
7       nnz_begin = S.Tj_tile[ii].rowptr[i]
8       nnz_end = S.Tj_tile[ii].rowptr[i+1]-1
9       for e = nnz_begin to nnz_end do
10      for k = kk*Tk to kbound do
11        j = S.colidx[e]
12        O[i][k] += S.value[e] * I[j][k]
```

IV. DATA MOVEMENT ANALYSIS AND OPTIMIZATION FOR TILED SPMM

Tiling is a well known technique to minimize data movement. A key consideration for tiled code is the choice of tile sizes along all tiled dimensions of the iteration space. In general, for a d-dimensional tile, d tile-size parameters must be chosen. The choice of tile size is usually driven by considerations of minimization of data movement and much work has focused on this problem for regular computations on dense arrays. Even for regular computations, optimal tile size selection is extremely hard and the state-of-practice relies heavily on auto-tuning, i.e., empirical search through the space of tile-size configurations, using actual execution of the tiled code on the target platform for different tile-size choices. Auto-tuning is often very time consuming, especially for high-dimensional loop nests because of number of tile-size combinations grows exponentially with the dimensionality of

---

**TABLE I: Definitions of some of the terms**

<table>
<thead>
<tr>
<th>Shorthand Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nars_tile(Ti)</td>
<td>number of active row segments in a column tile of size Ti</td>
</tr>
<tr>
<td>nacs_tile(Tj)</td>
<td>number of active column segments in a row tile of size Tj</td>
</tr>
<tr>
<td>nnz_per_row_seg(Ti)</td>
<td>average nnz per row segment in a column tile of size Ti</td>
</tr>
<tr>
<td>nnz_per_col_seg(Tj)</td>
<td>average nnz per column segment in a row tile of size Tj</td>
</tr>
<tr>
<td>interval[d]</td>
<td>a segment of a column in the sparse matrix with length d</td>
</tr>
<tr>
<td>vertical interval</td>
<td>a vertical column segment in the matrix with height Ti</td>
</tr>
<tr>
<td>non-active interval</td>
<td>an interval with no nnz in it</td>
</tr>
<tr>
<td>active interval</td>
<td>an interval with at least a nnz in it</td>
</tr>
<tr>
<td>aligned interval</td>
<td>an interval that starts at position k*Ti where k is an integer</td>
</tr>
<tr>
<td>unaligned interval</td>
<td>an interval that doesn’t start at position k*Ti where k is an integer</td>
</tr>
<tr>
<td>total segments</td>
<td>Total number of intervals</td>
</tr>
</tbody>
</table>

---

Fig. 1: Illustration of data access/reuse pattern for tiled SpMM: innermost tile-loop along I dimension
the loop nest. If 100 possible tile-sizes are considered along
each dimension, the total number of cases to consider is $10^6$.
We show below how this can be considerably reduced by virtue
of a special property that holds for SpMM and SDDMM.

A. Tiling with Different Tile Loop Orders

Alg. 5 shows high-level pseudocode for a tiled SpMM algorithm using a CSR data representation for the sparse matrix $\mathbf{S}$. There are three tiling loops ($ii$, $jj$, $kk$), and three intra-tile loops ($i$, $e$, $k$). While efficient access to the elements of the sparse matrix $\mathbf{S}$ in CSR representation does impose some constraints on loop ordering, the inherent data dependences of SpMM and SDDMM permit all possible permutations within the 3 tile-loops and among the 3 intra-
tile loops. Focusing on only the tile-loops, 6 permutations are
possible, of which a permutation with $\text{tile loops}$. Focusing on only the tile-loops, 6 permutations are
possible, of which a permutation with $ii$ innermost is shown
in Alg. 5. Fig. 1 illustrates the data access/reuse characteristics for tiled execution with this tile-loop order. The figure shows the data accessed by a single tile of size $T_k \times T_j \times T_i$ for tile-
loop order $\langle kk, jj, ii \rangle$. Seven non-zero elements are shown in
$\mathbf{S}$, within the index space along $\langle i,j \rangle$ covered by the tile. Each nonzero $S_{i,j}$ causes access to a slice of $T_k$ contiguous data elements in row $j$ of $\mathbf{I}$ to be read and a slice of $T_k$ contiguous
data elements in row $i$ of $\mathbf{O}$ to be modified. If two non-zero
elements in the tile footprint of $\mathbf{S}$ have the same column-
index $j$, they both will need access to the same elements in the
input matrix $\mathbf{I}$, thereby enabling intra-tile reuse for those elements in $\mathbf{I}$. Similarly, elements of $\mathbf{S}$ that have the same
row index $i$ will enable reuse of elements of $\mathbf{O}$ in cache. In
the example shown in Fig. 1, for processing all 7 non-zero
elements, only $3T_k$ elements of $\mathbf{I}$ and $4T_k$ elements of $\mathbf{O}$
will be accessed and not $7T_k$ elements of $\mathbf{I}$ and $\mathbf{O}$. Further, as
successive tiles are executed along $ii$, while distinct elements
of $\mathbf{O}$ are accessed in different tiles, any slice of $\mathbf{I}$ will only
be accessed once for all tiles along the inner-most tile loop $ii$
(assuming $T_j \times T_k \text{isnottoolargetofitincache}$).

We call the innermost tiling loop of a tiled loop nest as the
streaming loop and the corresponding dimension ($M$, $N$ or $K$)
as the streaming dimension. This term is used because, as
we explain later in this section, it is never beneficial to use a
tile size along the inner-most tiling loop to be larger than one.
A choice of tile-size of one means that we are traversing the
indices along that fastest varying index in a streamed fashion.
The streaming choices for SpMM (and their impact) can be
explained with the help of the tiled SpMM algorithm shown
in Alg. 5. A property of SpMM (and SDDMM) is that each
loop iterator is active in the indexing of two out of the three
matrices, but is completely absent in the indexing of the third
matrix: $i$ is used in indexing into $\mathbf{S}$ and $\mathbf{O}$ but not $\mathbf{I}$; $j$
is used in indexing into $\mathbf{S}$ and $\mathbf{I}$ but not $\mathbf{O}$; $k$ is used in indexing into
$\mathbf{I}$ and $\mathbf{O}$ but not $\mathbf{S}$. Thus, exactly the same set of elements
in a slice of the non-indexed matrix is accessed by successive
tiles as the inner-most tile iterator is varied, since that iterator
is absent in the indexing of data elements for that matrix. We
refer to that matrix as the stationary matrix. Any of the three
matrices ($\mathbf{S}$, $\mathbf{I}$, $\mathbf{O}$) can be chosen to be the stationary matrix,
based on the choice of streaming tile dimension:

- Streaming along $M(i)$: A tile of $\mathbf{I}$ of size $T_j \times T_k$ is
  kept stationary in fast memory;
- Streaming along $N(j)$: A tile of $\mathbf{O}$ of size $T_i \times T_k$ is
  kept stationary in fast memory;
- Streaming along $K(k)$: A tile of $\mathbf{S}$ of size $T_i \times T_j$ is kept
  stationary in fast memory.

Figure 2 depicts the streaming choices.
B. Pruning the search space

From the discussion about data access/reuse characteristics of SpMM for Fig. 1, we can see that all data reuses are along iteration-space axes, and each of the axes is a reuse direction for one of the three arrays: \( i \) is the reuse direction for the input matrix \( \mathcal{I} \), \( j \) is the reuse direction for the output matrix \( \mathcal{O} \), and \( k \) is the reuse direction for the input sparse matrix \( \mathcal{S} \). Now, let us consider the change in accessed data within tiles as the innermost tiling loop is traversed. Since that direction is a reuse direction for exactly one of the three arrays, the data footprint of that array will not change as we traverse that innermost tile, while that loop index indexes the other two arrays and therefore those two arrays will have a completely disjoint data footprint from the previous tile’s data footprint. So we can see that for any of the three possible permutations of the tiling loops, one array will have complete reuse, while the other two arrays will have no reuse as we traverse the innermost tiling loop. This means that as the entire innermost tiling loop is traversed, total reuse is achieved for the array which has the innermost loop as the reuse direction, while the other two arrays only achieve as much reuse corresponding to the number of active iteration space instances along the dimensions corresponding to the outer two tiling loops. The direct consequence of this observation is that the tile size does not affect the achieved reuse for one array (which has reuse along the innermost tiling dimension) and only affects reuse for the other two arrays, whose reuse directions correspond to the two outer tiling loops. Two important conclusions from the above analysis are:

- The tile size along the inner-most tiling dimension does not affect reuse of any of the arrays but affects the data footprint of multiple arrays. Hence it is best to choose that tile size as one (or a suitable fixed value, if necessary, to enable effective vectorized execution), to minimize its impact on the data footprint of arrays.
- The relative order of the outer two tiling loops only has a minor second-order effect on total data movement and therefore can be chosen in any order without much impact on data movement volume.

The above two observations enable a significant reduction on tiled configurations to be considered for performance optimization of SpMM. The same is true for SDDMM. If \( T \) possible tile sizes are to be considered along each dimension, instead of \( 3! \) permutations of tiled loops and \( T^3 \) configurations for different combinations of tile sizes (total of \( 6T^3 \) cases), the above two observations mean that we only need to consider \( 3 \) possible innermost tiling loops (streaming cases) and \( T^2 \) tile combinations (innermost tile size does not need exploration) for a total of \( 3T^2 \) cases. Further, the estimation of total data movement for an arbitrary sparse matrix can be efficiently estimated using sparse matrix signatures, whose efficient computation is described in the next section.

C. Data movement analysis

a) Streaming along \( M(i) \): In this scheme the \( \mathcal{I} \) matrix is kept stationary. Hence the total volume of data moved for \( \mathcal{I} \) is \( N \times K \). Each \( \mathcal{S} \) element is represented by a value and an index in CSR format. Each \( \mathcal{S} \) element is read in once for every \( T_k \) tile. Hence the data movement volume for \( \mathcal{S} \) is \((2 \times \text{nnz} \times K)/T_k\). A simple over-approximation for the volume of data movement due to \( \mathcal{O} \) is \((2 \times N)/T_j\) – for each tile of size \( T_j \) each \( \mathcal{O} \) element is read and written once. However, depending on the sparsity level and sparsity structure, there may be many empty row-segments in a tile of size \( T_j \), in which case the corresponding \( \mathcal{O} \) elements are not read/written. The total volume of \( \mathcal{O} \) elements, after accounting for empty rows of \( \mathcal{S} \), can be expressed as \((2 \times \text{nars}_{-\text{tile}}(T_j) \times K)\) where \( \text{nars}_{-\text{tile}}(T_j) \) represents the number of non-empty or active row-segments in all tiles combined, which is a function of \( T_j \). In other words, for every active row-segment, we have to read and write \( K \) elements of \( \mathcal{O} \). Thus the total volume is \((N + 2 \times \text{nnz}/T_k + 2 \times \text{nars}_{-\text{tile}}(T_j)) \times K\).

b) Streaming along \( N(j) \): This scheme is similar to streaming along \( M \). Here, we keep \( \mathcal{O} \) stationary, as depicted in Alg. 6. Hence, the total volume of data transferred for \( \mathcal{O} \) is \( M \times K \). Each \( \mathcal{S} \) element is bought into memory once for every \( T_k \) tile. Hence, the data movement volume for \( \mathcal{S} \) is \((2 \times \text{nnz} \times K)/T_k\). Similar to \( \mathcal{O} \) when streaming along \( M \), the total data transfer volume for \( \mathcal{I} \) can be expressed as \((\text{nacs}_{-\text{tile}}(T_i) \times K)\) where \( \text{nacs}_{-\text{tile}}(T_i) \) represents the number of active column-segments, which is a function of \( T_i \). Thus the total volume is \((M + 2 \times \text{nnz}/T_k + \text{nacs}_{-\text{tile}}(T_i)) \times K\).

c) Streaming along \( K(k) \): In this case each \( \mathcal{S} \) element is only read once and gets full reuse. Hence, the total volume of data transferred for \( \mathcal{S} \) is \( 2 \times \text{nnz} \). Similar to streaming along \( M \), the total volume of data moved transferred for \( \mathcal{O} \) is \((2 \times \text{nars}_{-\text{tile}}(T_i) \times K)\). Similar to when streaming along \( N \), the total data transfer volume for \( \mathcal{I} \) is \((\text{nacs}_{-\text{tile}}(T_i) \times K)\). Thus the total volume is \(2 \times \text{nnz} + (2 \times \text{nars}_{-\text{tile}}(T_i) + \text{nacs}_{-\text{tile}}(T_i)) \times K\).

d) Optimizing Tile Sizes: The analysis for case (a) and case (b) are similar. Here, we present the analysis for case (b). Let \( \rho \) be \( \text{de} \) density of the sparse matrix \( \mathcal{S} \), defined as the fraction of nonzero elements to the total number of elements in the matrix. In case (ii), a \( T_i \times T_k \) slice of \( \mathcal{O} \), \( T_i \times \rho \) slice of \( \mathcal{S} \) and \( 1 \times T_k \) slice of \( \mathcal{I} \) are kept in fast memory (cache or scratchpad). Thus the capacity constraint is:

\[
T_i \times T_k + 2 \times T_i \times \rho + T_k \leq C,
\]

where \( C \) is capacity of fast memory. The higher \( T_i \) is, the lower the data movement cost for \( \mathcal{I} \). Similarly, increasing \( T_k \) lowers the amount of data movement for \( \mathcal{S} \). Let \( \text{nnz}_{-\text{per} \_\text{col} \_\text{seg}}(T_i) \) be the average number of non-zero elements per active column of size \( T_i \) in \( \mathcal{S} \). Then, the total number of elements is:

\[
\text{nnz} = \text{nacs}_{-\text{tile}}(T_i) \times \text{nnz}_{-\text{per} \_\text{col} \_\text{seg}}(T_i)
\]
coarse-grained shared-memory parallelism is utilized. Stream method is shown in Alg. 6, which also shows how J-Stream along J (\( (b) \) is more memory efficient; therefore, we use streaming (2 and \( b \) is

By only making a single pass through the non-zeros of the given sparse matrix can be estimated. In the next section, we tested values for tile sizes, the best tile size to be used for a
data movement for SpMM.

signatures of a sparse matrix are sufficient to estimate the total

of a sparse matrix.

These 1D functions of tile sizes compactly represents the
data volume is a function of \( k \) (summation) dimension, and is omitted here.

subject to the constraint

\[
T_i \times T_k + 2 \times T_i \times \rho + T_k \leq C. \tag{5}
\]

Since 2, \( N \) and \( K \) are constants they can be removed from the minimization objective. Thus the minimization objective from eq. (4) can be re-written as:

\[
\min_{T_k, T_i} \left\{ 2 \times \frac{\text{nnz}}{T_k} + \text{nacs\_tile}(T_i) \right\} \tag{6}
\]

Equation (3) can be substituted in eq. (6) to obtain:

\[
\min_{T_k, T_i} \left\{ \text{nnz} \times \left( \frac{2}{T_k} + \frac{1}{\text{nnz\_per\_col\_seg}(T_i)} \right) \right\} \tag{7}
\]

Since \( \text{nnz} \) is constant, the minimization objective can be written as

\[
\min_{T_k, T_i} \left\{ \frac{2}{T_k} + \frac{1}{\text{nnz\_per\_col\_seg}(T_i)} \right\} \tag{8}
\]

A similar analysis can be done for case (a) to find optimization function:

\[
\min_{T_k, T_i} \left\{ \frac{1}{T_k} + \frac{1}{\text{nnz\_per\_row\_seg}(T_j)} \right\} \tag{9}
\]

Note that in eq. 9, the input dense matrix is read only once, while output dense matrix in eq. 8 moved twice (read and write). Therefore, numerator of the second term is halved in eq. 9. A similar analysis can be performed for streaming along the \( k \) (summation) dimension, and is omitted here.

The above analysis shows that the analytical estimate of data volume is a function of \( \text{nars\_tile}(T_j) \) or \( \text{nacs\_tile}(T_i) \). These 1D functions of tile sizes compactly represents the function signatures corresponding to the 2D sparsity structure of a sparse matrix.

As per the above analysis, these one-dimensional function signatures of a sparse matrix are sufficient to estimate the total data movement for SpMM.

By calculating the estimated data volume for a range of tested values for tile sizes, the best tile size to be used for a given sparse matrix can be estimated. In the next section, we present an efficient algorithm for computing a matrix signature by only making a single pass through the non-zeros of the matrix.

The data movement volume difference between methods a and b is \( (2 \times \text{nars\_tile}(T_j) - \text{nacs\_tile}(T_i)) \). According to our experiments, this amount is almost always positive, meaning (b) is more memory efficient; therefore, we use streaming along J (J-Stream) for our experiments. Pseudocode for J-Stream method is shown in Alg.6, which also shows how coarse-grained shared-memory parallelism is utilized.

From eq. (2):

\[
\text{nacs\_tile}(T_i) = \frac{\text{nnz}}{\text{nnz\_per\_col\_seg}(T_i)} \tag{3}
\]

Our objective is:

\[
\min_{T_k, T_i} \left\{ (2 \times N + 2 \times \frac{\text{nnz}}{T_k} + \text{nacs\_tile}(T_i)) \times K \right\} \tag{4}
\]

subject to the constraint

\[
T_i \times T_k + 2 \times T_i \times \rho + T_k \leq C. \tag{5}
\]

\( \text{V. EFFICIENT COMPUTATION OF SPARSE MATRIX SIGNATURE} \)

In this section, we present a novel algorithm for efficient estimation of the sparse matrix signatures. The naive generation of the signature for a matrix involves repeated scanning of the sparse matrix to count the number of active column/row segments as a function of row/column band size. Instead, the algorithm explained below generates the signature using a single pass over the sparse matrix to record a histogram of spacing between successive non-zero elements in a row (column).

\[\text{Algorithm 6: J-Stream SpMM}\]

\[
\begin{align*}
\text{input} & : \text{CSR } \mathcal{S}[M][N], \text{ dense } \mathcal{I}[N][K] \\
\text{output} & : \text{ dense } \mathcal{O}[M][K] \\
\text{for } \text{row\_seg} & = 1 \text{ to } \text{num\_row\_seg} \text{ do in parallel} \\
& \text{ for } \text{kk} = 0 \text{ to } N_k - 1 \text{ step } T_k \text{ do} \\
& \quad \text{ for } \text{jp} = \text{act\_cols[row\_seg]} \text{ to } \text{act\_cols[row\_seg+1]} \text{ do} \\
& \quad \quad j = \text{jp.column\_id} \\
& \quad \quad \text{ for } i \in \text{jp.row\_ids} \text{ do} \\
& \quad \quad \quad \text{ for } k = \text{kk} \text{ to } \min(\text{kk} + T_k, N_k) - 1 \text{ do} \\
& \quad \quad \quad \quad \text{ } \mathcal{O}[i][k] += \mathcal{S}[i][j]\mathcal{I}[j][k] \\
\end{align*}
\]

\[\text{Fig. 3: Examples of defined concepts.}\]

Assume we have a \( M \times N \) sparse matrix \( \mathcal{A} \). Tiled execution of SpMM or SDDMM corresponds to access of the sparse matrix data in partitioned row bands of height \( T_i \) (column bands of width \( T_j \)). If in a given row band, a column-segment is made-up of only zero elements, then the corresponding segment is said to be non-active. If it contains at least one non-zero element, then it is active. For a given value \( T_i \), a segment \( \mathcal{A}[i..(i + T_i - 1)][j] \) is said to be aligned if \( i \mod T_i = 0 \). For a given tiling of height \( T_i \), the matrix is partitioned into
The overall approach for computing the sparse matrix signature is to compute the total number of non-active segments. A pertinent property of vertical intervals is that a non-active segment is necessarily included in a vertical interval. More interestingly, we can precisely compute the non-active segments included in a vertical interval.

We next discuss how to compute those two distributions (notsmaller[\(T_i\)] and notsmallerweighted[\(T_i\)]) in linear time. Alg. 8 performs this task iteratively, by starting from \(T_i = M + 1\) and expressing notsmaller[\(T_i\)] and notsmallerweighted[\(T_i\)] as recurrence equations. Indeed, it may be observed that

\[
\text{notsmaller}[T_i] = \sum_{d \geq T_i} \text{intervals}[d] = \text{intervals}[T_i] + \sum_{d \geq T_i + 1} \text{intervals}[d] = \text{intervals}[T_i] + \text{notsmaller}[T_i + 1]
\]

and

\[
\text{notsmallerweighted}[T_i] = \sum_{d \geq T_i} d \times \text{intervals}[d] = T_i \times \text{intervals}[T_i] + \sum_{d \geq T_i + 1} d \times \text{intervals}[d] = T_i \times \text{intervals}[T_i] + \text{notsmallerweighted}[T_i + 1]
\]
### Algorithm 8: Linear time computation of \( \text{notsmaller}[T_i] \) and \( \text{notsmallerweighted}[T_i] \)

<table>
<thead>
<tr>
<th>( T_i )</th>
<th>( \text{notsmaller}[T_i] )</th>
<th>( \text{notsmallerweighted}[T_i] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( M+1 )</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( M+1 )</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( T_i )</td>
<td>intervals ( [T_i, T_i+1) )</td>
<td>intervals ( [T_i, T_i+1) )</td>
</tr>
<tr>
<td>( T_i \times \text{intervals}[T_i] )</td>
<td>( \text{notsmaller}[T_i] + \text{notsmallerweighted}[T_i+1] )</td>
<td>( \text{notsmallerweighted}[T_i] + \text{notsmallerweighted}[T_i+1] )</td>
</tr>
</tbody>
</table>

### Computing the signature:

To summarize, the computation of the signature \( p[T_i] \) begins by computing the number of vertical intervals of length \( d \), for each value of \( 0 \leq d \leq M \), using Alg. 7. The complexity of this pass is linear in the size of the matrix (number of non-zero elements). We then compute the cumulative distributions (with a complexity of \( O(M) \)) using Alg. 8. Those two distributions can then be used to compute the number of non-active segments of length \( T_i \) by directly applying eq. (14). The complexity is again \( O(M) \). Finally, eq. (11) can be used to derive the signature for each value of \( T_i \geq 1 \) (\( O(M) \) complexity). Hence, the overall complexity is linear, and dominated by the initial scan of the matrix performed by Alg. 7.

This enables efficient generation of approximations to the sparse matrix signatures as a function of column/row panel size. We find empirically that the fast approximate signatures track the exact signatures very closely. Figure 4 shows the exact and approximate signatures for one of the tested matrices.

![Active Column Estimation for hood](image1)

**Fig. 4:** Number of active panel segments estimation vs. actual for hood matrix.

### Tying it all together:

The performance of J-Stream is determined by two parameters: \( T_i \) and \( T_k \). Our objective is to select the best \( T_i \) and \( T_k \) values that minimize the total data volume (eq. (9)) under the cache capacity constraint (eq. (5)). In order to solve eq. (8), for any given \( T_i \), we need to compute the average number of non-zeros per column segment of size \( T_i \) (\( \text{nnz}_\text{per}_\text{col}_\text{seg}(T_j) \)). Naively computing \( \text{nnz}_\text{per}_\text{col}_\text{seg}(T_j) \) is very expensive as it requires \( O(\text{nnz}) \) operation per \( T_i \) value. Thanks to our matrix signature, we can compute the total number of active row segments \( (\text{nnz}_\text{tile}(T_i)) \) in \( O(1) \). \( \text{nnz}_\text{per}_\text{col}_\text{seg}(T_i) \) can then be computed as

\[
\frac{\text{nnz}}{\text{nnz}_\text{tile}(T_i)}.
\]

### VI. Experimental Evaluation

In this section, we compare the performance of the model-driven tiled J-Stream implementations of SpMM and SDDMM against state-of-the-art alternatives: (i) Intel’s MKL, (ii) TACO compiler [18], and (iii) Compressed Sparse Blocks (CSB) [3]. The experiments were performed on a dual-socket CPU platform, which has two \( \times \) Intel(R) Xeon(R) CPU E5-2680 v4 (Broadwell architecture 14 cores per socket, clocked at 2.40 GHz and 256KB L2 Cache) processors. We carried out the experimental evaluation using 22 sparse matrices. These datasets were selected based on previous papers that studied sparse matrix multiplication [23], [34]. All the datasets used in the experiments were downloaded from the publicly available SuiteSparse Matrix Collection [10]. The characteristics of the matrices are listed in Table III.

Figure 6 compares performance for two different feature \( (K) \) sizes, \( K = 128 \) and \( K = 1024 \). Each run was repeated 100 times, and the median value is reported. The performance of different approaches was normalized with respect to J-Stream. Each bar corresponds to the normalized GFLOPS achieved on a given dataset—the higher the bar, the better the performance. The last group of bars shows the geometric mean of each implementation on the 22 test matrices. The GFLOPS achieved by J-Stream is also shown in these charts. For example, as shown in Figure 6, J-Stream SpMM’s kernel’s performance ranges from 9 GFLOPS to 186 GFLOPS on the Broadwell platform for different matrices.

![Preprocessing Time Ratio to Single of a Kernel](image2)

**Fig. 5:** Ratio of pre-processing to execution time for J-Stream SpMM and SDDMM for \( K = 128 \) and \( K = 1024 \).

### A. SpMM Performance

For \( K = 128 \), J-Stream (using model-selected tile sizes) achieves 9%, 9%, and 50% speed-up over TACO, MKL, and CSB, respectively. MKL and TACO are faster than J-Stream
Fig. 6: Comparison of performance SpMM and SDMM kernels for $K = 128$ and $K = 1024$. The J-Stream performance bars correspond to model-selected tile-size; the spike in the middle of the J-Stream bar shows minimum and maximum performance over exhaustive search across tile sizes. Preprocessing times are not included for any of the methods.
TABLE II: Model-selected tile sizes for the test matrices

<table>
<thead>
<tr>
<th>Matrix Name</th>
<th>K=128</th>
<th>K=1024</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$T_i$</td>
<td>$T_k$</td>
</tr>
<tr>
<td>2cubes_sphere</td>
<td>512</td>
<td>64</td>
</tr>
<tr>
<td>cage12</td>
<td>1024</td>
<td>32</td>
</tr>
<tr>
<td>cant</td>
<td>256</td>
<td>128</td>
</tr>
<tr>
<td>conspH</td>
<td>256</td>
<td>128</td>
</tr>
<tr>
<td>cop20k_A</td>
<td>256</td>
<td>128</td>
</tr>
<tr>
<td>facebook_combined</td>
<td>145</td>
<td>128</td>
</tr>
<tr>
<td>filter3D</td>
<td>1024</td>
<td>32</td>
</tr>
<tr>
<td>hood</td>
<td>256</td>
<td>128</td>
</tr>
<tr>
<td>m133-b3</td>
<td>512</td>
<td>64</td>
</tr>
<tr>
<td>mac_econ_fwd500</td>
<td>1024</td>
<td>32</td>
</tr>
<tr>
<td>majorbasis</td>
<td>512</td>
<td>64</td>
</tr>
<tr>
<td>mc2depi</td>
<td>1024</td>
<td>32</td>
</tr>
<tr>
<td>offshore</td>
<td>256</td>
<td>128</td>
</tr>
<tr>
<td>patents_main</td>
<td>512</td>
<td>64</td>
</tr>
<tr>
<td>pdb1HYS</td>
<td>256</td>
<td>128</td>
</tr>
<tr>
<td>poisson3Da</td>
<td>483</td>
<td>64</td>
</tr>
<tr>
<td>ptk</td>
<td>256</td>
<td>128</td>
</tr>
<tr>
<td>rma10</td>
<td>256</td>
<td>128</td>
</tr>
<tr>
<td>scircuit</td>
<td>256</td>
<td>128</td>
</tr>
<tr>
<td>shipspec1</td>
<td>256</td>
<td>128</td>
</tr>
<tr>
<td>webbase-1M</td>
<td>256</td>
<td>128</td>
</tr>
<tr>
<td>web-BerkStan</td>
<td>256</td>
<td>128</td>
</tr>
</tbody>
</table>

TABLE III: Properties of the test matrices.

<table>
<thead>
<tr>
<th>Matrix Name</th>
<th>Rows</th>
<th>Columns</th>
<th>nnz</th>
</tr>
</thead>
<tbody>
<tr>
<td>2cubes_sphere</td>
<td>101,492</td>
<td>101,492</td>
<td>1,647,264</td>
</tr>
<tr>
<td>cage12</td>
<td>130,228</td>
<td>130,228</td>
<td>2,032,536</td>
</tr>
<tr>
<td>cant</td>
<td>64,541</td>
<td>64,451</td>
<td>4,007,383</td>
</tr>
<tr>
<td>conspH</td>
<td>83,134</td>
<td>83,134</td>
<td>6,010,480</td>
</tr>
<tr>
<td>cop20k_A</td>
<td>121,192</td>
<td>121,192</td>
<td>2,624,331</td>
</tr>
<tr>
<td>facebook_combined</td>
<td>4,039</td>
<td>4,039</td>
<td>88,234</td>
</tr>
<tr>
<td>filter3D</td>
<td>106,437</td>
<td>106,437</td>
<td>2,707,179</td>
</tr>
<tr>
<td>hood</td>
<td>220,542</td>
<td>220,542</td>
<td>10,768,436</td>
</tr>
<tr>
<td>m133-b3</td>
<td>200,200</td>
<td>200,200</td>
<td>800,800</td>
</tr>
<tr>
<td>mac_econ_fwd500</td>
<td>206,500</td>
<td>206,500</td>
<td>1,273,389</td>
</tr>
<tr>
<td>majorbasis</td>
<td>160,000</td>
<td>160,000</td>
<td>1,750,416</td>
</tr>
<tr>
<td>mc2depi</td>
<td>525,825</td>
<td>525,825</td>
<td>2,100,225</td>
</tr>
<tr>
<td>offshore</td>
<td>259,789</td>
<td>259,789</td>
<td>4,242,673</td>
</tr>
<tr>
<td>patents_main</td>
<td>240,547</td>
<td>240,547</td>
<td>560,943</td>
</tr>
<tr>
<td>pdb1HYS</td>
<td>36,417</td>
<td>36,417</td>
<td>4,344,765</td>
</tr>
<tr>
<td>poisson3Da</td>
<td>13,514</td>
<td>13,514</td>
<td>352,762</td>
</tr>
<tr>
<td>ptk</td>
<td>217,918</td>
<td>217,918</td>
<td>11,634,424</td>
</tr>
<tr>
<td>rma10</td>
<td>46,835</td>
<td>46,835</td>
<td>2,374,001</td>
</tr>
<tr>
<td>scircuit</td>
<td>170,900</td>
<td>170,900</td>
<td>958,936</td>
</tr>
<tr>
<td>shipspec1</td>
<td>140,874</td>
<td>140,874</td>
<td>7,813,404</td>
</tr>
<tr>
<td>webbase-1M</td>
<td>1,000,005</td>
<td>1,000,005</td>
<td>3,105,336</td>
</tr>
<tr>
<td>web-BerkStan</td>
<td>685,230</td>
<td>685,230</td>
<td>7,600,953</td>
</tr>
</tbody>
</table>

It may be observed that J-Stream SpMM performance drops as the feature size increases from $K = 128$ to $K = 1024$. Our initial analysis shows that this is due to prefetching effects. Consider the case of $K = 128$ and $T_k = 128$. As a row of data from the output dense matrix is updated, prefetching causes data from the next adjacent row of $O$ to be brought into cache, and it is likely to be used soon afterwards. However, in the case where $K = 1024$, the prefetched data corresponds to the tile along $K$, which is only executed after all the rows in the current tile are processed. Hence, the probability of reuse is much lower and thus the efficacy of the prefetcher is decreased.

B. SDDMM Performance

We compared the performance of the J-Stream SDDMM implementation with TACO’s SDDMM (Figure 6). On average, J-Stream achieved 70%, 52% speedup over TACO for $K = 128$ and $K = 1024$, respectively. J-Stream was faster than TACO for all cases in the SDDMM tests.

The geometric mean of TACO performance increased as $K$ was increased from 128 to 1024, whereas it decreased for J-Stream. This is likely due to the TACO code not being adversely affected by the prefetcher at large $K$ due to its data access pattern, whereas J-Stream is negatively affected by it.

C. Model Effectiveness

To evaluate the effectiveness of our model (Section V), we compared the performance achieved using the model selected tile-sizes and the performance obtained using exhaustive search across tile sizes. The spikes in the middle of the in J-Stream performance bars in Figure 6 show the minimum and maximum performance achieved over an exhaustive search across tile-sizes. In general, our model performs quite well. For example, for SpMM with $K = 128$, on average the gap between performance using our model and the maximum achievable performance with optimal empirically found tile sizes is around 10%, and it is less than 20% in all instances, except for one case (cage12), where it is 41%.

For the SDDMM kernel, on average, the performance gap between use of our model and empirically determined optimal tile size via exhaustive search is around 14%. The performance gap increases to 30% in some cases, where the number of rows is very small for the input. When the number of rows is small, the entire output matrix can fit inside the cache, which eliminates the benefits of tiling.

The performance gap between use of our model and use of the optimal tile sizes found by exhaustive auto-tuning is generally not very large. However, the data from the exhaustive searches also shows that there may still be room for improvement in the model and also the development of model-driven auto-tuning, where the model is augmented with a limited amount of execution on the target platform for a selected set of tile sizes guided by the model.

D. Preprocessing Overhead

Fig. 5 shows the preprocessing time for creation of the matrix signatures and selection of tile sizes, relative to SpMM
and SDDMM kernel execution time. As shown in fig. 5, normalized preprocessing time is quite negligible for \( K=1024 \). On average, the modeling time was 76% and 9% of the time of a single SpMM or SDDMM kernel execution for \( K=128 \) and 1024, respectively. The preprocessing Algorithms 7 and 8 has complexity \( O(nnz) \) and \( O(N) \), respectively, and total complexity of sequential preprocessing is \( O(nnz) \), which we plan to parallelize as part of future work.

E. Scalability

In order to compare scalability, we ran each SpMM implementation by varying the number of threads from 2 to 28 (the number of physical cores) and compared against the performance of the corresponding single-core run. Fig 7 and 8 show speedup as a function of the number of threads. For all cases except SpMM with \( K=128 \), J-Stream achieves the best scaling result. For the SpMM with \( K=128 \) case, MKL scales better.

VII. Conclusion

In this paper, we have developed an analytical approach to modeling data movement and tile size optimization for SpMM and SDDMM. The analysis is made possible by generation of compact one-dimensional function signatures that capture the impact of the 2D non-zero distribution pattern of a matrix on data movement for the applicable class of computations. Implementations of parallel tiled SpMM and SDDMM kernels using the model-driven tiling approach demonstrated effectiveness of the developed methodology.

VIII. Acknowledgments

We thank the reviewers for their valuable feedback. This work was supported in part by the U.S. National Science Foundation through awards 1946752 and 1919122.

References


