Bootes: Boosting the Efficiency of Sparse Accelerators Using Spectral Clustering

Sanjali Yadav
University of Maryland
College Park, USA
sanjali7@umd.edu

Bahar Asgari
University of Maryland
College Park, USA
bahar@umd.edu

Abstract

Sparse matrix-matrix multiplication (SpGEMM) is crucial in many applications, with numerous recent efforts focused on optimizing it. The row-wise product has emerged as a favorable SpGEMM dataflow due to its balanced performance, but it alone is insufficient to minimize data movement and off-chip traffic—key factors for efficient accelerator deployment. Previous studies face three key challenges: (1) reordering is often suboptimal, failing to maximize memory traffic reduction; (2) preprocessing steps are typically slow and inefficient, making the overhead hard to justify; and (3) certain sparsity patterns do not benefit from reordering, potentially increasing traffic, yet existing methods lack a mechanism to detect such cases. To address these challenges, we propose Bootes, a novel approach that leverages spectral clustering to optimally reorder the rows of matrix A, aligning data access patterns with matrix B to maximize reuse and reduce off-chip memory traffic during row-wise matrix multiplication. Our key insight lies in using a similarity matrix that captures the structural properties of matrix A to guide clustering, along with an optimized implementation of the spectral clustering algorithm to reduce preprocessing overhead. Additionally, Bootes incorporates a decision tree model trained on real-world matrices to predict when reordering is beneficial, enabling a cost-aware preprocessing strategy. Bootes achieves a geometric mean speedup of 11.61× in preprocessing time compared to existing row reordering techniques while maintaining scalability. It is deployed on state-of-the-art accelerators—Flexagon, GAMMA, and Trapezoid—where it reduces off-chip traffic by 2.31×, 1.67×, and 1.38×, respectively, thereby boosting their efficiency.

Keywords

Sparse Matrix-Matrix Multiplication, Matrix Reordering, Sparse Accelerators

ACM Reference Format:
Sanjali Yadav and Bahar Asgari. 2025. Bootes: Boosting the Efficiency of Sparse Accelerators Using Spectral Clustering. In *58th IEEE/ACM International Symposium on Microarchitecture (MICRO '25), October 18–22, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3725843.3756125

1 Introduction

Sparse General Matrix-Matrix Multiplication (SpGEMM) is a fundamental kernel across domains such as scientific computing, graph analytics, and deep learning, where sparsity is prevalent. Since sparsity causes performance degradation when running on traditional general-purpose computing systems that are often optimized for dense data, numerous studies have been exploring various techniques to enhance the efficiency of sparse computations [8–10, 12, 13, 16, 18, 25, 35–40, 42, 43, 46, 47, 52, 58–60, 65, 67, 68, 72, 74]. Sparse accelerators have explored a range of techniques such as re-purposing traditional architectures to handle sparsity [19, 20, 45, 63], using hardware/software co-designs with a focus on compression formats [10, 29, 41, 58, 59, 62, 78], or proposing techniques to target a certain category of applications such as sparsity in deep learning applications [8, 18, 25, 31, 35, 36, 40, 42, 67, 74].

More specifically, prior SpGEMM [50, 61, 77] accelerators employed different dataflows for kernel implementation. The most common dataflows are inner product, outer product, and row-wise product [17], each offering distinct trade-offs. For instance, the inner product provides good output reuse but poor input reuse and is inefficient for highly sparse matrices due to the overhead of intersecting indices of ineffective elements. In contrast, the outer product enables efficient input reuse but results in high memory traffic from large partial output matrices. The row-wise product strikes a balance between these approaches, offering reduced memory traffic with smaller partial products and eliminating the need for index matching. This makes the row-wise product particularly effective for highly sparse inputs, and it has emerged as the preferred dataflow in many state-of-the-art accelerators [32, 61, 77] as a result of its favorable trade-offs.

While the row-wise product minimizes memory traffic by balancing the extremes of inner and outer products, it still incurs significant data movement and memory traffic due to irregular memory access patterns, impacting the energy efficiency of novel accelerators. To address this, prior studies [4, 27, 77] have introduced a preprocessing step for row-wise product that reorders the rows of one input matrix to enhance data reuse and reduce memory traffic of the other input matrix. However, row reordering presents challenges. Although optimal reordering can drastically reduce memory traffic, it introduces substantial overhead in terms of preprocessing time and memory requirements on the host. Furthermore, existing approaches face three key limitations: (1) the reordering is often suboptimal, failing to maximize the potential reduction in memory traffic, (2) the preprocessing steps are typically slow and inefficient, making the cost of reordering unjustifiable, even when performed as a one-time operation, and (3) certain sparsity patterns do not benefit from reordering or may even experience increased memory traffic when reordered. Existing studies lack a mechanism to identify such scenarios, where the optimal strategy would be to avoid reordering altogether.

To address these challenges, we propose boosting the efficiency of sparse accelerators, through Bootes[1], which reduces the off-chip traffic in Flexagon [46], GAMMA [77], and Trapezoid [73] by a geomean factor of 2.31×, 1.67×, 1.38×, respectively while accelerating the preprocessing method by a geomean factor of 11.61×. Our key insight lies in leveraging spectral clustering with a similarity matrix that encodes the structural properties of the input matrix A. This enables optimal row reordering of matrix A, aligning access patterns to matrix B to maximize data reuse and minimize memory traffic. We optimize the implementation of the spectral clustering algorithm to reduce the overhead of row reordering, making it more efficient than existing solutions. Bootes also incorporates a decision tree model that evaluates the sparsity pattern of the input matrix before reordering to predict whether the preprocessing is worthwhile.

To showcase how Bootes enhances the efficiency of state-of-the-art sparse accelerators, we integrate it with Flexagon, GAMMA, and Trapezoid [73], all of which utilize the row-wise product for their SpGEMM implementations. We evaluate performance across a diverse set of matrices to demonstrate Bootes' impact. Additionally, our scalability studies show that Bootes maintains robust efficiency in both preprocessing time and memory footprint, outperforming prior approaches as the matrix size and density vary.
In summary, we make the following contributions:

- We analyze the row-wise product, which is recognized for generating the least memory traffic among common dataflows, and identify further opportunities for memory traffic reduction. We demonstrate that row reordering is an effective strategy for minimizing memory traffic, and conduct an in-depth analysis to show why existing row reordering techniques fail to fully capitalize on this potential.
- We introduce Bootes, a novel row reordering technique that leverages spectral clustering to minimize memory traffic and improve data locality. Moreover, Bootes optimizes preprocessing time and memory footprint, offering significant improvements over existing methods.
- We construct a dataset derived from a wide range of real-world matrices, capturing key features that characterize their sparsity patterns. Using this dataset, we train a decision tree model to conduct a cost-benefit analysis, predicting whether row reordering will yield sufficient performance gains to justify the associated overheads.
- We integrate Bootes with state-of-the-art accelerators to empirically demonstrate its efficacy in reducing memory traffic and data movement. We also evaluate Bootes across a broad spectrum of matrices with diverse sparsity patterns, showcasing its performance improvement over existing techniques, both in terms of memory traffic reduction and preprocessing efficiency.

2 Background and Challenges

This section first reviews dataflows for SpGEMM, focusing on the advantages of row-wise product. It then discusses the limitations of prior solutions for improving row-wise-product-based accelerators, highlighting gaps that Bootes addresses with further optimizations.

2.1 Dataflows and the Favorable One Used in Recent Sparse Accelerators: Row-wise Product

Three common dataflows exist for sparse matrix-matrix multiplication: inner product, outer product, and row-wise product, as summarized in Table 1. To compute $A_{M\times K} \times B_{K\times N} = C_{M\times N}$, all approaches use a triply-nested loop over dimensions M, N, and K. Their key difference lies in the placement of the K-loop: innermost in inner product, outermost in outer product, and middle in row-wise product. Inner product multiplies a row of A with a column of B, leading to efficient reuse of C but high over-fetching from B due to repeated accesses. Outer product pairs columns of A with rows of B, optimizing input reuse but often requiring large partial sums to be offloaded to off-chip memory, increasing data movement. Row-wise product multiplies the nonzero elements of a row in A with the corresponding elements in B, balancing the challenges of IP and OP. It avoids index matching, generates fewer unnecessary outputs, and efficiently reuses C. However, irregular access patterns to B caused by the distribution of nonzero elements in A limit its efficiency. As a result of these favorable trade-offs, the row-wise product is used in many state-of-the-art accelerator designs.

2.2 Drawbacks of Row-wise Product and Prior Solutions: Tiling and Row Reordering

While among the dataflows, the row-wise product offers the most favorable trade-offs, its primary drawback lies in poor input reuse for matrix B. Specifically, effective reuse of matrix B's rows occurs when multiple nonzero elements in matrix A share the same column coordinates and are located in adjacent rows. Unfortunately, this scenario is uncommon for two main reasons: ***(1) High row density:*** If a single row in matrix A contains many nonzeros, it will require accessing multiple rows of B. This leads to frequent cache thrashing and increased data movement; ***(2) Lack of structure in sparse matrices:*** Adjacent rows in matrix A often contain largely disjoint column coordinates, minimizing opportunities for reusing the same rows of B.

Previous work has addressed the challenge of high row density by proposing tiling strategies for the input matrix. ASpT [22] introduced an adaptive-tiling technique to improve data locality by dividing the sparse matrix into row panels. Within each panel, columns are reordered to separate densely populated columns from sparse ones. For densely populated columns, traditional tiling is

Table 1: Summary of how the dataflow selection impact various design aspects."✗" indicates a potential challenge and "✓" indicates no issue.

Aspect ↓ \ Dataflow →	Inner	Outer	Row-wise
Psum Granularity	✓	✗	✓
Input Format	✗	✗	✓
Index Intersection	✗	✓	✓
Input Reuse (B)	✗	✓	✗
Output Reuse (C)	✓	✗	✓

[1]**Bootes** is a constellation in the northern sky.

applied: dividing rows with many nonzeros into smaller sub-rows by tiling them along the column dimension. Instead of processing a dense row all at once, it now interleaves with other rows in the panel, encouraging shared access to matrix B and minimizing unnecessary cache reloads. Similarly, TileSpGEMM [48] employs a tiling approach to enhance data locality. However, its method differs by dividing the input matrix into fixed 16×16 sub-blocks, followed by block matrix multiplication. With a consistent tile size of 16×16, each block contains only a small number of zeros. This prevents a single row from becoming excessively long, helping to avoid cache thrashing and ensuring more efficient data reuse.

The second challenge – dealing with the lack of structure in sparse matrices – is more complex, as real-world sparse matrices often exhibit little to no structure. Previous work has attempted to address this through row reordering, which brings rows with similar column coordinates closer together [4, 27, 77]. They have shown that when the sparsity pattern allows for effective reordering, significant gains can be achieved by reducing memory traffic and data movement—making the preprocessing overhead worthwhile. Figure 1 depicts the sparsity pattern of the *invextr1_new* matrix from the SuiteSparse collection [7], revealing a notable pattern. We marked repeated column coordinate patterns across distant rows. Such repetitions indicate that these rows access the same rows of matrix B. Despite the visual proximity in the figure, the matrix's actual size of 30k by 30k means these rows are significantly separated. Therefore, by the time similar column coordinate patterns recur, the corresponding rows of B may no longer reside in the cache, necessitating a read from off-chip memory. This issue becomes more pronounced in larger matrices with higher density.

Consequently, reordering these rows to align their column coordinate patterns has emerged as a popular solution. However, reordering these rows requires analyzing the entire matrix to identify inter-row patterns, a process that is both computationally demanding and memory-intensive. The matrix A must be read into memory on the host, reordered, and then written back, adding significant preprocessing overhead. Given these resource demands, the benefits of reordering must justify the cost to ensure it is a worthwhile optimization. Prior studies [4, 27, 77] incorporate various optimizations to minimize preprocessing time and memory usage, ensuring that reordering remains a practical and efficient solution. However, after analyzing their methods in the following sub-sections, we show that there is still room for significant improvement. Motivated by these studies, Bootes leverages reordering and introduces an efficient implementation that offers a practical and effective solution to this challenge (more details in Section 3).

2.2.1 Gamma Row Reordering. Algorithm 1 outlines GAMMA's [77] row reordering algorithm, which improves matrix operations by enhancing cache efficiency. This greedy algorithm uses a priority queue Q to reorder rows based on similarity. All input matrix rows are initially added to the queue with zero priority. The algorithm starts by selecting a random row, adding it to P, the array of the final row permutation. At each step, it inspects the latest row in P and identifies columns with nonzero values. For each such column, it increases the priority of other rows not yet in P that also have nonzero values in that column. This favors rows with similar access patterns to matrix B, grouping them closer together. As detailed

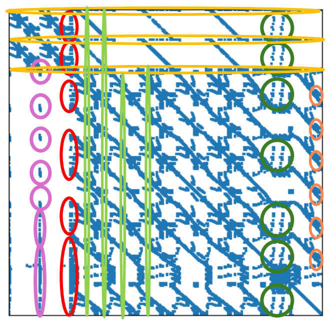
Figure 1: Opportunity for Reordering – The annotations indicate opportunities to reorder the rows of matrix A (nonzeroes illustrated by blue dots) to increase the similarity among their column coordinates, ensuring that rows with similar patterns are positioned closely together to optimize memory access for matrix B.

Algorithm 1 Gamma [77]

```
1: Input: Rows of matrix A, $M$ number of rows
2: Output: Permutation $P$ of row indices
3: for $r \in$ rows do
4:     $Q$.insert$(r, 0)$
5: end for
6: Select some row $r$ to start, $P[0] \leftarrow r$, $Q$.remove$(r)$
7: for $i \in [1, M]$ do
8:     for $u \in$ column coords of row $P[i-1]$ do
9:         for $r \in$ row coords of column $u$ do
10:            if $r \in Q$ then $Q$.incKey$(r)$
11:        end for
12:    end for
13:    if $i > W$ then
14:        for $u \in$ column coords of row $P[i-W-1]$ do
15:            for $r \in$ row coords of column $u$ do
16:                if $r \in Q$ then $Q$.decKey$(r)$
17:            end for
18:        end for
19:    end if
20:    $P[i] \leftarrow Q$.pop()
21: end for
```

in Table 2, the algorithm's complexity is dominated by Q^2, as each of the N rows may update priorities for up to Q entries, with each priority queue operation taking $O(logN)$. Performance degrades with matrix density due to increased priority updates.

A key feature is the window size W, defining how many rows fit in cache. As the algorithm progresses to a new window, it decreases the priority of rows similar to those from earlier windows, specifically those i-W-1 strides back, assuming their associated data has been evicted from cache. This biases the algorithm toward rows with patterns resembling those at the end of the previous window, under the assumption that their data is still cache-resident. This design aligns reordering with cache reuse, aiming to reduce cache misses and off-chip memory traffic.

A primary limitation is GAMMA's greedy nature. Starting from a random row can lead to suboptimal results, as early choices heavily

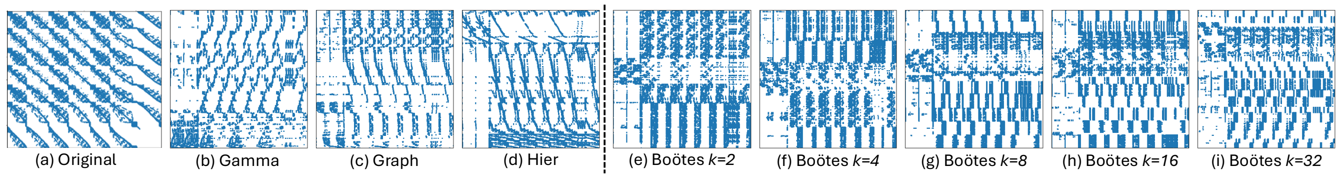

Figure 2: Visualized row reordering – Figures (b-d) display the results of applying Gamma, Graph, and Hier row reordering to Figure (a). Figures (e-i) illustrate the results of applying Bootes reordering to the matrix in Figure (a) for different k-values.

influence subsequent ones. While the window size W aims to balance local and global reordering, it imposes structural constraints. Within each window, the algorithm groups rows with similar column patterns to enhance local reuse and attempts to extend recent patterns for continuity. However, real-world matrices often have irregular sparsity, making pattern continuity across windows difficult. When new window patterns diverge from previous ones, the algorithm fails to align rows effectively, reducing reordering benefits. This limitation is visualized in Figure 2b, which shows GAMMA's result applied to figure 2a. In contrast, the ideal outcome in figure 2e would show vertically aligned columns, reflecting optimized access to matrix B.

Algorithm 2 Graph [4]

```
1: Input: Matrix A, $M$ number of rows
2: Output: Permutations $P$ of row indices
3: initialize $G(V, E)$ with $r$ vertices and no edges
4: initialize hash table visited[][] for $r$ vertices
5: for $u \in [1, M]$ do
6:     for $c \in$ column coords of row $u$ do
7:         for $v \in$ row coords of column $c$ and $u \neq v$ do
8:             if $(u, v) \notin E$ then
9:                 $E \leftarrow E \cup (u, v)$
10:                $w(u, v) \leftarrow 0$
11:            end if
12:            $w(u, v) \leftarrow w(u, v) + 1$
13:        end for
14:    end for
15: end for
16: $row\_idx \leftarrow$ index of a random row in A
17: $visited[row\_idx] \leftarrow 1$
18: $P[0] \leftarrow row\_idx$
19: for $i \leftarrow 1$ to $r - 1$ do
20:     $row\_idx \leftarrow$ maxPath$(P[i-1], visited)$
21:     $visited[row\_idx] \leftarrow 1$
22:     $P[i] \leftarrow row\_idx$
23: end for
```

2.2.2 *Graph Row Reordering.*

Algorithm 2 describes the graph-based row reordering algorithm proposed in prior work for an FPGA-based SpGEMM accelerator [4]. This method uses a graph-based approach to capture the similarity between matrix rows by constructing a weighted graph $G(V, E)$, where each vertex corresponds to a row, and edge weights represent the number of shared column coordinates between two rows.

Graph construction closely mirrors GAMMA's reordering logic. For each row in matrix A, the algorithm iterates over its column coordinates. For every coordinate, it identifies other rows containing that same value and creates edges (u, v) between the corresponding vertices. The weight $w(u, v)$ is incremented for each shared coordinate. As the sparsity of the matrix A increases, the resulting graph becomes proportionally sparser, which improves both storage efficiency and traversal speed. As shown in Table 2, the time complexity is dominated by the q^2 term. For each of the r non-empty rows, up to q nonzero elements may lead to connections with up to q other rows, resulting in q^2 edge operations per row. The graph construction phase dominates the complexity, as each nonzero element can potentially connect to q other elements in the same column. The r term reflects the reduced problem size, as the algorithm only processes non-empty rows.

Once the graph is constructed, the algorithm traverses it to determine the row permutation. A visited table tracks which rows have already been added to the permutation list. The process starts by randomly selecting a row from A, then repeatedly selects the next unvisited row that shares the highest weighted edge with the most recently added row. This is done using the following function:

$$\text{maxPath}(u, \text{visited}) = \arg\max_{v \in V} w(u, v) \quad \text{s.t.} \quad (u, v) \in E \text{ and visited}[v] \neq 1. \tag{1}$$

This greedy strategy prioritizes local similarity by choosing the most similar unvisited neighbor at each step. Unlike GAMMA, which attempts to preserve global structure through windowing, the graph-based method emphasizes strong local connections. While this can be advantageous for certain sparsity patterns, it may still yield suboptimal reordering for more irregular matrices. This tradeoff is evident in Figure 2c, showing improved vertical alignment and reduced fragmentation compared to GAMMA, but still retains some of the original disjoint patterns, indicating incomplete optimization.

2.2.3 *Hierarchical Cluster Row Reordering.*

Algorithm 3 summarizes the hierarchical clustering-based row reordering algorithm in recent work [27]. It uses locality-sensitive hashing (LSH) [34] to group rows into buckets, ensuring that rows in the same bucket share similar column coordinates, while rows in different buckets are less likely to be similar. LSH serves as an efficient approximation to a full similarity matrix, avoiding exhaustive pairwise comparisons. It uses fixed parameters, siglen and bsize, which remain constant across all matrices. The algorithm calculates similarity using the Jaccard score, defined as the ratio of shared column coordinates to the total number of unique coordinates between two rows.

As shown in Table 2, the algorithm's time complexity consists of three terms. The $(N + E)logE$ term arises from priority queue

Table 2: Time Complexity Analysis of Row Reordering Algorithms

Term Definitions	Algorithm	Time Complexity	Dominant Factor	Scalability
N – number of rows; Q – avg nonzeros per row; r – nonempty rows; q – avg nonzeros per row/col; E – candidate pairs from LSH; k – clusters & eigenvectors; d_j – nnz in column j of A; g – $nnz(S)/N$, $S = AA^T$; t – Krylov iterations	Gamma [77]	$O(N \log N \cdot Q^2)$	Q^2 density squared	Poor with density
	Graph [4]	$O(r \times q^2)$	q^2 density squared	Good (only nonempty rows)
	Hier [27]	$O(E \log N + (N + E) \log E + N)$	$ElogE$ candidate pairs	Moderate
	Bootes	$O\Big(\underbrace{\sum_j d_j^2}_{AA^T} + \underbrace{Ng}_{\text{Laplacian}} + \underbrace{Ngkt}_{\text{Eigensolve}} + \underbrace{Nk^2}_{\text{Kmeans}}\Big)$	Nk^2 or $Ngkt$ linear in matrix size	Excellent

Algorithm 3 Hierarchical Cluster (Hier) [27]

```
Input: Matrix $A$, $bsize$, $siglen$, $threshold\_size$
Output: Permutations $P$ of row indices
Initialize $candidate\_pairs$ using LSH($S$, $siglen$, $bsize$)
Use a max-heap $sim\_queue$ for $candidate\_pairs$
Define arrays: $cluster\_id$ (track root node), $cluster\_sz$ (size of
each cluster, init 1s), $deleted$ (track deleted nodes, init false)
procedure ROOT($i$)                ▷ Find representative row in cluster
    while $i \neq cluster\_id[i]$ do
        $i \leftarrow cluster\_id[cluster\_id[i]]$
    end while
    return $i$
end procedure
while not $sim\_queue$.empty() and $nclusters > 0$ do
    $(i, j) \leftarrow sim\_queue$.pop()        ▷ Pair with high similarity
    if $i$ and $j$ are the representing rows then
        merge smaller cluster into larger one
        if $cluster\_sz > threshold\_size$ then
            delete the cluster and nclusters -= 1
        end if
    else
        i=root(i); j=root(j);
        if $i$, $j$ are not in the same cluster then
            compute $(i, j)$ jaccard similarity
            add similarity to the queue
        end if
    end if
end while
Group rows into clusters, update $P$ with cluster values
```

operations, where N initial rows and E candidate pairs are managed in a max-heap, with each insertion or extraction taking $O(logE)$ time. The $ElogN$ term accounts for the LSH phase, where E candidate pairs are generated, and each hashing operation requires $O(logN)$ time. The final $O(N)$ term corresponds to the linear-time processing required to organize and output the final clusters.

The algorithm proceeds by generating a list of highly similar candidate row pairs using LSH and inserting them into a priority queue. Initially, each row gets treated as its own cluster. Iteratively, the algorithm extracts the most similar row pair (i, j) and merges their respective clusters. Each cluster is implemented as a tree, allowing efficient merging by appending one tree as a child of another. A representative row is chosen for each cluster to guide future merges. If a cluster contains only one row, that row serves as the representative. When two clusters merge, the representative of the larger cluster is used; if both are equal in size, the row with the smaller index is selected.

While implementation details are available in the original paper[27], the core idea approximates a similarity matrix using LSH and iteratively merges clusters based on similarity until a predefined size limit is reached. However, the use of a fixed $threshold\_size$ to constrain cluster size introduces a limitation. It does not adapt to varying sparsity patterns because highly similar rows may be split across different clusters, reducing the effectiveness of row grouping. Moreover, the method does not address global alignment across clusters—once clusters are formed, there is no reordering step to ensure structural coherence between them.

Another challenge is the strategy for choosing the representative row. Using the row with the smallest index can introduce bias. If that row is dense, it may align with many others, leading to oversized clusters. If it is sparse, it may result in underdeveloped clusters, reducing the quality of reordering. Figure 2d highlights many of the original patterns are still visible after applying this technique, indicating lack of optimal row merging and presence of disjoint groupings.

2.3 Summary of Our Targeted Challenges

The three algorithms discussed in Section 2 address the challenge of reordering the rows of a sparse input matrix A to mitigate the lack of structure inherent in sparse matrices. This structural deficiency causes adjacent rows in matrix A to have largely disjoint column coordinates, reducing the potential to reuse the same rows of matrix B during computations. While each algorithm takes a different approach, they all leverage the matrix sparsity pattern to reorder rows such that adjacent rows exhibit similar column coordinates.

Although these row reordering techniques improve the efficiency of the target accelerators, they introduce significant overheads in terms of preprocessing time (i.e., time complexity) and memory usage (i.e., storage complexity), particularly when transitioning from highly to moderately sparse inputs. Another issue, overlooked by most of the papers except [27], is that reordering can sometimes result in performance degradation. In certain scenarios, greedy heuristics may yield suboptimal configurations, and in other cases, the sparsity pattern may not allow any reordering to outperform the original row order. Therefore, it is essential to develop a mechanism for identifying such cases. This would ensure that resources are not wasted on reordering matrices where it offers no performance benefit and that the reordering process itself is not implemented sub-optimally, thereby preserving computational efficiency.

3 Bootes

Bootes introduces a novel approach to row reordering for matrices with highly diverse sparsity patterns. First, as Section 3.1.1 explains, Bootes showcases that with strategic optimizations, spectral clustering can be effectively applied to this domain, significantly reducing memory traffic, preprocessing time, and overall memory footprint compared to existing algorithms. A unique feature of Bootes is its integration of a decision tree model that performs a cost-benefit analysis to assess whether reordering will yield performance gains (detail in Section 3.2). If beneficial, the model dynamically selects optimal parameters tailored to the matrix-specific sparsity pattern to maximize memory efficiency. Our rationale for choosing decision trees is based on a balance between accuracy and storage efficiency. Although we experimented with random forests, XGBoost, and SVMs – with XGBoost achieving the highest accuracy – it required considerably more storage. Decision trees, while offering similar levels of accuracy, present a lightweight solution that better meets our objective for efficient storage in potential online deployments. Designed to enhance, not replace, existing hardware, Bootes operates atop state-of-the-art accelerators, boosting their performance without the need for specialized hardware.

3.1 Spectral Clustering

Spectral clustering is a machine learning technique that identifies clusters by analyzing the eigenvalues and eigenvectors of a similarity matrix. It is particularly effective for non-linearly separable or irregularly shaped data, with applications in image segmentation, gene clustering, and document organization. Unlike K-means, which relies on linear boundaries, spectral clustering captures complex patterns by embedding data into a new space via eigenvector decomposition. Compared to hierarchical clustering, which builds nested clusters through iterative merging, spectral clustering is more flexible. While hierarchical methods perform well on compact, evenly shaped clusters, they struggle with irregular structures, often found in sparse matrices. Spectral clustering, by contrast, captures global data structure, accommodating non-linear patterns and varying cluster sizes. In matrix reordering, spectral clustering offers a key advantage by leveraging global patterns rather than local similarities, making it especially effective for matrices with diverse or long-range dependencies where traditional methods may fall short.

3.1.1 Using Spectral Clustering for Row Reordering. Algorithm 4 outlines the spectral clustering algorithm, including four key steps to compute clusters of matrix rows:

Construct Similarity Matrix (line 12): Given a matrix A in Compressed Sparse Row (CSR) format, we calculate the similarity matrix by computing the dot product of the matrix with its transpose. Since the matrix is in binary format (with nonzeros replaced by 1s), the similarity matrix effectively captures the number of shared column coordinates between two rows. Each entry in the similarity matrix measures how many nonzero elements (columns) two rows have in common, which serves as a measure of row-wise similarity.

Algorithm 4 Spectral Clustering Algorithm in Bootes

```
1:  **Input:** Matrix $A$, $k$
2:  **Output:** Permutations $P$ of row indices
3:  **procedure** ComputeLaplacian(similarity_matrix)
4:      $degrees \leftarrow \text{Array}(similarity\_matrix.sum).flatten()$
5:      $inv\_sqrt\_degrees \leftarrow 1.0/\sqrt{degrees}$
6:      $D\_inv\_sqrt \leftarrow \text{csr\_matrix}(inv\_sqrt\_degrees)$
7:      $identity\_matrix \leftarrow \text{I}(similarity\_matrix.nrows)$
8:      $laplacian \leftarrow identity\_matrix - D\_inv\_sqrt$
            $@\ similarity\_matrix\ @\ D\_inv\_sqrt$
9:  **end procedure**
10: **return** $laplacian$
11: $A.data \leftarrow 1$
12: $similarity\_matrix \leftarrow A \cdot A^T$
13: $laplacian \leftarrow \text{ComputeLaplacian}(similarity\_matrix)$
14: $v0 \leftarrow \text{Array}(laplacian.nrows)$
15: $eigenvectors \leftarrow \text{scipy.sparse.linalg.eigsh}(laplacian, k, v0)$
16: $kmeans \leftarrow \text{sklearn.cluster.KMeans}(n\_clusters = k)$
17: $P \leftarrow kmeans.fit\_predict(eigenvectors)$
```

Compute Laplacian Matrix (line 13): The Laplacian matrix is a transformation of the similarity matrix that captures both the local and global structure of the data, making it easier to detect clusters. The normalized Laplacian matrix is defined as:

$$L = I - D^{-1/2} \cdot A \cdot D^{-1/2}$$

where I is the identity matrix, A is the similarity matrix, and D is the degree matrix, where each diagonal entry D_{ii} is the sum of the similarities for row i.

Intuitively, the Laplacian captures how each node differs from its neighbors by combining information from the similarity and degree matrices. This makes it well-suited for identifying clusters—groups of nodes more connected to each other than to the rest of the graph [66]. Its eigenvectors reveal these groupings, as nodes within a cluster tend to share similar eigenvector values, reflecting strong internal similarity and weak external links. Effectively, the Laplacian acts like a discrete second derivative, highlighting deviations from local averages and exposing the graph's underlying cluster structure [44].

Compared to simpler similarity metrics like Jaccard or commute time, Laplacian-based methods are more expressive, capturing both local affinities and global connectivity. They adapt naturally to variations in graph density and support multi-scale, hierarchical clustering by tuning spectral components, making them particularly robust for detecting complex or irregularly shaped clusters.

Calculate Eigenvectors of the Laplacian Matrix (line 15): This step involves extracting the top-k eigenvectors of the Laplacian matrix, corresponding to the smallest nonzero eigenvalues. These eigenvectors form the spectral embedding of the data, representing each row in a lower-dimensional space where clustering becomes easier. The top-k eigenvectors capture the most meaningful cluster structure by minimizing noise and focusing on stable relationships across rows. In this spectral space, the rows of the original matrix are better separated, even if they were non-linearly separable in the original space. This step helps prevent the clustering algorithm from being trapped in local minima, which is a common issue in traditional algorithms like K-means.

Figure 2(d-h) illustrates how the selection of k values affects the spectral clustering reordering algorithm. In this example, k = 2 is the optimal configuration, as it successfully reorders the rows to align their column coordinate patterns. The top 2 eigenvectors were sufficient to capture the structure of the input matrix. The pattern and size of the input matrix influence the optimal k value, and we will discuss how to select the value in section 3.2.

Perform K-means Clustering on Eigenvectors (line 16): Once the rows are transformed into the spectral embedding space, we apply K-means clustering to group similar rows based on their new coordinates. In this transformed space, each row corresponds to a data point, and each column of the eigenvector matrix represents an eigenvector that captures key structural relationships within the original data. These eigenvectors act as new features, with the top k eigenvectors providing a low-dimensional space where clusters are more separable. If two rows have very similar values across all eigenvectors, they are likely close in the original data and may belong to the same cluster. K-means leverages these coordinates to partition the data, ensuring that even rows with complex, non-linear patterns in the original space are grouped into coherent clusters in the spectral space.

The time complexity of Bootes, detailed in Table 2, consists of four primary stages: similarity matrix construction (O($\sum_j d_j^2$)), Laplacian generation (O(Ng)), iterative eigen solver (O($Ngkt$)), and k-means clustering (O(Nk^2)). The runtime is dominated by the trade-off between the eigensolver and k-means, as the initial matrix construction is not a bottleneck for our target sparse workloads. Since the number of clusters k is a small, fixed constant, the O(Nk^2) cost of k-means scales linearly with the number of rows N. The eigensolver's cost, O($Ngkt$), also exhibits linear scaling with N because the average row density of the similarity matrix (g) and the number of Krylov iterations (t) are typically small and well-behaved for sparse graphs. The bottleneck shifts from k-means on extremely sparse graphs to the eigensolver as graph density increases, but the scalability remains linear in matrix size. We conduct empirical validation in Section 5.3.

3.1.2 Optimizations. Spectral clustering aligns well with our problem because of the diverse, sparse patterns inherent in real-world matrices, which often violate the assumptions of traditional clustering algorithms—such as linear separability and predefined cluster shapes. However, spectral clustering introduces additional computational overhead. Recent advancements in the literature offer optimized algorithms that address these challenges, allowing us to implement spectral clustering efficiently and at scale. By leveraging these implementations, we achieved superior preprocessing performance compared to previous approaches, especially as matrix density and size increase (more in Section 5.3).

Our key optimization stems from the sparsity of matrix A, which ensures that the associated similarity matrix and Laplacian matrix also remain sparse. This translates into a reduced memory footprint since these matrices are maintained in CSR (Compressed Sparse Row) format throughout the computation. To compute pairwise similarity, we directly calculate the dot product between binary matrices in sparse format. Although LSH in hierarchical clustering reduces computational overhead, it sacrifices accuracy in similarity calculations, which is evident from figure 2c. Increasing LSH parameters such as siglen and b improves accuracy but introduces additional computational complexity. In contrast, graph and gamma reordering algorithms avoid storing the full similarity matrix explicitly as an optimization. However, they incur their own overhead by requiring tracking of row-column relationships, specifically keeping track of which rows share the same column coordinates. We provide a detailed discussion of these trade-offs in Section 5.4.

After constructing the similarity matrix, we compute the Laplacian matrix – an operation that is computationally intensive but manageable because of the sparsity maintained throughout. The degrees of the nodes D (i.e., rows of matrix A) are stored as an array, with each element representing the sum of the corresponding row. Another array, inv_sqrt_degrees, stores the inverse square root of the degrees. These values are then used to construct a diagonal matrix in CSR format. The identity matrix I is also represented in sparse format, minimizing memory usage throughout. With these components, we compute the Laplacian matrix using sparse arithmetic operations provided by highly optimized Python libraries. Once the Laplacian matrix is available, we extract eigenvectors using efficient sparse matrix eigensolvers. Our key optimization stems from the constraints on the parameter k, which controls both the number of eigenvectors to extract and the number of clusters for the k-means step. Keeping k small helps mitigate computational overhead. We experimented with various k-values across 500 matrices from the SuiteSparse [7] and SNAP [33] collections and identified that the values 2, 4, 8, 16, and 32 offer the most optimal trade-off. These values effectively limit problem size and computational overhead while achieving significant performance improvements.

3.2 Decision Tree

A critical component of Bootes is the decision tree model, which determines both whether matrix reordering should be applied and, if so, selects the optimal k-value for spectral clustering. Since spectral clustering is highly sensitive to parameter tuning, particularly to the choice of k, it is essential to develop a systematic approach to parameter selection that generalizes across matrix structures.

We began by constructing a decision tree using a broad set of candidate features derived from sparse matrices. To support this, we curated a dataset of over 500 matrices from the SuiteSparse and SNAP collections, with a diverse range of matrix sizes, structures, and sparsity patterns. After training the initial model, we analyzed feature importance and pruned the tree to retain only the most influential features. This process helped identify a compact yet expressive feature set.

The final model uses the following features: global sparsity (i.e., the ratio of nonzero to total elements); the variance of nonzeros per row and per column, which captures the uniformity or skewness in the distribution of sparsity; and intersection metrics that reflect structural overlap. Specifically, the intersection average quantifies the degree of shared nonzero positions between rows, while the variation in intersection indicates whether such overlap follows a consistent pattern or varies widely across the matrix. Together, these features serve as structural fingerprints that encode symmetry, clustering tendencies, and locality.

To generate labels for training, we performed spectral clustering on each matrix across a range of k-values, collecting performance metrics such as memory traffic and execution time. The k that yielded the best trade-off between these metrics was recorded as the optimal choice. The decision tree was trained using matrix features as inputs and these optimal k-values as targets. Additionally, the model predicts whether reordering should be applied, based on whether the expected memory traffic reduction exceeds a threshold (currently set to 10%). This threshold can be adjusted to suit specific hardware constraints or application requirements.

Our decision tree model is trained on data from three distinct accelerators, Flexagon, GAMMA, and Trapezoid, each with unique hardware characteristics, including different PE counts, cache sizes, and memory bandwidths (detailed in Section 4). The model predicts when reordering will be beneficial and selects the optimal cluster size for each architecture. By capturing these hardware-specific traits, the decision tree effectively acts as a black-box predictor, identifying both the applicability and the best configuration for data reordering to maximize performance.

The training process for our model requires only a few minutes, utilizing our dataset. To apply this method to a new accelerator, one simply needs to execute relevant workloads on the accelerator and collect performance metrics, typically execution latency and memory traffic. The decision tree model can then be trained on these collected metrics to automatically infer the underlying hardware characteristics of the accelerator and subsequently recommend optimal cluster sizes.

Bootes Workflow Summary: Before SpGEMM execution, Bootes applies a lightweight preprocessing step to determine if matrix reordering can reduce memory traffic and data movement. It extracts structural features, such as density and nonzero distribution, and feeds them into a decision tree model. The model predicts whether reordering will be beneficial and selects the optimal k-value for spectral clustering if needed. If reordering is advised, the matrix is reordered and sent for computation; otherwise, it is processed in its original form. With minimal inference overhead, Bootes ensures efficient, data-driven decisions for boosting the efficiency of matrix multiplication kernels.

4 Methodology

Baselines for Comparisons and Targeted Accelerators: We evaluate Bootes in comparison with three state-of-the-art row reordering algorithms including Gamma [77], Graph [4], and Hier [27], along with the baseline of no-reordering (Original). To demonstrate the adaptability of Bootes, we integrated it with three state-of-the-art hardware accelerators: GAMMA [77], Flexagon [46], and Trapezoid [73] – to clarify, our experiments include both the row reordering algorithm of Gamma and the hardware accelerator proposed in GAMMA. We evaluate our results on an AMD EPYC 7302 processor, which has 16 cores and 32 threads, 22MB L3 cache, and 128GB of DRAM with a bandwidth of 204.8 GB/s. Our decision tree was also trained on this system.

Simulator and Configurations: We use Trapezoid's simulator [73] to simulate our target accelerators and their components. This includes modeling processing elements (PEs), data distribution, local buffers, global caches, and HBM, as described in their respective papers. The simulator meticulously tracks contention and stalls, providing precise measurements of data movement and memory traffic impacts in different accelerators. Our intention with testing on different accelerators is to explore the relationship between the reduction in memory traffic and underlying hardware, as each accelerator has different cache sizes and number of PEs. We adjust configurations – such as the number of PEs, on-chip SRAM, and HBM main memory – across various baseline scenarios to demonstrate Bootes' robust performance consistency under different hardware configurations. Flexagon features a 1MB cache with 67 PEs, GAMMA includes a 3MB cache with 64 PEs, and Trapezoid boasts a 4MB cache with 128 PEs.

It's important to note that our reordering algorithm is not limited to accelerators with specific cache sizes, PE counts, or memory bandwidth. Its main requirement is the use of a row-wise matrix multiplication approach, common in modern SOTA sparse accelerators, where rows of matrix B are accessed based on the column indices of matrix A. While implementations may vary across architectures, our method remains applicable as long as irregular memory access from sparsity introduces notable overhead. Additionally, our decision tree model functions as a black-box predictor that captures hardware-specific characteristics, enabling accurate selection of optimal cluster sizes for effective data reordering.

Analysis of Algorithm 1 indicates that the Gamma reordering method is significantly affected by variations in cache size. However, Bootes maintains consistent, robust performance across these diverse configurations, distinguishing it from previous implementations. Additionally, we assess Bootes against a baseline where the input matrix A is not reordered.

Table 3: Sparse Matrices

ID	Matrix	Size	Density
ET	EternityII_Etilde	10k × 204k	5.70e-4
PO	poisson3Da	14k × 14k	1.93e-3
IN	invextr1_new	30k × 30k	1.94e-3
MI	mixtank_new	30k × 30k	2.22e-3
CI	cit-HepPh	35k × 35k	3.53e-4
BC	bcircuit	69k × 69k	7.91e-5
CO	copter2	55k × 55k	2.47e-4
NC	ncvxqp5	63k × 63k	1.09e-4
SP	sparsine	50k × 50k	6.20e-4
RA	rajat15	37k × 37k	3.19e-4
K4	k49_norm_10NN	39k × 39k	4.16e-4
E4	e40r0100	17k × 17k	1.85e-3
HE	helm3d01	32k × 32k	4.13e-4
EX	ex3sta1	17k × 17k	2.41e-3
EA	EAT_RS	23k × 23k	6.04e-4
MA	Maragal_6	21k × 10k	2.49e-3
VI	vibrobox	12k × 12k	1.99e-3
MS	msc23052	23k × 23k	2.15e-3
OR	Oregon-1	11k × 11k	3.55e-4
SH	ship_001	35k × 35k	3.20e-3
SM	sme3Da	13k × 13k	5.60e-3
TO	tomographic1	73k × 59k	1.49e-4
OL	olesnik0	88k × 88k	9.55e-5
MR	mri2	63k × 147k	6.10e-05
DU	Dubcova2	65k × 65k	2.44e-04
FO	fome20	33k × 108k	6.35e-05

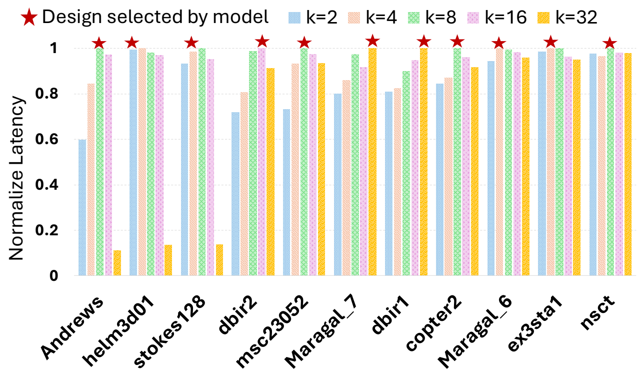

Figure 3: Performance of various workloads for different cluster sizes (normalized to best cluster size per workload).

Workloads: For the experimental evaluation, we use matrices with diverse sparsity patterns from the SuiteSparse collection [7] and SNAP [33] as listed in Table 3. We selected matrices that display unique sparsity patterns and are representative of real-world matrices and graphs. By unique sparsity patterns, we mean matrices representing diverse domains, dimensions, and symmetrical patterns from SuiteSparse. This ensures the decision tree learns a diverse range of patterns to be able to effectively evaluate the need for preprocessing. Finally, in our evaluations, B matrices are identical to A but not reordered. For non-square A, we use B's transpose for multiplication. This methodology aligns with prior studies such as GAMMA and Trapezoid, which we adopted for consistency in evaluation.

Metrics: The primary objective of Bootes is to have a faster and more efficient preprocessing to reduce off-chip memory traffic. To achieve this, in addition to measuring metrics such as the ratio of preprocessing time to actual computation time and the memory footprint of preprocessing, we track memory traffic separately for input matrices A and B, output matrix C, and total traffic. Memory traffic is defined as the number of reads and writes to off-chip memory during SpGEMM execution. Additionally, we record the total compute cycles for SpGEMM processing. For preprocessing, we track the time spent, including reading and writing operations, and use memory profiling tools to monitor the minimum memory allocation needed to avoid out-of-memory errors, which we define as the memory footprint.

5 Results and Analysis

In this section, we present a comprehensive evaluation of our approach by examining several key dimensions. We begin with a decision tree analysis. Next, we investigate memory traffic to highlight how our approach optimizes data movement and its associated overheads. We then explore scalability by assessing both the time and memory footprint of the preprocessing phase, which is critical for handling large and complex datasets. Finally, we quantify the speedup achieved by our method, demonstrating its effectiveness in enhancing overall computational efficiency.

5.1 Decision Tree Analysis

Our dataset is divided into a training set comprising 70% of the data and a validation set comprising the remaining 30%. Many matrices in SuiteSparse and SNAP exhibit a common structural pattern, where most nonzero elements are concentrated along the diagonals and minimal column intersections. As a result, a significant portion of the data points in our dataset are labeled as "no reorder." This inherent class imbalance, where the majority of the data tends towards not requiring row reordering, can introduce bias into our model, skewing predictions towards the majority class. To mitigate this bias, we employed class balancing techniques during the training process of our decision tree model, ensuring that each class (reordering vs. not reordering) is weighted equally to prevent the model from being biased towards the dominant "no reorder" class and ensure it can effectively predict whether reordering is necessary.

Figure 3 presents execution time for matrices in the validation set across different cluster sizes, normalized to the best-performing configuration. The cluster size predicted by our model is indicated with a star. In most cases, the model successfully identifies the optimal configuration; in a few cases (e.g., helm3d01 and stokes128), it selects a suboptimal size, but with minimal slowdowns of only 1.01× and 1.05×, respectively. The figure underscores the impact of cluster size selection—for instance, Andrews sees a 9.08× speedup when using the optimal size over the worst-case. The model achieves 88% accuracy and yields an overall geometric mean speedup of 1.38× across the test set, relative to a baseline without clustering, by first deciding whether to reorder and then selecting the cluster size.

Beyond accuracy and speedup, the figure highlights the complexity of selecting optimal cluster sizes. For example, Maragal6 and Maragal7 share sparsity patterns but differ in size, leading to different optimal choices. While larger matrices might seem to favor larger clusters (e.g., $k = 32$), Andrews, despite being larger than Maragal7, performs best with a smaller size. Similarly, msc23052 and nsct both prefer $k = 4$ despite differing in shape and sparsity. These cases illustrate that cluster size selection depends on a nuanced interplay of sparsity, structure, and hardware behavior, underscoring the need for data-driven models to make robust, generalizable decisions.

Additionally, we chose a decision tree model because of its minimal storage requirement of only 11KB, making it highly efficient for dynamic prediction. This compact size is critical as we aim to minimize the memory footprint on the host during preprocessing. A lightweight model ensures that it does not overburden the on-chip memory, allowing for faster execution without consuming excessive resources.

5.2 Memory Traffic

Figure 4 shows traffic normalized to the compulsory traffic, which refers to the traffic that all designs would incur with unbounded on-chip memory, equivalent to reading the input matrices and writing the output matrix. Each bar in the bar chart is divided into three components: normalized traffic for reading Matrix A (in green), normalized traffic for reading Matrix B (in red), and normalized traffic for writing Matrix C (in blue).

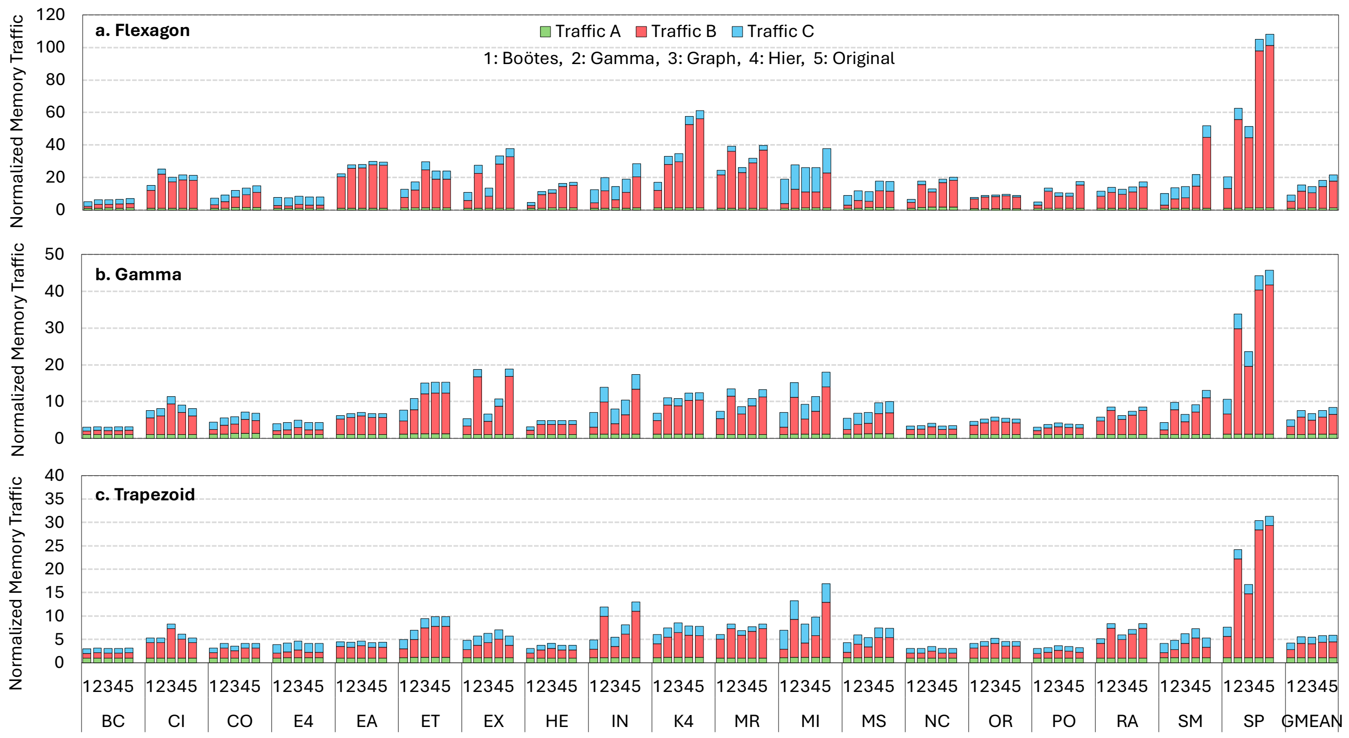

Figure 4: Adaptability Analysis – Memory traffic breakdown when different row reordering methods, including Bootes, Gamma, Graph, and Hier, are applied to three state-of-the-art accelerators, (a) Flexagon [46], (b) Gamma [77], and (c) Trapezoid [73].

Bootes significantly reduces off-chip memory traffic across three accelerators: Flexagon, GAMMA, and Trapezoid. Specifically, Bootes reduces traffic in Flexagon by factors of 1.67×, 1.55×, 1.95×, and 2.31× compared to Gamma, Graph, Hier, and Original, respectively; in GAMMA, the reductions are 1.50×, 1.35×, 1.51×, and 1.67×; and in Trapezoid, the improvements are 1.30×, 1.28×, 1.36×, and 1.38×.

We observe that Bootes demonstrates the highest improvement on the Flexagon accelerator compared to the baselines, which aligns with expectations given Flexagon's smaller cache size. The limited cache space results in more frequent evictions of rows from matrix B, as it cannot hold a significant number of rows simultaneously. In contrast, Trapezoid, with roughly four times the cache size of Flexagon, exhibits relatively smaller improvements. The larger cache enables it to retain more rows of matrix B, allowing even suboptimal reordering techniques to benefit from reduced memory access, as the cache can still store sufficient rows for reuse.

Among the three baselines, Hier performs the least efficiently, likely because of the fixed parameters used in candidate pair generation in LSH. Hier performs well when the parameters are well-suited, as the initial candidate pairs exhibit high similarity, guiding a more optimal clustering process. However, when the parameters are misaligned, its performance deteriorates. Gamma, on the other hand, is constrained by the window size W. In certain cases, the window size aligns well with matrix patterns, leading to effective sorting, while in other cases, it is less optimal. For smaller and highly sparse matrices such as *Oregon-1* (OR), *bcircuit* (BC), and *e40r0100* (E4), the algorithm is more effective because of the smaller window size. As discussed in Gamma's analysis, the global consistency between windows is higher in such cases because the top and bottom rows within each window are more similar, minimizing discrepancies during the next-row selection for subsequent windows. Lastly, Graph follows a greedy search strategy, selecting the row with the highest similarity to the current one and adding it to the final permutation. Like many greedy algorithms, Graph can be hit-or-miss. Some matrix patterns, such as *ex3sta1* (EX), *invertex_new* (IN), and *rajat15* (RA), are well-suited to this approach, leading to optimal reordering. However, in other cases, such as *EternityII_Et* (ET), *Sparsine* (SP), and *citHepPh* (CI), suboptimal reordering increases off-chip memory traffic.

Bootes leverages a spectral clustering algorithm to map matrix rows to a lower dimension to simplify the clustering problem, allowing for more efficient reordering, and ultimately achieves higher performance on these accelerators compared to existing methods. By reducing off-chip memory traffic, Bootes potentially enhances the efficiency of integrating the state-of-the-art accelerators, as fetching data from the off-chip memory and moving data to the compute units consumes significantly (e.g., up to ∼ 4000× to 64000×) more energy than pure computation [5, 6]. As a result, by reducing memory traffic by 2.01×, 2.05×, and 1.69× compared to the original use case of accelerators (without reordering the input matrix), Bootes would potentially improve its energy efficiency.

5.3 Scalability

Figure 5 presents the preprocessing time and memory footprint for Bootes and our baseline methods as we vary matrix density (bubble

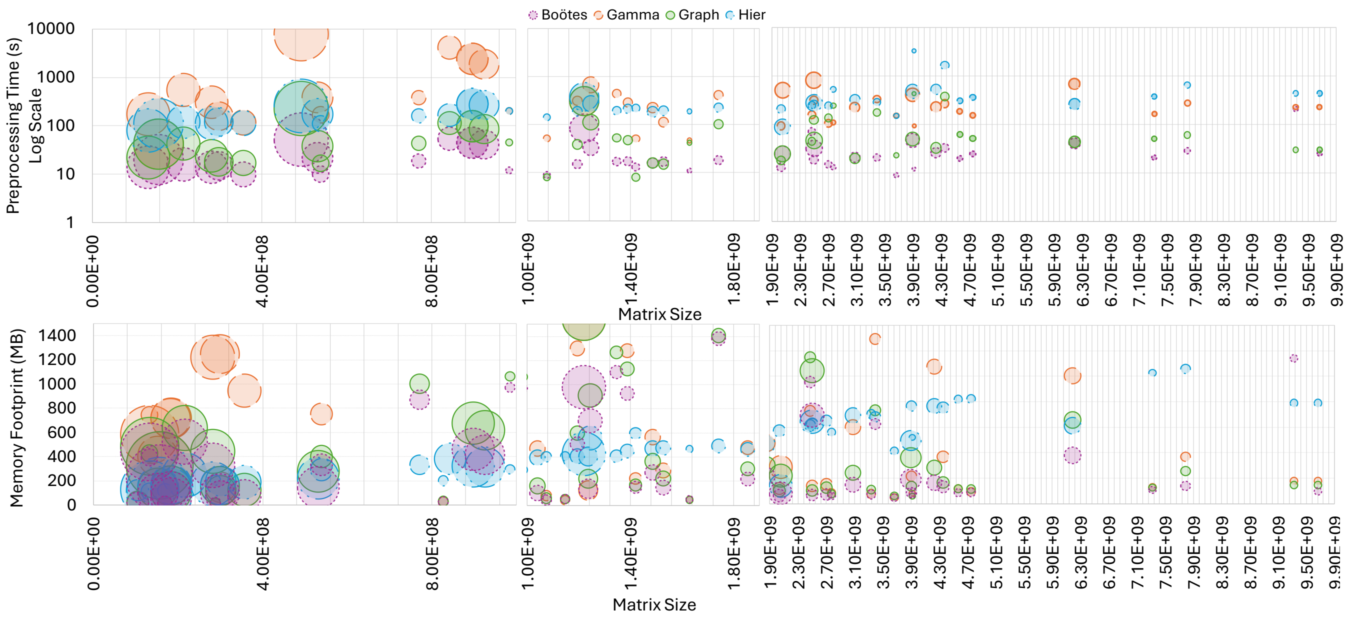

Figure 5: Scalability Analysis – The prepossessing time (top) and memory footprint (bottom) when the matrix size (x-axis) and density (size of the bubbles) vary. A larger bubble means denser and the density varies from 3.78E-05 to 6.54E-03.

size) and size (on the x-axis). Bootes consistently achieves the lowest preprocessing time as the matrices become denser (indicated by larger bubbles) and as their size increases. The matrices, sourced from real-world datasets in SuiteSparse [7] and SNAP [33], naturally exhibit increased sparsity as their size grows, resulting in smaller bubble sizes as the matrix size increases.

Bootes outperforms the baselines by reducing preprocessing times by geometric mean factors of 10.2×, 1.95×, and 11.61× compared to Gamma, Graph, and Hier, respectively. Notably, even as the matrices become denser and larger, Bootes maintains consistent preprocessing efficiency, showcasing its scalability across a wide range of matrix types. This scalability stems from the same factors contributing to its lower preprocessing-to-computation time ratio, as discussed earlier. Preprocessing time is a critical factor in overall computational efficiency, as it can be up to a thousand times more expensive than a single matrix multiplication. This means that even substantial improvements in the multiplication phase, such as making it five times faster, may not compensate for the heavy upfront cost unless the same sparsity pattern is reused in thousands of multiplications. While methods like SpMM and SpMV often benefit iterative computations, the same efficiency gains are not as readily apparent for SpGEMM. Consequently, accelerating the preprocessing stage is essential—it directly enhances performance by reducing the initial computational burden and ultimately allows for more balanced and resource-efficient processing in applications where large-scale, sparse matrix operations are necessary.

In addition to reducing preprocessing time, Bootes boasts a lower memory footprint, making it a widely applicable solution. Even in offline scenarios, memory usage is a critical concern because large, sparse matrices often require significant memory allocation when using other preprocessing methods, which can frequently lead to out-of-memory errors. Bootes effectively mitigates this issue. The bottom part of Figure 5 illustrates the memory footprint, defined as the minimum memory allocation required on the host. Bootes reduces the memory footprint by a geometric mean factor of 2.63×, 1.35×, and 2.10× compared to Gamma, Graph, and Hier, respectively. The figure demonstrates that as matrix size and density increase, Bootes exhibits a lower memory footprint compared to its counterparts, because of its more efficient memory allocation and management strategies. This makes Bootes a more favorable algorithm when scaling up the problem size for reordering.

In the bottom part of Figure 5, we can see that Hier shows an almost linear increase in memory usage as matrix size increases. This is because of the costly heap construction and the space needed to store the hash tables and signatures. Gamma also experiences high memory demands mainly because of the priority queue, which must track priorities, and Gamma also keeps track of how many other rows share a nonzero value in the same column coordinate. Graph faces similar issues, as it must store this information as well for the first iterative loop. However, instead of using a priority queue, Graph uses a graph representation, which needs to store the vertices, edges, and weight information.

Additionally, Gamma is the only algorithm where the final permutations list P is populated during the iterative loop rather than at the end. This increases the maximum memory required at a specific point in execution since P must be allocated simultaneously before other structures can be freed. In Bootes, the primary memory overhead comes from storing the similarity matrix and Laplacian matrix. However, these matrices are kept in compressed formats to reduce the memory footprint. Furthermore, the similarity matrix is de-allocated after the Laplacian construction is complete to ensure efficient memory management.

5.4 Speedup

Figure 6 displays the end-to-end speedup, considering both preprocessing time and compute time, of the SpGEMM kernel for Bootes

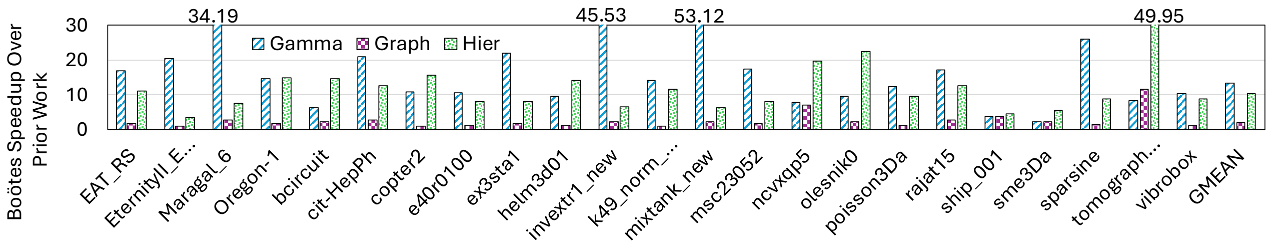

Figure 6: The end-to-end speedup of Bootes over the prior studies.

and our baseline algorithms. Across various matrix patterns, sizes, and sparsity levels, Bootes consistently demonstrates a speedup. In particular, Bootes significantly reduces the ratio of preprocessing time to actual compute time by factors of 13.41×, 1.96×, and 10.34× compared to Gamma, Graph, and Hier, respectively. Preprocessing comprises row reordering, matrix multiplication, and restoring the matrix rows to their original order (post-processing). The post-processing times are uniform across algorithms. Furthermore, the row reordering has a minimal impact on the multiplication phase execution time, as its primary objective is to minimize off-chip memory access and data movement, thereby enhancing overall processing efficiency. In other words, this approach does not adversely affect performance; in particular, it does not increase execution time. Rather, its impact on the multiplication phase is indirect: by reducing memory traffic, data reaches the processing elements with greater bandwidth, enabling more simultaneous computations and ultimately enhancing computational throughput.

Bootes achieves high computational efficiency by limiting the range of cluster sizes (k), reducing time complexity, and avoiding iterative passes when k is small. Its closest competitor, Graph, leverages matrix sparsity to efficiently construct and traverse sparse graphs. In contrast, Gamma incurs overhead from maintaining a priority queue with O(logn) updates. Hier is computationally expensive due to its need to initialize a max-heap based on pairwise similarities, delaying merging until all candidates are processed. While LSH in Hier efficiently narrows down candidate pairs via hash-based bucketing, it does not compute similarity scores. These must be calculated separately using Jaccard similarity, which involves costly set operations (intersections and unions). Additionally, during the merging process, many of these pairs may not contribute meaningfully if they are not chosen as representative rows, which inflates the algorithm's overall complexity. Bootes avoids these major bottlenecks, making it significantly more efficient than the baselines.

Table 4 summarizes the geometric mean speedup of each reordering algorithm across different accelerators. The values reflect the speedup achieved by applying row-reordering preprocessing

Table 4: Geomean Speedup of each reordering algorithm across accelerators.

Accelerator	Bootes	Gamma	Graph	Hier
Flexagon	1.74×	1.28×	1.30×	1.12×
GAMMA	1.35×	1.09×	1.15×	1.07×
Trapezoid	1.22×	1.05×	1.07×	1.02×

compared to no preprocessing. These results complement our earlier findings, showing that Bootes consistently delivers the highest speedup across all accelerators relative to other reordering methods.

6 Prior Work

The related studies discussed throughout this paper are just a few examples among numerous recent efforts focusing on sparse problems [1–3, 8–10, 12, 14–16, 18, 21, 23, 25, 26, 28, 29, 31, 35–42, 46, 47, 49, 51–55, 57–60, 62, 64, 65, 67, 68, 70, 72, 74, 76, 78]. Sparsity-related studies employ a variety of techniques. Some focus on traditional methods such as systolic arrays [19, 20, 45, 63] and adder trees [12, 55]. For instance, innovations such as DTC-SpMM [11] optimize sparse operations for Tensor Cores, whereas Conveyor [30] addresses mismatches between dense systolic arrays and unstructured sparsity. Others propose hardware/software co-designs that focus on prefetching or novel compression formats [10, 29, 58, 59, 62, 69, 78]. Architectures such as HiRAC [56] and Spada [38] apply hierarchical and adaptive designs to accelerate SpGEMM. While these works aim to speed up sparse computation, they often overlook preprocessing overhead—a key focus of Bootes. Finally, recent efforts in sparse compilation [24, 71, 75] are orthogonal to our approach and may offer complementary benefits.

7 Conclusions

This paper introduced Bootes, which emphasizes the importance of preprocessing steps for efficiently integrating sparse accelerators. In particular, it focused on the preprocessing required for reducing off-chip memory traffic, a crucial factor in the efficiency of today's accelerators. Bootes addressed this gap by leveraging a novel spectral clustering approach to optimize row reordering, significantly accelerating preprocessing time. Through integration with state-of-the-art accelerators such as Flexagon, GAMMA, and Trapezoid, we demonstrated Bootes' adaptability and ability to consistently reduce memory traffic across diverse workloads with varying sizes and levels of density. In short, our results highlighted Bootes' novelty in combining preprocessing efficiency with memory traffic optimization, setting a new standard for integrating sparse accelerators in modern computing systems.

Acknowledgments

We gratefully acknowledge the support of US Department of Energy (DoE), Office of Science under the Advanced Scientific Computing Research (ASCR) Early Career Research Program (ECRP), Award DE-SC0024079.

References

[1] Bahar Asgari, Ramyad Hadidi, Jiashen Cao, Sung-Kyu Lim, Hyesoon Kim, et al. 2021. Fafnir: Accelerating sparse gathering by using efficient near-memory intelligent reduction. In *HPCA*. 908–920.

[2] Ubaid Bakhtiar, Helya Hosseini, and Bahar Asgari. 2024. Acamar: A dynamically reconfigurable scientific computing accelerator for robust convergence and minimal resource underutilization. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1601–1616.

[3] Ubaid Bakhtiar, Donghyeon Joo, and Bahar Asgari. 2025. Pipirima: Predicting Patterns in Sparsity to Accelerate Matrix Algebra. In *Proceedings of the 62nd ACM/IEEE Design Automation Conference (DAC)*.

[4] Erfan Bank Tavakoli, Michael Riera, Masudul Hassan Quraishi, and Fengbo Ren. 2024. FSpGEMM: A Framework for Accelerating Sparse General Matrix–Matrix Multiplication Using Gustavson's Algorithm on FPGAs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 32, 4 (2024), 633–644. https://doi.org/10.1109/TVLSI.2024.3355499

[5] William Dally. 2021. The Future of Computing: Domain-Specific Architecture. https://www.clsac.org/uploads/5/0/6/3/50633811/2021-clsac-dally.pdf. [Online; accessed May-2023].

[6] William Dally. 2022. On the Model of Computation: Point: We Must Extend Our Model of Computation to Account for Cost and Location. https://cacm.acm.org/magazines/2022/9/263792-on-the-model-of-computation-point/abstract. [Online; accessed October-2022].

[7] Timothy A. Davis and Yifan Hu. 2011. The university of Florida sparse matrix collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (Dec. 2011), 25 pages. https://doi.org/10.1145/2049662.2049663

[8] Chunhua Deng, Yang Sui, Siyu Liao, Xuehai Qian, and Bo Yuan. 2021. Gospa: An energy-efficient high-performance globally optimized sparse convolutional neural network accelerator. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 1110–1123.

[9] Matthew Denton and Herman Schmit. 2022. Direct Spatial Implementation of Sparse Matrix Multipliers for Reservoir Computing. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 1–11.

[10] Yixiao Du, Yuwei Hu, Zhongchun Zhou, and Zhiru Zhang. 2022. High-Performance Sparse Linear Algebra on HBM-Equipped FPGAs Using HLS: A Case Study on SpMV. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 54–64.

[11] Ruibo Fan, Wei Wang, and Xiaowen Chu. 2024. DTC-SpMM: Bridging the Gap in Accelerating General Sparse Matrix Multiplication with Tensor Cores. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 253–267.

[12] Siying Feng, Xin He, Kuan-Yu Chen, Liu Ke, Xuan Zhang, David Blaauw, Trevor Mudge, and Ronald Dreslinski. 2022. MeNDA: a near-memory multi-way merge solution for sparse transposition and dataflows. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 245–258.

[13] Gelin Fu, Tian Xia, Shaoru Qu, Zhongpei Luo, Shuyu Li, Pengyu Cheng, Runfan Guo, Yitong Ding, and Pengju Ren. 2023. PrSpMV: An Efficient Predictable Kernel for SpMV. In *2023 IEEE 41st International Conference on Computer Design (ICCD)*. IEEE, 448–456.

[14] Armin Gerami and Bahar Asgari. 2024. Gust: Graph edge-coloring utilization for accelerating sparse matrix vector multiplication. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*. 127–141.

[15] Ashish Gondimalla, Noah Chesnut, Mithuna Thottethodi, and TN Vijaykumar. 2019. SparTen: A sparse tensor accelerator for convolutional neural networks. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 151–165.

[16] Sumanth Gudaparthi, Sarabjeet Singh, Surya Narayanan, Rajeev Balasubramonian, and Visvesh Sathe. 2022. CANDLES: Channel-Aware Novel Dataflow-Microarchitecture Co-Design for Low Energy Sparse Neural Network Acceleration. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 876–891.

[17] Fred G Gustavson. 1978. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Transactions on Mathematical Software (TOMS)* 4, 3 (1978), 250–269.

[18] Xin He, Kuan-Yu Chen, Siying Feng, Hun-Seok Kim, David Blaauw, Ronald Dreslinski, and Trevor Mudge. 2022. Squaring the circle: Executing Sparse Matrix Computations on FlexTPU—A TPU-Like Processor. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. 148–159.

[19] Xin He, Kuan-Yu Chen, Siying Feng, Hun-Seok Kim, David Blaauw, Ronald Dreslinski, and Trevor Mudge. 2023. Squaring the Circle: Executing Sparse Matrix Computations on FlexTPU—A TPU-Like Processor. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques* (Chicago, Illinois) *(PACT '22)*. Association for Computing Machinery, New York, NY, USA, 148–159. https://doi.org/10.1145/3559009.3569665

[20] Xin He, Subhankar Pal, Aporva Amarnath, Siying Feng, Dong-Hyeon Park, Austin Rovinski, Haojie Ye, Yuhan Chen, Ronald Dreslinski, and Trevor Mudge. 2020. Sparse-TPU: Adapting systolic arrays for sparse matrices. In *Proceedings of the 34th ACM international conference on supercomputing*. 1–12.

[21] Xin He, Subhankar Pal, Aporva Amarnath, Siying Feng, Dong-Hyeon Park, Austin Rovinski, Haojie Ye, Yuhan Chen, Ronald Dreslinski, and Trevor Mudge. 2020. Sparse-TPU: Adapting systolic arrays for sparse matrices. In *Proceedings of the 34th ACM International Conference on Supercomputing*. 1–12.

[22] Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and P. Sadayappan. 2019. Adaptive sparse tiling for sparse matrix multiplication. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming* (Washington, District of Columbia) *(PPoPP '19)*. Association for Computing Machinery, New York, NY, USA, 300–314. https://doi.org/10.1145/3293883.3295712

[23] Helya Hosseini, Ubaid Bakhtiar, Donghyeon Joo, and Bahar Asgari. 2025. Segin: Synergistically Enabling Fine-Grained Multi-Tenant and Resource Optimized SpMV. *IEEE Computer Architecture Letters* (2025).

[24] Olivia Hsu, Maxwell Strange, Ritvik Sharma, Jaeyeon Won, Kunle Olukotun, Joel S. Emer, Mark A. Horowitz, and Fredrik Kjølstad. 2023. The Sparse Abstract Machine. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. Association for Computing Machinery, 710–726.

[25] Chao-Tsung Huang. 2021. Ringcnn: Exploiting algebraically-sparse ring tensors for energy-efficient cnn-based computational imaging. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 1096–1109.

[26] Abhishek Kumar Jain, Hossein Omidian, Henri Fraisse, Mansimran Benipal, Lisa Liu, and Dinesh Gaitonde. 2020. A domain-specific architecture for accelerating sparse matrix vector multiplication on fpgas. In *2020 30th International conference on field-programmable logic and applications (FPL)*. IEEE, 127–132.

[27] Peng Jiang, Changwan Hong, and Gagan Agrawal. 2020. A novel data transformation and execution strategy for accelerating sparse matrix multiplication on GPUs. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Diego, California) *(PPoPP '20)*. Association for Computing Machinery, New York, NY, USA, 376–388. https://doi.org/10.1145/3332466.3374546

[28] Donghyeon Joo, Ramyad Hadidi, Soheil Feizi, and Bahar Asgari. 2024. Endor: Hardware-Friendly Sparse Format for Offloaded LLM Inference. *arXiv preprint arXiv:2406.11674* (2024).

[29] Konstantinos Kanellopoulos, Nandita Vijaykumar, Christina Giannoula, Roknoddin Azizi, Skanda Koppula, Nika Mansouri Ghiasi, Taha Shahroodi, Juan Gomez Luna, and Onur Mutlu. 2019. SMASH: Co-designing Software Compression and Hardware-Accelerated Indexing for Efficient Sparse Matrix Operations. In *MICRO*. ACM, 600–614.

[30] S. Kim, G. Byeon, S. Kim, H. Kim, and S. Hong. 2023. Conveyor: Towards Asynchronous Dataflow in Systolic Array to Exploit Unstructured Sparsity. In *2023 IEEE 41st International Conference on Computer Design (ICCD)*. IEEE Computer Society, 423–431. https://doi.org/10.1109/ICCD58817.2023.00070

[31] HT Kung, Bradley McDanel, and Sai Qian Zhang. 2019. Packing sparse convolutional neural networks for efficient systolic array implementations: Column combining under joint optimization. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 821–834.

[32] Jong Hun Lee, Beomjin Park, Joonho Kong, and Arslan Munir. 2022. Row-Wise Product-Based Sparse Matrix Multiplication Hardware Accelerator With Optimal Load Balancing. *IEEE Access* 10 (2022), 64547–64559. https://doi.org/10.1109/ACCESS.2022.3184116

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

[34] Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. 2014. *Mining of Massive Datasets* (2nd ed.). Cambridge University Press, USA.

[35] Gang Li, Weixiang Xu, Zhuoran Song, Naifeng Jing, Jian Cheng, and Xiaoyao Liang. 2022. Ristretto: An Atomized Processing Architecture for Sparsity-Condensed Stream Flow in CNN. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1434–1450.

[36] Shiyu Li, Edward Hanson, Xuehai Qian, Hai" Helen" Li, and Yiran Chen. 2021. ESCALATE: Boosting the efficiency of sparse CNN accelerator with kernel decomposition. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 992–1004.

[37] Shiqing Li, Di Liu, and Weichen Liu. 2021. Optimized Data Reuse via Reordering for Sparse Matrix-Vector Multiplication on FPGAs. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 1–9.

[38] Zhiyao Li, Jiaxiang Li, Taijie Chen, Dimin Niu, Hongzhong Zheng, Yuan Xie, and Mingyu Gao. 2023. Spada: Accelerating Sparse Matrix Multiplication with Adaptive Dataflow. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 747–761. https://doi.org/10.1145/3575693.3575706

[39] Bowen Liu and Dajiang Liu. 2023. Towards High-Bandwidth-Utilization SpMV on FPGAs via Partial Vector Duplication. In *Proceedings of the 28th Asia and South Pacific Design Automation Conference (ASP-DAC)*. 33–38.

[40] Hang Lu, Liang Chang, Chenglong Li, Zixuan Zhu, Shengjian Lu, Yanhuan Liu, and Mingzhe Zhang. 2021. Distilling bit-level sparsity parallelism for general purpose deep learning acceleration. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 963–976.

[41] Kai Lu, Zhaoshi Li, Leibo Liu, Jiawei Wang, Shouyi Yin, and Shaojun Wei. 2019. Redesk: A reconfigurable dataflow engine for sparse kernels on heterogeneous platforms. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–8.

[42] Liqiang Lu, Yicheng Jin, Hangrui Bi, Zizhang Luo, Peng Li, Tao Wang, and Yun Liang. 2021. Sanger: A co-design framework for enabling sparse attention using reconfigurable architecture. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 977–991.

[43] Yuechen Lu and Weifeng Liu. 2023. DASP: Specific Dense Matrix Multiply-Accumulate Units Accelerated General Sparse Matrix-Vector Multiplication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 1–14.

[44] Peter Macgregor and He Sun. 2022. A Tighter Analysis of Spectral Clustering, and Beyond. arXiv:2208.01724 [cs.DS] https://arxiv.org/abs/2208.01724

[45] Euripides Montagne and Rina Surós. 2019. Systolic Sparse Matrix Vector Multiply in the Age of TPUs and Accelerators. In *2019 Spring Simulation Conference (SpringSim)*. 1–10. https://doi.org/10.23919/SpringSim.2019.8732860

[46] Francisco Muñoz Martínez, Raveesh Garg, Michael Pellauer, José L. Abellán, Manuel E. Acacio, and Tushar Krishna. 2023. Flexagon: A Multi-dataflow Sparse-Sparse Matrix Multiplication Accelerator for Efficient DNN Processing. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (, Vancouver, BC, Canada,) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 252–265. https://doi.org/10.1145/3582016.3582069

[47] 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.

[48] 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* (Seoul, Republic of Korea) *(PPoPP '22)*. Association for Computing Machinery, New York, NY, USA, 90–106. https://doi.org/10.1145/3503221.3508431

[49] Nebil Ozer, Gregory Kollmer, Ramyad Hadidi, and Bahar Asgari. 2025. La Superba: Leveraging a Self-Comparison Method to Understand the Performance Benefits of Sparse Acceleration Optimizations. In *2025 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 1–12.

[50] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok Kim, David Blaauw, Trevor Mudge, and Ronald Dreslinski. 2018. OuterSPACE: An Outer Product Based Sparse Matrix Multiplication Accelerator. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 724–736. https://doi.org/10.1109/HPCA.2018.00067

[51] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok Kim, David Blaauw, Trevor Mudge, and Ronald Dreslinski. 2018. OuterSPACE: An Outer Product based Sparse Matrix Multiplication Accelerator. In *HPCA*. IEEE, 724–736.

[52] Eric Qin, Geonhwa Jeong, William Won, Sheng-Chun Kao, Hyoukjun Kwon, Sudarshan Srinivasan, Dipankar Das, Gordon E Moon, Sivasankaran Rajamanickam, and Tushar Krishna. 2021. Extending sparse tensor accelerators to support multiple compression formats. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 1014–1024.

[53] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. 2020. SIGMA: A Sparse and Irregular GEMM Accelerator with Flexible Interconnects for DNN Training. In *HPCA*.

[54] Dheeraj Ramchandani, Bahar Asgari, and Hyesoon Kim. 2023. Spica: Exploring FPGA Optimizations to Enable an Efficient SpMV Implementation for Computations at Edge. In *2023 IEEE International Conference on Edge Computing and Communications (EDGE)*. IEEE, 36–42.

[55] Fazle Sadi, Joe Sweeney, Tze Meng Low, James C Hoe, Larry Pileggi, and Franz Franchetti. 2019. Efficient SpMV Operation for Large and Highly Sparse Matrices using Scalable Multi-way Merge Parallelization. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 347–358.

[56] Hesam Shabani, Abhishek Singh, Bishoy Youhana, and Xiaochen Guo. 2023. HIRAC: A Hierarchical Accelerator with Sorting-based Packing for SpGEMMs in DNN Applications. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 247–258. https://doi.org/10.1109/HPCA56546.2023.10070977

[57] Björn Sigurbergsson, Tom Hogervorst, Tong Dong Qiu, and Razvan Nane. 2019. Sparstition: a partitioning scheme for large-scale sparse matrix vector multiplication on FPGA. In *2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, Vol. 2160. IEEE, 51–58.

[58] Linghao Song, Yuze Chi, Licheng Guo, and Jason Cong. 2022. Serpens: A high bandwidth memory based accelerator for general-purpose sparse matrix-vector multiplication. In *Proceedings of the 59th ACM/IEEE design automation conference*. 211–216.

[59] Linghao Song, Yuze Chi, Atefeh Sohrabizadeh, Young-kyu Choi, Jason Lau, and Jason Cong. 2022. Sextans: A streaming accelerator for general-purpose sparse-matrix dense-matrix multiplication. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 65–77.

[60] Nitish Srivastava, Hanchen Jin, Jie Liu, David Albonesi, and Zhiru Zhang. 2020. Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 766–780.

[61] Nitish Srivastava, Hanchen Jin, Jie Liu, David Albonesi, and Zhiru Zhang. 2020. MatRaptor: A Sparse-Sparse Matrix Multiplication Accelerator Based on Row-Wise Product. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 766–780. https://doi.org/10.1109/MICRO50266.2020.00068

[62] Nitish Srivastava, Hanchen Jin, Shaden Smith, Hongbo Rong, David Albonesi, and Zhiru Zhang. 2020. Tensaurus: A versatile accelerator for mixed sparse-dense tensor computations. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 689–702.

[63] Minjin Tang, Mei Wen, Yasong Cao, Junzhong Shen, Jianchao Yang, Jiawei Fei, Yang Guo, and Sheng Liu. 2023. Mentha: Enabling Sparse-Packing Computation on Systolic Arrays. In *Proceedings of the 51st International Conference on Parallel Processing* (Bordeaux, France) *(ICPP '22)*. Association for Computing Machinery, New York, NY, USA, Article 18, 11 pages. https://doi.org/10.1145/3545008.3545053

[64] Chaithanya Krishna Vadlamudi and Bahar Asgari. 2024. Electra: Eliminating the Ineffectual Computations on Bitmap Compressed Matrices. *IEEE Computer Architecture Letters* (2024).

[65] Hanrui Wang, Zhekai Zhang, and Song Han. 2021. Spatten: Efficient sparse attention architecture with cascade token and head pruning. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 97–110.

[66] Xiaoqian Wang, Feiping Nie, and Heng Huang. 2016. Structured Doubly Stochastic Matrix for Graph Based Clustering: Structured Doubly Stochastic Matrix. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (San Francisco, California, USA) *(KDD '16)*. Association for Computing Machinery, New York, NY, USA, 1245–1254. https://doi.org/10.1145/2939672.2939805

[67] Yang Wang, Chen Zhang, Zhiqiang Xie, Cong Guo, Yunxin Liu, and Jingwen Leng. 2021. Dual-side sparse tensor core. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 1083–1095.

[68] Xinfeng Xie, Zheng Liang, Peng Gu, Abanti Basak, Lei Deng, Ling Liang, Xing Hu, and Yuan Xie. 2021. SpaceA: Sparse matrix vector multiplication on processing-in-memory accelerator. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 570–583.

[69] Z. Xue, M. Wen, Z. Chen, Y. Shi, M. Tang, J. Yang, and Z. Luo. 2023. Releasing the Potential of Tensor Core for Unstructured SpMM using Tiled-CSR Format. In *2023 IEEE 41st International Conference on Computer Design (ICCD)*. IEEE Computer Society, 457–464. https://doi.org/10.1109/ICCD58817.2023.00076

[70] Sanjali Yadav and Bahar Asgari. 2025. DynaFlow: An ML Framework for Dynamic Dataflow Selection in SpGEMM accelerators. *IEEE Computer Architecture Letters* (2025).

[71] Tao Yang, Yiyuan Zhou, Qidong Tang, Feng Xu, Hui Ma, Jieru Zhao, and Li Jiang. 2023. SpMMPlu: A Compiler Plug-in with Sparse IR for Efficient Sparse Matrix Multiplication. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1109/DAC56929.2023.10247957

[72] Yifan Yang, Joel S Emer, and Daniel Sanchez. 2021. Spzip: architectural support for effective data compression in irregular applications. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 1069–1082.

[73] Y. Yang, J. S. Emer, and D. Sanchez. 2024. Trapezoid: A Versatile Accelerator for Dense and Sparse Matrix Multiplications. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE Computer Society, Los Alamitos, CA, USA, 931–945. https://doi.org/10.1109/ISCA59077.2024.00072

[74] Amir Yazdanbakhsh, Ashkan Moradifirouzabadi, Zheng Li, and Mingu Kang. 2022. Sparse Attention Acceleration with Synergistic In-Memory Pruning and On-Chip Recomputation. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 744–762.

[75] Zihao Ye, Ruihang Lai, Junru Shao, Tianqi Chen, and Luis Ceze. 2023. Sparsetir: Composable abstractions for sparse compilation in deep learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 660–678.

[76] Shulin Zeng, Yujun Lin, Shuang Liang, Junlong Kang, Dongliang Xie, Yi Shan, Song Han, Yu Wang, and Huazhong Yang. 2019. A fine-grained sparse accelerator for multi-precision DNN. In *Proceedings of the 2019 ACM/SIGDA International*

Symposium on Field-Programmable Gate Arrays. 185–185.

[77] Guowei Zhang, Nithya Attaluri, Joel S. Emer, and Daniel Sanchez. 2021. Gamma: leveraging Gustavson's algorithm to accelerate sparse matrix multiplication. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 687–701. https://doi.org/10.1145/3445814.3446702

[78] Zhekai Zhang, Hanrui Wang, Song Han, and William J Dally. 2020. Sparch: Efficient architecture for sparse matrix multiplication. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 261–274.