SPALM: A Sparsity-Pattern-Adaptive Library for Matrices

JUNYOUNG KIM, Columbia University, USA
KENNETH A. ROSS, Columbia University, USA

Matrix multiplication is commonly used in many data-intensive applications, such as in analytics, machine learning, and scientific computing. While the data inputs and data outputs of matrix multiplication often originate from and are stored in relational databases, the matrix multiplication process itself is typically performed outside of the database in order to use existing libraries and tools that are not part of the database. This introduces the overhead of data export and import in the data processing pipeline, while raising data privacy concerns.

We propose SPALM (Sparsity-Pattern-Adaptive Library for Matrices), a library that performs sparsity-sensitive matrix multiplication. SPALM utilizes a novel matrix multiplication algorithm that applies different matrix multiplication techniques depending on the sparsity of fine-grained regions of the matrices, and does not require the user to have any prior knowledge of the sparsity patterns to get good performance. We show through experiments that SPALM achieves up to 34× speedup on tasks involving both real-world and synthetic matrices with varying sparsity patterns compared to popular matrix multiplication libraries on CPUs. We also integrate SPALM as an operator in DuckDB, which allows users to declaratively specify tasks involving matrix multiplication in the context of a database.

CCS Concepts: • **Information systems** → **Query operators**.

Additional Key Words and Phrases: Matrix multiplication, Query processing, Sparsity

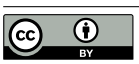
ACM Reference Format:
Junyoung Kim and Kenneth A. Ross. 2026. SPALM: A Sparsity-Pattern-Adaptive Library for Matrices. *Proc. ACM Manag. Data* 4, 3 (SIGMOD), Article 236 (June 2026), 27 pages. https://doi.org/10.1145/3802113

1 Introduction

Matrix multiplication is used in many domains, such as subgraph matching [56], shortest-path finding [8], cycle detection [7], linear solvers [3] and Machine Learning (ML) [6, 43, 61]. Many of the data inputs and outputs of applications involving matrix multiplication frequently come from and are stored in relational databases [37, 52], especially with the widespread adoption and increasing scale of cloud data storage and database services such as Amazon Redshift, Azure SQL, and Google BigQuery. Typically, however, the matrix multiplication process itself is done outside of the database in order to utilize highly optimized libraries and tools that have been developed independently of relational databases [37, 50]. Using these external libraries and tools introduces performance and usability drawbacks as importing and exporting data into a separate execution framework is expensive and adds significant operational complexity, as well as raising privacy concerns [1]. Executing matrix multiplication outside of the database also puts the burden of finding a performant matrix multiplication algorithm for the input data on the user, which can be a nontrivial task for sparse matrices.

Popular applications involving sparse matrices include graph processing [7, 8, 25, 56], and natural joins and group-by aggregates on relational data [14, 23, 24]. In ML, users may want to introduce

Authors' Contact Information: Junyoung Kim, Columbia University, New York, USA, junyoung2@cs.columbia.edu; Kenneth A. Ross, Columbia University, New York, USA, kar@cs.columbia.edu.



ACM 2836-6573/2026/6-ART236
https://doi.org/10.1145/3802113

sparsity to reduce storage costs, and improve energy and inference/training times [2]. Sparsity can also be introduced to reduce overfitting when training a model [48, 58]. With the popularity of the transformer deep learning architecture, which is used in many common applications such as Large Language Models, much research effort has been given to introducing sparsity to the attention mechanism [10, 11, 42, 62], which is the transformer's core operation and can be done via matrix multiplication.

In order to get good performance for matrix multiplication involving sparse matrices, the user must choose the right matrix multiplication algorithm depending on the sparsity of the data. Different algorithms are used depending on whether one or both of the matrices are sparse, or if both matrices are dense. However, choosing the right algorithm is not a trivial task, as users need to know about the sparsity of their data in advance, which becomes an even harder task to do if the user does some processing on the data before the actual matrix multiplication. For example, a graph dataset user may prune out vertices they are not interested in. This is also nontrivial to do in a data pipeline involving several matrix multiplications, where the user may not be able to predict the sparsity of intermediate result matrices. While it is possible to simply count the number of non-zero elements in a matrix and calculate sparsity automatically, not only does the sparsity threshold at which matrix multiplication algorithm becomes optimal differ from machine to machine, the user may leave performance on the table as sparsity is not considered at a fine-grained level. Different sparsity patterns affect matrix multiplication algorithms differently, and a quick look at the SuiteSparse Matrix Collection [13, 29] shows a variety of different sparsity patterns in real-world data. Thus, it is not an easy task for the user to find or implement the optimal matrix multiplication algorithm for their application, especially if they are working with several different machines and datasets.

Previous work [14, 24] aims to address these issues by implementing sparsity-sensitive matrix multiplication algorithms to accelerate queries on relational data that can be mapped to matrix multiplication. They use coarse sparsity analysis methods that rely on counting the number of non-zero elements in an entire row or column, which cannot pick up on more detailed sparsity patterns in the data. They also either do not use low-level loop optimization techniques present in fast matrix multiplication algorithms for CPUs, or call optimized matrix multiplication libraries as a black box, which limits their applications in scenarios where the input data is neither very dense nor very sparse.

We introduce SPALM (Sparsity-Pattern-Adaptive Library for Matrices), a library that enables the user to achieve fast matrix multiplication without requiring the user to know about the sparsity characteristics of their data or their machine. SPALM groups subrows of the matrix into partitions, called 'slices' in the paper, and then uses different algorithms to multiply the slices based on the slice's density. While the idea of partitioning a matrix and applying a different matrix multiplication algorithm depending on partition density thresholds is not new, this is only a part of what SPALM does. An overview of SPALM's technical novelties is as follows:

Novel inter-partition processing algorithm: SPALM uses a novel inter-partition processing algorithm which is more sophisticated than previous methods of simply iterating through pairs of partitions and applying a specific matrix multiplication algorithm depending on the density of the partitions. For partitions requiring dense matrix multiplication, SPALM uses a tiled partition iteration algorithm that goes hand in hand with its small partition sizes to get good memory locality. For partitions requiring sparse matrix multiplication, SPALM creates a temporary data structure whose elements are ordered by the density of the partitions they originate from, which allows for fast performance by amortizing the cost of processing slice boundaries while also getting good locality.

Flexible choice of matrix multiplication algorithm depending on the combination of slice densities: In previous methods, partitions of the matrix are labeled as either 'sparse' or 'dense' which results in rigid algorithmic boundaries where there is no overlap between partitions used in sparse-sparse matrix multiplication and dense-dense matrix multiplication. In contrast, SPALM determines the matrix multiplication algorithm using the densities from *both* partitions, which can result in overlap between partitions participating in sparse and dense matrix multiplication. This design allows for more granular algorithmic boundaries that can be automatically fine-tuned for each machine. This overlap is facilitated by SPALM's sparse matrix multiplication algorithm that orders matrix elements by their corresponding partition's densities and uses subrange processing to perform multiplications.

By integrating SPALM into a relational database, users are able to declaratively write and efficiently execute queries involving matrix multiplication. By working in the context of a database, users will also have the benefit of not exporting their data outside of their database if they want to perform fast matrix multiplication, and will also have better data privacy. Query optimization also potentially benefits from SPALM. A self-optimizing matrix multiplication operator in which local optimizations are based on the actual input data to the operator reduces the burden on the query optimizer. An alternative in which multiple matrix algorithms (dense, sparse, etc.) are available forces the optimizer to choose the preferred algorithm, likely based on very approximate predicted statistics.

Our main contributions are as follows:

- We introduce a novel sparsity-aware matrix multiplication algorithm that can handle a wide range of sparsity patterns, from very sparse matrices to fully dense matrices and implement it as a library called SPALM. SPALM's algorithm is based on grouping subrows of the matrix into matrix slices, and then using different algorithms to multiply the slices based on the slices' density.
- We show experimentally that SPALM achieves performance that is better or competitive with popular, hand-coded matrix multiplication libraries on a variety of different combinations of matrix sparsity patterns without requiring the user to identify the optimal matrix multiplication algorithm. We show SPALM's effectiveness on matrices from real-world domains ranging from ML to biology, and on synthetic data with sparsity patterns commonly found in real-world applications.
- We integrate SPALM as an operator in DuckDB [40][1]. A DuckDB user can declaratively specify tasks that use matrix multiplication and get fast performance without having to know about the characteristics of their data or their machine.

2 Related Work

ATMULT [27], DHK [14] and DIM3 [24] are most related to SPALM in that their goal is to perform matrix multiplication or a task similar to matrix multiplication for a wide variety of sparsity ranges by partitioning the input matrices and applying different algorithms based on the densities of the partitions. All three methods use partition sizes much larger than SPALM and are thus less sparsity-sensitive. DHK labels entire rows and columns as sparse or dense depending on their density, and defines dense partitions as the intersection of dense rows and dense columns. DIM3 horizontally partitions one of the input matrices based on the density of only its rows, and does not do any sparsity-based partitioning on the other matrix. As such, both DHK and DIM3 are unable to accurately capture local sparsity patterns that do not span entire rows or columns.

[1]Source code is available at https://github.com/junyoungkim22/spalm-release/tree/main
and https://github.com/junyoungkim22/duckdb-spalm-release/tree/main

ATMULT partitions the matrix into submatrices whose minimum size is determined by the system's last-level cache size, which on their test machine yields a minimum partition size of 1024×1024, which is 2048 times bigger than the smallest unit of matrix area considered in SPALM, which is a subrow of size 1×512. Each partition in ATMULT is labeled as sparse or dense, and multiplication between partitions is deferred to existing library functions when possible. Because each matrix multiplication via a library call is independent of each other, in order to get good performance each partition-partition multiplication must be big enough for the library to utilize the underlying system's memory hierarchy, which forces ATMULT to use large partition sizes, or risk performance degradation due to hardware underutilization and frequent boundary processing. All three systems also have rigid algorithm boundaries based on a binary labeling of partitions, which differs from SPALM's method of choosing matrix multiplication algorithms based on the participating partition's actual densities.

There have been various works on optimizing sparse-sparse or sparse-dense matrix multiplication. [47] divides a sparse matrix into tiles, and uses a different sparse algorithm depending on the density. Unlike SPALM, it does not consider sparse-dense or dense-dense matrix multiplication. [63] uses bitmaps to represent non-zero elements in sparse matrices and uses them in multiplication, similar to the bitsets used in very dense slices in SPALM. It differs from our work in that it mainly targets sparse matrices and is suboptimal for dense matrices. [59] shows how register reuse can be used for sparse-dense matrix multiplication. It differs from our work in that it does not consider sparsity patterns on the dense matrix.

There is various work on mixing sparse and dense matrix multiplication on different hardware. [25] divides and reorders a matrix into sparse and dense regions, and implements hardware to optimize memory movement. [36] stores sparse matrices in tiles and uses a sparsified version of dense matrix multiplication to multiply tiles together, which is similar to SPALM. However, its data structure targets GPUs instead, and its dense matrix multiplication algorithm, if translated to a CPU setting, is not optimized for locality for each level of the cache.

Recently, there has been much work on performing ML in a database. [49] does LLM inference on CPUs using DuckDB as the runtime, and represents matrices as chunks similar to how SPALM uses slices to represent matrices. However, the way the chunks are stored in the index space is different, and sparsity of the matrices is not considered. [46] shows how users can define and operate on their own defined data structure, so that a database can operate on a wide variety of storage formats, such as CSR. [5] shows how Einstein notation, which is used to represent tensor operations such as matrix multiplication, can be mapped to SQL. [9] shows how linear algebra can be applied to normalized data. [39] optimizes ML pipelines in a database by combining database operators and ML operators into a single graph and introduces optimizations that span both types of operators. [45] learns a linear regression model over factorized joins in a database, and involves efficiently computing a similarity matrix between elements in normalized tables.

There has been increasing interest in performing ML on CPUs due to demand for AI on edge devices [19] or privacy or cost issues. Google has released Gemma [51], a family of lightweight LLMs for CPUs, and Meta maintains the FBGEMM project [28], a high-performance library for inference on machines including CPUs. Amazon AWS is developing Graviton [31], a CPU optimized for ML workloads. With the increasing popularity of LLMs, CPUs are attracting more attention as hardware accelerators face challenges due to limited memory capacity [34]. We expect SPALM to go hand in hand with this trend and accelerate ML tasks on CPUs.

There has been previous work on optimizing other join + group by patterns that do not map to matrix multiplication. Techniques including pushing down aggregates [16, 44, 60] and computing aggregates in hash tables used for joins [33] and parallelizing such techniques [17].

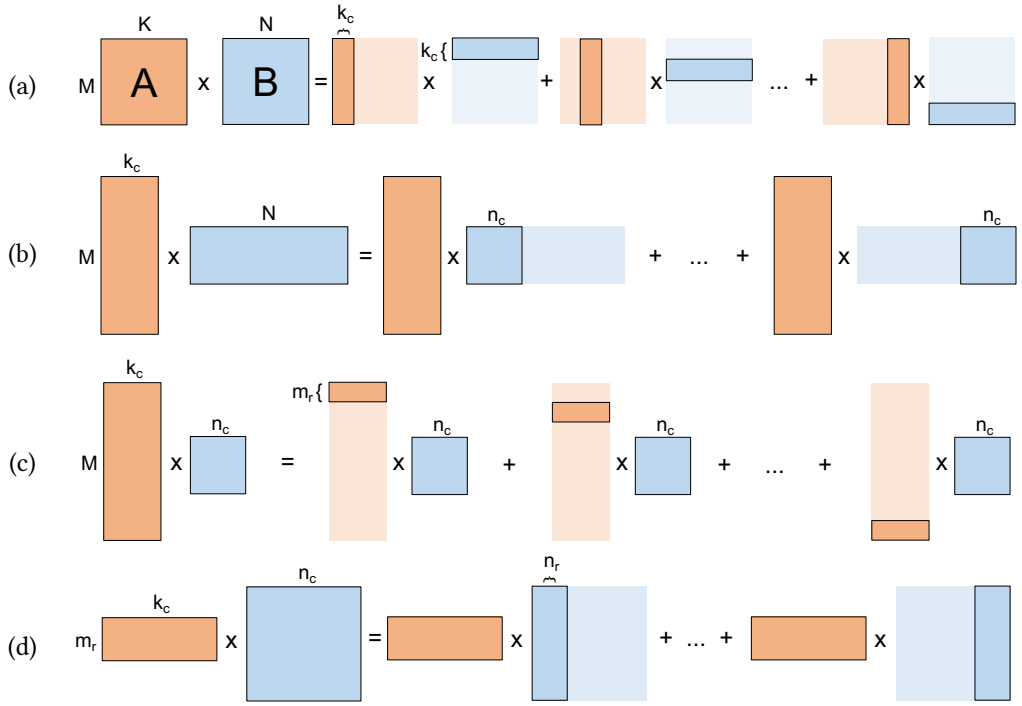


Fig. 1. Partitioning matrix multiplication into subtasks

3 Preliminaries

We outline existing matrix multiplication algorithms for CPUs.

3.1 Dense-Dense Matrix Multiplication (GEMM)

Fast multiplication of two dense matrices, referred to as GEMM (General Matrix Multiplication) in the literature, is efficiently done on CPUs by doing several layers of loop tiling to amortize the cost of data movement over a large number of computations, for every layer of the memory hierarchy. This approach is explained in the algorithm by Goto et al. [20], and is widely used in popular linear algebra libraries such as OpenBLAS [38].

The tiling process is shown in Figure 1. A matrix multiplication between two matrices A and B of dimensions $M \times K$ and $K \times N$, respectively, can first be divided into the sum of matrix multiplications of panels of dimensions $M \times k_c$ and $k_c \times N$. (Figure 1a) Each panel-panel matrix multiplication is then divided into the sum of panel-block multiplications, where each block has dimensions $k_c \times n_c$ (Figure 1b). Each panel-block matrix multiplication is then divided into the sum of slice-block multiplications, where each slice has dimensions $m_r \times k_c$ (Figure 1c). The purpose of this division is to keep the $k_c \times n_c$ size block in the L2 cache while multiplying it with the slices in the $M \times k_c$ panel, so that the cost of moving the block into the L2 cache is amortized over the computation with the slices in the panel. Each slice-block matrix multiplication is then further divided into the sum of slice-slice multiplications, where the slices from B have dimensions $k_c \times n_r$ (Figure 1d). Following the same principle as the previous step, the purpose of this subdivision is to amortize the cost of moving a slice from A (of dimensions $m_r \times k_c$) into the L1 cache over the computation with the slices from a block in B. To further optimize matrix multiplication, a slice-slice matrix

```
inner_kernel_2x8(
    double A[M][K], double B[K][N], double C[M][N],
    int k_c, int i, int k, int j) {
    // Initialize registers
    simd_reg a_reg, b_reg, c0, c1;
    simd_setzero(c0); simd_setzero(c1);
    // calculate and store 2x8 subresults in c0 and c1
    for(int kk = 0; kk < k_c; kk++) {
        b_reg = simd_load(&B[k+kk][j]);
        a_reg = simd_broadcast(&A[i][k+kk]);
        c0 = simd_fmadd(a_reg, b_reg, c0);
        a_reg = simd_broadcast(&A[i+1][k+kk]);
        c1 = simd_fmadd(a_reg, b_reg, c1);
    }
    // add subresults to C
    c0 = simd_add(&C[i][j],c0); simd_store(&C[i][j],c0);
    c1 = simd_add(&C[i+1][j],c1); simd_store(&C[i+1][j],c1);
}
```

Fig. 2. 2×8 Inner kernel function

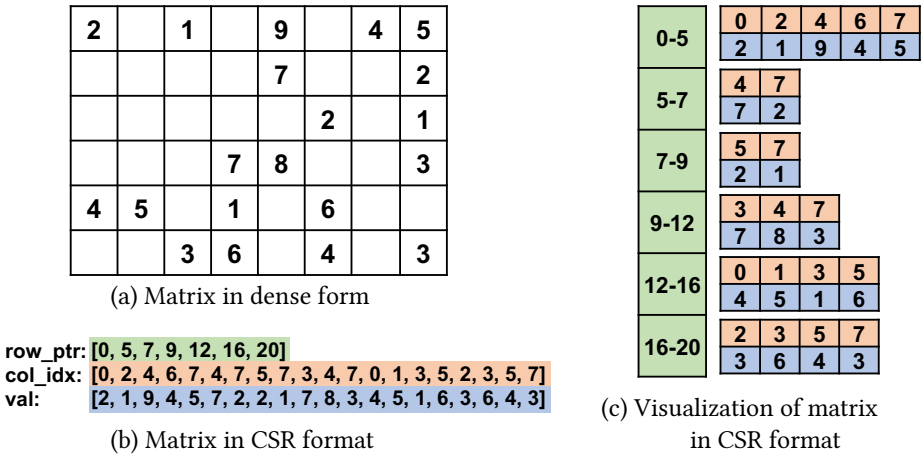
multiplication is done so that the final multiplication result of size $m_r \times n_r$ fits entirely in the SIMD registers of the CPU in order to minimize data movement between memory while accumulating multiplication results in the $m_r \times n_r$ submatrix. The high performance function for multiplying two slices to produce $m_r \times n_r$ subresults is referred to as the **inner kernel**, and is crucial for fast multiplication. Pseudocode for an inner kernel where $m_r = 2$ and $n_r = 8$ is shown in Figure 2. In practice, the values of m_r and n_r are larger (e.g. $m_r = 12$, $n_r = 16$ for double data types).

Fast matrix multiplication libraries also do **packing**. Before the matrix multiplication, matrix elements are copied into a contiguous format in a temporary buffer in the same sequence they would be accessed during computation. This optimization reduces TLB misses and cache misses, while enabling better code vectorization.

The optimal values of the parameters k_c, n_c, m_r and n_r depend on the characteristics of the machine as well as the data type of the matrices. Generally, m_r and n_r are chosen to be as large as possible while being within the SIMD register budget of the machine, and k_c and n_c are chosen to be as large as possible while a panel of dimensions $m_r \times k_c$ fits in the L1 cache, while a block of dimensions $k_c \times n_r$ fits in the L2 cache. In practice, libraries save and use tuned parameters for each machine.

3.2 Sparse-Sparse Multiplication (SpGEMM)

We explain how fast matrix multiplication is done on two matrices stored in **Compressed sparse row (CSR)** format, the most frequently used format to store sparse matrices [18], and the sparse representation we also use in this work. The CSR format uses three one-dimensional arrays to represent the non-zero elements of the matrix: *val*[], *col_idx*[] and *row_ptr*[]. *val*[] stores the non-zero values of the matrix in row-major order, and *col_idx*[] stores the column index of each non-zero value in the *val*[] array. *row_ptr*[] stores the starting index of each row's data in the *val*[] and *col_idx*[] arrays. In other words, the values of the non-zero elements in row i of the



(a) Matrix in dense form

(b) Matrix in CSR format

(c) Visualization of matrix in CSR format

Fig. 3. CSR matrix example

matrix are stored in the subarray $val[row\_ptr[i]{:}row\_ptr[i+1]]$, and the column indices of those non-zero elements correspond to the integers stored in subarray $col\_idx[row\_ptr[i]{:}row\_ptr[i+1]]$. An example of a matrix in CSR format is shown in Figure 3. It is common for the non-zero elements for each row in a matrix in CSR format to be sorted by their column index to allow for good performance on common operations, and fast lookups via binary search within a row. This format, called 'canonical format', is supported in popular libraries such as SciPy [57].

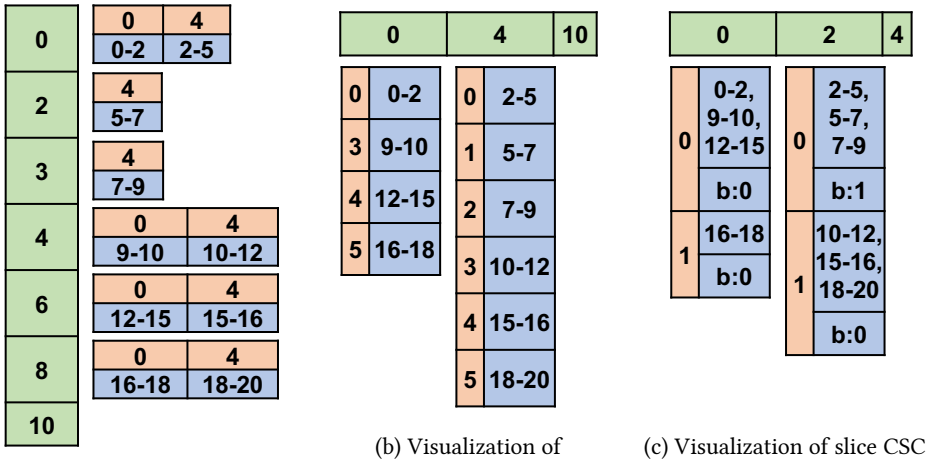
In the case where the common dimension between the two CSR matrices to be multiplied is the columns for the first matrix and is the rows for the second matrix (e.g. Multiplying a $M \times K$ matrix with a $K \times N$ matrix), fast matrix multiplication is done by iterating through each element in the first matrix, and using the element's column index to index into the corresponding row in the second matrix, and multiplying the element in the first matrix with all elements in the corresponding row in the second matrix.

In the case where the common dimension between the two CSR matrices is the columns for both the first matrix and second matrix (e.g. Multiplying a $M \times K$ matrix with a $N \times K$ matrix), the matrix with fewer non-zero elements is transposed into CSC (Compressed Sparse Column) format, which is the same as the CSR format, but with the dimensions for the rows and columns swapped, and then the two matrices are multiplied as previously described. We omit the process of transposing a CSR matrix for brevity, and it is possible to find open-source implementations of the transpose process such as in SciPy [57].

4 Matrix Multiplication in SPALM

We explain how SPALM performs fast sparsity-sensitive matrix multiplication. The principles of SPALM's algorithm are based on the following observations from the GEMM algorithm explained in Section 3.1.

- *Rows that have only zero elements in a panel/block can be skipped:* One can observe that the tiled GEMM algorithm does not need to consider 'empty' subrows (i.e. subrows that have no non-zero elements) of length k_c during the matrix multiplication process, and what matters is to process non-empty subrows in groups at a time (m_r non-empty subrows at a time for the left matrix, and n_r non-empty subrows at a time for the right matrix). This effectively reduces the number of



(a) Visualization of matrix in CSSR format where $k_c = 4$

(b) Visualization of transposed matrix in CSSR format

(c) Visualization of slice CSC matrix where $m_r = 3$ and $B = 2$

Fig. 4. Matrix from Figure 3 in CSSR and slice CSC formats

computations in the dense matrix multiplication to be proportional to the number of non-empty subrows in the matrix, as opposed to the total number of subrows.

- *Different inner kernel implementations can be used:* One can also observe that different inner kernel implementations can be used to multiply two matrix slices (i.e. submatrices of dimensions $m_r \times k_c$ and $k_c \times n_r$) as how the matrix multiplication is done does not affect the locality benefits on the L1 and L2 cache from the previous layers of loop tiling.

Based on these two observations, we design SPALM's matrix multiplication algorithm, which at a high level, does the following.

(1) Non-empty subrows of each matrix are grouped into slices.
(2) Relatively dense slices are selectively multiplied with a dense, tiled matrix multiplication algorithm. The choice of inner kernel algorithm depends on the density of the slices.
(3) Relatively sparse slices are selectively multiplied with a sparse matrix multiplication algorithm.

Each of these steps will be explained in detail in the upcoming sections. SPALM receives as input two matrices in CSR format in canonical form, which is the most common way to represent sparse matrices, and outputs its results in either a dense matrix or a sparse matrix depending on the circumstances.

4.1 Grouping Non-empty Subrows into Slices

Given an input matrix in CSR format whose column indices are sorted for each row, SPALM first identifies non-empty subrows of size k_c, where k_c is a tiling parameter chosen in advance that corresponds to kc in Figure 1. Identification of subrows is done by iterating through the column indices of each row and keeping track of the start and end column indices of each subrow. This information is stored in a separate CSR matrix, which we will refer to as the CSSR (Compressed Sparse SubRow) matrix, where each subrow is treated as an independent element. The column indices of this new CSR matrix correspond to the offset of the subrow, and values are a 2-tuple that correspond to the start and end indices of the subrow. An example of a CSSR matrix that is created

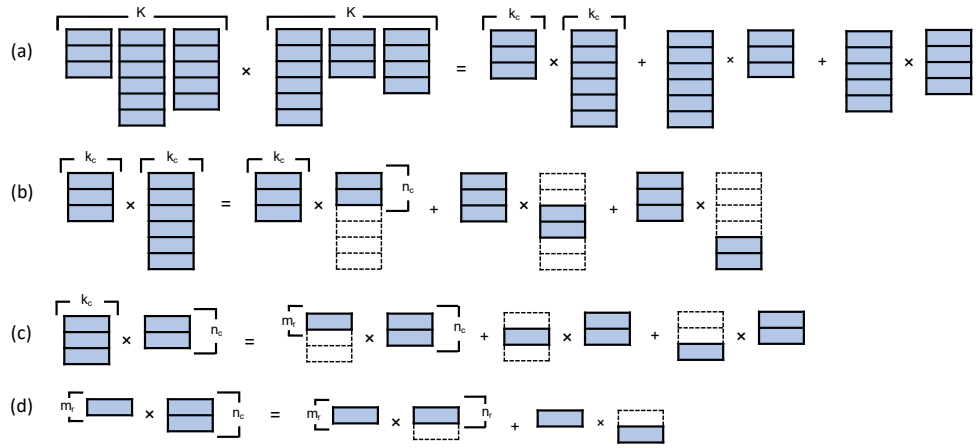


Fig. 5. Multiplying two slice CSC matrices in a tiled fashion

where $k_c = 4$ from the example matrix in Figure 3 is shown in Figure 4a. In practice, the value of k_c is larger, such as 512 or 1024. Creation of the CSSR matrix is lightweight as the creation process requires only two scans through the elements in the CSR matrix (one for counting the number of subrows, and one for initializing the elements of the CSSR matrix), and only needs to be done once and can be reused for all subsequent matrix multiplications involving the matrix.

Once the CSSR matrix is created, to group subrows with the same column offsets into slices, we then transpose the CSSR matrix (an example transpose is shown in Figure 4b), and then group adjacent subrows within the same column and store them in a separate CSC matrix, similar to how we grouped adjacent elements into subrows when creating the CSSR matrix. For the left matrix we group m_r subrows at a time, and for the right matrix we group n_r subrows at a time, where m_r and n_r are chosen according to the SIMD register budget of the system ahead of time. For each slice, we assign a bucket to it depending on its density (i.e. number of non-zero elements in the slice divided by the capacity of the slice (e.g. $m_r \times k_c$ for the left matrix)). Using B buckets, where the number of non-zero elements in the slice is nnz, and the capacity of the slice is C, the assigned bucket number is $\lfloor ((nnz - 1)/C) * B \rfloor$. We will refer to this matrix as the slice CSC matrix. An example slice CSC matrix is shown in Figure 4c where $m_r = 3$ and $B = 2$. In practice, SPALM uses a value of $B = 100$ to be more sensitive to slice densities.

Given two slice CSC matrices created from the two input matrices in CSR format, in the following sections we describe how matrix multiplication is done in SPALM. In order to get good performance, SPALM's goal is to apply the optimal matrix multiplication between pairs of slices to be multiplied depending on their density, while keeping the benefits of locality of the fast matrix multiplication algorithms explained in Section 3. Generally speaking, we wish to apply a sparse matrix multiplication between slices that are relatively sparse, and a dense matrix multiplication between slices that are relatively dense. To do this, matrix multiplication is done in two steps, sparsity-sensitive dense matrix multiplication, and sparsity-sensitive sparse matrix multiplication. In the following sections we explain each step in more detail.

4.2 Sparsity-Sensitive Dense Multiplication

SPALM performs sparsity-sensitive dense matrix multiplication by iterating through the slice CSC matrix for both matrices and multiplying pairs of slices together. This process is shown in Figure 5,

and is similar to the dense matrix multiplication shown in Figure 1. A matrix multiplication between the slices from two matrices can be divided into matrix multiplications between pairs of columns of slices that share the same indices along the common dimension (Figure 5a). For each pair of columns, prior to the matrix multiplication, we convert each slice in the column into a dense, column-order matrix, so that we can use SIMD instructions to quickly load data from columns in each slice. For each slice, we store a bitset of size k_c that indicates which columns have at least one non-zero element in the slice. Converting slices into a temporary dense format corresponds to the packing process explained in Section 3.1. The memory buffer used for this packing is allocated ahead of time, and is reused across matrix multiplications between pairs of columns of slices. A matrix multiplication between columns of slices can be further divided into matrix multiplications between a column of slices and a block of slices of size $k_c \times n_c$ (Figure 5b), where n_c is a tiling parameter set before matrix multiplication. A matrix multiplication between a column of slices and a block of slices can be further divided into matrix multiplications between a slice and a block of slices (Figure 5c), and finally a matrix multiplication between a slice and a block of slices can be divided into matrix multiplications between slices (Figure 5d). For the same reasons explained in Section 3.1, when the height of a slice from the left matrix is m_r, the parameters k_c and n_c are chosen so that a slice from the left matrix, whose size is $m_r \times k_c$, fits in the L1 cache while being as big as possible, and a block of slices from the right matrix, whose size is $n_c \times k_c$, fits in the L2 cache while being as big as possible.

For a matrix multiplication on two slices, depending on the assigned bucket of the slices, SPALM uses a different algorithm to multiply the slices together. We implement a function *GET_ALG()* that takes in the two buckets of the slices to multiply and outputs the algorithm to use. Details of *GET_ALG()* will be explained in Section 4.5. If *GET_ALG()* returns *DENSE_DENSE*, which is when both of the slices are relatively dense, we use a work-skipping version of the inner kernel function used in dense matrix multiplication to multiply the two slices. The pseudocode for this work-skipping kernel where $m_r = 2$ and $n_r = 8$ is shown in Figure 6. Because the bitset stored with each slice indicates which of the k_c columns of numbers have non-zero elements, we can use the AND of the two bitsets from each matrix to determine which columns of numbers to actually process in the inner loop. This is done by doing a SIMD compress instruction, which returns the indices of the bits that are set to one (e.g. 101101 returns [0, 2, 3, 5]). We then use these indices to only process the common columns in the very dense slices that have non-zero elements. By implementing this work-skipping inner kernel, SPALM is able to be sparsity-sensitive on the common dimension between two matrices within two slices, which allows it to achieve speedups on sparsity patterns where many columns of numbers in a slice do not contain non-zero elements. This is a pattern that appears in many workloads, such as when a section of a matrix shifts from a sparse region to a dense region, or around the diagonal in a matrix whose elements are clustered around the diagonal. Both of the aforementioned patterns can be easily found in real-world matrices shown in the SuiteSparse Matrix Collection [13, 29]. In the case where the bitsets in both slices are all set to 1s (i.e., there is no work to be skipped), we run the vanilla inner kernel used in standard dense matrix multiplication, which is slightly faster in this situation than the work-skipping inner kernel as it does not do any bitset processing.

If *GET_ALG()* returns *SPARSE_DENSE* or *DENSE_SPARSE*, which is when one of the slices is relatively sparse and one of the slices is relatively dense, we use a different inner kernel algorithm that multiplies the two slices by iterating through each element in each row in the CSR representation of the relatively sparse slice, and uses the column index of each element to directly access the corresponding column in the relatively dense slice to multiply. Direct access is possible because an $n_r \times k_c$ slice that is packed in memory can be viewed as an array of k_c n_r-sized columns of numbers. In the process of multiplying a row in the relatively sparse slice with a $n_r \times k_c$ slice, n_r subresults

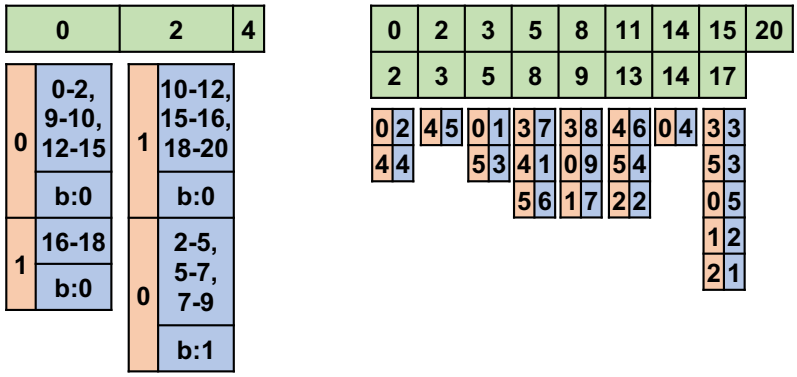
```
inner_kernel_2x8_work_skipping(
  double* slice_A, uint64_t* A_bitset,
  double* slice_B, uint64_t* B_bitset,
  double C[M][N], int k_c, int i, int j) {

  // Store 8 bit integers from 0 to 63 in a SIMD register
  simd_reg bit_indices = [0, 1, 2, 3, 4, ... , 62, 63]
  // Memory to write indices of columns to multiply
  uint8_t indices[64];
  simd_reg a_reg, b_reg, c0, c1;
  simd_setzero(c0); simd_setzero(c1);

  // Work on 64 columns of numbers at a time
  for (int k = 0; k < k_c; k += 64) {
    uint64_t and_bitset = A_bitset[k/64] & B_bitset[k/64];
    simd_reg column_indices =
      simd_compress_8bit(and_bitset, bit_indices);
    simd_store(indices, column_indices);
    columns_to_process = popcount(and_bitset);

    for (int kk = 0; kk < columns_to_process; kk++) {
      b_reg = simd_load(&slice_B[16*(k+indices[kk])]);
      a_reg = simd_broadcast(&slice_A[2*(k+indices[kk])]);
      c0 = simd_fmadd(a_reg, b_reg, c0);
      a_reg = simd_broadcast(&slice_A[2*(k+indices[kk])+1]);
      c1 = simd_fmadd(a_reg, b_reg, c1);
    }
  }
  c0 = simd_add(&C[i][j],c0); simd_store(&C[i][j],c0);
  c1 = simd_add(&C[i+1][j],c1); simd_store(&C[i+1][j],c1);
}
```

Fig. 6. 2×8 Inner kernel function with work-skipping

are stored in SIMD registers, before being written out to memory after iterating through all the elements in the row. We will refer to the two inner kernels used in this case as the *sparse-dense inner kernel* (when the left slice is relatively sparse) or the *dense-sparse inner kernel* (when the right slice is relatively sparse).

If *GET_ALG()* returns *SPARSE_SPARSE*, which is when both the slices to be multiplied are relatively sparse, we do not multiply the slices, as they will be handled separately when SPALM does sparse matrix multiplication in a later step.

4.2.1 Parallelization. We implement parallelism for this process by dividing up the panel-block multiplications in Figure 5b among different threads, as the result locations of each panel-block multiplication do not overlap and thus do not need synchronization. In the case where there are not enough panel-block multiplications for all the threads, we further divide the slice-block multiplications in Figure 5c among different threads. We use OpenMP for parallelization.



(a) Slice CSC matrix where slices are sorted according to bucket

(b) Bucketed CSC matrix

Fig. 7. Slice CSC matrix from Figure 4 sorted by bucket and converted to bucketed CSC format

4.3 Sparsity-Sensitive Sparse Multiplication

As explained in Section 3.2, given two CSR matrices A and B, fast matrix multiplication is done by first transposing B to be column-major, and then multiplying each element in A with the elements in the corresponding column in B by using the column index of the element from A to directly access the corresponding column in B. This is fast as an element from A is multiplied with a continuous array of elements from B, which allows for good locality on both the input and output, and easy vectorization of code.

In this section we explain SPALM's novel algorithm of multiplying relatively sparse slices from both matrices (every slice-slice multiplication that was not done during the dense matrix multiplication process explained in Section 4.2) by transposing B so that elements in a column in the resulting CSC matrix are ordered according to the density of the slices they belong to. This novel ordering allows for selective multiplication of elements within a column that are from slices below a certain density based on the density of a slice an element from A comes from, while still getting the locality and vectorization benefits a standard sparse matrix multiplication algorithm benefits from.

4.3.1 Creating a bucketed CSC matrix. With the slice CSC representation of the right matrix, we create a CSC matrix where the elements of each column are ordered according to the assigned bucket of their slices. This process is shown in Figure 7. To do this, given a slice CSC matrix, we first sort each column in the slice CSC matrix according to each slice's bucket in increasing bucket order (Figure 7a). Since the number of buckets is finite, we use a bucket sort on the slices to quickly sort the buckets. Then for each column, we iterate through each slice, and for each slice iterate through each subrow, and for each subrow iterate through the corresponding elements in the original CSR matrix and append each element to its corresponding column in the result CSC matrix using its column index. For example, the elements in the first subrow of the first slice, which are 2 and 1, are appended to their corresponding columns, column 0 and 2. The second subrow of this slice, which has a single element 7, is appended to column 3. For each b in range 0 to B-1, where B is the number of buckets used to bucket the slices, for each column in the resulting CSC matrix we save the start index of the first element from slices with bucket b, for a total of B indices per column. An example bucketed CSC matrix where $B = 2$ is shown in Figure 7b.

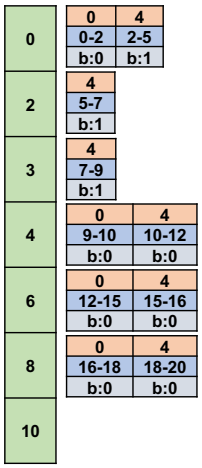


Fig. 8. CSSR matrix from Figure 4b with bucket indices

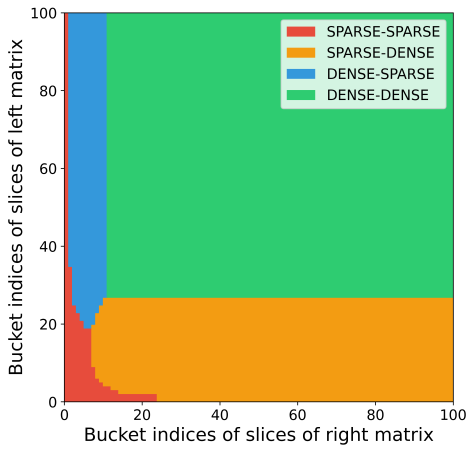


Fig. 9. Optimal algorithm when B=100

We parallelize this process by dividing the columns in the slice CSC matrix among threads, which does not require synchronization as the elements in each column will be written to non-overlapping locations. Again, we use OpenMP for parallelization.

4.3.2 Labeling subrows with densities. The next step is to iterate through the rows of the left matrix, and multiply each element with a subarray of the corresponding column in the bucketed CSC matrix based on the bucket of the slice the element comes from. To know the assigned bucket of each element during matrix multiplication, we first label each subrow in the left matrix with the bucket of the subrow's slice. This is done by iterating through each slice in the slice CSC matrix, and labeling each subrow in the CSSR matrix with the bucket index of the slice. An example of this labeling is shown in Figure 8, using our running matrix example. In practice, the process of creating a bucketed CSC matrix is done on the right matrix, and the process of labeling subrows is done on the left matrix.

4.3.3 Sparse matrix multiplication. After transforming the left matrix into a labeled CSSR matrix, and the right matrix into a bucketed CSC matrix, we then iterate through each subrow in the left matrix and multiply elements from each subrow with a subarray of the corresponding column in the right matrix. We use a function, *GET_SPARSE_THRES()*, that takes as input a bucket index, and outputs the minimum bucket index of the slices that it should not be multiplied with via a sparse matrix multiplication, to determine the subarray of the corresponding column to be multiplied. As an example, if the right matrix is the bucketed CSC matrix in Figure 7b, and an element from the left matrix has a column index of 4, has a bucket index of 0, and *GET_SPARSE_THRES(0)* = 2, the element will be multiplied with elements 8-11 (i.e. the entire 4th column) of the bucketed CSC matrix. If another element from the left matrix also has a column index of 4, but has a bucket index of 1, and *GET_SPARSE_THRES(1)* = 1, the element will be multiplied with elements 8-9 (i.e. elements from the 4th column that come from slices with a bucket index < 1) from the bucketed CSC matrix. In general, *GET_SPARSE_THRES()* returns a smaller bucket index the larger the input bucket index is, as it becomes more inefficient to multiply slices using a sparse matrix multiplication the denser the slices are. How *GET_SPARSE_THRES()* is implemented is explained in Section 4.5.

We parallelize this process using OpenMP by dividing the rows in the CSSR matrix among threads, which requires no synchronization as the location of the matrix multiplication results does not overlap for each row.

4.4 Determining Tiling Parameters

We explain how the optimal values of m_r, n_r, k_c and n_c are determined for a machine. Since m_r and n_r depend solely on the number of SIMD registers in a machine, we determine the values of m_r and n_r by choosing values that allow for an inner kernel function that stores the largest number of subresults while being within the machine's SIMD register budget, which is 32 for a machine that supports the AVX512 instruction set. For double data types, we use an inner kernel implementation where $m_r = 12$ and $n_r = 16$, and for float data types, we use values $m_r = 12$ and $n_r = 32$.

To determine the optimal value of k_c and n_c, we first determine the optimal value of k_c, as the optimal value of k_c does not depend on the optimal value of n_c as explained in Section 3.1. We determine k_c by measuring the matrix multiplication time of multiplying two fully dense 4096×4096 matrices using the dense matrix multiplication algorithm while varying the value of k_c to be a power of 2 ranging from 16 to 4096 while n_c is set to 64, and then choosing the value of k_c that resulted in the shortest matrix multiplication time. We then determine the optimal value of n_c using a similar process, by fixing k_c to be the optimal value found in the previous step, and by choosing the value of n_c out of values that are a power of 2 ranging from 16 to 4096 that resulted in the fastest matrix multiplication time on the aforementioned matrices. The process of determining k_c and n_c is done only once when SPALM is installed on a machine, and is used for all future matrix multiplications.

4.5 Determining Algorithm Thresholds

We now explain the implementation of the two functions *GET_ALG()* and *GET_SPARSE_THRES()*, which are used for determining which algorithm to use when multiplying slices. The following process is done once upon installation of SPALM after the optimal tiling parameters are determined, for every data type SPALM supports. Given a value of B, the number of buckets SPALM plans to use when bucketing the slices in the slice CSC matrix during matrix multiplication, we first create B 2048×2048 matrices where the ith matrix has a uniform density (i.e. the matrices have a uniform distribution of non-zero values) of $((i+1)/B) \times 100\%$, where i starts from 0. We then multiply all $B \times B$ combinations of matrices using the four matrix multiplication algorithms that SPALM supports: Sparse matrix multiplication (Section 4.3), and three different versions of dense matrix multiplication: one using a sparse-dense inner kernel, a dense-sparse inner kernel, and the work-skipping inner kernel (Section 4.2). For every combination, SPALM saves the algorithm that resulted in the fastest matrix multiplication time, and stores these results in a metadata file. We use 2048×2048 matrices as they are larger than the L2 cache which ensures the testing involves the tiling parameters found in the previous step while keeping calibration time relatively short.[2] We show a visualization of the result of this process on our test machine, when $B = 100$, in Figure 9. The four algorithms are represented as SPARSE-SPARSE, SPARSE-DENSE, DENSE-SPARSE, and DENSE-DENSE. SPALM defaults to $B = 100$ as the bucketing of slices is fine-grained enough so that choosing the wrong matrix multiplication algorithm at an algorithm boundary results in less than 5% slowdown. Using more buckets would have diminishing gains in speedups while using more memory; using fewer buckets would result in reduced sparsity sensitivity. The user is free to configure B to the needs and constraints of their system.

[2]The calibration phase took around 10 minutes for our test machine.

At runtime, SPALM reads the saved metadata file and initializes two arrays, one that stores the optimal algorithm given two input buckets, and one that stores, for every bucket, the largest bucket for which a sparse matrix multiplication algorithm is optimal. *GET_ALG()* is implemented via a simple array lookup using the two input buckets on the first array, and *GET_SPARSE_THRES()* is implemented via an array lookup using the input bucket on the second array.

While our method of determining algorithm thresholds is relatively simple in that it does not consider sparsity patterns within a slice, we think it is sufficient in the majority of cases because the slices are relatively small (e.g. dimensions of 16×512) and thus not big enough to contain sparsity patterns at a high resolution. It also has the advantage of having relatively small slice metadata overhead, while having fast and simple algorithm decision logic at runtime. We recognize the potential for improvement and leave investigating the tradeoffs associated with implementing a more sophisticated, sparsity-pattern-aware model to future work.

When there are too few operations performed via either the sparse or dense matrix multiplication algorithms, the overhead of starting the algorithm could be greater than the time taken doing actual multiplications. To determine which algorithm to use in these cases, SPALM employs a cost estimation model where it first estimates the number of element multiplications done via each matrix multiplication algorithm, and then uses the information to make a decision. To estimate the number of element multiplications, SPALM iterates through pairs of columns in both the slice CSC matrices (in the same fashion as Figure 5a), and estimates the number of element multiplications for doing each slice-slice multiplication. This estimation step is fast, linear in the number of slices in each slice CSC matrix, and can be easily parallelized by dividing up pairs of slice columns (Figure 5a) among threads.

If the number of multiplications done via the dense matrix multiplication (*dense_m*) is small, we observe that the matrix multiplication time is slow compared to doing only a sparse matrix multiplication, even if the slices are dense, due to less efficient reuse between slices during the dense matrix multiplication. To address this, if *dense_m* is below a certain threshold, SPALM defaults to using only a sparse matrix multiplication algorithm. We determine this threshold at installation time by performing tests where we multiply two 2048×2048 matrices with the same uniform density that increases from 1% to 100% across tests using only a sparse matrix multiplication algorithm and using only a dense matrix multiplication algorithm, and saving the value of *sparse_m* at the first test where using a dense matrix multiplication algorithm is consistently faster than using a sparse matrix multiplication algorithm.

If, within a single multiplication task, the number of multiplications done via sparse matrix multiplication (*sparse_m*) is small compared to *dense_m*, we notice that matrix multiplication is slower compared to using a dense matrix multiplication algorithm exclusively. Elements that would be multiplied using a sparse matrix multiplication algorithm could 'piggyback' off the locality and reuse done in the dense matrix multiplication algorithm. As a heuristic, SPALM uses an exclusively dense matrix multiplication algorithm when $sparse\_m < dense\_m \times 0.1$ as we see approximately a 10-20% slowdown in these cases, and checking whether *sparse_m* is smaller than a tenth of *dense_m* comfortably covers the majority of cases where there is a slowdown. Choosing a smaller threshold (e.g. 0.05) results in more cases that incur slowdowns, and a larger threshold (e.g. 0.2) inhibits SPALM's ability to achieve fast performance via applying different algorithms to different data.

When SPALM determines that the matrix multiplication is to be done via only a sparse matrix multiplication or only a dense matrix multiplication based on the estimation process, we fall back to a highly optimized library such as Intel oneMKL, as these libraries generally offer better performance for doing a purely sparse or purely dense matrix multiplication. SPALM will always fall back to a SpGEMM library function when only sparse matrix multiplication is to be done. When only dense matrix multiplication is to be done, SPALM falls back to a GEMM library function when

```
SELECT E.row, P.row, SUM(E.v * P.v)
FROM E, P WHERE E.col = P.col
GROUP BY E.row, P.row
```

Fig. 10. Example matrix multiplication query

$dense\_m > 0.8 \times M \times K \times N$, where $M \times K$, $K \times N$ are the dimensions of the matrices to be multiplied. Because SPALM's dense matrix multiplication implementation is generally 10-30% slower than library functions for fully dense matrices, choosing the threshold as 0.8 allows SPALM to fall back to library functions in the majority of cases where it would have otherwise been slower than a library function. Choosing a smaller threshold would mean we lose potential speedups, and a larger threshold would result in more cases where using SPALM sees a slowdown compared to libraries.

4.6 Output Format

If possible, SPALM will write its results in a dense matrix, as writing to a dense matrix is faster and it is beneficial to have a format that supports direct access in the later stages of the data pipeline. However, in the case that the system does not have enough memory to support a dense output, SPALM writes its results in a CSR matrix instead. It does this by first doing a matrix multiplication-like step with the same process explained in Section 4.2 and 4.3, but instead of doing multiplication, SPALM counts the number of unique locations the results are written to. This number is used to allocate memory for the result CSR matrix, after which SPALM does actual matrix multiplication and writes its results to the aforementioned CSR matrix.

5 DuckDB Integration

We integrate SPALM into an open-source relational database by implementing SPALM as a standalone operator in DuckDB [40]. We use the *extensions* functionality provided by DuckDB to extend DuckDB's query optimizer to replace the join and aggregation operator in a query with the form shown in Figure 10 with a custom SPALM operator that performs the join and aggregation using a matrix multiplication. We are aware that this replacement does not always improve performance, such as when the input tables are very small or when SPALM breaks otherwise deeply pipelined operators because of its need to materialize matrices. A more complete integration of SPALM into a database would involve a cost-based decision when rewriting a query plan to use SPALM, as well as not treating SPALM as a black-box operator to consider other optimization opportunities, such as pushing down a filter below the SPALM operator. The implementation of these features is beyond the scope of this paper, and is left to future work.

6 Experimental Evaluation

6.1 Baselines

In the following sections, we compare SPALM with **DIM3** [24] and **DHK** [14] (see Section 2). Both DIM3 and DHK are optimized for the Join-Project operation instead of true matrix multiplication. DIM3 processes bits rather than numbers, and work-skipping optimizations can reduce the computational complexity of some workloads. For a fair comparison with DIM3, we implemented a version of DIM3 that uses fast matrix multiplication libraries to multiply the resulting matrices created by its partitioning scheme.[3] We compare with DHK directly as it does not do any work-skipping.

[3]While DIM3 does have code for performing matrix multiplication, unlike its join-project algorithm it is single threaded and does not apply any sparsity-sensitive partitioning during multiplication.

While we are not able to compare against ATMULT [27] as its code is proprietary, we expect SPALM will outperform ATMULT in cases with sparsity patterns that are more fine-grained than the relatively large partition sizes used in ATMULT. We also compare SPALM against **OpenBLAS** [38], an open-source library for performing linear algebra, and **Intel oneMKL** [26], a library containing optimized math routines for modern CPUs. Specifically, we use oneMKL's SPGEMM (sparse-sparse matrix multiplication), SPMM (sparse-dense matrix multiplication), and GEMM (dense-dense matrix multiplication) routines as baselines. All are popular libraries used in various domains such as ML and scientific computing, and are also used as the (sometimes default) backend in other commonly used libraries such as SciPy [57] and NumPy [21].

6.2 Execution Time on Synthetic Datasets

6.2.1 Dataset description. We test SPALM on a variety of synthetic datasets with common sparsity patterns that appear in real-world use cases and data. Each sparsity pattern is tunable with a numeric parameter p, so that we can create multiple matrices of the same sparsity pattern, but with varying degrees of sparsity. Figure 11 shows a visualization of several examples, where a nonzero element is a black pixel and a zero element is a white pixel. The sparsity patterns are as follows:

Uniform density: Each element in the matrix has an equal probability $p\%$ of being non-zero. This sparsity pattern is seen in regularization techniques in ML such as DropConnect [58].

Random row: Each row of the matrix has an equal probability $p\%$ of being all non-zero, and is all zero otherwise. This kind of sparsity pattern is used in pruning in ML where whole neurons in a fully connected layer are dropped [22, 32], and can arise in situations where the user applies a filter to rows in their dataset.

Random column: Each column of the matrix has an equal probability $p\%$ of being all non-zero, and is all zero otherwise. This sparsity pattern appears in ML pruning techniques where entire features are removed [22], and can appear in situations where users remove columns from their dataset.

Diagonal: Matrix entries outside of the diagonal band of width $2 \times p - 1$ are zero. This pattern is common in real-world scientific datasets [53, 54], and matrices that feature diagonal bands of widths larger than 1 are used in sparse attention mechanisms in LLMs [4, 62] which allows each token to only consider tokens around it, enabling greater computational efficiency while capturing necessary contextual information.

Edge: Matrix entries that are within distance p from the boundaries of the matrix are non-zero, while all other entries are zero. This sparsity pattern is seen in sorted datasets where heavily populated rows and columns are moved to the edges of the matrix. This sparsity pattern is also used in sparse attention [62], where it allows certain tokens to consider the entire input sequence at once.

Square: A square where each side is of length p in the upper left corner is fully non-zero, while other entries are zero. Sparsity patterns where a large subrectangle of the matrix is mostly non-zero are common in datasets where certain combinations of rows and columns are more popular than others [53, 55].

Triangular: The upper triangular of the matrix is non-zero, whereas the lower triangular of the matrix has a uniform density of $p\%$. Triangular matrices are commonly used in many scientific computations such as linear systems and matrix decompositions, and as a result appear in many applications such as signal processing [41].

Gradual row: The rows of the matrix gradually become sparse to dense from the first row to the last row. The first row has a density of zero, and the last row has a uniform density of $p\%$. Datasets with these density patterns are common in scientific datasets.

Gradual column: The same as the gradual row sparsity pattern, but applied to columns instead of rows.

Block 16 or 128: Non-zero elements are distributed with a uniform density of $p\%$ at the granularity of 16×16 (or 128×128) blocks. This is a common structured sparsity pattern used in ML, for the purpose of model compression and efficient training and inference, and specialized hardware has been introduced [12] to take advantage of this sparsity pattern.

Sparsity Pattern	Parameters
Uniform density	1, 5, 10, 20, 50, 75, 100
Random row	1, 5, 20, 50
Random column	1, 5, 20, 50
Diagonal	5, 100, 500, 1000, 2000
Edge	10, 50, 100, 1000, 2000
Square	500, 1000, 2000, 3000
Triangular	0, 1, 5, 10, 50
Gradual row	5, 20, 100
Gradual column	5, 20, 100
Block 16	1, 5, 10, 50
Block 128	1, 5, 10, 50

Table 1. Synthetic datasets used for experiments

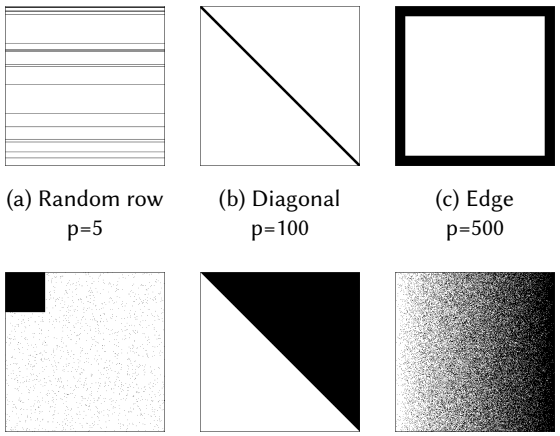
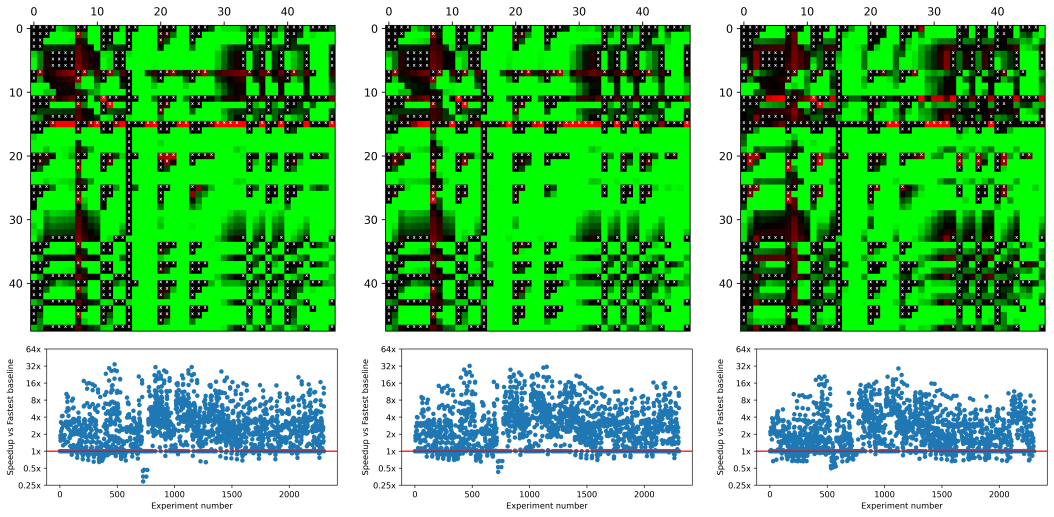


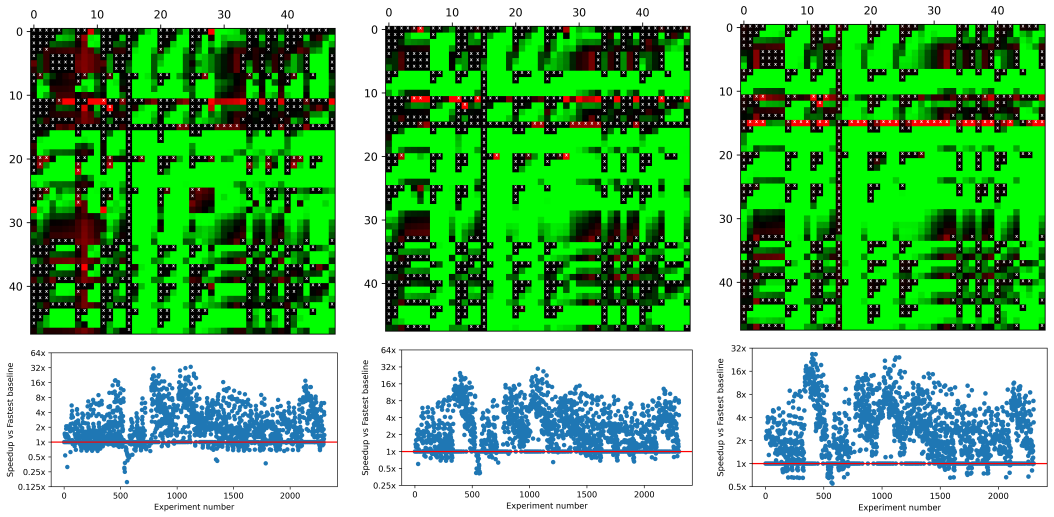
Fig. 11. Visualization of example synthetic matrices

6.2.2 Experiment results on 32-bit floats. For experiments, we used the parameters for each sparsity pattern shown in Table 1 to create 48 different matrices, each with dimensions 8000×8000, which is around the size of the matrices used in LLM inference on CPUs in [30], and is near the average dimensions used in the real-world matrices in Section 6.3. The matrices use 32-bit floats for values, which is the most commonly used data type in ML. For every possible combination of matrix multiplication using the 48 matrices, we compare only the matrix multiplication time of SPALM against every baseline except for DHK as DHK assumes the two input matrices are identical. Every method of execution has its matrices prepared in the correct format, which means for SPALM we do not include the time of creating slice CSC matrices, and we write results to a dense matrix. The experiments were conducted on a Linux server with a 24-core 2.4GHz Xeon Gold 6312U processor, and we report the median of 5 runs for each matrix multiplication.

Results are shown in Figure 12a-c. For each data type, we ran experiments with 1, 4, and 20 thread(s). We did not use 24 threads as it resulted in unstable performance, for both the baselines and SPALM. For each thread and data type configuration, we visualize the experiment results in two ways: on the top we display a 48×48 grid where the color of each cell represents the speedup of SPALM compared to the fastest baseline for that particular combination of input matrices. Cells with white Xs represent experiments where SPALM falls back onto a library call, where the entire matrix is multiplied with either a sparse-sparse matrix multiplication or a dense-dense matrix multiplication via a library call. For cells that do not have Xs, the color of the cells falls on a spectrum from bright red to black to bright green, where bright red represents a $0.5\times$ speedup (i.e. SPALM $2\times$ slower than the fastest baseline), and bright green represents a $2\times$ speedup. We cap the brightness scale so that speedups larger than $2\times$, or smaller than $0.5\times$, have the same shade of



(a) 1 thread (b) 4 threads (c) 20 threads



(d) 64-bit doubles, 20 threads (e) 32-bit floats, Intel 8488C, 48 threads (f) End-to-end time, 32-bit floats, 20 threads

Fig. 12. Speedups for SPALM compared to the fastest baseline for all 48 × 48 experiments on various configurations.

Data type, Configuration/Measurement	# of threads	Avg speedup	% of tests and avg speedup when speedups > 1	% of tests and avg speedup when speedups < 1	% of fallbacks
32-bit Float	1	3.33×	72.66% / 4.23×	10.72% / 0.82×	21.74%
	4	3.39×	73.48% / 4.28×	11.76% / 0.85×	
	20	2.71×	68.71% / 3.53×	17.19% / 0.86×	
64-bit Double	20	2.68×	62.50% / 3.72×	19.87% / 0.85×	23.95%
32-bit Float on Intel 8488C	48	2.51×	65.23% / 3.35×	17.53% / 0.86×	24.86%
32-bit Float, E2E time	20	3.29×	73.10% / 4.17×	26.90% / 0.90×	21.74%

Table 2. Average speedup and fallback percentages for experiments on synthetic datasets

bright green or bright red as 2× and 0.5× speedup. Underneath, we display a scatterplot using the values of the speedups of the individual experiments displayed on a log scale.

For each experiment in Figure 12a-c, we report the average speedup, percentage of tests and average speedup of cases where SPALM has a speedup larger than 1, percentage of tests and average speedup of cases where SPALM has a speedup smaller than 1, and the percentage of tests where SPALM falls back onto a library function in the first 3 rows of Table 2. SPALM achieves speedups larger than 2× for the majority of cases, and generally successfully falls back to a fast library function in other cases.

The majority of cases where SPALM is slower than the fastest baseline are for experiments on rows 11 and 15 on the grid. These correspond to cases where either the *diagonal* matrix with parameter 5, or the *random column* matrix with parameter 1 is the left matrix. The fastest baseline in these cases is generally the Intel oneMKL SPMM function (multiplying a sparse matrix with a dense matrix), and these sparsity patterns are particularly amenable to SPMM algorithms because 1) they are very sparse and 2) many non-zero elements have overlapping columns which allows for good locality when iterating over the corresponding column in the right matrix. It is difficult to match the performance in specifically these cases as we would need a more sophisticated cost estimation model that takes into account the locality of element accesses as well, and we leave the handling of these cases to future work.

Inspecting the 48×48 speedup grid shows that SPALM successfully falls back to highly optimized matrix multiplication libraries when the input matrices are comparatively sparse (overall density less than or equal to 5%) or comparatively dense (overall density larger than or equal to 50%) and that SPALM mainly achieves large speedups in cases that are on the boundary of sparse matrix multiplication and dense matrix multiplication, which are cases that previous methods are not proficient at handling, and that users would have a difficult time choosing a performant matrix multiplication algorithm.

For experiments involving multithreading, at higher thread counts we see smaller speedups in general, and slightly fewer cases where SPALM is faster than the fastest baseline. This is due to our parallelization implementation not being as performant as Intel oneMKL's parallel GEMM implementation, thus reducing the performance gap between SPALM and Intel oneMKL and DIM3, which relies on calls to Intel oneMKL. While we were not able to look at Intel oneMKL's code as it is not open source, we speculate that oneMKL may be using more sophisticated optimizations such as changing tile sizes depending on the number of threads. Implementing these optimizations is beyond the scope of this paper; oneMKL is a hand-optimized, machine-specific private library managed by Intel that incorporates 30 years of cumulative optimizations. Nevertheless, even at 20 threads, SPALM is slower than the fastest baseline in fewer than 18% of the cases, in which SPALM is on average only about 14% slower, and SPALM is faster than the fastest baseline for the majority of the experiments, in which SPALM is on average more than 250% faster.

We also performed experiments with non-square matrices where we vary the non-common dimension between matrices among 2000, 4000 and 8000. We omit the results as the trends were similar, and we get comparable speedups to square matrices.

6.2.3 Experiment results on 64-bit doubles. We repeat the same experiment, using 64-bit double data types, the most popular data type used in scientific computing. Results are shown for experiments run with 20 threads in Figure 12d and the fourth row of Table 2. SPALM has the same trends and comparable speedups to when operating on 32-bit floats with 20 threads, showing that SPALM can apply its techniques to multiple data types. SPALM also achieves its highest speedup (34×) across all synthetic dataset experiments here on the (edges, 500) × (diagonal 100) dataset, which have sparsity patterns with both horizontal and vertical locality patterns that are not well handled

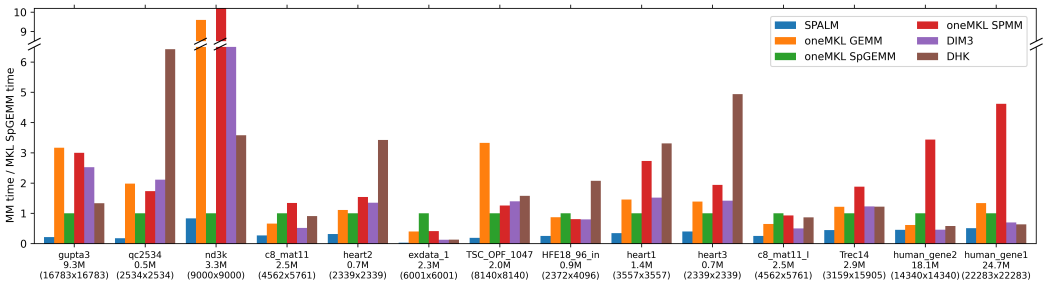


Fig. 13. Relative execution times for SuiteSparse matrices

by our baselines. Compared to 32-bit floats with 20 threads, there are some cases in the bottom left quarter of the grid where SPALM is slower than the fastest baseline. Upon inspection, they are cases where SPALM's cost estimation model chooses the wrong algorithm, and choosing the correct algorithm would have prevented a slowdown. However, these cases amount to less than 7% of the experiments, which in most cases have a slowdown of less than 10%. Similar to the tradeoffs encountered during database query optimization, we acknowledge that SPALM's cost estimation model cannot make the right decision 100% of the time, and these slowdowns are the result of the tradeoff between having a more sophisticated cost estimation model that takes longer and is harder to implement, versus a simpler cost model.

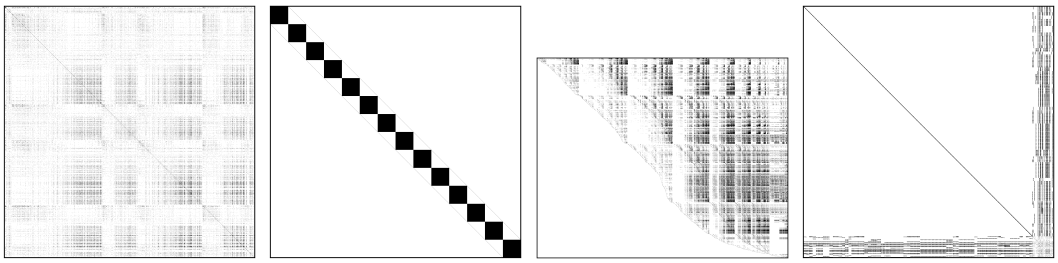
6.2.4 Experiment results on a different machine. We test SPALM on a different machine with more threads. After initializing the libraries with a short warmup run by multiplying two small (200×200) matrices,[4] we run experiments on a Linux server with a 48-core Intel Xeon Platinum 8488C processor, and report results for 32-bit floats, using 48 threads. Results are shown in Figure 12e and the fifth row of Table 2. We can see that the trends are similar to previous experiments, thus showing that SPALM can effectively adapt its parameters and cost model to new machines.

6.2.5 Experiments measuring end-to-end time. We measure the end-to-end time of multiplying two matrices that are given in CSR format. For SPALM this includes the time creating the slice CSC matrices, as well as running the cost estimation model. For baselines involving dense matrices, this includes the time spent converting the CSR matrix into a dense format. We do not include the time taken for memory allocation when doing this dense format conversion, and assume the database uses memory pre-allocated for matrix operations. Results are shown in Figure 12f and the sixth row of Table 2. The trends are similar to previous experiments, and the overhead of creating the slice CSC matrices and running the cost estimation model is small, as SPALM is only around 11% slower than the fastest baseline on average in cases where SPALM's speedup is below 1. Many slowdowns come from row 15 of the grid, which is similar to the results seen in Section 6.2.2, although now there are more slowdowns due to including the cost of transposing the right matrix during sparse matrix multiplication.

6.3 Execution Time on Real-World Datasets

We test SPALM on real-world data from the SuiteSparse matrix collection [13, 29]. To evaluate our previous claim that SPALM is effective in covering cases that are on the boundary of sparse matrix multiplication and dense matrix multiplication, we obtained a subset of matrices from the SuiteSparse matrix collection that are potentially on the verge of this threshold, by choosing

[4]This process takes less than half a second and is done only once, before running all $48 \times 48 = 2304$ experiments.



(a) human_gene1 (b) qc2534 (c) c8_mat11_l (d) gupta3

Fig. 14. Visualizations of four example matrices from the SuiteSparse matrix collection.

matrices that have at least 1000 rows and 1000 columns, and have an overall density equal to or greater than 3%, and for each matrix we evaluate the performance of multiplying the matrix with its transpose, which is a common method of evaluation [15, 35]. We run our experiments using 20 threads, on the same machine described in Section 6.2.2, and use the median of 5 runs when measuring execution time. For the 7 matrices where the original data type of the matrix was not float or double, we cast the matrix to the double data type, which was the most common data type of the other matrices. Out of the 29 matrices in the SuiteSparse matrix collection that satisfy the above conditions, SPALM is faster than the fastest baseline for 15 of those cases, and falls back to Intel oneMKL's SpGEMM function in the remaining 14 cases, in all of which Intel oneMKL's SpGEMM function was the fastest baseline. Experiment results for the 15 cases where SPALM is faster than the fastest baseline are shown in Figure 13, along with the name, dimensions, and number of non-zero elements of each matrix. SPALM achieves on average 2.93× speedup over the fastest baseline, with its largest speedup of 5.63× coming from multiplying the qc2534 matrix. We can see that the fastest baseline is not the same across all the experiments, and there is not a strict correlation between the density of the matrices and the fastest baseline, which makes it hard for a user to choose a fast matrix multiplication algorithm when sparse matrices are involved. For example, it is faster to multiply matrix heart2 with a sparse algorithm, despite heart2 being denser than matrix exdata_1, where it is faster to do multiplication with a dense algorithm. We can see that for a variety of different matrix shapes, SPALM's matrix multiplication algorithm is adaptable and is able to automatically apply a dense matrix multiplication algorithm or a sparse matrix multiplication algorithm on selected regions of the matrix to get good performance without requiring the user to manually choose an algorithm. For reference, visualizations of the sparsity pattern of four example matrices reported in the experiments are shown in Figure 14.

6.4 Preprocessing Overheads

Dimension size	CSSR	Slice CSC	Cost estimation	MM
dim < 5000	3.09	7.70	0.22	88.99
dim >= 5000	0.58	2.14	0.06	97.22

Table 3. Relative time (%) taken for each step in SPALM

We measure the overhead of creating CSSR and slice CSC matrices, as well as doing matrix multiplication cost estimation, compared to the end-to-end execution time in SPALM. We report the average proportions for these times for the real-world matrices whose dimensions are (1) less

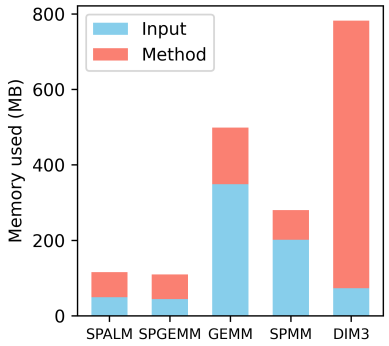


Fig. 15. Average memory usage in SPALM

than 5,000 and (2) 5,000 or greater. Results are shown in Table 3. The time to create slice CSSR and slice CSC matrices is manageable for smaller matrices (collectively 10.8% of e2e time) and scales well to larger matrices (collectively 2.7% of e2e time) as the slice CSC matrices see more reuse during matrix multiplication. In both cases, the time of doing cost estimation is negligible (less than 0.3% of e2e time).

6.5 Memory Usage

We compare the memory usage of SPALM with other baselines. For each method, we measure the memory footprint of both the input matrices and the matrix multiplication process itself, for each real-world dataset, and report the average memory usage. Results are shown in Figure 15. We omit results for GEMM with OpenBLAS as its memory usage patterns were similar to GEMM with oneMKL, and we omit memory to store the output matrix as it is the same for every method. We can see that SPALM's average memory usage is similar to the least memory-intensive baseline, oneMKL SPGEMM, due to not needing a dense input format, and using lightweight data structures. SPALM tends to use more memory than SPGEMM when sparse matrix multiplication is done during matrix multiplication due to creating additional data structures along with a CSC (i.e. transposed CSR) matrix, but sometimes needs less memory than SPGEMM when there is no need to create a CSC matrix.

6.6 Execution Time with DuckDB

We evaluate the effectiveness of SPALM in the context of a real-world database management system. To do this, we store matrices as relational tables where each non-zero element corresponds to a (row index, column index, value) tuple in the table, and then execute the queries in Figure 16. We compare the end-to-end times of query execution using SPALM, using the fastest baseline for matrix multiplication for the join-aggregate portion of the query, and unmodified DuckDB, on 20 threads.

Experiment results are shown in Table 4. Using SPALM in DuckDB achieves an average end-to-end query time speedup of 2.26× compared to using other methods of matrix multiplication for join-aggregation, which is made possible by the fact that matrix multiplication originally took up the majority of time in these queries. We can also see that our custom SPALM operator is compatible with existing query optimization rules, as the SPALM operator is introduced in the query plan after filter pushdown, and is able to directly use the preceding operator's output as input.

```
-- Q1: Matrix multiplication on human_gene1 (24.7M rows)
SELECT E.row, P.row, SUM(E.v * P.v)
FROM E, P WHERE E.col = P.col
GROUP BY E.row, P.row

-- Q2: Value filter on human_gene1 (24.7M rows)
SELECT E.row, P.row, SUM(E.v * P.v) FROM E, P
WHERE E.col = P.col AND P.v >= 0.04 AND E.v >= 0.04
GROUP BY E.row, P.row

-- Q3: Diagonal pattern on 16k x 16k dense matrix (256M rows)
SELECT E.row, P.row, SUM(E.v * P.v) FROM E, P
WHERE E.col = P.col
AND ABS(E.row - E.col) <= 2000 AND ABS(P.row - P.col) <= 2000
GROUP BY E.row, P.row

-- Q4: Row pruning on 16k x 16k dense matrix (256M rows)
SELECT E.row, P.row, SUM(E.v * P.v) FROM E, P
WHERE E.col = P.col AND HASH(E.row)%4=0 AND HASH(P.row)%4=0
GROUP BY E.row, P.row
```

Fig. 16. Test matrix multiplication queries

Query / Method	DuckDB w/ SPALM	DuckDB w/ fastest baseline	DuckDB (unmodified)
Q1	6.45 (5.98)	10.52 (10.01)	230.18 (–)
Q2	2.53 (2.20)	3.38 (3.07)	46.27 (–)
Q3	1.50 (0.71)	6.61 (5.81)	495.65 (–)
Q4	1.67 (0.79)	2.77 (1.95)	511.50 (–)

Table 4. E2E execution time (unit: s) for queries on DuckDB. Numbers in ( ) denote time spent doing matrix multiplication.

7 Conclusions

We propose SPALM, a matrix multiplication library utilizing a novel, sparsity-aware matrix multiplication algorithm. SPALM's matrix multiplication algorithm groups subranges of rows, and uses different matrix multiplication techniques depending on the dynamically-computed density of the row groups. We show through experiments that SPALM achieves speedups up to 34× compared to existing methods on a wide variety of sparsity patterns from both synthetic matrices and real-world matrices from various domains.

For future work, we plan to extend SPALM to GPUs. We also plan to investigate how SPALM, when integrated inside a DBMS engine can be co-optimized with more complex queries and data pipelines.

Acknowledgments

This work was supported by the National Science Foundation under grant III-2312991.

References

[1] 2025. *Personal Data Protection Act.* Retrieved October 6, 2025 from https://www.pdpc.gov.sg/overview-of-pdpa/the-legislation/personal-data-protection-act

[2] Subutai Ahmad and Luiz Scheinkman. 2019. How can we be so dense? The benefits of using highly sparse representations. *arXiv preprint arXiv:1903.11257* (2019).

[3] Hasan Metin Aktulga, Aydin Buluç, Samuel Williams, and Chao Yang. 2014. Optimizing sparse matrix-multiple vectors multiplication for nuclear configuration interaction calculations. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 1213–1222.

[4] Iz Beltagy, Matthew E Peters, and Arman Cohan. 2020. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150* (2020).

[5] Mark Blacher, Julien Klaus, Christoph Staudt, Sören Laue, Viktor Leis, and Joachim Giesen. 2023. Efficient and portable einstein summation in SQL. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–19.

[6] Davis Blalock and John Guttag. 2021. Multiplying matrices without multiplying. In *International Conference on Machine Learning*. PMLR, 992–1004.

[7] Keren Censor-Hillel, Petteri Kaski, Janne H Korhonen, Christoph Lenzen, Ami Paz, and Jukka Suomela. 2015. Algebraic methods in the congested clique. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*. 143–152.

[8] Timothy M Chan. 2007. More algorithms for all-pairs shortest paths in weighted graphs. In *Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*. 590–598.

[9] Lingjiao Chen, Arun Kumar, Jeffrey Naughton, and Jignesh M Patel. 2016. Towards linear algebra over normalized data. *arXiv preprint arXiv:1612.07448* (2016).

[10] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. 2019. Generating long sequences with sparse transformers. *arXiv preprint arXiv:1904.10509* (2019).

[11] Krzysztof Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Davis, Afroz Mohiuddin, Lukasz Kaiser, et al. 2020. Rethinking attention with performers. *arXiv preprint arXiv:2009.14794* (2020).

[12] Paolo D'Alberto, Taehee Jeong, Akshai Jain, Shreyas Manjunath, Mrinal Sarmah, Samuel Hsu, Yaswanth Raparti, and Nitesh Pipralia. 2024. Weight block sparsity: Training, compilation, and ai engine accelerators. *arXiv preprint arXiv:2407.09453* (2024).

[13] Timothy A Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 1–25.

[14] Shaleen Deep, Xiao Hu, and Paraschos Koutris. 2020. Fast join project query evaluation using matrix multiplication. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1213–1223.

[15] Mehmet Deveci, Christian Trott, and Sivasankaran Rajamanickam. 2018. Multithreaded sparse matrix-matrix multiplication for many-core and GPU architectures. *Parallel Comput.* 78 (2018), 33–46. doi:10.1016/j.parco.2018.06.009

[16] Marius Eich, Pit Fender, and Guido Moerkotte. 2018. Efficient generation of query plans containing group-by, join, and groupjoin. *The VLDB Journal* 27, 5 (Oct. 2018), 617–641. doi:10.1007/s00778-017-0476-3

[17] Philipp Fent and Thomas Neumann. 2021. A practical approach to groupjoin and nested aggregates. *Proc. VLDB Endow.* 14, 11 (July 2021), 2383–2396. doi:10.14778/3476249.3476288

[18] Jianhua Gao, Weixing Ji, Fangli Chang, Shiyu Han, Bingxin Wei, Zeming Liu, and Yizhuo Wang. 2023. A systematic survey of general sparse matrix-matrix multiplication. *Comput. Surveys* 55, 12 (2023), 1–36.

[19] Georgi Gerganov and the ggml-org contributors. 2023. llama.cpp: LLM inference in C/C++. https://github.com/ggml-org/llama.cpp. GitHub repository; Initial release: March 10, 2023; Accessed: October 17, 2025.

[20] Kazushige Goto and Robert A. van de Geijn. 2008. Anatomy of High-Performance Matrix Multiplication. *ACM Trans. Math. Softw.* 34, 3, Article 12 (may 2008), 25 pages. doi:10.1145/1356052.1356053

[21] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585, 7825 (Sept. 2020), 357–362. doi:10.1038/s41586-020-2649-2

[22] Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. 2021. Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks. *Journal of Machine Learning Research* 22, 241 (2021), 1–124.

[23] Yu-Ching Hu, Yuliang Li, and Hung-Wei Tseng. 2022. TCUDB: Accelerating database with tensor processors. In *Proceedings of the 2022 International Conference on Management of Data*. 1360–1374.

[24] Zichun Huang and Shimin Chen. 2022. Density-optimized intersection-free mapping and matrix multiplication for join-project operations. *Proceedings of the VLDB Endowment* 15, 10 (2022), 2244–2256.

[25] Ranggi Hwang, Minhoo Kang, Jiwon Lee, Dongyun Kam, Youngjoo Lee, and Minsoo Rhu. 2023. GROW: A row-stationary sparse-dense GEMM accelerator for memory-efficient graph convolutional neural networks. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 42–55.

[26] Intel Corporation. 2020–present. Intel® oneAPI Toolkits. https://software.intel.com/content/www/us/en/develop/tools/oneapi.html. Version: 2024.2.1, Accessed: March 16, 2026.

[27] David Kernert, Wolfgang Lehner, and Frank Köhler. 2016. Topology-aware optimization of big sparse matrices and matrix multiplications on main-memory systems. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE, 823–834.

[28] Daya Khudia, Jianyu Huang, Protonu Basu, Summer Deng, Haixin Liu, Jongsoo Park, and Mikhail Smelyanskiy. 2021. FBGEMM: Enabling High-Performance Low-Precision Deep Learning Inference. *arXiv preprint arXiv:2101.05615* (2021).

[29] Scott P Kolodziej, Mohsen Aznaveh, Matthew Bullock, Jarrett David, Timothy A Davis, Matthew Henderson, Yifan Hu, and Read Sandstrom. 2019. The suitesparse matrix collection website interface. *Journal of Open Source Software* 4, 35 (2019), 1244.

[30] Eldar Kurtic, Denis Kuznedelev, Elias Frantar, Michael Goin, and Dan Alistarh. 2023. Sparse Fine-tuning for Inference Acceleration of Large Language Models. arXiv:2310.06927 [cs.CL] https://arxiv.org/abs/2310.06927

[31] Vihan Lakshman, Anshumali Shrivastava, and Tharun Medini. 2024. Accelerating Large-Scale Neural Network Training on CPUs with ThirdAI and AWS Graviton. AWS Machine Learning Blog. https://aws.amazon.com/blogs/machine-learning/accelerating-large-scale-neural-network-training-on-cpus-with-thirdai-and-aws-graviton/ Published on February 29, 2024.

[32] Paul Michel, Omer Levy, and Graham Neubig. 2019. Are sixteen heads really better than one? *Advances in neural information processing systems* 32 (2019).

[33] Guido Moerkotte and Thomas Neumann. 2011. Accelerating queries with group-by and join by groupjoin. *Proceedings of the VLDB Endowment* 4, 11 (2011), 843–851.

[34] Seonjin Na, Geonhwa Jeong, Byung Hoon Ahn, Jeffrey Young, Tushar Krishna, and Hyesoon Kim. 2024. Understanding performance implications of llm inference on cpus. In *2024 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 169–180.

[35] Yusuke Nagasaka, Satoshi Matsuoka, Ariful Azad, and Aydın Buluç. 2018. High-performance sparse matrix-matrix products on Intel KNL and multicore architectures. In *Workshop Proceedings of the 47th International Conference on Parallel Processing*. 1–10.

[36] Yuyao Niu, Zhengyang Lu, Haonan Ji, Shuhui Song, Zhou Jin, and Weifeng Liu. 2022. TileSpGEMM: A tiled algorithm for parallel sparse general matrix-matrix multiplication on GPUs. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 90–106.

[37] Dan Olteanu. 2020. The relational data borg is learning. *arXiv preprint arXiv:2008.07864* (2020).

[38] OpenBLAS Project. 2011–present. OpenBLAS. http://www.openblas.net/. Version: 0.3.27, Accessed: March 16, 2026.

[39] Kwanghyun Park, Karla Saur, Dalitso Banda, Rathijit Sen, Matteo Interlandi, and Konstantinos Karanasos. 2022. End-to-end optimization of machine learning prediction queries. In *Proceedings of the 2022 International Conference on Management of Data*. 587–601.

[40] Mark Raasveldt and Hannes Mühleisen. 2019. Duckdb: an embeddable analytical database. In *Proceedings of the 2019 international conference on management of data*. 1981–1984.

[41] Florian Ries, Tommaso De Marco, Matteo Zivieri, and Roberto Guerrieri. 2009. Triangular matrix inversion on graphics processing unit. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. 1–10.

[42] Aurko Roy, Mohammad Saffar, Ashish Vaswani, and David Grangier. 2021. Efficient content-based sparse attention with routing transformers. *Transactions of the Association for Computational Linguistics* 9 (2021), 53–68.

[43] Pau San Juan, Pedro Alonso-Jordá, and Enrique S Quintana-Ortí. 2021. High performance and energy efficient integer matrix multiplication for deep learning. In *2021 29th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. IEEE, 122–125.

[44] Maximilian Schleich, Dan Olteanu, Mahmoud Abo Khamis, Hung Q. Ngo, and XuanLong Nguyen. 2019. A Layered Aggregate Engine for Analytics Workloads. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1642–1659. doi:10.1145/3299869.3324961

[45] Maximilian Schleich, Dan Olteanu, and Radu Ciucanu. 2016. Learning linear regression models over factorized joins. In *Proceedings of the 2016 International Conference on Management of Data*. 3–18.

[46] Maximilian Schleich, Amir Shaikhha, and Dan Suciu. 2023. Optimizing tensor programs on flexible storage. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–27.

[47] Gwanghwi Seo and Sungju Ryu. 2025. SpecBoost: Accelerating Tiled Sparse Matrix Multiplication via Dataflow Speculation. *IEEE Access* 13 (2025), 45568–45576.

[48] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research* 15, 1 (2014), 1929–1958.

[49] Wenbo Sun, Qiming Guo, Wenlu Wang, and Rihan Hai. 2025. Accessible and Portable LLM Inference by Compiling Computational Graphs into SQL. arXiv:2502.02818 [cs.DB] https://arxiv.org/abs/2502.02818

[50] Wenbo Sun, Asterios Katsifodimos, and Rihan Hai. 2023. Accelerating machine learning queries with linear algebra query processing. In *Proceedings of the 35th International Conference on Scientific and Statistical Database Management*. 1–12.

[51] Gemma Team, Thomas Mesnard, Cassidy Hardin, Robert Dadashi, Surya Bhupatiraju, Shreya Pathak, Laurent Sifre, Morgane Rivière, Mihir Sanjay Kale, Juliette Love, et al. 2024. Gemma: Open models based on gemini research and technology. *arXiv preprint arXiv:2403.08295* (2024).

[52] Daniel ten Wolde, Tavneet Singh, Gábor Szárnyas, and Peter A Boncz. 2023. DuckPGQ: Efficient Property Graph Queries in an analytical RDBMS.. In *CIDR*.

[53] The University of Florida/Texas A&M Sparse Matrix Collection. 2025-10-11. *GHS_indef/exdata_1 Matrix Description*. http://sparse.tamu.edu/GHS_indef/exdata_1

[54] The University of Florida/Texas A&M Sparse Matrix Collection. 2025-10-11. *Gupta/gupta3 Matrix Description*. http://sparse.tamu.edu/Gupta/gupta3

[55] The University of Florida/Texas A&M Sparse Matrix Collection. 2025-10-11. *Newman/power Matrix Description*. http://sparse.tamu.edu/Newman/power

[56] Virginia Vassilevska, Ryan Williams, and Raphael Yuster. 2010. Finding heaviest H-subgraphs in real weighted graphs, with applications. *ACM Transactions on Algorithms (TALG)* 6, 3 (2010), 1–23.

[57] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. 2020. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* 17 (2020), 261–272. doi:10.1038/s41592-019-0686-2

[58] Li Wan, Matthew Zeiler, Sixin Zhang, Yann Le Cun, and Rob Fergus. 2013. Regularization of neural networks using dropconnect. In *International conference on machine learning*. PMLR, 1058–1066.

[59] Lucas Wilkinson, Kazem Cheshmi, and Maryam Mehri Dehnavi. 2023. Register tiling for unstructured sparsity in neural network inference. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 1995–2020.

[60] Weipeng P. Yan and Per-Åke Larson. 1995. Eager Aggregation and Lazy Aggregation. In *Proceedings of the 21th International Conference on Very Large Data Bases (VLDB '95)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 345–357.

[61] Zhiwei Yang, Lu Lu, and Ruimin Wang. 2022. A batched GEMM optimization framework for deep learning. *The Journal of Supercomputing* 78, 11 (2022), 13393–13408.

[62] Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, et al. 2020. Big bird: Transformers for longer sequences. *Advances in neural information processing systems* 33 (2020), 17283–17297.

[63] Jianting Zhang and Le Gruenwald. 2018. Regularizing irregularity: bitmap-based and portable sparse matrix multiplication for graph data on GPUs. In *Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*. 1–8.

Received October 2025; revised January 2026; accepted February 2026