Rethinking Tiling and Dataflow for SpMM Acceleration: A Graph Transformation Framework

Amir Ghazizadeh Ahsaei*
University of Central Florida
Orlando, USA
amir.g@ucf.edu

Lingxiang Yin*
University of Central Florida
Orlando, USA
lingxiang.yin@ucf.edu

Shilin Tian
University of Central Florida
Orlando, USA
shilin.tian@ucf.edu

Fangzhou Ye
University of Central Florida
Orlando, USA
fa011718@ucf.edu

Fan Yao
University of Central Florida
Orlando, USA
fan.yao@ucf.edu

Hao Zheng
University of Central Florida
Orlando, USA
hao.zheng@ucf.edu

Abstract

Sparse Matrix Dense Matrix Multiplication (SpMM) is a fundamental computation kernel across various domains, including scientific computing, machine learning, and graph processing. Despite extensive research, existing approaches optimize SpMM using loop transformations and linear algebra principles, which (1) poorly handle unstructured sparsity patterns, (2) rely on empirical methods to explore data reuse opportunities, and (3) enforce rigid coordinate alignment, compromising data locality.

In this paper, we demonstrate that these limitations stem from the fundamental matrix representation and traditional dataflows of SpMM (e.g., inner-product, outer-product, and Gustavson). We propose Aquila, a graph transformation framework that reformulates SpMM computations as a graph optimization problem, leveraging graph theory to reinterpret tiling and dataflow. First, on the theoretical side, we introduce vertex decomposition and adaptive depth traversal (ADT) to enable non-contiguous tiling, where nonzero elements from discontinuous rows and columns are clustered by connectivity rather than following matrix dimensionality. This approach quantifies data reuse and improves data locality beyond traditional loop transformations while maintaining output equivalence. Second, on the algorithm side, we develop a pull-after-push (PaP) dataflow that simultaneously enhances the dense matrix data reuse while eliminating synchronization issues in output matrix accumulation. Third, building on our theoretical approach and dataflow, we present a versatile accelerator architecture that handles a variety of SpMM kernels with diverse data sizes and sparsity patterns in a unified architecture. Additionally, we introduce a bidirectional fiber tree (BFT) format to support the proposed graph-oriented dataflow in contrast to traditional column or row-major access. Evaluation across diverse sparse datasets shows Aquila achieves speedups of 4.3×, 3.4×, 3.7×, 2.9×, and 2.7× in execution time and up to 4.8× improvements in energy efficiency compared to state-of-the-art accelerators.

ACM Reference Format:
Amir Ghazizadeh Ahsaei, Lingxiang Yin, Shilin Tian, Fangzhou Ye, Fan Yao, and Hao Zheng. 2025. Rethinking Tiling and Dataflow for SpMM Acceleration: A Graph Transformation Framework. In *58th IEEE/ACM International Symposium on Microarchitecture (MICRO '25), October 18–22, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3725843.3756128

1 Introduction

Sparse Matrix Dense Matrix Multiplication (SpMM) is a critical computation kernel in numerous domains such as machine learning and scientific computing [2, 5, 6, 18, 20, 40–42, 45, 66]. For example, in Graph Neural Networks [35], a sparse matrix is used to represent the graph connectivity, with nonzero elements indicating connections between pairs of vertices [33, 58, 62]. While such sparse data representation can be leveraged to eliminate unnecessary computations and reduce storage overheads, the irregular data patterns pose challenges to data locality and computation regularity.

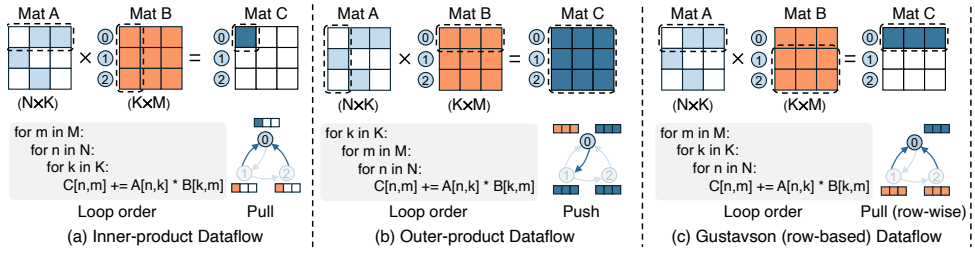
Prior work [1, 8, 57, 63, 64] has employed conventional loop transformation techniques to optimize data reuse and parallelism in SpMMs. However, these techniques rely on dense linear algebra, which fails to accurately capture data reuse within unstructured sparse patterns. For example, inner [21, 22, 47] and row-wise product [34, 50, 67] (i.e., pull-based dataflow) have been used to improve the data reuse of the output matrix, where multiple blocks of the input matrix are simultaneously retrieved to generate a single block of the output matrix. However, the data reuse for the dense input matrix remains suboptimal because of the unstructured sparse matrix. Additionally, accumulating partial results of the output matrix imposes strict synchronization requirements. Outer-product dataflow [44, 69] (i.e., push-based dataflow) enhances the data reuse of input dense matrix but may reduce the reuse efficiency of the output matrix. However, current dataflows cannot simultaneously optimize data reuse of dense matrices while avoiding the synchronization issue. This limitation is primarily due to the loop-based representation of dataflows.

Moreover, significant research [3, 28, 32, 36, 61] aimed to regularize the sparse matrices by leveraging matrix reordering or condensing nonzero elements. However, reordering the distribution of nonzero elements in matrix representation has been proven to be an NP-hard problem [3, 31], and the resulting solutions are specific

*Both authors contributed equally to this research.

MICRO '25, Seoul, Republic of Korea

ACM ISBN 979-8-4007-1573-0/25/10
https://doi.org/10.1145/3725843.3756128



	Inner	Outer	Gustavson
B reuse	Low	High	Moderate
C reuse	High	Low	Moderate
C granularity	Scalar	Matrix	Vector
On-chip buffer requirement	Low	High	Moderate

(d) Dataflow Comparison

Figure 1: Traditional matrix-based dataflows and their corresponding graph representations.

to a given sparsity pattern. In addition, while graph clustering algorithms, such as I-GCN [15] and GCoD [65], can enhance the density in specific matrix regions, efficient data reuse ultimately depends on the tiling strategy and dataflow design. For example, several well-known sparse matrix reordering algorithms, including RCM [37], column approximate minimum degree ordering [9], graph islandization [15], and column permutation based on nonzero counts [24], are only applicable to a certain sparsity pattern or application. Condensing and repositioning nonzero elements [11, 68, 69] at runtime can effectively balance the workload, but disrupting the coordinate information can lead to irregular dependencies with prohibitive synchronization overheads. Furthermore, adaptive tiling [34] is effective for balancing the quantity of nonzero elements across matrix partitions, but varying matrix dimensions could in turn affect the data locality of dense matrices.

In this paper, we argue that the fundamental issue of existing SpMM optimizations results from their matrix representation. The linear representation of SpMMs is theoretically ineffective in analyzing and identifying the optimal data reuse and parallelism opportunities. As opposed to matrix representation, our key idea is to use graph representation to analyze the relationship between nonzero elements rather than where they are indexed. Upon this central idea, we propose a graph transformation framework for designing efficient SpMM accelerators.

Specifically, this paper makes the following contributions:

- On the theory side, we repurpose graph transformation to rethink matrix tiling and dataflow, wherein the SpMM kernel is abstracted as a graph. We introduce vertex decomposition and adaptive depth traversal, which permit non-contiguous tiling, grouping nonzero elements from rows and columns that are not consecutive in their indices. This serves as a theoretical foundation to quantify and enhance inter- and intra-tile data reuse while retaining the mathematical equivalence to matrix representation.
- On the algorithm side, we propose a graph-oriented dataflow to enhance data reuse. Specifically, we propose a parent-and-child aggregation dataflow, resulting from the proposed vertex decomposition, to increase the data reuse between matrix partitions. Additionally, we propose a pull-after-push dataflow, as opposed to the current push-based (outer-product) or pull-based (inner and row-wise product) model, to increase the data reuse of both input and output dense matrices while avoiding the synchronization issue in partial sum accumulation.
- On the architecture side, backed by our framework and dataflow, we propose a versatile accelerator architecture that can efficiently handle various SpMM kernels with varying data sizes and sparsity patterns. Specifically, We introduce a new sparse data compression format, bidirectional fiber tree (BFT), to support graph-oriented dataflows, dedicated processing logic to enable dynamic vertex decomposition and adaptive depth traversal, and a unified architecture that decouples inter- and intra-tile computation into a child-parent aggregator and customized processing element (PE) engine.

We conduct a detailed performance and energy evaluation through simulation and show that the proposed accelerator achieves 4.3×, 3.4×, 3.7×, 2.9×, and 2.7× reductions in execution time when compared to state-of-the-art-accelerators, Sextans [49], SPADE [17], HotTiles [16], ReGNN [7] and I-GCN [15], respectively. Aquila can further achieve up to 4.8× improvements in energy efficiency.

2 Background and Motivation

2.1 SpMM Dataflows

There are several dataflow models to compute SpMM kernels [15, 21, 33, 38], each offering distinct tradeoffs based on sparsity structure and reuse opportunities. Given a sparse matrix $A \in \mathbb{R}^{N\times K}$, a dense matrix $B \in \mathbb{R}^{K\times M}$, and the output $C \in \mathbb{R}^{N\times M}$, the kernel computes: $C = A \times B$. Following linear algebra principles, as illustrated in Figure 1, loop reordering over (N, K, M) exposes different memory access patterns and reuse behaviors depending on the sparsity distribution and matrix dimensions.

For instance, as illustrated in Figure 1(a), Extensor [21] and Sigma [47] adopt the inner-product dataflow to optimize SpMM execution. Here, each output element $C[n, m]$ is computed by taking the dot product between the n-th row of the sparse matrix A and the m-th column of the dense matrix B. This loop structure promotes temporal reuse of the output row $C[n, :]$, which remains local during the inner loop iteration. However, neither the sparse matrix A nor the dense matrix B benefits from effective reuse. The nonzero pattern of each row in A dictates which rows of B must be fetched. Specifically, computing $C[i, :]$ requires accessing $B[k, :]$ for every k where $A[i, k] \neq 0$. This access is row-specific and irregular, preventing the reuse of B across iterations. As a result, B is reloaded for each row of A, leading to $O(N)$ redundant reads.

On the other hand, outer-product [43, 69] is adopted to maximize the reuse of the input matrix B. As shown in Figure 1(b), the k-th column of A is broadcast across all rows of C, and multiplied with the k-th row of B to generate contributions to C. This traversal exploits the temporal reuse of $B[k, :]$ across all output rows. However, it incurs a high cost in output accumulation: partial sums for $C[n, m]$ are generated out-of-order and must be buffered or synchronized until all k contributions are complete. This places significant pressure on the on-chip memory to retain large portions of C throughout the computation.

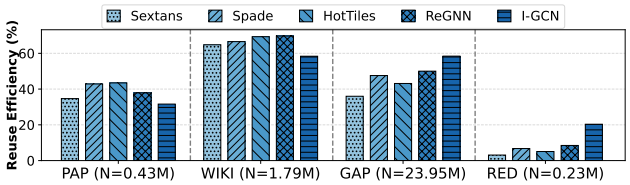


Figure 2: Data reuse of traditional SpMM accelerators.

Similarly, as shown in Figure 1(c), the row-based product (i.e., Gustavson) shares the inner-product's goal of optimizing the reuse of the output matrix C, but mitigates some of the inefficiencies in accessing B. This dataflow processes A row by row: for each row n, it iterates over nonzeros $A[n,k]$, fetches the corresponding row $B[k,:]$, scales it by $A[n,k]$, and accumulates the result into $C[n,:]$. While this improves access locality compared to inner-product, the reuse of B remains limited due to irregular sparsity, and the accumulation into C introduces synchronization overheads (e.g., multiple writes to the same row of C) when parallelized [25, 36].

2.2 Traditional Tiling Methods and Limitations

Tiling is a critical technique for organizing computation and memory access in SpMM to match the on-chip buffer capacity constraints. It partitions the sparse and dense matrices in the SpMM kernel into manageable blocks to minimize redundant transfers and enhance reuse. However, in SpMM, traditional tiling faces unique challenges due to unstructured sparsity. Tiles with identical dimensions can exhibit vastly different memory access patterns depending on nonzero distribution, making workload scheduling and reuse difficult to model. The tiling strategy must therefore not only consider matrix dimensions and buffer size, but also sparsity structure. That is, the reuse potential lies in the relationships among nonzeros, not merely where they are indexed. Traditional tiling methods ignore this aspect, as they partition matrices into fixed rows and columns based on consecutive index ranges.

Position-based tiling [21, 23, 29, 53] slices matrices into fixed row or column blocks. While simple to implement, these tiles often contain large regions of zeros, resulting in poor compute utilization and wasted memory bandwidth when moved across the memory hierarchy. Additionally, being agnostic to sparsity structure, tile boundaries offer no guarantee about the locality or relevance of nonzeros, leading to unpredictable and irregular accesses to dense matrices B and C. Even worse, B and C must conform to the tile dimensions of A, forcing entire rows or columns to be fetched regardless of actual access pattern. This increases on-chip storage demand and further complicates the exploitation of reuse in on-chip buffers. Adaptive tiling methods [21, 23] attempt to balance nonzeros across tiles but still operate on index-aligned regions, resulting in irregular tile shapes, unpredictable reuse patterns, and high control overhead. Consequently, large segments of B and C are fetched speculatively without guaranteed reuse.

On the other hand, loop reordering faces two key limitations in determining temporal reuse. First, due to uneven nonzero distributions and varying tile sizes, measuring optimal reuse is difficult. Second, loop-based SpMM dataflows inherently favor either pull or push, preventing simultaneous reuse of both input and output matrices. To study the interaction between tiling and dataflow, we characterized dense matrix reuse across four scientific domains

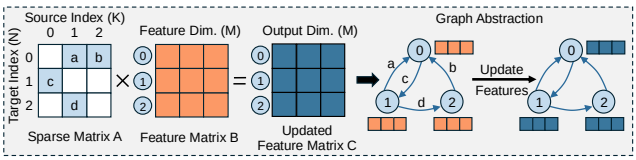


Figure 3: Graph interpretation of SpMM dataflows.

with diverse sparsity and dimensions. We measure how often rows of B and C are reused after being fetched on-chip (tile size 64×64), spanning five state-of-the-art SpMM accelerators. As shown in Figure 2, the average reuse is 39.9%, with up to $\approx$ 92% of potential reuse unexploited in some cases, underscoring the need for new reuse-optimized execution strategies.

3 Graph Abstraction of SpMM kernels

Traditional matrix representations of SpMM impose a rigid structure on scheduling nonzero elements following row- or column-wise coordinate traversal, where the sparsity patterns are overlooked. Instead, we leverage a graph abstraction that preserves the mathematical semantics of SpMM while exposing sparsity through connectivity. As shown in Figure 3, we reinterpret the SpMM kernel $C = A \times B$ as a directed graph $G = (V, E)$, where each nonzero entry $A[n,k] \neq 0$ is an edge from source vertex k to target vertex n. Here, the column dimension K of A defines the source vertices and aligns with the row indices of B, while the row dimension N of A defines the target vertices and maps to the row indices of C. The dense matrix $B \in \mathbb{R}^{K \times M}$ assigns M-dimensional feature vectors to source vertices, and the output $C \in \mathbb{R}^{N \times M}$ aggregates these features into target vertices via multiply-accumulate operations per edge. *Pull*-based dataflows [21, 46, 56] (inner-product and row-wise) correspond to target nodes n pulling features $B[k,:]$ via incoming edges. *Push*-based dataflows [14, 15, 46] (outer-product) correspond to source nodes k broadcasting $B[k,:]$ along outgoing edges to target nodes n.

However, source and target vertex counts determine dense matrix storage requirements. When vertices span multiple partitions, the corresponding dense matrix rows require multiple accesses. We use two graph theory primitives for non-contiguous tiling: *vertex decomposition*, which breaks high-degree vertices into lower-degree ones (previewed in Figure 4(c)), and *adaptive depth traversal (ADT)*, which clusters highly connected vertices. The decomposed vertex structure can reduce the K and N dimensions of the produced tiles for dense matrices, whereas the adaptive depth traversal can cluster nonzero elements within each row and column into a single tile, improving the data reuse of each tile.

3.1 Non-contiguous Tiling

Matrix tiling partitions a large matrix to meet the limited storage capacity. Current SpMM tiling techniques, such as nonzero based scheduling and adaptive tiling [12, 23, 49], emphasize the quantity of nonzero elements while neglecting their coordinate information. This often incurs (1) increased on-chip storage demand for dense and output matrices or (2) compromised data reuse due to the redundant data fetches. For example, as shown in Figure 4(a) and (b), if matrix A employs either row or column-wise tiling, the row or column dimensions of matrices B and C must be adjusted accordingly to ensure dimension matching. This results in compromised

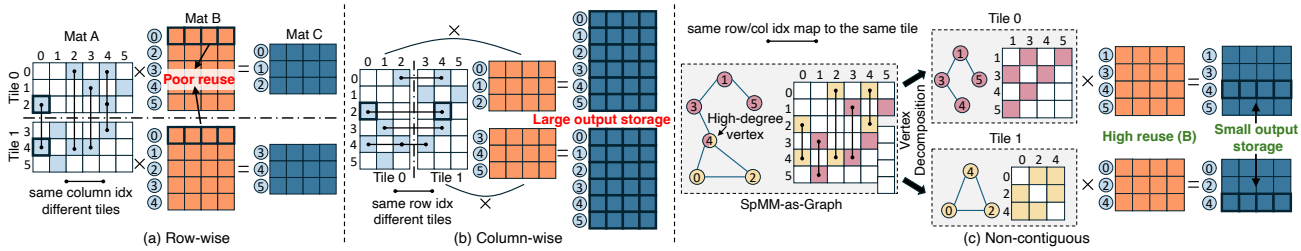


Figure 4: The impacts of matrix tiling on SpMM: (a) row-wise, (b) column-wise, and (c) proposed non-contiguous tiling.

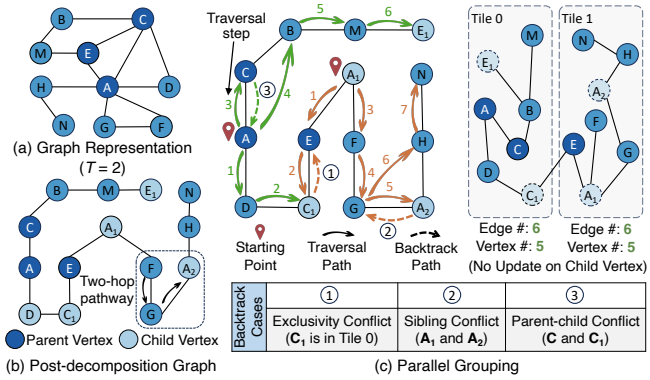


Figure 5: Vertex decomposition and adaptive depth traversal.

data reuse and increased on-chip storage demands. To address this issue, we propose non-contiguous tiling that does not adhere to strict row or column index ordering as shown in Figure 4(c).

3.1.1 Vertex Decomposition. A key challenge in SpMM is that rows or columns with many nonzero elements (i.e., high-degree vertices in the graph abstraction) skew both compute and memory usage: the same row from the dense matrix B must be fetched repeatedly to handle all associated nonzeros. To mitigate this, we first use *vertex decomposition* to split any high-degree vertex v^* with degree $\deg(v^*) > T$ into multiple child vertices $N = \lceil \deg(v^*)/T \rceil$, each inheriting a fraction of v^*'s edges. Consequently, each child vertex refers to the same row of **B** (via the parent's coordinate) but can be mapped to different tiles to avoid overwhelming a single tile.

The original vertex becomes a *parent* vertex, which later accumulates outputs from its children. Figure 5(b) illustrates the result of applying vertex decomposition to the input graph in Figure 5(a). Vertices A, C, and E are decomposed into child vertices (e.g., A into A_1 and A_2), with each child assigned a subset of the original edges according to the vertex loading order. For instance, edges from A to E and A to F are assigned to A_1, while edges from A to G and A to H are assigned to A_2. This transformation ensures that: (1) all vertices have the degree at most T, enforcing an upper bound on the number of nonzeros per row or column and thus regularizing sparsity, and (2) dense operand accesses can now be localized and reused across subgraphs, since child vertices (which duplicate the parent's dense row of matrix B) may be mapped to different tiles. Specifically, rows or columns with a high number of nonzeros, which would otherwise be reused across too many tiles, are now broken into predictable blocks that improve data locality.

Algorithm 1 illustrates the process of offloading edges from v^* to its children. To set T, we perform Deterministic Skip Sampling (DSS) [54] with $m = 1000$ to estimate degree distributions within 3% error. We choose T so that the expected set of decomposed vertices (and their partial results) meets the on-chip buffer capacity (which will be discussed in the following section).

Algorithm 1 Graph Vertex Decomposition

Input: Graph $G(V, E)$, Threshold T
Output: Decomposed Graph $G'(V', E')$, list *child-parent*

```
function DECOMPOSE(G, T)
    Initialize children, keep, split, child_info
    for v ∈ V do
        if DEGREES(v) > T then
            neighbors ← GETNEIGHBORS(G, v)
            keep, split ← SPLITNEIGHBORS(neighbors, T)
            EDGEOFFLOAD(G, v, keep)
            children ← CHILDDESCEND(v, split, T)
            EDGELOAD(child_info, children)
            child-parent ← CHILDDESCEND(v, children)
        end if
    end for
    G' ← GRAPHRECONSTRUCT(G(V, E), children)
    return G', child-parent
end function
```

3.1.2 Quantifiable Data Reuse in Graph Abstraction: After decomposition, we form *non-contiguous tiles* by grouping vertices that maximize data reuse, independent of their original coordinate order. In particular, we co-locate vertices that share common neighbors, capturing reuse via *two-hop pathways*. When two vertices share neighbor v, they both require access to row $B[v, :]$ and contribute to overlapping rows in C. Grouping such vertices into the same tile allows reuse of both dense input and partial output data. We define a reuse-maximization objective across all partitions p as:

$$\max \sum_{i=0}^{|p|} \sum_{v \in V_p} \binom{\deg(v)}{2} \tag{1}$$

Each term $\binom{\deg(v)}{2}$ quantifies reuse opportunities centered at vertex v: unique vertex pairs in partition p that share v as a neighbor. Placing such triplets, v and its neighbors into the same tile enables both operands (i.e., matrix B rows) and outputs (i.e., rows of matrix C) to be reused locally. For example, in Figure 5(b), vertices E and F share neighbor A_1, forming a two-hop pathway suggesting these vertices should be grouped to maximize data locality.

3.1.3 Adaptive Depth Traversal for Non-contiguous Tiling: To solve this optimization problem while maintaining balanced partitions, we introduce Adaptive Depth Traversal (ADT), a modified depth-first search (DFS) traversal. ADT initiates multiple parallel traversals from different starting points in the decomposed graph G', exploring each with limited depth D. This captures the benefits

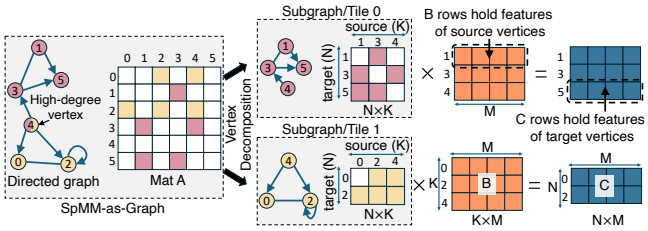


Figure 6: Directed graph of asymmetric sparse matrix.

of DFS by including two-hop pathways while staying within local neighborhoods like BFS. Unlike traditional DFS, ADT does not explore branches to their maximum depth but instead balances depth exploration with breadth coverage. This strategy increases the likelihood of capturing connected communities within each subgraph, where multiple vertices can share access to common neighbors, translating to improved data locality in SpMM, where different nonzeros access the same row of B. By limiting exploration depth to parameter D (set similarly to threshold T from vertex decomposition), we balance vertex and edge distribution across partitions.

3.1.4 Conflict Management. Implementing ADT requires adherence to specific traversal rules to maximize reuse opportunities. Beyond standard DFS backtracking, additional backtracking instances, termed *conflicts*, may arise. Figure 5(c) illustrates how ADT navigates a graph to achieve partitioning in the presence of these conflicts. These conflicts include:

Exclusivity Conflict: The search hits a vertex already claimed by another partition (backtrack case ① in Figure 5(c)), triggering a backtrack to explore different branches. **Sibling Conflict:** A newly visited child vertex such as A_2 conflicts with its sibling (e.g., A_1) already in the same partition (backtrack case ②). Since the sibling already ensures the required row/column reuse, the traversal backtracks to another branch. **Parent-Child Conflict:** A parent vertex C lands in the same partition as its child C_1 (backtrack case ③). To improve the vertex diversity and balance workload, either the parent or child vertex is thus reassigned to a different partition.

3.1.5 Impact of Column Tiling on Off-chip Memory Access. In practice, the column dimension (M) of dense matrices B and C may exceed on-chip buffer capacity, necessitating column-wise tiling. Smaller M_t values enable more vertices (rows from sparse matrix A) to fit within on-chip buffers, enhancing vertex connectivity exploitation through ADT. However, this increases the number of passes through sparse matrix A, as each pass processes only a fraction of the output columns. Conversely, larger M_t values reduce the required passes through A but limit the number of vertices that can be processed concurrently, constraining ADT's ability to capture graph connectivity patterns. To quantify this relationship, we analyze the buffer capacity constraint that determines the maximum number of vertices $|V_p^{'}|$ that can reside on-chip within partition p:

$$|V_p^{'}| \cdot M_t \cdot 2 \cdot 4 + (2 \cdot |E_I^p| + |E_X^p|) \cdot s \leq B_{cap} \tag{2}$$

where factor 2 represents both dense input and output matrices (B and C), 4 bytes represents the size of each 32-bit floating-point element, s is the size in bytes for each nonzero element in the sparse matrix, and B_{cap} is the on-chip buffer capacity. E_I^p represents internal edges (both endpoints in partition p), and E_X^p represents cut edges (one endpoint in partition p).

After vertex decomposition, each vertex's degree is bounded by threshold T, and we let V' and E denote the vertex and edge sets of the transformed graph. The average degree is $\mu' = \frac{|E|}{|V'|}$. Assuming $|V_p|$ vertices per partition and p total partitions, we estimate:

$$E_I^p \approx |V_p^{'}| \cdot \mu' \cdot \frac{1}{|p|}, \quad E_X^p \approx |V_p^{'}| \cdot \mu' \cdot \frac{|p|-1}{|p|} \tag{3}$$

The factor $\frac{1}{|p|}$ represents the probability that both endpoints of an edge fall within the same partition, while $\frac{|p|-1}{|p|}$ indicates the probability of an edge crossing partition boundaries as $|p|$ increases. Substituting these estimations into Equation 2 and solving for M_t:

$$M_t = \frac{B_{cap} - |V_p^{max'}| \cdot \mu' \cdot \left(1 + \frac{1}{|p|}\right) \cdot s}{|V_p^{max'}| \cdot 8} \tag{4}$$

This equation reveals the inverse relationship between column tile size M_t and vertex capacity $|V_p^{max'}|$. To evaluate ADT's effectiveness in minimizing cross-partition edges, we define an edge locality factor λ, which represents the fraction of edges that cross partition boundaries. Lower λ values indicate better locality capture by ADT, resulting in fewer redundant memory accesses. The total data volume accessed from DRAM per SpMM operation can then be expressed as:

$$D_{total} = 8 \cdot |V'| \cdot M + \frac{M}{M_t} \cdot |E| \cdot s \cdot 2 \cdot \lambda \tag{5}$$

where the first term represents accesses to dense matrices B and C, and the second term represents accesses to sparse matrix A, adjusted by the ADT effectiveness factor. The λ term reflects that as external edged decrease, fewer redundant accesses are needed across tiles.

3.1.6 Non-contiguous Tiling for Asymmetric Sparse Matrix. We use an example to demonstrate the applicability of non-contiguous tiling to the asymmetric sparse matrix as shown in Figure 6, where vertex 4 connects to vertices 0 and 2. Vertex 4 pushes its feature vector $B[4,:]$ to both targets, which in turn pull and accumulate these into $C[0,:]$ and $C[2,:]$, respectively. Asymmetric sparse matrices can be formulated as directed graphs in our abstraction, with K source and N target vertices could result in rectangular matrices ($N \neq K$). Applying vertex decomposition and ADT can still partition the rectangular matrices into different tiles.

4 Pull-after-Push Dataflow

The next challenge is to design a dataflow that supports such sparse tiles and reframes the push–pull dichotomy to fully exploit the reuse exposed by ADT. Conventional SpMM dataflows follow either a Push (outer-product) or Pull (inner- and row-wise) execution model. Push emphasizes reuse of dense input matrix rows via column-wise broadcasting but incurs scattered and uncoordinated writes to the output. Pull improves locality for output writes but suffers from poor reuse of dense inputs due to irregular sparsity. To overcome these limitations, we propose a novel hybrid dataflow named *Pull-after-Push (PaP)*, integrating the complementary advantages of PUSH and PULL paradigms. At its core, PaP exploits the structural information exposed by our graph abstraction of a sparse tile. PaP operates through a coordinated two-phase traversal.

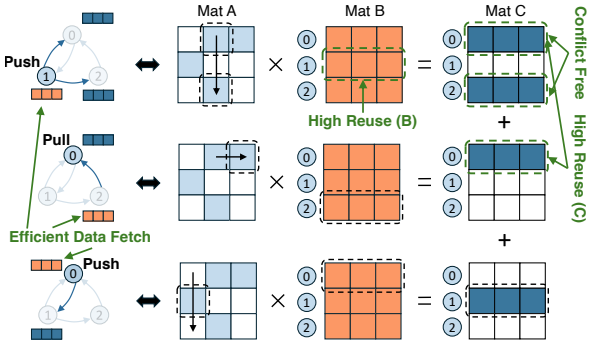


Figure 7: Proposed Pull-after-Push dataflow.

Push Phase (column-wise): Upon encountering a nonzero element $A[i, j]$, the dataflow retrieves the corresponding row $B[j, :]$ from the dense matrix and multiplies it with all nonzeros in column j of the sparse matrix. This operation efficiently exploits the temporal reuse of $B[j, :]$ across multiple multiplications and distributes partial products to distinct output buffers, eliminating write conflicts. This is shown in Figure 7.

Pull Phase (row-wise): After completing the column traversal, PaP immediately transitions into the PULL phase, revisiting the first nonzeros found on A's j column, and horizontally processing the remaining nonzeros along row i in A to finalize the accumulation into output row $C[i, :]$. Such interleaved switching naturally captures both dense input and output reuse opportunities, efficiently utilizing on-chip buffering for partial sums.

Unlike static, loop-oriented dataflows like inner-product, outer-product, or row-wise schemes that commit to fixed traversal patterns, PaP adapts dynamically to the unstructured state of sparsity. The hybrid traversal allows each PE to exploit both dense matrix row reuse (Push phase) and output row reuse (Pull phase) without sacrificing either benefit, substantially reducing off-chip memory access compared to pure Pull or Push implementations. For example, figure 7 illustrates the PaP dataflow from a graph abstraction perspective. Given nonzero entries $\{(0, 1), (0, 2), (1, 0), (2, 1)\}$, PaP initiates with column 1 (PUSH), computing partial sums for rows 0 and 2. It then switches to row 0 (PULL), accumulates remaining values, and proceeds dynamically, guided by graph connectivity. This dynamic, graph-guided traversal ensures predictable reuse, conflict-free execution, and efficient local buffer utilization, characteristics unattainable with traditional matrix-centric dataflows. This behavior holds within each tile, as decomposition and ADT regularize the sparsity pattern.

Benefits: PaP's adaptive traversal directly leverages graph structure to achieve (1) maximal reuse for dense rows of B (during the Push phase), (2) reuse of output buffers (localized row-wise aggregation during Pull phase), and (3) conflict-free writes to different rows of C concurrently.

4.1 Bidirectional Fiber Tree (BFT) Format

Efficiently implementing PaP requires a sparse storage format capable of adaptive, bidirectional traversal along both rows and columns simultaneously. Conventional formats, including CSR, CSC, and traditional fiber trees, optimize data accesses strictly along a single dimension—either rows or columns [51]. Such rigidity fundamentally restricts the efficiency of PaP's traversal. To overcome this limitation, we introduce the *Bidirectional Fiber Tree (BFT)*, a specialized compressed format explicitly designed for PaP's simultaneous multi-dimensional traversal.

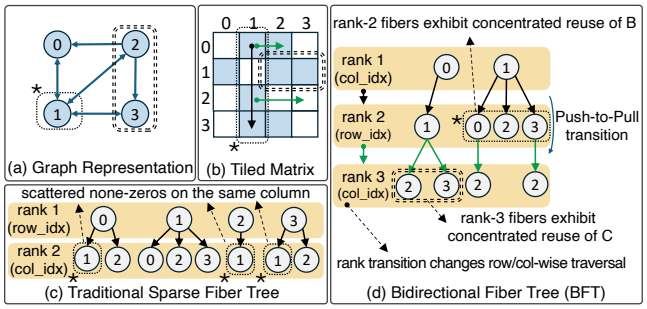


Figure 8: An example of the proposed BFT format.

As shown in Figure 8, the BFT format extends the fiber tree concept by explicitly interleaving row and column indices, matching PaP's traversal semantics. Specifically, BFT begins traversal from the first encountered nonzero (i, j) at rank-1 (column), branching to rank-2 fibers (rows sharing the column index j) and subsequently connecting to rank-3 fibers (columns sharing each row index i). This hierarchical structure explicitly encodes both reuse opportunities for dense input (column-centric) and output accumulation (row-centric). In traditional fiber trees, rigidly traversing all nonzero elements in an entire row or column could potentially affect the spatial data locality. The proposed BFT format explicitly enables bidirectional (row-to-column and column-to-row) and partial traversal. In particular, fibers at rank 2 concentrate reuse opportunities for matrix B (identical col_index), while fibers at rank 3 concentrate reuse opportunities for matrix C (identical row_index). As illustrated in Figure 8(c), traditional fiber trees scatter these reuse opportunities across separate fibers, whereas Figure 8(d) shows that the BFT structure organizes reuse opportunities into single fibers in a concentrated manner, significantly enhancing data locality and reducing redundant memory accesses.

Overhead Analysis. The storage overhead and efficiency of the compression format depend on the sparsity ratio and distribution. BFT, similar to COO, records the full set of coordinate information of each nonzero element. However, BFT further reduces the storage overheads of COO by encoding row and column indexes similar to CSR and CSC. In the worst case scenario, when no nonzero elements share the same column or column index, BFT incurs approximately 33% of storage overheads relative to CSR or CSC.

5 Aquila Accelerator

This section presents the accelerator design that enables our proposed graph-based tiling and dataflow. We introduce a *Non-Contiguous Tiling (NCT) Engine* that dynamically performs vertex decomposition followed by adaptive depth traversal (ADT) to generate non-contiguous tiles at runtime. To support inter-tile accumulation driven by dependencies in high-degree sparse rows, we design a dedicated *Child-Parent Aggregator* unit. Finally, we introduce a custom *Processing Element (PE)* microarchitecture that executes the pull-after-push (PaP) dataflow directly over the BFT format, enabling fine-grained reuse and conflict-free parallel execution.

Figure 9 presents the high-level architecture of Aquila, comprising a Non-Contiguous Tiling (NCT) Engine, a Processing Element

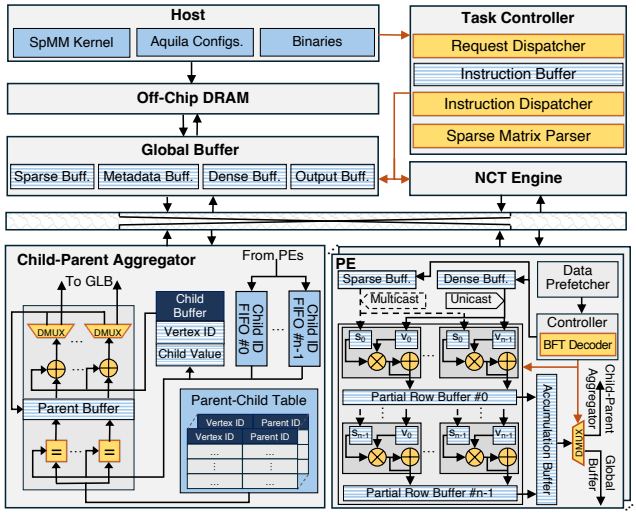


Figure 9: Overview of Aquila accelerator architecture.

(PE) Array, a Child-Parent Aggregator (CPA), and a unified Control and Instruction Dispatch subsystem. The **Global Buffer (GLB)** acts as an intermediary between off-chip DRAM and the on-chip compute fabric, provisioning operands for both the sparse matrix A and dense matrices B and C in the SpMM kernel. A crossbar interconnect connects GLB to the PE array, enabling parallel operand distribution. The **Control Unit** interfaces with the host through a request dispatcher, which compiles high-level kernel invocations into a sequence of micro-operations issued to the **Instruction Buffers**. An **Instruction Dispatcher** coordinates the issue and retirement of instructions across the NCT Engine and PE Array, maintaining execution coherence.

At runtime, the NCT Engine streams subgraphs from the input adjacency matrix into on-chip memory that fits within GLB capacity. Each subgraph is transformed to mitigate irregularity through vertex decomposition and partitioned via ADT, forming p non-contiguous tiles matched to the number of available PEs. Once the tiles are generated, they are dispatched to the PE array, where each PE is assigned one tile for processing. While the PEs execute the SpMM kernel over the current tile set using the PaP dataflow, the NCT Engine concurrently begins processing the next subgraph in a pipelined manner, ensuring sustained throughput. Operand tiles are streamed from the GLB via a crossbar fabric, and partial sums are locally accumulated in PE-side buffers. However, if a row of the output matrix corresponds to a parent vertex that receives partial contributions from multiple child vertices across different tiles, final aggregation cannot occur within a single PE. In this case, the PE forwards its partial result to the Child-Parent Aggregator (CPA), which queues and accumulates these contributions using a lightweight parent-child tracking table. Once all child contributions have been received, the CPA finalizes the reduction and flushes the result to DRAM.

This architecture enables decoupled execution of decomposition, computation, and aggregation. By pipelining tile generation and SpMM execution, and offloading inter-tile reduction to the CPA, Aquila eliminates global synchronization barriers, improves PE utilization, and maximizes locality for both data and compute across the irregular sparse computation. The Aquila accelerator

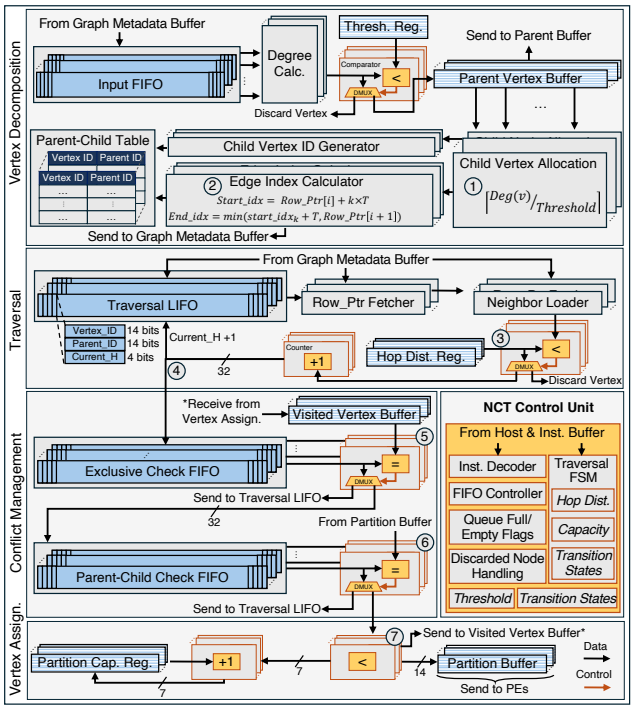


Figure 10: Non-contiguous tiling (NCT) engine architecture.

utilizes a three-stage pipeline to maximize throughput. First, the Non-Contiguous Tiling (NCT) Engine dynamically generates optimized tiles at runtime using vertex decomposition and ADT. Next, a parallel PE array processes these tiles, with each PE executing the SpMM kernel using the efficient Pull-after-Push (PaP) dataflow. A decoupled CPA then asynchronously finalizes results for decomposed vertices, avoiding synchronization stalls.

5.1 Non-contiguous Tiling (NCT) Engine

Facilitating the real-time execution of Vertex Decomposition followed by ADT is essential to generate non-contiguous tiles in a streaming manner. The engine is architected around four tightly coupled units: (1) a Vertex Decomposition Unit that identifies and splits high-degree vertices, (2) a Traversal Unit that performs bounded-depth exploration to maximize local reuse, (3) a Conflict Management Unit that enforces partition correctness via fast exclusion checks, and (4) a Vertex Assignment Unit that finalizes load-balanced tile generation. All units operate in parallel over a streamed graph representation of the sparse workload, ensuring minimal stalls and consistent throughput, which is shown in Figure 10.

5.1.1 Vertex decomposition Unit: This unit traverses the SpMM graph as described in Algorithm 1. The *Graph Metadata*, stored in CSR format, is first streamed into multiple *Input FIFOs*. Vertices are forwarded to the *Degree Calculator* module, which identifies parent vertices whose degrees exceed threshold T. A multi-bank buffer records their IDs, and the corresponding rows of matrix B—i.e., the features of parent vertices—are placed in the *Parent Vertex Buffer* in global memory. Since parent vertices are frequently accessed across tiles, caching their features reduces off-chip memory traffic during execution.

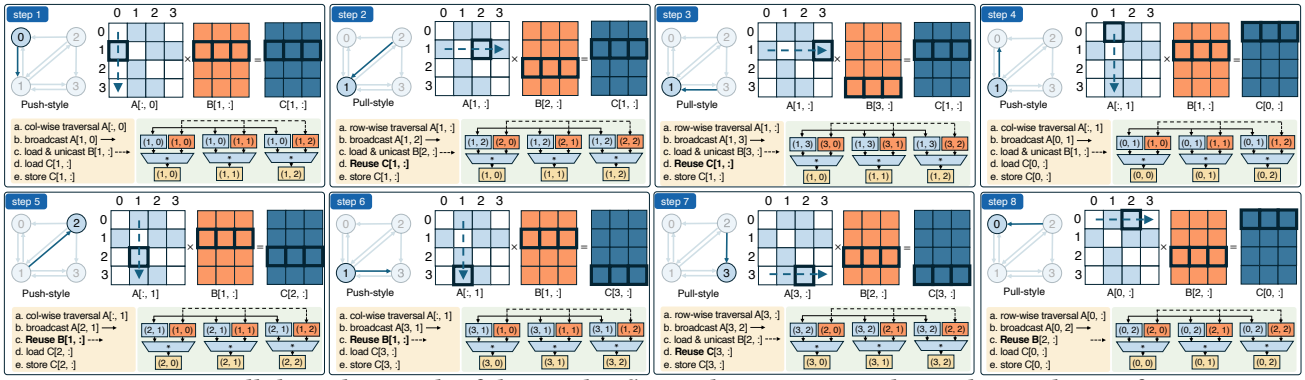


Figure 11: Walkthrough example of the PaP dataflow and matrix traversal according to the BFT format.

As shown in Figure 10 (①), decomposed vertices are passed to the *Child Vertex Allocation* module, which computes the number of child vertices required per parent based on edge count. The *Edge Index Calculator* then redistributes excess edges from each parent to its assigned children, as shown in ②. To manage dependencies between child and parent vertices efficiently, we utilize a lightweight *Parent-Child Table*. This table records each *Vertex_ID* alongside its corresponding *Parent_ID*. This organized structure ensures the accurate and efficient accumulation of partial results for parent vertices whose accumulation workload is distributed across child vertices.

5.1.2 Traversal Unit. The Traversal Unit implements the ADT algorithm to construct reuse-aware partitions via bounded-depth exploration. This unit dequeues vertices from *Traversal LIFOs*, populated by the *Graph Metadata Buffer*. Each LIFO maintains depth via a stack-like structure, with entries consisting of *Vertex_ID*, *Parent_ID*, and *current_H*. The first two fields are 14 bits, supporting subgraphs with up to 16,384 vertices; *current_H* is 4 bits, allowing traversal depths up to 16. Empirically, SpMM-abstracted graphs in many domains exhibit subgraph diameters rarely exceeding 5 [39, 52], keeping traversal within this bound.

Each LIFO is assigned to a distinct graph partition. To initiate traversal, it dispatches the top vertex to the *Row_Ptr Fetcher*, which retrieves the start and end offsets for its adjacency list. The *Neighbor Loader* then streams the neighbors and checks whether their *current_H* exceeds the limit stored in the *Hop Distance Register* ③. Vertices exceeding the limit are dropped. At ④, valid neighbors increment their *current_H*, are pushed back into the LIFO for continued traversal, and forwarded to the *Exclusive Check FIFOs* for conflict handling before final partition assignment.

5.1.3 Conflict Management Unit: This unit enforces the conflict rules described in Section 3.1.4. At ⑤, each vertex ID is checked against the *Visited Vertex Buffer* to ensure exclusivity within partitions; duplicates are discarded, while valid entries proceed to the *Parent-Child Check FIFOs*. At ⑥, the unit checks for sibling and parent-child conflicts by comparing a child's *Parent_ID* with the parent IDs in its partition's *Partition Buffer*. Child vertices that share a parent or sibling within the same partition are dropped.

5.1.4 Vertex Assignment Unit: At ⑦, a capacity check verifies whether the partition can accommodate the incoming vertex, using limits defined in the *Partition Capacity Register*. Vertices that exceed capacity are discarded; otherwise, they are added to the *Partition Buffer* and logged in the *Visited Vertex Buffer*, where each entry uses 14 bits for the *Vertex_ID*. Once partitioning completes, all vertices within a partition are mapped to a dedicated processing element. Additionally, the *NCT Engine Controller* handles metadata requests from the host and issues instructions from the *Instruction Buffer*, configuring traversal parameters such as *Hop Distance*, *Partition Capacity*, and transition logic. It also coordinates control signals for multiplexers and manages FIFO load/store operations.

5.2 PE Microarchitecture for PaP Dataflow

Each Processing Element (PE) in Aquila is designed to natively support the proposed Pull-after-Push (PaP) dataflow through a hardware-software co-optimized microarchitecture. At the core of this design is a dual-buffer interface, consisting of a *Sparse Buffer* and a *Dense Buffer*, that respectively ingest the compressed sparse tile (encoded in BFT format) and the corresponding dense matrix rows. During the **Push** phase, the PE exploits reuse by multicasting the nonzeros located in the same column of the sparse matrix to a MAC array cascade, while spatially pinning the corresponding dense row of matrix B across the MAC lanes. This spatial pinning ensures that the dense row $B[j,:]$ is reused across multiple MAC operations without reloading it from the buffer, as all nonzeros $A[i,j]$ sharing the same j index are consumed in a single phase. This mode is highly effective at amortizing the cost of accessing B.

Upon completion of the column traversal, the PE transitions to the **Pull** phase. Here, the PE reorients to a row-wise aggregation model where it processes the partial sums corresponding to $C[i,:]$. These partial results, previously generated during the Push phase and stored in the Partial Row (PR) buffer, are incrementally updated as the PE traverses additional nonzero elements in the same row of A (i.e., $A[i,:]$). This enables localized accumulation of output values without global synchronization or redundant memory movement. The bidirectional traversal logic in the PE is driven entirely by the BFT-encoded coordinates, which expose both the vertical and horizontal fiber views within a tile. Once a row's aggregation is finalized—either within the PE or across PEs via child-parent reduction—the result is flushed to the output buffer or forwarded to the CPA for final accumulation. Overall, the PE architecture allows seamless integration of graph-abstracted reuse into dense computation pipelines, achieving high MAC utilization and minimizing buffer pressure across a range of sparsity regimes.

Walkthrough Example: Figure 11 shows an 8-step execution of the proposed PaP dataflow on a 1D MAC array of size 3. Step

1 begins at coordinate $(0, 0)$ in the A tile. The nonzero at $A[1, 0]$ triggers a push-style execution: the value is broadcast across the MAC array, and $B[1, :]$ elements are unicasted to compute $C[1, :]$. Steps 2 and 3 switch to pull-style. PaP searches for nonzeros in A whose row indices match that of $A[1, 0]$, enabling reuse of the active output row $C[1, :]$ to accumulate remaining partial sums. Step 4 returns to push-style to process $A[0, 1]$, initiating computation of $C[0, :]$ with $B[1, :]$. Steps 5 and 6 process $A[2, 1]$ and $A[3, 1]$, which share the same column index as $A[0, 1]$. The dataflow reuses the already loaded $B[1, :]$ without re-fetch. Step 7 switches back to pull-style. Starting from the last element in $A[:, 1]$, it enables the reuse of row $C[3 :]$. Step 8 concludes the pull phase by consuming remaining nonzeros in rows corresponding to previously active column indices, finalizing the computation of partial sums in C. The dataflow dynamically alternates between push and pull styles to maximize reuse of both B and C operands.

5.3 Child-Parent Aggregator Unit

The *Child-Parent Aggregator (CPA)* is a specialized unit responsible for accumulating partial sums from decomposed child vertices to their corresponding parent vertices. As shown in Figure 9, this decouples irregular inter-tile reductions from the PE datapath and avoids cross-PE synchronization. During SpMM execution, each PE processes a non-contiguous tile (subgraph). If a vertex is identified as child within the PE logic, the partial output vector $C_{i,:}$ is redirected to the CPA unit rather than being finalized. The CPA receives partial sums tagged with both the child and parent vertex IDs. It uses the *Parent-Child Table* to aggregate all partial results corresponding to a given parent vertex. These partial sums are buffered in the *Parent Buffer*, which holds in-flight reductions. To track completion, each parent vertex is associated with a counter that reflects the expected number of child contributions—determined statically from the decomposition process. Upon receipt of each partial sum, the CPA updates the accumulator entry and decrements the counter. Once all children have reported their results, the final accumulated output is flushed to the output buffer and written back to DRAM.

By offloading parent-child reductions to the CPA, Aquila allows each PE to process tiles independently, alleviating the inter-PE synchronization requirements. This avoids introducing irregular accumulation logic within the PE datapath. Moreover, this design ensures correctness in the presence of decomposed vertices, while preserving scalability by bounding buffer requirements and tracking logic to only the subset of high-degree vertices identified during tiling. However, the mixed row- and column-wise access patterns of the PaP dataflow might lead to bank conflicts within the GLB. A high-degree vertex needed by multiple PEs simultaneously could cause bank conflicts, as concurrent requests to the same bank force serialization, stalling PEs and reducing memory-level parallelism. Aquila mitigates this through Vertex Decomposition, which splits the workload of high-degree vertices into logical child vertices. The ADT algorithm then assigns these children to non-contiguous tiles mapped to different PEs, naturally spreading memory requests across GLB banks and avoiding contention. During accumulation, instead of multiple PEs issuing conflicting write operations to the same bank in the GLB, each PE forwards its partial result to a dedicated CPA unit. This shifts the many-to-one write-back bottleneck into independent transfers handled asynchronously by the CPA.

Table 1: Evaluated dataset properties

Dataset	Dim.	Application	Symm.	NNZ	E2V
Delaunay n24 (DELL)	16.78M	Numerical Simulations	Yes	100.66M	6.0
wiki-topcats (WIKI)	1.79M	Wikipedia Hyperlinks	No	28.51M	15.9
mycielskian17 (MYC)	0.10M	Graph Coloring	Yes	100.25M	1023.0
Serena (SER)	1.39M	Geomechanical Modeling	Yes	64.13M	46.1
coPapersCiteseer (PAP)	0.43M	Citation Co-authorship	Yes	32.07M	73.9
GAP-road (GAP)	23.95M	Road Network	Yes	57.71M	2.4
Reddit (RED)	0.23M	Social Media	Yes	114.62M	492.8
PubMed (PMED)	0.02M	Citation Network	Yes	0.09M	4.5
MInference 1.0 (MIN)	0.12M	Machine Learning	No	1,638.4M	12,800
SeerAttention (SEE)	0.03M	Machine Learning	No	122.88M	3,840

Offloading irregular accumulation preserves GLB bandwidth for predictable operand fetches and maintains high throughput across the PE array. Once all child contributions for a parent vertex are received, the CPA performs a single final write to the output buffer, minimizing GLB write traffic.

6 Evaluation

6.1 Simulation Setup

Configurations: Simulation Setup. We built a cycle-accurate simulator for Aquila, following methodologies from prior work [34, 55, 59]. The simulator integrates with Ramulator [30] to model HBM with 256 GB/s bandwidth and captures the cycle-level behavior of all compute and memory components. Aquila comprises 32 PEs, each with a 4×8 FP32 MAC array (1K MACs total), operating at 1 GHz. The 1.25 MB global buffer includes 512 KB sparse, 512 KB dense, and 256 KB auxiliary storage. Local PE buffers sum to 35 KB. For ASIC evaluation, we synthesize in Verilog using TSMC 32nm with Synopsys Design Compiler. Switching activity is captured via waveform traces and analyzed using PrimeTime PX. On-chip buffer energy and area are modeled with CACTI 7.0 [4].

Datasets. We evaluate Aquila using eight datasets spanning numerical simulation (DEL, SER), network analysis (WIKI, PAP), graph algorithms (GAP, MYC [10]), and GNN workloads (RED [19], PMED [48]). These datasets vary in sparsity ratio (7.14×10^{-6} to 2.11×10^{-3}) and structure (i.e., symmetric and asymmetric), providing a broad benchmark suite. Further, to assess Aquila at moderate sparsity (10%), we include two sparse matrices from intermediate attention maps of SeerAttention [13] and MInference [27] LLMs. Dataset properties are summarized in Table 1.

Baseline Platforms: We evaluate Aquila against three state-of-the-art SpMM accelerators—Sextans [49], SPADE [17], and HoTtiles [16], and two GNN accelerators, I-GCN [15] and ReGNN [7], whose aggregation phases performs SpMM. To ensure fairness, we isolate and simulate only the aggregation kernel for both GNN accelerators and match their architectural parameters with Aquila. For I-GCN [15], we replicate the islandization mechanism for streaming dense tiles and also replicate its redundancy elimination mechanism. For ReGNN, we implement the redundancy elimination methodology [26]. For SpMM accelerators, Sextans utilizes out-of-order scheduling within rows to balance workload, mitigate RAW dependencies, and optimize pipeline utilization. SPADE implements tile-based scheduling with barrier synchronization, and we replicated its bypass buffers, which avoid cache interactions. HoTtiles applies an analytical model exploiting intra-matrix heterogeneity [16], partitioning matrices into dense regions processed by compute-intensive "Hot Workers" (analogous to Sextans' PEs) and sparse

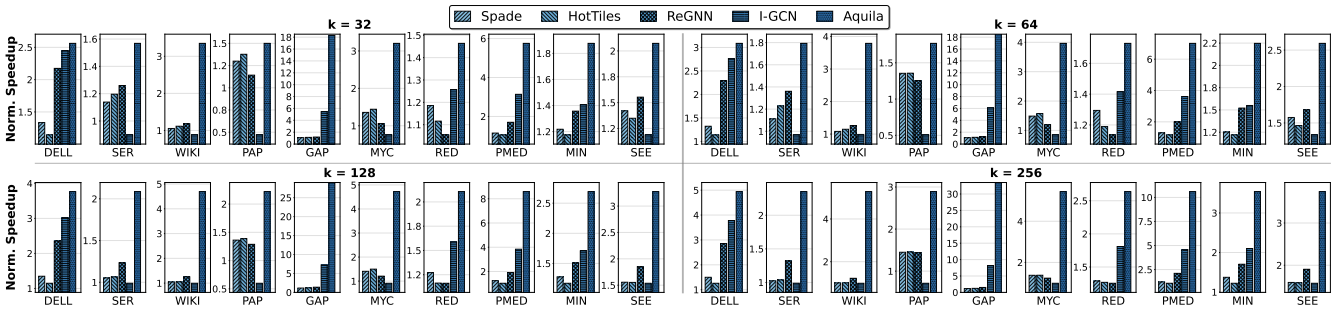


Figure 12: Normalized speedup over Sextans with varying dense input matrix dimension (k) across datasets.

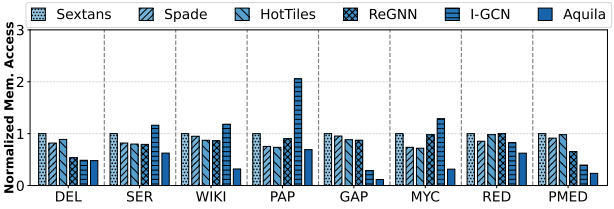


Figure 13: Off-chip memory access normalized to Sextans.

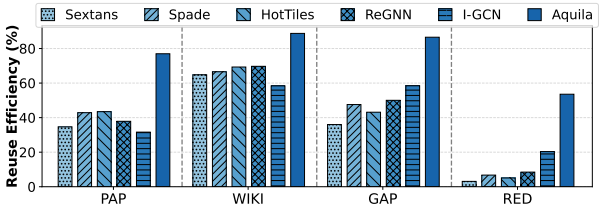


Figure 14: Data Reuse Comparison of Aquila vs. prior works.

regions handled by latency-tolerant "Cold Workers" (similar to SPADE's PEs). For a comprehensive comparison, Aquila's evaluation incorporates kernel execution along with host-to-device and device-to-host memory transfer times. For a fair comparison, all designs are scaled to a 1K-MAC array organized into 32 PEs, running at a projected 1 GHz. The GLB size is uniformly set to 1.25 MB. We also replicate the architecture and dataflow of the baselines as described in their work.

6.2 Performance Analysis

Figure 12 presents the normalized speedup of Aquila and the baselines relative to Sextans across four feature dimensions: $K \in \{32, 64, 128, 256\}$. The results demonstrate Aquila's consistent performance advantages across diverse sparsity patterns. Averaged across all datasets and dimensions, Aquila achieves 4.88× speedup. Aquila exhibits improving relative performance as K increases. This trend stems from Aquila's core design principles: vertex decomposition regularizes high-degree vertices that would otherwise create bottlenecks at larger K, while ADT-generated non-contiguous tiles expose reuse patterns that become more valuable as arithmetic intensity rises. The PaP dataflow amplifies both B-row and C-row locality simultaneously, allowing these reuse opportunities to effectively amortize growing DRAM traffic. In contrast, coordinate-aligned tiling schemes and rigid dataflows in prior work fail to capture such reuse potential as K grows.

On GAP, Aquila demonstrates exceptional dominance with speedups growing from 18.3× at $K = 32$ to 33.5× at $K = 256$ over Sextans. This low-density dataset challenges traditional approaches: ReGNN's redundancy elimination provides limited benefit when edge density is insufficient for vertex clustering, while HotTiles' heterogeneity-aware partitioning finds few dense regions to exploit. Aquila's vertex decomposition and ADT, however, discover and co-locate reuse opportunities within sparse neighborhoods, maintaining high utilization even in low-density regimes. Conversely, on MYC, where higher connectivity enables more effective traditional optimizations, Aquila's advantages are consistent: 3.2× to 5.4× across K values. Here, I-GCN's islandization and ReGNN's redundancy reduction gain traction due to abundant vertex connectivity, yet Aquila's unified approach to both input and output reuse through PaP maintains superior performance. As K increases, the reuse quantification in Equation 1 becomes more impactful, since two-hop pathways expose greater data sharing opportunities. This theoretical prediction aligns with the empirical observation that Aquila's relative advantage strengthens with feature dimension growth, while matrix-centric approaches plateau or degrade.

6.3 Off-Chip Memory Access

Figure 13 shows normalized off-chip memory accesses across accelerators (normalized to Sextans). On average, among SpMM accelerators, Aquila reduces off-chip memory accesses by 3.23×, 2.85×, and 2.81× over Sextans, SPADE, and HotTiles, respectively. It also outperforms ReGNN and I-GCN with 2.67× and 2.38× reductions.

In RED and PMED, Sextans, SPADE, and HotTiles incur higher off-chip memory accesses, as techniques like NNZ scheduling, tile-based strategies, and hot/cold region separation are less effective on power-law sparse matrices like RED and PMED. ReGNN reduces memory traffic by caching frequently shared vertex sets, common in such graphs. I-GCN improves locality by colocating high-degree vertices with its neighbors, mitigating irregular accesses. However, GNN accelerators face limitations: in PAP and MYC, their off-chip accesses exceed those of SpMM accelerators on average due to unpredictable sparsity and corresponding access patterns. Although prior work employs sophisticated tiling and scheduling, these techniques react inconsistently to sparsity variations, yielding unstable memory behavior. Aquila overcomes this by using a parent buffer to cache high-degree vertices on-chip, consistently reducing off-chip accesses—especially for vertices spanning multiple tiles, thus improving memory efficiency across diverse datasets.

In addition, we tile the Cora graph [60] into 64 × 64 tiles. Figure 15(b) shows the share of ineffectual (all-zero) tiles after vertex decomposition and ADT. Only 17 of ∼ 3K vertices exceed the degree threshold, so duplication overhead is <0.005%. ADT partitions vertices by connectivity rather than coordinate order, forming reuse-rich effectual tiles while isolating sparse, ineffectual ones for

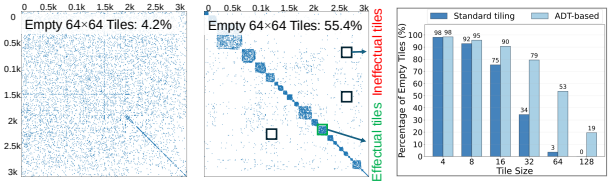


Figure 15: vertex decomposition and ADT performance.

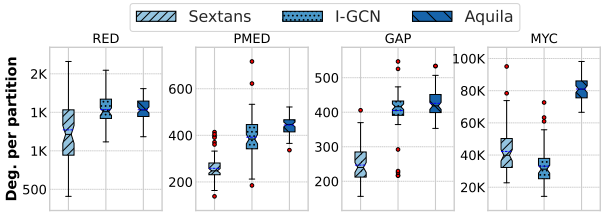


Figure 16: Average and range of the workload (sum of degree) across partitions (PEs).

elimination. As a result, ≈ 55% of tiles are ineffectual, far above the 4.2% under conventional tiling [16, 17, 49] (Figure 15(a)). Figure 15(c) confirms the advantage across tile sizes, noting that larger tiles naturally yield fewer fully ineffectual tiles due to decreased sparsity granularity.

6.4 Data Reuse Analysis

We compute on-chip data reuse as the number of accesses per row of the dense input (B) and output (C) matrices in SpMM before eviction, expressed as a percentage of the theoretical reuse assuming an unlimited GLB without tiling. As shown in Figure 14, Aquila achieves 72.12% of theoretical reuse on average. It improves reuse by 32.2% over prior work on average, outperforming HotTiles and I-GCN by 31.8% and 29.9%, respectively. This stems from vertex decomposition, which records parent and child reuse in different tiles; adaptive depth traversal, which aligns NNZs across non-contiguous tiles; and push-after-pull dataflow, which maximizes reuse within each tile for both dense and output matrices.

6.5 Workload Balance

Figure 16 compares workload balance across datasets, focusing on the distribution of NNZ elements per PE. Aquila demonstrates a significantly more balanced degree distribution compared to Sextans and I-GCN. While the mean degree per partition for Aquila (1548.94) is similar to I-GCN (1544.00), its standard deviation is much smaller at 105.44, compared to 175.51 for I-GCN and 336.87 for Sextans. This indicates that Aquila's partition degrees are tightly clustered around the mean, ensuring more uniformity. In contrast, the higher standard deviations in Sextans and I-GCN reflect greater variability, with some partitions having disproportionately high or low workloads. Sextans lacks a mechanism to handle high-degree vertices, making its workload balance highly dependent on the graph's degree distribution. I-GCN exacerbates this imbalance by clustering high-degree vertices into a few partitions using a BFS strategy, which creates dense regions while leaving other partitions sparsely connected compared to early-forming partitions. Aquila's superior balance stems from its vertex decomposition strategy, which resolves skewed degree distributions by ensuring vertices

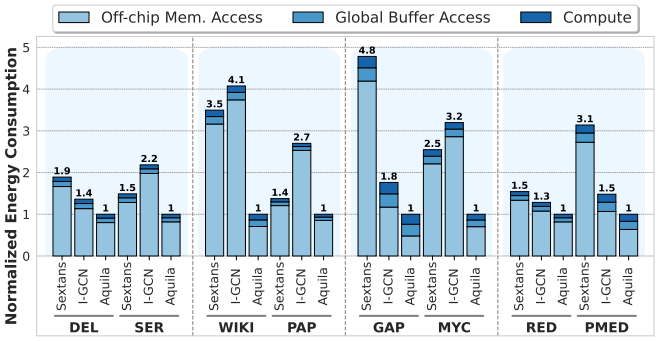


Figure 17: Energy consumption normalized to Aquila.

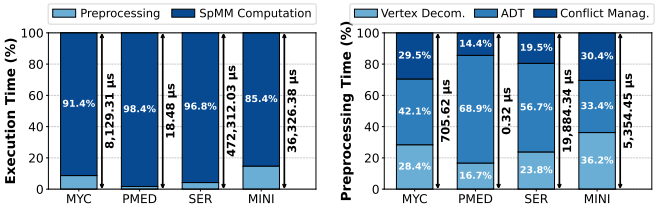


Figure 18: Preprocessing breakdown in Aquila.

have equalized degrees regardless of partitioning, leading to better workload balance.

6.6 Energy Efficiency

We compare the energy consumption of Aquila with Sextans and I-GCN, two representative baselines, as shown in Figure 17. Across all datasets, Aquila reduces energy consumption by an average of 60.5% and 55.7% when compared to Sextans and I-GCN, respectively. The reduction is attributed to Aquila's strategy of retaining frequently accessed vertices in parent buffer, achieving an order-of-magnitude reduction of off-chip memory access. Additionally, the ADT algorithm increases the edge-to-vertex ratio per PE, ensuring that neighboring vertices are often processed within the same PE, thereby minimizing on-chip data movement. Moreover, distributing child vertices across multiple PEs as replicas of frequently accessed vertices significantly decreases global buffer access.

6.7 Preprocessing Analysis

To assess preprocessing overhead, Figure 18 (left) shows it accounts for an average of 7.34% of total runtime across all datasets, demonstrating the NCT engine's efficiency in generating tiles before SpMM execution. As density increases (e.g. in the MIN dataset with 10% sparsity), preprocessing reaches 14.74%. The overhead grows due to the longer traversal required for vertex decomposition and ADT in denser matrices. Further, to analyze preprocessing overhead across Vertex Decomposition, ADT, and Conflict Management units, we use the Edge-to-Vertex (E2V) ratio as the key indicator, as shown in Figure 18 (right). PMED, with a low E2V of 4.5, spends 16.7% of time on vertex decomposition, since few vertices exceed the maximum degree threshold. In contrast, MIN's extreme E2V of 12,800 leads to 36.2% spent on decomposition, as more vertices exceed the degree threshold, and 30.4% on conflict management due to complex parent-child structures and denser connectivity. SER, with a moderate E2V of 46.1, achieves balanced overhead across phases thanks to its regular geometric structure. Overall,

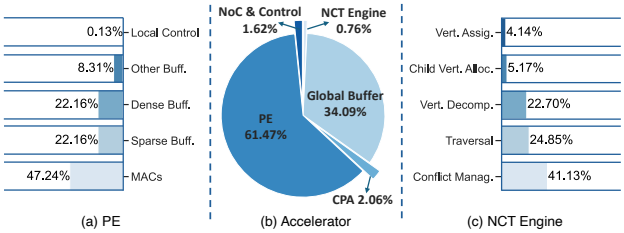


Figure 19: Area Breakdown.

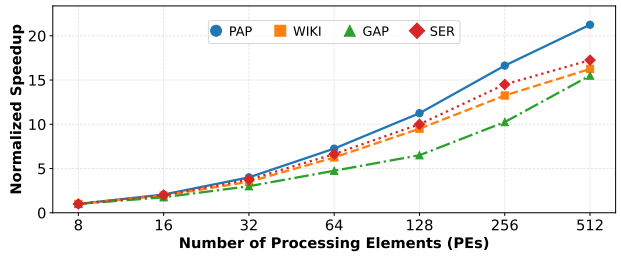


Figure 20: Normalized speedup for varying number of PEs.

LLM workloads incur the highest preprocessing costs (≈ 13%) due to dense connectivity and SpMM kernels involving a graph (e.g., PMED) benefit from efficient traversal with minimal decomposition due to their community-based structures.

6.8 Area Consumption Analysis

Figure 19 shows the area distribution of Aquila's accelerator. The global buffer accounts for 34.09% and the PE array uses 61.47% of the total chip area. In contrast, the CPA and NCT engine together occupy only about 2.82%, highlighting the minimal area overhead of Aquila's auxiliary hardware components. The detailed area breakdown for the PE is illustrated in Figure 19(a), where buffers consume 52.63% and MACs array uses 47.24% of the PE area. Additionally, Figure 19(c) provides a breakdown of the NCT engine's area. The *Conflict Management Unit* dominates the NCT engine's area, occupying about 41.13%. This unit includes buffers and FIFOs to stage vertex streams, handle potential conflicts during the ADT process, and assign vertices to partitions.

6.9 Sensitivity, Scalability and Ablation Study

Sensitivity Study on GLB Size. Figure 21 shows how GLB size scaling affects off-chip memory access across datasets. We observe an inverse relationship between the density of the sparse matrix and the benefit from GLB scaling: datasets with higher density see less reductions in memory access as GLB size increases. For example, RED, with a 492 edge to vertex (E2V) ratio, achieves 1 to 4.2× improvement when scaling GLB from 1MB to 8MB, whereas GAP, the dataset with the largest dimension 23.95M and an E2V ratio of 2.4, sees a 1 to 6.0× gain. This is because the less density of this dataset reduces number of dependency between the partitions (tiles) that should be loaded to the on-chip memory each time.

Scalability Study on PE Numbers. Figure 20 illustrates the normalized speedup of four representative datasets when scaling from 8 to 512 PEs in Aquila. The results demonstrate a correlation between graph density and scalability. PAP, with the highest edge-to-vertex ratio (E2V=73.9), achieves the best scaling with 20× speedup at 512 PEs, while the sparse GAP dataset reaches only

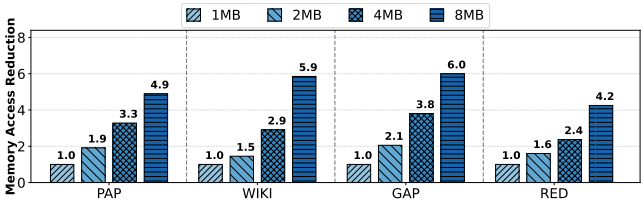


Figure 21: Memory access reduction with varying GLB size (Normalized to the memory access count of GLB = 1MB).

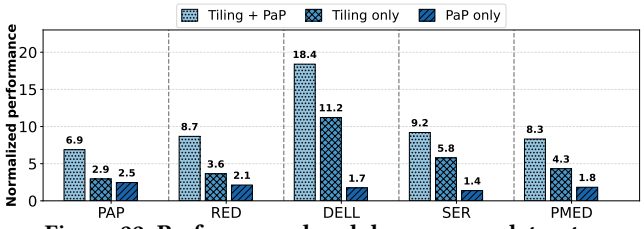


Figure 22: Performance breakdown across datasets.

15× speedup. WIKI and SER datasets exhibit intermediate scaling behavior proportional to their respective densities, with SER outperforming WIKI across all PE configurations.

Ablation Study. We performed an ablation study to evaluate the individual benefits of non-contiguous tiling and PaP dataflow, as well as their combined effectiveness, measuring improvements over a baseline implementation using simple index-based tiling with pull-based dataflow, as shown in Figure 22. Our results demonstrate that the combined approach achieves an average of 10.30× performance improvement across all evaluated datasets, significantly outperforming tiling-only (5.58× average) and PaP-only (1.91× average) implementations. The superior performance of PaP dataflow when combined with non-contiguous tiling, compared to its standalone application, stems from the increased tile density achieved through non-contiguous tiling, as illustrated in Figure 16.

7 Conclusion

In this paper, we propose a set of algorithms built on top of a graph abstraction framework to reengineer SpMM kernel execution. We also introduce *Aquila*, a specialized accelerator that supports these graph transformations in real time. Algorithmically, we reinterpret SpMM tiling and reuse using its graph abstraction beyond traditional loop transformations. We propose a *non-contiguous tiling* technique combined with a novel *pull-after-push* dataflow, which enhances both temporal and spatial reuse across all matrices, eliminates partial result accumulation and write conflicts, and ensures balanced workloads. Architecturally, we design a *Bidirectional Fiber Tree* format to match the access patterns of the pull-after-push dataflow, replacing rigid row- or column-major dataflows. Built on this foundation, Aquila supports diverse SpMM kernels with varying dimensions and sparsity patterns under a unified architecture. Simulation results show that Aquila achieves average speedups of 4.3×, 3.4×, 3.7×, 2.9×, and 2.7× reductions in execution time and up to 4.8× improvements in energy efficiency across multiple sparse datasets, compared to state-of-the-art accelerators [7, 15–17, 49].

Acknowledgments

This work is supported by the U.S. National Science Foundation under CAREER Award CCF-2441973.

References

[1] Khalid Ahmad, Anand Venkat, and Mary Hall. 2016. Optimizing LOBPCG: Sparse matrix loop and data transformations in action. In *Proceedings of International Workshop on Languages and Compilers for Parallel Computing (LCPC)*. Springer, 218–232.

[2] Hasan Metin Aktulga, Aydin Buluç, Samuel Williams, and Chao Yang. 2014. Optimizing sparse matrix-multiple vectors multiplication for nuclear configuration interaction calculations. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 1213–1222.

[3] Vignesh Balaji, Neal C Crago, Aamer Jaleel, and Stephen W Keckler. 2023. Community-based matrix reordering for sparse linear algebra optimization. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 214–223.

[4] Rajeev Balasubramonian, Andrew B Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New tools for interconnect exploration in innovative off-chip memories. *ACM Transactions on Architecture and Code Optimization (TACO)* 14, 2 (2017), 1–25.

[5] Nathan Bell, Steven Dalton, and Luke N Olson. 2012. Exposing fine-grained parallelism in algebraic multigrid methods. *SIAM Journal on Scientific Computing* 34, 4 (2012), C123–C152.

[6] 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 (STOC)*. ACM, 590–598.

[7] Cen Chen, Kenli Li, Yangfan Li, and Xiaofeng Zou. 2022. ReGNN: A redundancy-eliminated graph neural networks accelerator. In *Proceedings of IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 429–443.

[8] Guohao Dai, Guyue Huang, Shang Yang, Zhongming Yu, Hengrui Zhang, Yufei Ding, Yuan Xie, Huazhong Yang, and Yu Wang. 2022. Heuristic adaptability to input dynamics for spmm on gpus. In *Proceedings of ACM/IEEE Design Automation Conference (DAC)*. IEEE, 595–600.

[9] Timothy A Davis, John R Gilbert, Stefan I Larimore, and Esmond G Ng. 2004. A column approximate minimum degree ordering algorithm. *ACM Transactions on Mathematical Software (TOMS)* 30, 3 (2004), 353–376.

[10] 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.

[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 ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 253–267.

[12] Sanjay Gandham, Lingxiang Yin, Hao Zheng, and Mingjie Lin. 2023. SAGA: Sparsity-Agnostic Graph Convolutional Network Acceleration with Near-Optimal Workload Balance. In *Proceedings of IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 1–9.

[13] Yizhao Gao, Zhichen Zeng, Dayou Du, Shijie Cao, Peiyuan Zhou, Jiaxing Qi, Junjie Lai, Hayden Kwok-Hay So, Ting Cao, Fan Yang, et al. 2024. Seerattention: Learning intrinsic sparse attention in your llms. *arXiv preprint arXiv:2410.13276* (2024).

[14] Tong Geng, Ang Li, Runbin Shi, Chunshu Wu, Tianqi Wang, Yanfei Li, Pouya Haghi, Antonino Tumeo, Shuai Che, Steve Reinhardt, and Martin C. Herbordt. 2020. AWB-GCN: A graph convolutional network accelerator with runtime workload rebalancing. In *Proceedings of IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 922–936.

[15] Tong Geng, Chunshu Wu, Yongan Zhang, Cheng Tan, Chenhao Xie, Haoran You, Martin Herbordt, Yingyan Lin, and Ang Li. 2021. I-GCN: A graph convolutional network accelerator with runtime locality enhancement through islandization. In *Proceedings of IEEE/ACM international symposium on microarchitecture (MICRO)*. IEEE, 1051–1063.

[16] Gerasimos Gerogiannis, Sriram Aananthakrishnan, Josep Torrellas, and Ibrahim Hur. 2024. HotTiles: Accelerating SpMM with Heterogeneous Accelerator Architectures. In *Proceedings of IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 1012–1028.

[17] Gerasimos Gerogiannis, Serif Yesil, Damitha Lenadora, Dingyuan Cao, Charith Mendis, and Josep Torrellas. 2023. Spade: A flexible and scalable accelerator for spmm and sddmm. In *Proceedings of ACM/IEEE International Symposium on Computer Architecture (ISCA)*. IEEE, 1–15.

[18] John R Gilbert, Steve Reinhardt, and Viral B Shah. 2006. High-performance graph algorithms from parallel sparse matrices. In *Proceedings of International Workshop on Applied Parallel Computing (PARA)*. Springer, 260–269.

[19] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *Proceedings of International Conference on Neural Information Processing Systems (NIPS)*. ACM, 1025–1035.

[20] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

[21] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W Fletcher. 2019. Extensor: An accelerator for sparse tensor algebra. In *Proceedings of IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 319–333.

[22] Kartik Hegde, Jiyong Yu, Rohit Agrawal, Mengjia Yan, Michael Pellauer, and Christopher Fletcher. 2018. UCNN: Exploiting computational reuse in deep neural networks via weight repetition. In *Proceedings of ACM/IEEE International Symposium on Computer Architecture (ISCA)*. IEEE, 674–687.

[23] Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and P Sadayappan. 2019. Adaptive sparse tiling for sparse matrix multiplication. In *Proceedings of ACM SIGPLAN symposium on principles and practice of parallel programming (PPoPP)*. ACM, 300–314.

[24] Yi-Jou Hsiao, Chin-Fu Nien, and Hsiang-Yun Cheng. 2021. ReSpar: Reordering Algorithm for ReRAM-based Sparse Matrix-Vector Multiplication Accelerator. In *Proceedings of IEEE 39st International Conference on Computer Design (ICCD)*. IEEE, 260–268.

[25] Ranggi Hwang, Minhoo Kang, Jiwon Lee, Dongyun Kam, Youngjoo Lee, and Minsoo Rhu. 2022. GROW: A Row-Stationary Sparse-Dense GEMM Accelerator for Memory-Efficient Graph Convolutional Neural Networks. In *Proceedings of IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 42–55.

[26] Zhihao Jia, Sina Lin, Rex Ying, Jiaxuan You, Jure Leskovec, and Alex Aiken. 2020. Redundancy-free computation for graph neural networks. In *Proceedings of ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD)*. ACM, 997–1005.

[27] Huiqiang Jiang, Yucheng Li, Chengruidong Zhang, Qianhui Wu, Xufang Luo, Surin Ahn, Zhenhua Han, Amir H Abdi, Dongsheng Li, Chin-Yew Lin, et al. 2024. Minference 1.0: Accelerating pre-filling for long-context llms via dynamic sparse attention. *arXiv preprint arXiv:2407.02490* (2024).

[28] 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 ACM SIGPLAN symposium on principles and practice of parallel programming (PPoPP)*. ACM, 376–388.

[29] Jinkwon Kim, Myeongjae Jang, Haejin Nam, and Soontae Kim. 2023. HARP: Hardware-Based Pseudo-Tiling for Sparse Matrix Multiplication Accelerator. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 1148–1162.

[30] Yoongu Kim, Weikun Yang, and Onur Mutlu. 2015. Ramulator: A Fast and Extensible DRAM Simulator. *IEEE Computer architecture letters* 15, 1 (2015), 45–49.

[31] Paolo Sylos Labini, Massimo Bernaschi, Werner Nutt, Francesco Silvestri, and Flavio Vella. 2022. Blocking Sparse Matrices to Leverage Dense-Specific Multiplication. In *Proceedings of IEEE/ACM Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. IEEE, 19–24.

[32] Eunji Lee, Yoonsang Han, and Gordon Euhyun Moon. 2024. Accelerated block-sparsity-aware matrix reordering for leveraging tensor cores in sparse matrix-multivector multiplication. In *Proceedings of European Conference on Parallel Processing (Euro-Par)*. Springer, 3–16.

[33] Jiajun Li, Ahmed Louri, Avinash Karanth, and Razvan Bunescu. 2021. GCNAX: A flexible and energy-efficient accelerator for graph convolutional neural networks. In *Proceedings of IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 775–788.

[34] 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 ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 747–761.

[35] Zewen Li, Fan Liu, Wenjie Yang, Shouheng Peng, and Jun Zhou. 2021. A survey of convolutional neural networks: analysis, applications, and prospects. *IEEE transactions on neural networks and learning systems* 33, 12 (2021), 6999–7019.

[36] Xiaoyang Lu, Boyu Long, Xiaoming Chen, Yinhe Han, and Xian-He Sun. 2024. ACES: Accelerating Sparse Matrix Multiplication with Adaptive Execution Flow and Concurrency-Aware Cache Optimizations. In *Proceedings of ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 71–85.

[37] Chris Mueller. 2004. Sparse matrix reordering algorithms for cluster identification. *Machune Learning in Bioinformatics* 23 (2004).

[38] 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 ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 252–265.

[39] Mark EJ Newman, Duncan J Watts, and Steven H Strogatz. 2002. Random graph models of social networks. *Proceedings of the national academy of sciences* 99, suppl_1 (2002), 2566–2572.

[40] Michael K Ng and Zhaochen Zhu. 2019. Sparse matrix computation for air quality forecast data assimilation. *Numerical Algorithms* 80 (2019), 687–707.

[41] Israt Nisa, Aravind Sukumaran-Rajam, Sureyya Emre Kurt, Changwan Hong, and P Sadayappan. 2018. Sampled dense matrix multiplication for high-performance machine learning. In *Proceedings of 25th International Conference on High Performance Computing (HiPC)*. IEEE, 32–41.

[42] Amir Hossein Nodehi Sabet, Junqiao Qiu, and Zhijia Zhao. 2018. Tigr: Transforming irregular graphs for gpu-friendly graph processing. In *Proceedings of ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 622–636.

[43] S. Pal, J. Beaumont, D. Park, A. Amarnath, S. Feng, C. Chakrabarti, H. Kim, D. Blaauw, T. Mudge, and R. Dreslinski. 2018. OuterSPACE: An outer product based sparse matrix multiplication accelerator. In *Proceedings of IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 724–736.

[44] 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)*. IEEE, 724–736.

[45] Xavier Pinel and Marc Montagnac. 2013. Block Krylov methods to solve adjoint problems in aerodynamic design optimization. *AIAA journal* 51, 9 (2013), 2183–2191.

[46] Yingjie Qi, Jianlei Yang, Ao Zhou, Tong Qiao, and Chunming Hu. 2023. Architectural implications of GNN aggregation programming abstractions. *IEEE Computer Architecture Letters* 23, 1 (2023), 125–128.

[47] 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 *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 58–70.

[48] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. 2008. Collective classification in network data. *AI magazine* 29, 3 (2008), 93–93.

[49] 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 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM, 65–77.

[50] 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 *Proceedings of IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 766–780.

[51] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. 2017. Efficient processing of deep neural networks: A tutorial and survey. *Proc. IEEE* 105, 12 (2017), 2295–2329.

[52] Johan Ugander, Brian Karrer, Lars Backstrom, and Cameron Marlow. 2011. The anatomy of the facebook social graph. *arXiv preprint arXiv:1111.4503* (2011).

[53] Richard W Vuduc and Hyun-Jin Moon. 2005. Fast sparse matrix-vector multiplication by exploiting variable block structure. In *Proceedings of High Performance Computing and Communications (HPCC)*. Springer, 807–816.

[54] Bradley Worley and Robert Powers. 2015. Deterministic multidimensional nonuniform gap sampling. *Journal of magnetic resonance* 261 (2015), 19–26.

[55] Yannan Nellie Wu, Po-An Tsai, Angshuman Parashar, Vivienne Sze, and Joel S Emer. 2022. Sparseloop: An analytical approach to sparse tensor accelerator modeling. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1377–1395.

[56] Mingyu Yan, Lei Deng, Xing Hu, Ling Liang, Yujing Feng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie. 2020. HyGCN: A GCN accelerator with hybrid architecture. In *Proceedings of IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 15–29.

[57] Jiaqi Yang, Hao Zheng, and Ahmed Louri. 2022. Adapt-flow: A flexible dnn accelerator architecture for heterogeneous dataflow implementation. In *Proceedings of the Great Lakes Symposium on VLSI (GLSVLSI)*. ACM, 287–292.

[58] Jiaqi Yang, Hao Zheng, and Ahmed Louri. 2024. Aurora: A versatile and flexible accelerator for graph neural networks. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 890–902.

[59] Yifan Yang, Joel S Emer, and Daniel Sanchez. 2024. Trapezoid: A Versatile Accelerator for Dense and Sparse Matrix Multiplications. In *Proceedings of ACM/IEEE International Symposium on Computer Architecture (ISCA)*. IEEE, 931–945.

[60] Zhilin Yang, William Cohen, and Ruslan Salakhudinov. 2016. Revisiting semi-supervised learning with graph embeddings. In *International conference on machine learning*. PMLR, 40–48.

[61] Fangzhou Ye, Lingxiang Yin, Amir Ahsaei Ghazizadeh, and Hao Zheng. 2024. EGMA: Enhancing Data Reuse and Workload Balancing in Message Passing GNN Acceleration via Gram Matrix Optimization. In *Proceedings of ACM/IEEE Design Automation Conference (DAC)*. IEEE.

[62] Lingxiang Yin, Sanjay Gandham, Mingjie Lin, and Hao Zheng. 2024. SCALE: A Structure-Centric Accelerator for Message Passing Graph Neural Networks. In *Proceedings of IEEE/ACM international symposium on microarchitecture (MICRO)*. IEEE, 580–593.

[63] Lingxiang Yin, Amir Ghazizadeh, Shilin Tian, Ahmed Louri, and Hao Zheng. 2023. Polyform: A versatile architecture for multi-dnn execution via spatial and temporal acceleration. In *Proceedings of IEEE 39st International Conference on Computer Design (ICCD)*. IEEE, 166–169.

[64] Lingxiang Yin, Jun Wang, and Hao Zheng. 2023. Exploring Architecture, Dataflow, and Sparsity for GCN Accelerators: A Holistic Framework. In *Proceedings of the Great Lakes Symposium on VLSI (GLSVLSI)*. ACM, 489–495.

[65] Haoran You, Tong Geng, Yongan Zhang, Ang Li, and Yingyan Lin. 2022. Gcod: Graph convolutional network acceleration via dedicated algorithm and accelerator co-design. In *Proceedings of IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 460–474.

[66] Raphael Yuster and Uri Zwick. 2004. Detecting short directed cycles using rectangular matrix multiplication and dynamic programming.. In *Proceedings of the annual ACM-SIAM symposium on Discrete algorithms (SODA)*. ACM, 254–260.

[67] Guowei Zhang, Nithya Attaluri, Joel S Emer, and Daniel Sanchez. 2021. Gamma: Leveraging Gustavson's algorithm to accelerate sparse matrix multiplication. In *Proceedings of ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 687–701.

[68] Xiaoyu Zhang, Zerun Li, Rui Liu, Xiaoming Chen, and Yinhe Han. 2024. GAS: General-Purpose In-Memory-Computing Accelerator for Sparse Matrix Multiplication. *IEEE Trans. Comput.* (2024).

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