Accelerating Sparse Matrix-Matrix Multiplication on GPUs with Processing Near HBMs

Shiju Li*, Younghoon Min*, Timothy Lo*, Soohong Ahn§, Joonseop Sim§, Chul-Ho Lee‡, Jongryool Kim*
*SOLAB, SK hynix America, San Jose, USA
§AMS, SK hynix, Icheon, Korea
‡Texas State University, San Marcos, USA
shiju.li, younghoon.min, timothy.lo, soohong.ahn, joonseop.sim, jongryool.kim@sk.com
chulho.lee@txstate.edu

***Abstract*—Sparse General Matrix-Matrix Multiplication (SpGEMM) is a fundamental operation in numerous scientific computing and data analytics applications, often bottlenecked by irregular memory access patterns. This paper introduces the Acceleration of Indirect memory Access (AIA) technique, a novel approach to optimize SpGEMM on HBM GPUs. We develop a Ranged Indirect Access method and integrate it with a hash-based SpGEMM algorithm. Our Processing-Near-HBM based SpGEMM implementation demonstrates significant performance improvements over state-of-the-art methods, particularly in handling complex, application-specific workloads.**

We evaluate our approach on various graph workloads, including Graph Neural Networks (GNNs), graph contraction, and Markov clustering, showcasing its practical applicability. Our hash-based SpGEMM with AIA achieves an average of 77.27% speedup over cuSPARSE and 12.53% improvement over hash-based SpGEMM without AIA across diverse matrix datasets. For GNN training applications, our approach delivers 53.2% faster training time compared to cuSPARSE baselines, with a hybrid implementation (AIA forward + optimized backward) achieving up to 67.3% speedup while maintaining comparable accuracy. Results show consistent enhancements in cache utilization (L1 cache hit ratio improvements from 64.41% to 88.15%), runtime performance, and computational throughput across diverse datasets. Graph application benchmarks demonstrate up to 15.8% performance improvements, and our AIA gathering phase constitutes only 17.2% of total execution time, indicating significant optimization potential. Our findings position AIA as a powerful tool for high-performance graph processing on GPU architectures, with particular effectiveness for iterative algorithms that rely heavily on sparse matrix operations.

***Index Terms*—Indirect Memory Access, Sparse General Matrix-Matrix Multiplication, Near Memory Processing, Graph Neural Networks**

I. Introduction

Sparse General Matrix-Matrix Multiplication (SpGEMM) plays a pivotal role in computational science, machine learning, and graph analysis. While general matrix-matrix multiplication (GEMM) multiplies a matrix A (m × k) with matrix B (k × n) to produce matrix C (m × n), many applications such as algebraic multigrid methods [1], multi-source breadth-first search [2], recursive formulations of all-pairs shortest-paths algorithms [3], clustering coefficient [4], graph contraction [5], Markov clustering [6], and Graph Neural Networks (GNNs) [7]–[9] require exploiting sparsity in both input and output matrices to optimize storage and computational efficiency. Graph Neural Networks represent a particularly compelling application domain for SpGEMM optimization, as they fundamentally rely on sparse matrix operations for neighborhood aggregation and feature propagation across graph structures, with recent advances demonstrating that optimized sparse operations can achieve an average of 3.2 speedup over state-of-the-art frameworks [10].

However, the efficient execution of SpGEMM on modern GPUs faces several fundamental challenges, despite GPUs offering superior peak floating-point performance and memory bandwidth compared to CPUs. The primary challenges arise from the irregular nature of sparse computations. While compressed representations of sparse matrices reduce memory footprint, they incur significant overheads in accessing and processing metadata. These challenges manifest in three critical aspects: the unknown number of non-zero entries in the result matrix before computation, expensive parallel insert operations at random positions in the result matrix, and load balancing issues when handling input matrices with diverse sparsity structures. The irregular memory access patterns create a strong mismatch between data access patterns and memory layout, leading to limited spatial locality and cache effectiveness.

Previous GPU SpGEMM methods have attempted to address these challenges through various approaches, including dedicated hardware accelerators and software optimizations [11], [12]. However, these solutions have shown limited success, either performing well only for relatively regular sparse matrices or introducing significant memory overhead for specific sparsity patterns [1], [13]. Moreover, existing GPU implementations have struggled to consistently outperform well-optimized CPU approaches, highlighting the difficulty in fully utilizing GPU's massive parallelism for sparse matrix operations [14].

The advancement in 3D integration technologies has made the concept of coupling compute units close to the memory—called near memory processing (NMP) [15]—more viable. High Bandwidth Memory (HBM) [16] adopts a vertically stacked architecture that allows for greater capacity and bandwidth. Processing right at the "home" of data can significantly

diminish the data movement problem of SpGEMM. Thus, we adopt the idea of processing near GPU HBM to tackle the challenges of SpGEMM.

In this paper, we propose and implement the Acceleration of Indirect Memory Access (AIA) system, a novel Processing-Near-HBM approach to enhance performance in handling indirect memory access patterns in SpGEMM operations. We introduce the AIA technique, designed to assist the primary computational core in sparse data computations, specifically targeting SpGEMM operations on HBM GPUs. Our main contributions include:

- Development of a novel ranged indirect access technique for SpGEMM that effectively addresses the fundamental memory wall challenge in sparse matrix computations, transforming irregular memory access patterns into more efficient sequential streams and optimizing the two critical levels of indirection in SpGEMM;
- Integration of AIA functionality within GPU HBM architecture, leveraging high-bandwidth capabilities while mitigating traditional access pattern limitations, with AIA units placed between GPU cores and HBM stacks to enable efficient data gathering and significantly improve cache utilization—demonstrated by L1 cache hit ratio improvements from 64.41% to 75.14% in numeric phases and 64.66% to 88.15% in symbolic phases;
- Optimization of a sophisticated hash-based SpGEMM algorithm that effectively utilizes GPU resources through adaptive thread assignment and efficient shared memory hash tables, achieving an average of 77.27% speedup over cuSPARSE and 12.53% improvement over hash-based SpGEMM without AIA across diverse matrix datasets;
- Comprehensive evaluation demonstrating AIA's practical impact across diverse applications, with performance improvements up to 13.3% for Graph Contraction and 15.8% for Markov Clustering, and particularly notable results for GNN applications where MaxK-GNN systems achieve 3.22×-4.24× speedup over DGL frameworks and 4.15×-6.39× speedup over cuSPARSE implementations while maintaining comparable accuracy to state-of-the-art GNNs.** Our implementation achieves over 60 GFLOPS compared to 52 GFLOPS for non-AIA and less than 10 GFLOPS for cuSPARSE, validating AIA's scalability and efficiency for iterative graph algorithms requiring multiple SpGEMM operations.
- Optimization of a sophisticated hash-based SpGEMM algorithm that effectively utilizes GPU resources through adaptive thread assignment and efficient shared memory hash tables, achieving an average of 77.27% speedup over cuSPARSE and 12.53% improvement over hash-based SpGEMM without AIA across diverse matrix datasets;
- Comprehensive evaluation demonstrating AIA's practical impact across diverse applications, including performance improvements up to 13.3% for Graph Contraction and 15.8% for Markov Clustering, and particularly significant results for GNN training where our approach achieves up to 67.3% training time reduction while maintaining accuracy comparable to state-of-the-art frameworks. Our analysis reveals that the AIA gathering phase constitutes only 17.2% of total execution time on average, with projected improvements of 8.58%-12.87% achievable through FPGA acceleration, validating AIA's scalability and efficiency for iterative graph algorithms requiring multiple SpGEMM operations.

II. Background

- **Memory Wall and Indirect Memory Access.** Memory wall represents a critical bottleneck in modern computing systems, particularly affecting graph analysis, sparse linear algebra, and various other applications [17]–[19]. This challenge primarily stems from irregular memory access patterns, where data are arbitrarily accessed with poor locality, resulting in frequent cache misses and high latency when fetching data from the memory hierarchy. A key pattern contributing to this bottleneck is indirect memory access, typically appearing as x[a[i]], where array a's values serve as indices to access array x. This pattern becomes even more complex in the form of ranged indirect access (x[a[i]], x[a[i]+1], x[a[i]+2], ...), commonly seen in graph algorithms processing adjacency lists, where a[i] might point to the start of a vertex's neighbor list in array x, and multiple consecutive elements are accessed from that point. Such patterns pose significant challenges for conventional hardware prefetchers, as they require understanding both irregular access patterns and variable-length ranges of accessed data.
- **GPU-Centric Near Memory Processing and Challenges.** The integration of near-memory processing with GPU HBM architectures presents both opportunities and significant challenges. Several notable architectures have explored this concept in GPU contexts, with TOM [20] pioneering a novel approach using lightweight GPU cores integrated into 3D-stacked memories, supported by an automatic compiler framework for offload decisions. Building on this foundation, Pattnaik et al. [21] advanced the concept with an NMC-assisted GPU architecture featuring an affinity prediction model for kernel execution placement. Recent research has particularly focused on leveraging HBM's unique capabilities for near-memory processing. DNN-PIM [22] introduced a heterogeneous architecture incorporating both programmable ARM cores and fixed-function units in the HBM logic layer. Complementing this work, Boroumand et al. [23] evaluated PNM architectures for consumer applications, carefully considering the practical constraints of area and power budgets in HBM implementations.

  Despite these advances, four fundamental challenges must be addressed for effective implementation. First, the programming model must efficiently integrate with existing GPU frameworks while providing mechanisms to determine optimal code placement between host and near-memory units [22], [23]. Second, memory man-

agement requires sophisticated data mapping strategies between GPU cores and HBM stacks, as demonstrated by MONDRIAN [24], which emphasized the importance of hardware-software co-design in optimizing memory access patterns. Third, virtual memory support must align with GPU memory management systems while maintaining efficient address translation mechanisms, a challenge effectively addressed in TESSERACT [25] and subsequent works [26]. Finally, cache coherency remains a critical challenge in maintaining consistency between GPU cores and near-memory processing units, with recent work by Farmahini et al. [27] proposing innovative solutions for optimizing HBM bandwidth utilization and minimizing unnecessary off-chip traffic.

- **Prior SpGEMM Algorithms.** The evolution of SpGEMM algorithms reflects ongoing efforts to optimize sparse matrix multiplication across different hardware architectures. Traditional approaches can be categorized into three main strategies: outer product, inner product, and row-wise methods. The outer product method, exemplified by Dalton et al. [12], computes the contribution of each nonzero element in matrix A and B independently, requiring efficient merging of intermediate results. Inner product methods, as implemented by Demouth [13],compute each element of the output matrix through dot products of corresponding rows and columns, facing challenges with memory locality. Row-wise methods, adopted by Naumov et al. [11], process each row of the output matrix independently, offering better parallelism but requiring careful load balancing strategies.

  On GPU platforms, SpGEMM implementations have evolved from basic CSR-based approaches to sophisticated hybrid formats and specialized algorithms. Early implementations struggled with the irregular nature of sparse matrix operations, but recent advances have introduced various optimizations. Hash-based approaches, as demonstrated by Liu and Vinter [28], use hash tables to accumulate partial products efficiently. Sort-based methods, exemplified by Nagasaka et al. [29], offer better memory coalescing but incur sorting overhead. Hybrid methods, such as those proposed by Gremse et al. [30], combine multiple strategies to balance computation and memory access efficiency.

  Despite these advancements, existing implementations still face fundamental challenges in fully utilizing GPU capabilities for SpGEMM operations. The irregular memory access patterns create a strong mismatch between data access patterns and memory layout, leading to limited spatial locality and cache effectiveness, which causes frequent data movement between memory hierarchies. Processing-Near-HBM solutions offer a promising approach to address these challenges by reducing data movement and enhancing memory access efficiency.
- **SpGEMM Accelerates GNN.** Recent works have explored several promising approaches to accelerate Graph Neural Networks (GNNs) through sparse-sparse matrix multiplication (SpGEMM). SpGEMM accelerates GNN through strategic transformation of computational patterns and exploitation of induced sparsity. MaxK-GNN [10] demonstrates the most dramatic approach, converting traditional SpMM operations into SpGEMM by introducing MaxK nonlinearity that sparsifies feature matrices to 87.5% sparsity, storing them in a custom CBSR (Compressed Balanced Sparse Row) format. This transformation reduces global memory traffic by over 90% and achieves 3-7× kernel-level speedups on NVIDIA GPUs, with L1 cache hit rates improving from 1.53% to 22.16%. shivdikar2024neurachip. Shivdikar et al. [31] also propose a novel GNN spatial accelerator using spgemm on the sparsified feature matrix. PruneGNN [32] takes a complementary approach through structured dimension-wise pruning, selectively using SpGEMM for backpropagation steps where both weight and gradient matrices are sparse, achieving 2× average speedup on A100 GPUs through SIMD-aware kernel design. Tripathy et al. [33] reframes GNN neighborhood sampling as a matrix-based bulk sampling technique, expressing sampling as SpGEMM operations to efficiently process multiple minibatches simultaneously. This matrix-based approach allows complex sampling procedures to be represented through a series of SpGEMM operations: computing probabilities, performing rejection sampling, and executing row and column extractions, enabling 2.5-8.46× speedups for distributed training by processing multiple minibatches simultaneously. These implementations address SpGEMM's fundamental challenges—irregular memory access patterns, load imbalance, and accumulation bottlenecks—through techniques including warp-level edge partitioning, shared memory buffering, and row-wise product algorithms that reduce computational complexity from $O(Ed)$ to $O(Ek)$ where $k << d$, which collectively demonstrate SpGEMM's critical role in addressing the computational bottlenecks in GNNs, especially for large-scale graphs, while maintaining comparable accuracy to dense implementations.

III. SpGEMM on GPU

Our optimized SpGEMM algorithm accelerates sparse matrix multiplication while minimizing memory usage. The algorithm uses a shared memory hash table and intelligently groups matrix rows to maximize GPU resource utilization. The algorithm consists of three main phases, each designed to efficiently handle different aspects of sparse matrix multiplication.

A. Row Grouping

First, we count the non-zero elements that will appear in the output matrix. Since different rows require varying amounts of computation in SpGEMM, naive parallelization can lead to workload imbalance. To improve load balancing and optimize GPU resource usage (especially shared memory), our algorithm performs grouping in two stages. The initial

grouping phase organizes rows based on their number of intermediate products, while the final grouping phase arranges rows according to the number of non-zero elements in their output. Rather than reordering the entire input matrix, which would be computationally expensive, we generate an array of row indices for each group. This approach only requires additional memory for storing the row index arrays.

Algorithm 1: Count Intermediate Products for i-th row

```
1 intermediateCount ← 0
2 for j = rpt_A[i] to rpt_A[i + 1] do
3 |   intermediateCount ← intermediateCount +
  |     (rpt_B[colind_A[j] + 1] − rpt_B[colind_A[j]])
4 end
```

B. Non-zero Element Counting

We count non-zero elements in each output matrix row while handling overlapping column indices. Our approach optimizes GPU memory access patterns and uses hash tables for efficient column index management. We assign different hash table and thread block sizes to each row group for optimal GPU resource utilization. Multiple CUDA kernels are launched with different streams, enabling parallel execution across groups.

1) Thread Assignment Strategies: Our implementation provides two distinct thread assignment approaches as shown in algorithm 2 and algorithm 3. The first approach, named PWPR, uses partial warp per row, which assigns four threads per row. In this configuration, each thread processes a portion of the non-zero elements from matrix A and their corresponding rows from matrix B. This approach is particularly effective for rows with smaller numbers of non-zero elements or intermediate products.

Algorithm 2: Count Non-zeros Using PWPR

```
1 #thread per row ← 4
2 threadPos ← threadIdx% 4
3 for j ← rpt_A[i] to rpt_A[i + 1] stride 4 do
4 |   colindA ← colind_A[j + threadPos]
5 |   for k ← rpt_B[colindA] to rpt_B[colindA + 1] do
6 |   |   //Perform hash table operations (see Alg. 4)
7 |   end
8 end
```

The second approach, TBPR, assigns an entire thread block to process each row of matrix A. Within this configuration, one warp handles each non-zero element in matrix A, and individual threads within the warp process elements from matrix B. This approach proves more efficient for rows with larger numbers of non-zero elements or intermediate products.

2) Hash Table: Algorithm 4 implements a sophisticated collision-resolving hash table using linear probing. The algorithm begins with an empty hash table (initialized to -1) and a counter to track unique elements. When processing a new column index, it first computes a hash position using multiplication and modulo operations to ensure good distribution. The algorithm then enters a loop that handles three possible scenarios: finding an existing entry, inserting a new entry, or resolving a collision.

Algorithm 3: Count the number of non-zero elements of i-th row by TBPR

```
1 threadInWarp ← threadIdx%warpsize
2 warpNumber ← threadIdx/warpsize
3 totalWarps ← blockDim/warpsize
4 for j ← rpt_A[i] + warpNumber to rpt_A[i + 1]
    stride totalWarps do
5 |   columnIndA ← colind_A[j]
6 |   for k ← rpt_B[d] + threadInWarp to rpt_B[d + 1]
  |     stride warpsize do
7 |   |   // Perform hash table operations (see Alg. 4)
8 |   end
9 end
```

The hash table employs atomic Compare-And-Swap operations to safely handle concurrent insertions in a parallel environment. $TableSize$ is first set by $intermediateCount$ in algorithm 1 and then determined by $uniqueCount$ after the non-zero element counting phase. For rows with large numbers of non-zero elements, the algorithm uses a two-phase approach: first attempting to use a shared memory hash table, then falling back to global memory if the shared memory capacity is exceeded. This adaptive approach ensures efficient handling of varying row sizes while maximizing performance.

Algorithm 4: Hash Algorithm

```
1  table[] ← −1
2  uniqueCount ← 0
3  key ← colind_B[k]
4  hashPos ← (key ∗ multiplier)%tableSize
5  while true do
6  |   if table[hashPos] = key then
7  |   |   break
8  |   else
9  |   |   if table[hashPos] = −1 then
10 |   |   |   oldValue ←
   |   |   |     atomicCAS(table + hashPos, −1, key)
11 |   |   |   if oldValue = −1 then
12 |   |   |   |   uniqueCount ← uniqueCount + 1
13 |   |   |   |   break
14 |   |   |   end
15 |   |   else
16 |   |   |   hashPos ← (hashPos + 1)%tableSize
17 |   |   end
18 |   end
19 end
```

C. Output Matrix Calculation

The final phase combines intermediate products to form output matrix elements through three carefully orchestrated steps. The value computation step uses the hash table to accumulate values for each non-zero element, maintaining separate tables for column indices and values. Following this, the element gathering step collects non-zero elements from the hash table and prepares them for the final sorting step. In the column index sorting step, each thread determines its element's position by comparing with other elements, after which elements are written to the final output array in sorted column index order. When possible, all these operations are performed in shared memory to maximize performance. This organization enables efficient parallel processing while maintaining proper output matrix structure, ensuring both correctness and high performance in the final result.

IV. System Architecture

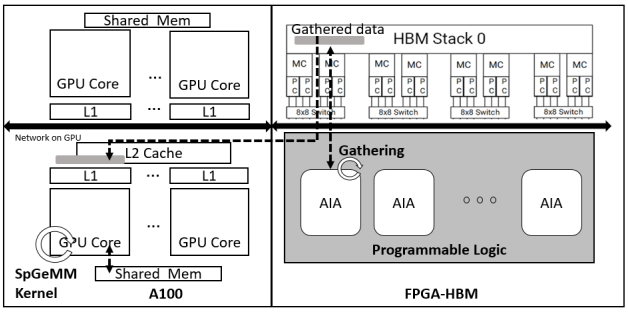

Fig. 1. GPU with AIA HBM

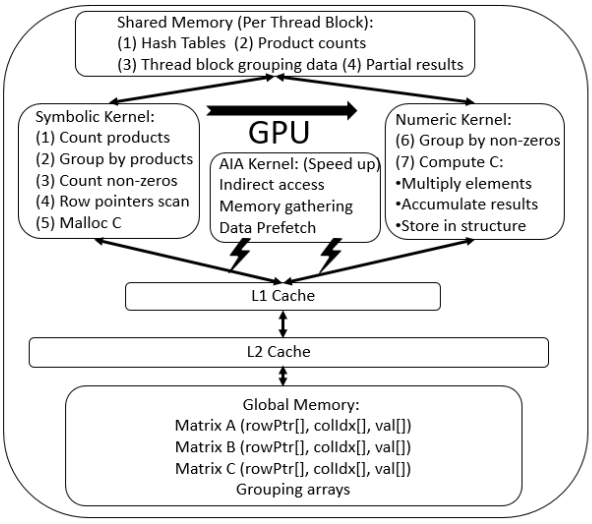

Fig. 2. SpGeMM GPU flow

Our proposed system integrates Indirect Access capabilities with the GPU's High Bandwidth Memory (HBM) to accelerate Sparse General Matrix-Matrix Multiplication (SpGEMM). This integration leverages the GPU's parallel processing power while addressing the irregular memory access patterns inherent in SpGEMM operations.

A. HBM-AIA Integration

We utilize an evaluation board that supports HBM and a Processing System (PS). The application runs on the PS, while the AIA functionality is implemented in the Programmable Logic (PL) section of the FPGA-HBM (V80) chip. This architecture allows the application to access gathered data in HBM through AIA functions, effectively bridging the gap between the processing system and the high-bandwidth memory.

The HBM stack is connected to the GPU cores via NV link. The AIA units are placed between the GPU cores and the HBM stack, enabling efficient gathering of data from HBM based on indirect access patterns.

B. AIA Ranged index Implementation

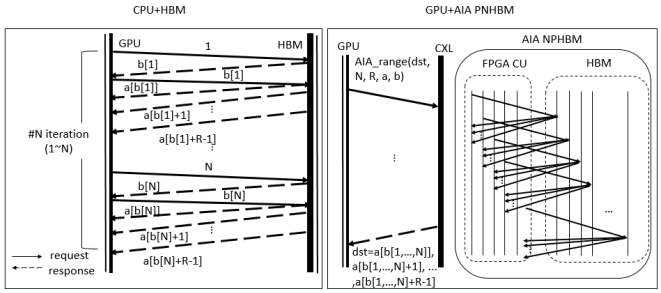

Fig. 3. AIA ranged index

We implement AIA ranged index functionality, which allows for efficient handling of SpGEMM's irregular access patterns. The AIA ranged index is particularly useful for SpGEMM as it can handle the two levels of indirection present in the algorithm: 1. Mapping over rearranged node index (load balance) and pointer array of matrix A 2. Mapping over column indices of matrix A and pointer array of matrix B.

The AIA ranged index implementation is based on the loop structure of SpGEMM, with an additional index mapping to ensure load balancing across GPU threads. This mapping is crucial for handling the irregular nature of sparse matrix operations.

C. GPU Memory Hierarchy Utilization

Our system takes advantage of the GPU's memory hierarchy to optimize SpGEMM performance:

L1 cache (128KB) local to each Streaming Multiprocessor (SM) L2 cache (6MB) shared between all SMs HBM (32GB) as global memory

The AIA units work in conjunction with this hierarchy, helping to reduce the number of memory requests and increase cache hit rates. By gathering relevant data from HBM using AIA functions, we can more efficiently utilize the L1 and L2 caches, reducing the impact of irregular memory access patterns.

D. SpGEMM Optimization

To optimize SpGEMM performance, we implement the following strategies: 1. Indirect Access Handling: We use AIA to efficiently handle the indirect access patterns in SpGEMM, reducing memory latency and improving bandwidth utilization. 2. Load Balancing: We implement a dynamic mapping scheme that distributes work evenly across GPU threads, addressing the load imbalance issues common in sparse matrix operations. 3. Memory Access Optimization: By using AIA range 2, we can gather data more efficiently, reducing the number of memory requests and the amount of unused data loaded into caches. 4. Cache Utilization: We optimize the use of L1 and L2 caches by ensuring that gathered data is more likely to be reused, increasing cache hit rates.

E. AIA on SpGEMM

The GPU kernel for SpGEMM is integrated using the AIA (ranged index R=2) approach. Each thread processes a row of matrix A, using AIA to efficiently gather the required data from matrix B. The integrated kernel structure is as shown in Algo. 5. The AIA-range2 technique effectively addresses the fundamental challenge of irregular memory access patterns inherent in SpGEMM operations by transforming the two-level indirection problem into optimized sequential data streams. In SpGEMM, computing the result matrix $C = AB$ requires accessing matrix B's rows corresponding to the non-zero column indices in each row of matrix A, creating a complex indirect access pattern. As illustrated in Fig. 4, the AIA system maps GPU block IDs to matrix rows through load-balanced binning, where each bin groups rows with similar computational complexity. For block i processing row j, the first AIA-range2 function computes $aia\_1[2i] = a\_row\_ptr[d\_map[i]]$ and $aia\_1[2i+1] = a\_row\_ptr[d\_map[i]+1]$ to define the range of non-zero elements in matrix A, while the second AIA-range2 function $aia\_2[2j] = b\_row\_ptr[d\_acol[j]]$ and $aia\_2[2j+1] = b\_row\_ptr[d\_acol[j]+1]$ define the corresponding ranges in matrix B. This approach enables the GPU cores to efficiently gather the required data from HBM in a more predictable pattern, significantly improving cache utilization and reducing memory latency. The binning strategy further enhances load balancing by ensuring that threads within the same block process rows with similar sparsity characteristics, while the ranged indirect access mechanism converts the traditionally irregular SpGEMM memory access patterns into optimized sequential streams that better match the GPU's memory hierarchy design.

This kernel structure allows for efficient parallelization of the SpGEMM operation while leveraging the AIA capabilities to handle irregular memory accesses. By integrating AIA functionality into the HBM-equipped GPU architecture, our system provides a powerful platform for accelerating SpGEMM operations, addressing the key challenges of irregular memory access and load balancing in sparse matrix computations.

Algorithm 5: SpGEMM Kernel integration with AIA

```
for j ← AIA_1[2rid] + wid to AIA_1[2rid+1] stride wnum do
    for k ← AIA_2[2j] + tid to AIA_2[2j+1] stride warp_size_num do
        //hash operation;
    end
end
```

V. Benchmarks/Applications

This section introduces the benchmarks and applications we evaluate to demonstrate the effectiveness of our AIA-accelerated SpGEMM implementation. We focus on key graph algorithms and sparse matrix operations that frequently occur in scientific computing and data analytics. Typical candidates include Markov Clustering (MCL), and Graph Contraction and the widely used Graph Neural Network (GNN). In each application, we explain where and how SpGEMM can be applied.

A. MCL

The Markov Cluster (MCL) algorithm is an unsupervised graph clustering method that simulates stochastic flows on graphs to detect natural clusters. It operates on the principle that random walks on a graph will tend to get trapped within densely connected regions, corresponding to clusters. The algorithm begins by converting the input graph into a column stochastic matrix. It then iteratively applies two main operations: expansion and inflation. The expansion step, implemented as matrix multiplication or exponentiation, simulates random walks and allows flow to spread out. The inflation step, realized through the Hadamard power followed by normalization, strengthens intra-cluster connections while weakening inter-cluster connections. A crucial intermediate step, pruning, is applied to maintain sparsity by removing small entries and retaining only the top-k entries in each column. This process is repeated until convergence, typically when changes between successive iterations become negligible. The final matrix is interpreted to extract clusters, often represented as the connected components of the graph implied by the matrix. MCL's effectiveness in detecting clusters in various network types, coupled with its mathematical elegance, has made it a popular choice in bioinformatics and other fields dealing with complex network data.

B. Graph Contraction

Graph contraction is a fundamental operation in graph theory and algorithm design that reduces the size of a graph by merging nodes with shared labels. This process is particularly useful in iterative graph algorithms where the problem can be solved on progressively smaller subgraphs. The algorithm efficiently implements contraction through matrix multiplication, utilizing a strategically constructed sparse matrix S. Given a graph G and a set of node labels, the algorithm first determines the number of nodes n and the maximum label value m. It then

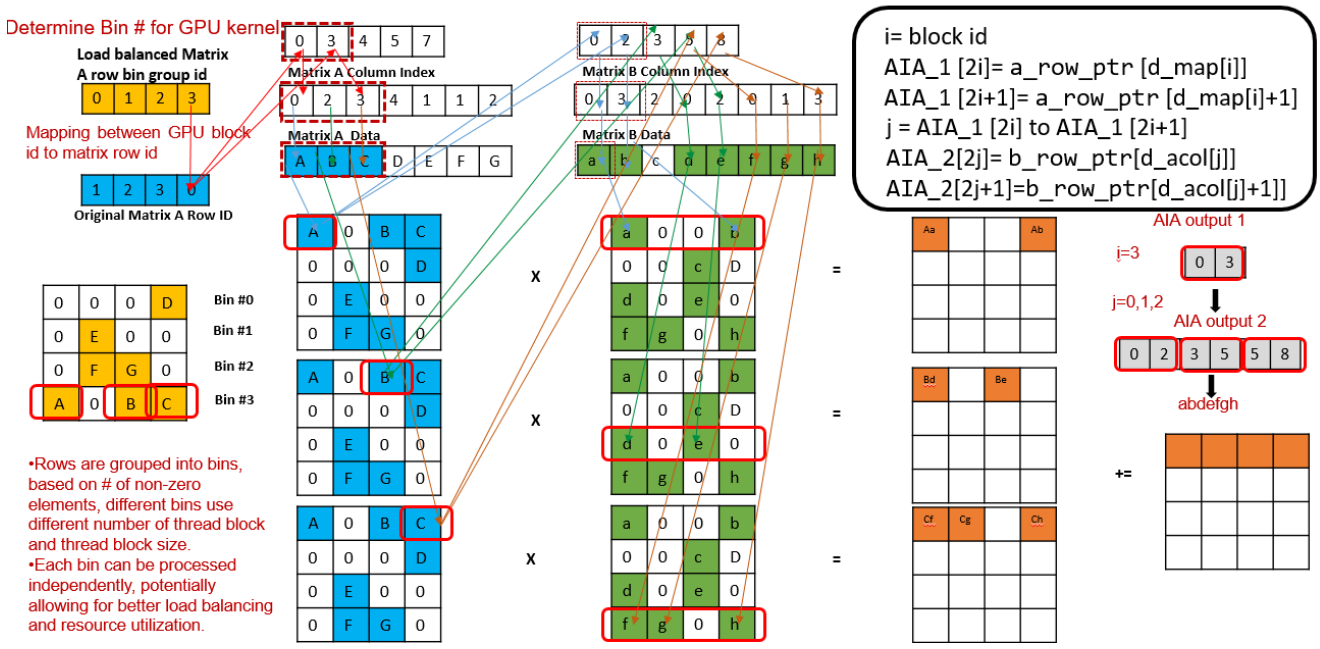

Fig. 4. AIA on SpGeMM: A Toy Example

Algorithm 6: Markov Cluster (MCL) Algorithm

Input: Weighted network G, expansion parameter e, inflation parameter r, pruning threshold θ, k
Output: $clusters$

AddSelfLoops(G)
$A \leftarrow$ ColumnStochasticMatrix(G)
Column Normalize(A)
while *change in successive iterations* **do**
  `// Expansion: random walks from all vertices`
  $B \leftarrow A^e$
  `// Prune: Sparsify the matrix`
  $C \leftarrow$ Prune(B, θ, k)
  for *each column* j *in* C **do**
    Remove entries $C_{ij} < \theta$
    Keep only top-k entries in column j
  end
  `// Inflation: strengthen intra-cluster connections`
  for *each entry* C_{ij} *in* C **do**
    $C_{ij} \leftarrow (C_{ij})^r$
  end
  $A \leftarrow$ Column Normalize(C)
end
$clusters \leftarrow$ InterpretMatrix(A)
return $clusters$

creates the sparse matrix S of dimensions m × n, where each column corresponds to a node's original label, and each row represents a new label in the contracted graph. The matrix S contains only ones, with their positions determined by the node labels. The contraction is then performed by the matrix multiplication $C = SGS^T$, where G is the original graph's adjacency matrix and S^T is the transpose of S. This operation effectively combines rows and columns of G that share labels, resulting in a new, smaller adjacency matrix C representing the contracted graph. The left multiplication by S combines rows with the same label, while the right multiplication by S^T combines columns, ensuring that edges between merged nodes are properly accounted for in the contracted graph.

Algorithm 7: Graph Contraction

Input: Graph G, node labels $labels$
Output: Contracted graph C

$n \leftarrow$ length(G);
$m \leftarrow \max(labels)$;
$S \leftarrow$ sparse($labels, 1:n, 1, m, n$);
$C \leftarrow S \times G \times S^T$;
return C

C. Graph Neural Network

There are two promising approaches to accelerate Graph Neural Networks (GNNs) through sparse-sparse matrix multiplication (SpGEMM), one is to introduce MaxK nonlinearity that sparsifies feature matrices, and the other is to formulate neighborhood sampling operations as Matrix-based Bulk sampling, where SpGEMM can be used in both cases.

Since Graph Neural Networks (GNNs) represent a challenging computational domain requiring SpGEMM operations,

particularly in forward and backward propagation steps. Consider a graph $G = (V, E, A)$ containing $|V|$ nodes and $|E|$ edges, where the adjacency matrix $A \in \mathbb{R}^{|V|\times|V|}$ exhibits high sparsity. Traditional GNN forward propagation follows $X_l = A \cdot h(X_{l-1})$, where X_l represents the node features at layer l, and $h(X_{l-1})$ denotes transformed features from the previous layer. These operations typically employ SpMM (Sparse-Dense Matrix Multiplication), becoming computationally intensive for large graphs.

MaxK-GNN introduces a paradigm shift by applying a nonlinear sparsification function to feature matrices, effectively transforming conventional SpMM operations into more efficient SpGEMM operations. This approach inserts a MaxK nonlinearity that selects only the top-k elements per node embedding, resulting in a sparsified feature matrix. The forward computation is mathematically reformulated as:

$$X_l = A \cdot \text{MaxK}(X_{l-1}W_l)$$

where $\text{MaxK}(\cdot)$ retains only the k largest values in each row, replacing others with zeros. This transforms the operation from SpMM to row-wise product-based SpGEMM, significantly reducing memory traffic. For the Reddit dataset with original hidden dimension of 256 and $k = 16$, this approach achieves 90.6% reduction in global memory traffic compared to traditional SpMM. The corresponding backward propagation also benefits from sparsity, expressed as:

$$\frac{\partial L}{\partial h(X_{l-1})} = A^T \cdot \frac{\partial L}{\partial X_l}$$

where only non-zero elements from the forward path require gradient computation.

GNN training requires efficient neighborhood sampling to overcome the prohibitive costs of the "neighborhood explosion" problem in large graphs. The matrix-based bulk sampling approach formulates sampling as a series of sparse-sparse matrix multiplication (SpGEMM) operations, enabling simultaneous processing of multiple minibatches. Given an input graph with adjacency matrix and sampler-dependent matrix Q^L, the algorithm follows a three-step process expressed through matrix operations.

For each layer $l = L$ to 1, the probability distribution matrix P is computed via SpGEMM as:

$$P = Q^l A$$

where Q^l encodes the sampling context for layer l. After normalization of P, the algorithm applies Inverse Transform Sampling (ITS) to select s non-zeros per row:

$$Q^{l-1} = \text{SAMPLE}(P, b, s)$$

where b is batch size and s is the sampling parameter. The final adjacency matrix A^l for layer l is extracted through:

$$A^l = \text{EXTRACT}(A, Q^l, Q^{l-1})$$

This approach is particularly advantageous for distributed training as it can process k batches simultaneously by vertically stacking individual matrices:

$$Q^l = \begin{pmatrix} Q_1^l \\ \vdots \\ Q_k^l \end{pmatrix}, P = \begin{pmatrix} P_1 \\ \vdots \\ P_k \end{pmatrix}, A^l = \begin{pmatrix} A_1^l \\ \vdots \\ A_k^l \end{pmatrix}$$

The dimensions for stacked Q^l and P^l are $kbs^l \times n$, while stacked A^l are $kbs^l \times s^{l+1}$. This matrix formulation enables the use of communication-avoiding SpGEMM algorithms that significantly reduce data movement in distributed environments. For node-wise sampling algorithms like GraphSAGE, each row of P represents the neighborhood distribution of a batch vertex, while for layer-wise algorithms like LADIES, it represents the distribution across the aggregated neighborhood of the entire batch.

VI. Experimental Results

Hardware Configuration. All experiments were conducted on a Linux server with a 3.0GHz 144-core Intel Xeon Platinum 8452YL CPU (Sapphire Rapids) and 1 TB RAM. Our AIA system is implemented on an NVIDIA A100 GPU. **Methodology.** We compare our AIA-based implementation against a version without AIA and the state-of-the-art cuSPARSE library. Performance is evaluated based on FLOPS for matrix squaring, calculated as twice the number of intermediate products divided by execution time. **Datasets.** We selected 10 square matrices from the University of Florida Sparse Matrix Collection, commonly used for evaluating sparse matrix computations on GPUs. Table I summarizes the characteristics of these matrices, which vary significantly in size, sparsity, and non-zero element distribution.

A. Performance Analysis of SpGeMM

Cache Hit Ratio: Figure 5 shows the L1 cache hit ratio for the scircuit and cage15 datasets, comparing implementations with and without AIA across numeric and symbolic phases. AIA significantly improves cache utilization in both phases for

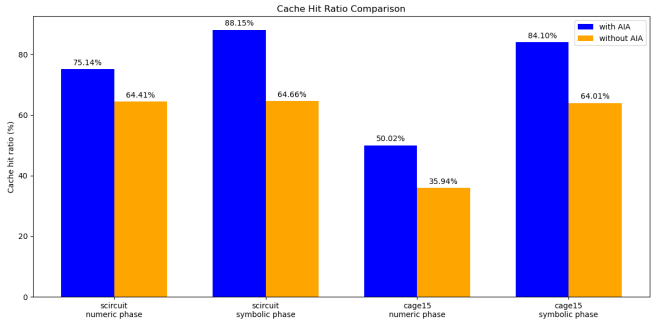

Fig. 5. L1 Cache Hit Ratio Comparison

both datasets. For scircuit, the hit ratio increases from 64.41% to 75.14% in the numeric phase and from 64.66% to 88.15% in the symbolic phase. Similarly, for cage15, improvements are observed from 35.94% to 50.02% in the numeric phase and

Name	Row	Non-zero	Nnz/row	Max nnz/row	Intermediate product of A^2	Nnz of A^2
RoadTX	1,393,383	3,843,320	2.8	51	12099370	3843320
p2p-Gnutella04	10,879	39,994	3.7	497	180230	39994
amazon0601	403,394	3,387,388	8.4	100	32373599	16,258,436
web-Google	916,428	5,105,039	5.6	4334	60687836	29,710,164
scircuit	170,998	958,936	5.6	353	8,676,313	5,222,525
cit-Patents	3,774,768	16,518,948	4.4	770	82,152,992	68,848,721
Economics	206,500	1,273,389	6.2	44	7,556,897	6,704,899
webbase-1M	1,000,005	3,105,536	3.1	4700	69,524,195	51,111,996
wb-edu	9,845,725	57,156,537	5.8	3841	1,559,579,990	630,077,764
cage15	5,154,859	99,199,551	19.2	47	2,078,631,615	929,023,247
Wind Tunnel	217,918	11,634,424	53.4	180	626,054,402	32,772,236
Protein	36,417	4,344,765	119.3	204	555,322,659	19,594,581

TABLE I
MATRIX DATA

from 64.01% to 84.10% in the symbolic phase. These results demonstrate AIA's effectiveness in optimizing memory access patterns, particularly in the symbolic phase.

Runtime and GFLOPS Analysis: Figures 6 present runtime and GFLOPS comparisons for SpGEMM implementations.AIA consistently reduces execution time across all datasets, with particularly pronounced improvements for larger datasets such as cage15, wb-edu, and cit-Patents. The GFLOPS comparison further illustrates AIA's performance benefits, with the Wind Tunnel dataset showing a 2.5x performance boost (from 5.5 to 14 GFLOPS). Compared to cuSPARSE, our AIA implementation demonstrates superior performance in both runtime and GFLOPS. The performance gap is especially significant for datasets like Wind Tunnel and Protein, where AIA achieves substantially higher GFLOPS.

B. Graph workload Performance

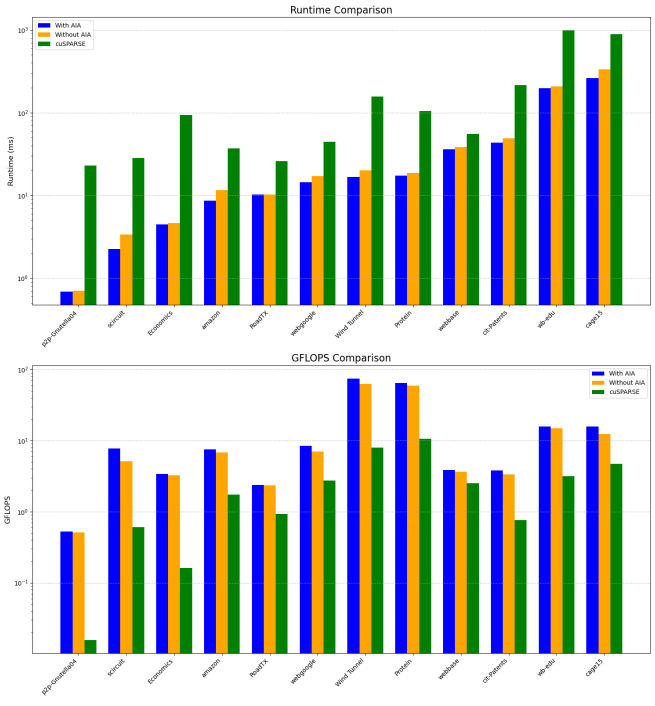

Fig. 6. Runtime and GFLOPS Comparison (AIA vs. Without AIA)

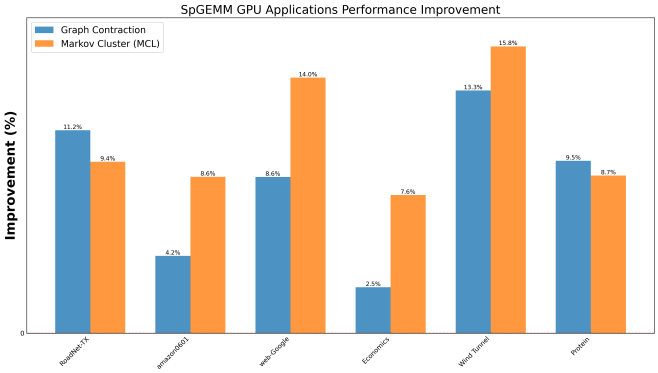

Fig. 7. SpGeMM Graph Applications Performance Improvement

Figure 7 illustrates the performance improvements achieved by AIA for two graph applications: Graph Contraction and Markov Clustering (MCL). The results demonstrate consistent performance gains across diverse datasets, with improvement percentages varying based on application type and dataset characteristics. For Graph Contraction, AIA shows significant improvements ranging from 2.5% to 13.3%. The most notable improvements are observed with Wind Tunnel (13.3%) and RoadNet-TX (11.2%) datasets. Web-Google and amazon0601 datasets show moderate improvements of 8.6% and 4.2% respectively, while Economics demonstrates a 2.5% improvement. Markov Clustering exhibits even more pronounced improvements in several cases. Wind Tunnel dataset achieves the highest improvement at 15.8%, followed by web-Google at 14.0%. Other datasets show consistent gains: RoadNet-TX (9.4%), Protein (8.7%), amazon0601 (8.6%), and Economics (7.6%). MCL's higher improvement percentages, particularly for complex datasets like Wind Tunnel and web-Google, suggest that AIA is especially effective for iterative graph algorithms with multiple SpGEMM operations.

C. Graph Neural Network Training Performance

Graph Neural Networks represent a critical application domain for SpGEMM optimization, where sparse matrix operations directly impact training efficiency. We evaluate our AIA-enhanced hash-based SpGEMM on GNN training workloads using the Flickr dataset with GraphSAGE architecture, comparing against three distinct baselines to demonstrate

comprehensive performance improvements. Table II presents a comprehensive comparison of our approach against three key baselines: cuSPARSE (standard sparse library), MaxK-GNN (state-of-the-art GNN acceleration), and hash-based SpGEMM without AIA. The forward pass times reflect SpGEMM operations for neighborhood aggregation and SpMM operations for aggregated feature matrix and weight matrix, while backward pass times represent SpMM operations for adjacent matrix and weight matrix.Our hash-based SpGEMM with AIA achieves significant performance improvements across all metrics. In the forward pass, our approach delivers 77.0% speedup over cuSPARSE (5.97ms vs 26.01ms) and 68.7% improvement over MaxK-GNN (5.97ms vs 19.10ms). When combined with optimized backward pass techniques from MaxK-GNN, the hybrid approach achieves the most dramatic results, delivering 67.3% total training time reduction compared to cuSPARSE baselines.Table III analyzes the contribution of AIA gathering operations to total execution time, revealing optimization opportunities and projected improvements with HBM. The analysis reveals that AIA gathering operations constitute an average of 17.2% of total execution time, indicating significant room for optimization. HBM acceleration of the AIA component projects additional improvements of 8.58% (2× faster) to 12.87% (4× faster), with particularly notable gains for matrices with higher AIA ratios such as scircuit (up to 32.15% improvement).

TABLE II
GNN TRAINING PERFORMANCE COMPARISON ON FLICKR DATASET (GRAPHSAGE)

Implementation	Forward (ms)	Backward (ms)	Total (ms)	Speedup vs cuSPARSE
cuSPARSE (Baseline)	26.01	11.68	37.69	-
MaxK-GNN	19.10	6.36	25.46	32.4%
Hash w/o AIA	6.82	11.68	18.50	50.9%
Hash with AIA	**5.97**	**11.68**	**17.65**	**53.2%**
AIA + MaxK Backward	**5.97**	**6.36**	**12.33**	**67.3%**

TABLE III
AIA COMPONENT ANALYSIS AND PROJECTED HBM IMPROVEMENTS

Matrix	AIA Ratio (%)	2× HBM Improvement	4× HBM Improvement	Current Total (ms)
amazon	15.5	7.76%	11.63%	12.02
webgoogle	12.4	6.20%	9.30%	21.60
scircuit	42.9	21.44%	32.15%	4.84
cit-Patents	11.1	5.56%	8.34%	48.81
Average	**17.2**	**8.58%**	**12.87%**	**-**

The experimental results demonstrate three critical findings: (1) Hash-based SpGEMM without AIA already provides substantial improvements over standard libraries (50.9% vs cuSPARSE), indicating the algorithmic contribution; (2) AIA adds meaningful incremental gains (4.6% over hash without AIA, 53.2% over cuSPARSE), validating the Processing-Near-HBM approach; (3) Hybrid optimization combining AIA forward pass with optimized backward techniques delivers the most dramatic results (67.3% total speedup), suggesting that comprehensive system-level optimization across the entire training pipeline provides maximum benefit for GNN applications.

These improvements consistently demonstrate AIA's effectiveness in handling complex, application-specific workloads beyond basic SpGEMM operations, achieving an average of 77.27% speedup over cuSPARSE and particularly significant results for Graph Neural Network training with 53.2% to 67.3% faster training time. The results are particularly impressive for larger, more complex datasets and iterative algorithms like Markov clustering, validating AIA's scalability and practical applicability in real-world graph processing scenarios. AIA gathering operations constitute only 17.2% of total execution time, indicating substantial room for further optimization and positioning our Processing-Near-HBM approach as a powerful tool for high-performance sparse matrix computations on GPU architectures.

VII. CONCLUSION

This paper presents the Acceleration of Indirect Memory Access (AIA) System, a novel Processing-Near-HBM approach to enhance performance in handling indirect memory access patterns in SpGEMM operations. Our solution integrates AIA functionality directly into GPU's HBM architecture, effectively bridging the gap between processing requirements and memory access patterns. The implementation achieves significant cache utilization improvements. Our comprehensive evaluations demonstrate substantial performance gains across different application domains. Consistent improvements across diverse datasets underscore AIA's effectiveness in handling irregular memory access patterns and load-balancing challenges inherent in sparse matrix computations. With the continuous evolution of GPU architectures and the growth of data size, the AIA system presents a promising direction for future optimizations in high-performance computing, particularly in the domains of graph processing and sparse matrix operations.

In conclusion, the AIA-SpGEMM approach introduced in this paper not only advances the state of the art in SpGEMM performance but also opens new avenues for optimizing indirect memory access in a wide range of applications dealing with irregular sparse data structures on GPU architectures.

REFERENCES

[1] N. Bell, S. Dalton, and L. N. Olson, "Exposing fine-grained parallelism in algebraic multigrid methods," *SIAM Journal on Scientific Computing*, vol. 34, no. 4, pp. C123–C152, 2012.

[2] A. Buluç and J. R. Gilbert, "The combinatorial blas: Design, implementation, and applications," *The International Journal of High Performance Computing Applications*, vol. 25, no. 4, pp. 496–509, 2011.

[3] P. D'alberto and A. Nicolau, "R-kleene: A high-performance divide-and-conquer algorithm for the all-pair shortest path for densely connected networks," *Algorithmica*, vol. 47, pp. 203–213, 2007.

[4] A. Azad, A. Buluç, and J. Gilbert, "Parallel triangle counting and enumeration using matrix algebra," in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. IEEE, 2015, pp. 804–811.

[5] J. R. Gilbert, S. Reinhardt, and V. B. Shah, "A unified framework for numerical and combinatorial computing," *Computing in Science & Engineering*, vol. 10, no. 2, pp. 20–25, 2008.

[6] S. Van Dongen, "Graph clustering by flow simulation," *PhD thesis, University of Utrecht*, 2000.

[7] Y. Ma and J. Tang, *Deep learning on graphs*. Cambridge University Press, 2021.

[8] W. L. Hamilton, *Graph representation learning*. Morgan & Claypool Publishers, 2020.

[9] X. Huang, J. Kim, B. Rees, and C.-H. Lee, "Characterizing the efficiency of graph neural network frameworks with a magnifying glass," in *2022 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2022, pp. 160–170.

[10] H. Peng, X. Xie, K. Shivdikar, M. A. Hasan, J. Zhao, S. Huang, O. Khan, D. Kaeli, and C. Ding, "Maxk-gnn: Extremely fast gpu kernel design for accelerating graph neural networks training," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2024, pp. 683–698.

[11] M. Naumov, L. Chien, P. Vandermersch, and U. Kapasi, "Cusparse library," in *GPU Technology Conference*, vol. 12, 2010.

[12] S. Dalton, L. Olson, and N. Bell, "Optimizing sparse matrix—matrix multiplication for the gpu," *ACM Transactions on Mathematical Software (TOMS)*, vol. 41, no. 4, pp. 1–20, 2015.

[13] J. Demouth, "Sparse matrix-matrix multiplication on the gpu," in *Proceedings of the GPU technology conference*, vol. 3, no. 7, 2012.

[14] E. Wang, Q. Zhang, B. Shen, G. Zhang, X. Lu, Q. Wu, Y. Wang, E. Wang, Q. Zhang, B. Shen *et al.*, "Intel math kernel library," *High-Performance Computing on the Intel® Xeon Phi™: How to Fully Exploit MIC Architectures*, pp. 167–188, 2014.

[15] G. Singh, L. Chelini, S. Corda, A. J. Awan, S. Stuijk, R. Jordans, H. Corporaal, and A.-J. Boonstra, "Near-memory computing: Past, present, and future," *Microprocessors and Microsystems*, vol. 71, p. 102868, 2019.

[16] D. U. Lee, K. W. Kim, K. W. Kim, H. Kim, J. Y. Kim, Y. J. Park, J. H. Kim, D. S. Kim, H. B. Park, J. W. Shin *et al.*, "25.2 a 1.2 v 8gb 8-channel 128gb/s high-bandwidth memory (hbm) stacked dram with effective microbump i/o test methods using 29nm process and tsv," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. IEEE, 2014, pp. 432–433.

[17] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze *et al.*, "{TVM}: An automated {End-to-End} optimizing compiler for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 578–594.

[18] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan, "Meet the walkers: Accelerating index traversals for in-memory databases," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, 2013, pp. 468–479.

[19] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee, "Measurement and analysis of online social networks," in *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, 2007, pp. 29–42.

[20] K. Hsieh, E. Ebrahimi, G. Kim, N. Chatterjee, M. O'Connor, N. Vijaykumar, O. Mutlu, and S. W. Keckler, "Transparent offloading and mapping (tom) enabling programmer-transparent near-data processing in gpu systems," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 204–216, 2016.

[21] A. Pattnaik, X. Tang, A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, and C. R. Das, "Scheduling techniques for gpu architectures with processing-in-memory capabilities," in *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, 2016, pp. 31–44.

[22] J. Liu, H. Zhao, M. A. Ogleari, D. Li, and J. Zhao, "Processing-in-memory for energy-efficient neural network training: A heterogeneous approach," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 655–668.

[23] A. Boroumand, S. Ghose, Y. Kim, R. Ausavarungnirun, E. Shiu, R. Thakur, D. Kim, A. Kuusela, A. Knies, P. Ranganathan *et al.*, "Google workloads for consumer devices: Mitigating data movement bottlenecks," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018, pp. 316–331.

[24] M. Drumond, A. Daglis, N. Mirzadeh, D. Ustiugov, J. Picorel, B. Falsafi, B. Grot, and D. Pnevmatikatos, "The mondrian data engine," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2, pp. 639–651, 2017.

[25] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015, pp. 105–117.

[26] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "Pim-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3S, pp. 336–348, 2015.

[27] A. Farmahini-Farahani, J. H. Ahn, K. Morrow, and N. S. Kim, "Nda: Near-dram acceleration architecture leveraging commodity dram devices and standard memory modules," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2015, pp. 283–295.

[28] W. Liu and B. Vinter, "An efficient gpu general sparse matrix-matrix multiplication for irregular data," in *2014 IEEE 28th international parallel and distributed processing symposium*. IEEE, 2014, pp. 370–381.

[29] Y. Nagasaka, A. Nukada, and S. Matsuoka, "High-performance and memory-saving sparse general matrix-matrix multiplication for nvidia pascal gpu," in *2017 46th International Conference on Parallel Processing (ICPP)*. IEEE, 2017, pp. 101–110.

[30] F. Gremse, A. Hofter, L. O. Schwen, F. Kiessling, and U. Naumann, "Gpu-accelerated sparse matrix-matrix multiplication by iterative row merging," *SIAM Journal on Scientific Computing*, vol. 37, no. 1, pp. C54–C71, 2015.

[31] K. Shivdikar, N. B. Agostini, M. Jayaweera, G. Jonatan, J. L. Abellán, A. Joshi, J. Kim, and D. Kaeli, "Neurachip: Accelerating gnn computations with a hash-based decoupled spatial accelerator," in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2024, pp. 946–960.

[32] D. Gurevin, M. Shan, S. Huang, M. A. Hasan, C. Ding, and O. Khan, "Prunegnn: Algorithm-architecture pruning framework for graph neural network acceleration," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2024, pp. 108–123.

[33] A. Tripathy, K. Yelick, and A. Buluç, "Distributed matrix-based sampling for graph neural network training," *Proceedings of Machine Learning and Systems*, vol. 6, pp. 253–265, 2024.