2025 IEEE 36th International Conference on Application-specific Systems, Architectures and Processors (ASAP) | 979-8-3315-9552-4/25/$31.00 ©2025 IEEE | DOI: 10.1109/ASAP65064.2025.00011

AIRES: Accelerating Out-of-Core GCNs via Algorithm-System Co-Design

Shakya Jayakody, Youpeng Zhao, Jun Wang
University of Central Florida
Email: {shakya, youpeng.zhao, jun.wang}@ucf.edu

***Abstract*—Graph convolutional networks (GCNs) are fundamental in various scientific applications, ranging from biomedical protein-protein interactions (PPI) to large-scale recommendation systems. An essential component for modeling graph structures in GCNs is sparse general matrix-matrix multiplication (SpGEMM). As the size of graph data continues to scale up, SpGEMMs are often conducted in an out-of-core fashion due to limited GPU memory space in resource-constrained systems. Albeit recent efforts that aim to alleviate the memory constraints of out-of-core SpGEMM through either GPU feature caching, hybrid CPU-GPU memory layout, or performing the computation in sparse format, current systems suffer from both high I/O latency and GPU under-utilization issues.**

In this paper, we first identify the problems of existing systems, where sparse format data alignment and memory allocation are the main performance bottlenecks, and propose AIRES, a novel algorithm-system co-design solution to accelerate out-of-core SpGEMM computation for GCNs. Specifically, from the algorithm angle, AIRES proposes to alleviate the data alignment issues on the block level for matrices in sparse formats and develops a tiling algorithm to facilitate row block-wise alignment. On the system level, AIRES employs a three-phase dynamic scheduling that features a dual-way data transfer strategy utilizing a tiered memory system: integrating GPU memory, GPU Direct Storage (GDS), and host memory to reduce I/O latency and improve throughput. Evaluations show that AIRES significantly outperforms the state-of-the-art methods, achieving up to 1.8× lower latency in real-world graph processing benchmarks.

***Index Terms*—Graph Convolutional Networks, Sparse General Matrix Multiplication, Compressed Sparse Formats, Memory Optimization**

I. Introduction

Graph convolutional networks (GCNs) have emerged as a powerful tool for processing structured data, particularly in the field of natural recommendation systems, and biomedical applications [7], [19], [30]. One of the most critical operations in GCNs is the sparse general matrix multiplication (SpGEMM) for modeling graph structures [18], [19], [34]. However, the computational demands of this operation can be formidable, especially when dealing with large-scale graphs, motivating the need for efficient acceleration techniques [18]. The high computation costs of SpGEMM stem largely from the enormous size and inherent sparsity of matrices. For instance, in protein-protein interaction (PPI) networks, there are approximately 68 million nodes and 1.5 billion edges, exemplifying such challenges [18], [19]; the multiplication and addition operations for each non-zero element could amount to over 1.36 billion.

The advent of high-performance computing and the ever-increasing demand for processing large datasets have propelled the development of more efficient computational methods and hardware accelerators [23], [25]. Among these advancements, GPUs have emerged as a pivotal technology, offering substantial parallel processing capabilities that far exceed those of traditional CPUs. Such a technological leap has proven particularly beneficial in matrix operations, which are foundational elements integral to the broader field of machine learning [6], [24]. Sparse matrices, characterized by their predominantly zero-valued elements, present unique challenges and opportunities for optimizing GCN system performance [1], [19], [21]. Unlike traditional dense matrix operations, sparse matrices require specialized handling, particularly when scaled to large-scale datasets. This necessity becomes critical in out-of-core scenarios where the data could easily exceed in-memory capacities, driving the need for innovative approaches that leverage the raw computational power and memory hierarchy of modern computer systems to maintain or enhance performance without the significant overhead [27], [29], [32].

Several prior works have attempted to alleviate the memory and computation bottlenecks for large graph-based networks. Notably, UCG [22] proposes a unified CPU-GPU protocol that dynamically balances the workload between the CPU and GPU, incorporates GPU feature caching, unified shared memory, and overlaps communication with computation. ETC [16] further introduces a new batching scheme with a specialized data access strategy for matrices in sparse formats to avoid redundant data transfers. However, despite their promising results, today's GCN systems continue to struggle to achieve high end-to-end throughput, especially in out-of-core environments with a limited GPU memory capacity.

Sparse matrices are often stored in compressed formats to reduce memory usage. In SpGEMM, one common method is to decode the compressed matrix into a dense format before performing the multiplication. Unfortunately, it incurs significant decoding latency and imposes large GPU memory space. An alternative strategy is to perform SpGEMM directly on the compressed (sparse) data on the fly, thereby reducing both memory usage and decoding latency. Nevertheless, even in the compressed form, matrices derived from large, intensive graph data can be extremely large, potentially exceeding available GPU memory and triggering out-of-core processing. To conduct a practical out-of-core SpGEMM with compressed formats, it poses new challenges: 1) **lack of data alignment**

2160-052X/25/$31.00 ©2025 IEEE
DOI 10.1109/ASAP65064.2025.00011

Authorized licensed use limited to: University of Central Florida. Downloaded on September 24,2025 at 13:07:24 UTC from IEEE Xplore. Restrictions apply.

TABLE I: Comparison of prior works and AIRES. DMA and UM denote direct memory access and unified memory, respectively.

	UCG [22]	ETC [16]	AIRES (Ours)
Alignment	✗	✗	✓
DMA	✗	✓	✓
UM reads	✓	✗	✗
Dual-way	✗	✗	✓
Co-Design	✗	✗	✓

control, and 2) **inflexible static memory allocation strategy**. More specifically, large sparse data is generally transferred incrementally by a chunk of data called segments from host memory to the GPU. One straightforward way is to transfer as many rows/columns to maximize GPU memory usage. However, we observe that such a method induces significant CPU pre-processing and I/O overheads: after each compute cycle, the next portion of data often needs to be packed with the last portion of data already transferred to the GPU for merging and staging in the host memory. This is because, upon receiving a non-integer number of rows, the GPU performs matrix operations on complete row data while returning incomplete row data back to the CPU host. Moreover, different from computing in dense formats, the memory required for output allocation is usually dynamic instead of static, decided by both the matching process between the compressed format of the first matrix row and the second matrix column. As the exact number of matches cannot be pre-determined, the memory allocation remains unknown beforehand. Therefore, flexible and adaptive memory management is needed to avoid GPU resource under-utilization and lengthy execution times. Addressing these challenges is essential for enhancing the capabilities of out-of-core GCN systems, ensuring they are equipped to efficiently manage the growing size and complexity of graph data in diverse real-world applications.

In this paper, we introduce AIRES, a novel algorithm-system co-design approach to tackle the above-mentioned challenges. Our focus is on accelerating SpGEMM for GCNs represented in sparse formats, optimized for out-of-core processing on GPUs. Specifically, on the algorithm level, AIRES incorporates a block-wise data alignment solution to alleviate data staging overheads and expedite computation. Built upon such a data alignment strategy, AIRES features a specialized tiling for block-wise partitioned data to optimize compressed matrix multiplication; on the systems level, AIRES employs a three-phase dynamic scheduling protocol that leverages a dual-way path strategy to perform dynamic output memory allocation to maximize GPU resources and further reduce I/O latency.

We implement and validate ARIES across numerous workload scenarios. Extensive evaluations demonstrate that ARIES outperforms existing GCN systems in both end-to-end latency and overall throughput. Specifically, ARIES sustains up to 1.8× and 1.5× runtime improvement against the state-of-the-art solutions, UCG [22], ETC [16] respectively, across various graph datasets.

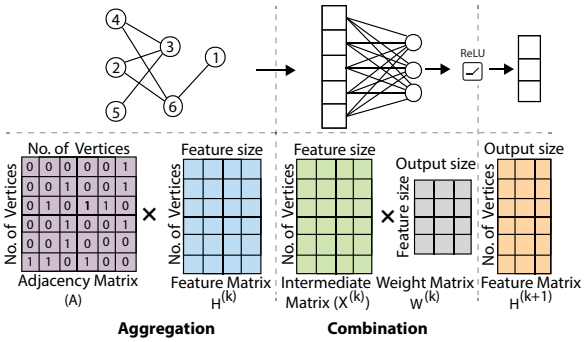

Fig. 1: Illustration of chain matrix multiplication in graph convolutional networks (GCNs) during the aggregation and combination phases.

In summary, this paper makes the following contributions:

- We identify the data alignment and memory allocation challenges in prior GCN systems and propose an algorithm-system co-design solution, AIRES, for efficient out-of-core GCN training in memory-constrained systems.
- On the algorithm level, AIRES features a new block-wise partitioning tailored for compressed matrix formats and a specialized tiling optimization strategy for block-wise partitioned data.
- On the system level, we develop a dynamic scheduling strategy that enables efficient data transfer across a tiered memory system using a dual-path data transfer strategy to avoid resource under-utilization.
- We evaluate AIRES over diverse graph workloads. Experiments demonstrate that AIRES can significantly reduce end-to-end latency against existing systems.

II. Background

A. Graph Convolutional Networks (GCNs)

Graph convolutional networks (GCNs) exhibit parallels to traditional convolutions in the realm of graph analysis, particularly in their approach of sharing "filter" parameters across all graph locations [12], [19]. Figure 1 shows the chain matrix multiplication during the GCN computation. The framework of GCN can be broken down into two primary stages:

1) Aggregation: Each node constructs a feature vector, representing the message it intends to disseminate to all its adjacent nodes. The aggregation equation is defined as:

$$X^{(k)} = \tilde{A}H^{(k)} \tag{1}$$

$$\tilde{A} = \hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}} \tag{2}$$

where $\tilde{A}$ and $\hat{A}$ are the normalized and augmented adjacency matrix, respectively, where $\hat{A} = A + I$, with A being the

Authorized licensed use limited to: University of Central Florida. Downloaded on September 24,2025 at 13:07:24 UTC from IEEE Xplore. Restrictions apply.

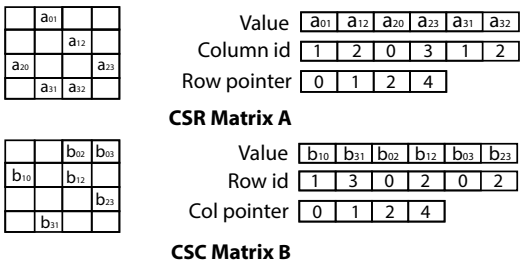

Fig. 2: Examples of the compressed format of matrix A (CSR) and matrix B (CSC).

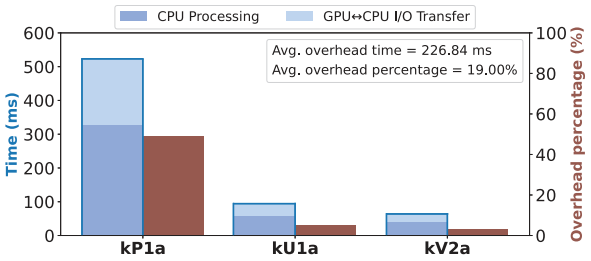

Fig. 3: Merging overhead observed across three different datasets. The latency overhead includes merging the partial segments, and data transfer time between the GPU and host memory. The percentage is measured over the computation latency.

original adjacency matrix and I being the identity matrix, which includes self-loops. The degree matrix $\hat{D}$ is a diagonal matrix where each diagonal element corresponds to the degree of the respective node, ensuring that the feature aggregation is normalized by the connectivity of each node [31].

2) Combination: These messages are sent to neighboring nodes. Consequently, each node receives one message from every adjacent node. The combination equation with the transformation through weight parameters and an activation function is given by:

$$H^{(k+1)} = \sigma\left(X^{(k)}W^{(k)}\right) \tag{3}$$

where $W^{(k)}$ represents the weight matrix at layer k and σ is the activation function, such as ReLU [19]. This step learns trainable parameters that allow the network to combine the aggregated features effectively, adjusting how each node processes its local neighborhood.
Altogether, the computation within a standard GCN layer can be expressed as follows.

$$H^{(k+1)} = \sigma\left(\hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}H^{(k)}W^{(k)}\right) \tag{4}$$

B. Compressed Matrix Formats

In the landscape of data-intensive computing applications, the multiplication of large sparse matrices is a fundamental operation that often presents significant computational challenges [3], [21]. Traditional dense matrix multiplication methods can be highly inefficient when applied to sparse matrices due to unnecessary processing of zero elements, leading to increased computational time and resource consumption [5]. Compressed matrix formats, such as compressed sparse row (CSR) and compressed sparse column (CSC), have been developed to address these inefficiencies [14], [17]. These formats significantly reduce the storage and computation overhead by only storing and processing non-zero elements of the matrices An illustrative example of CSR and CSC matrices is shown in Figure 2.

III. AIRES Design

A. Algorithm Design

Row block-wise (RoBW) Data Alignment. In the era of big data, existing works mainly aim to address the memory and computation challenges through GPU-CPU coordination [22] and batching schemes for graph data [16] to improve system throughput. As the GCN computations are often conducted in an out-of-core fashion, large matrices must be transferred in segments to circumvent GPU out-of-memory (OOM) errors. A naive way to maximize the available GPU memory space is to send out as many rows or columns as possible. Nevertheless, we observe that segments of the original matrices often contain partial rows/columns, which cannot be processed at the current computation cycle. Therefore, such partial data must be repetitively transferred back to host memory to merge with the remaining data, which could lead to significant processing latency overheads.

To corroborate our intuition, we conduct exploratory experiments for three real-world graph datasets, namely kmer_P1a (kP1a), kmer_U1a (kU1a), and kmer_V2a (kV2a) from SuiteSparse [11] and showcase the overheads of merging partial data in Figure 3. We have two key observations. First, the merging overheads are non-negligible. For instance, in the kV2a dataset, the merging overhead accounts for a whopping 50% of the total computation latency. Second, the smaller the allocated GPU memory, the higher the overheads, e.g. the latency overhead of kV2a is about 6× higher than that of kP1a.

To effectively alleviate the above overhead, we propose performing alignment for sparse data segments. Specifically, we employ a row block-wise (RoBW) partitioning strategy to avoid unnecessary merging and I/O operations. Figure 4 illustrates our RoBW partitioning method for CSR A. Instead of naively maximizing the available memory space, we divide the sparse data into blocks, each containing complete and unfragmented data.

The block size p is decided by the dataset on the fly, as each data segment depends on the input CSC B matrix and output matrix CSR C. We first approximate the GPU memory usage M_C for output matrix CSR C as follows:

$$M_C = \frac{3 \cdot \alpha_A \cdot (100 - s_A)}{100} \cdot \left(1 + \frac{\alpha_B}{\alpha_A} + \frac{100 - s_B}{100}\right) \tag{5}$$

where α_A and α_B is the value size for CSR A and CSC

Authorized licensed use limited to: University of Central Florida. Downloaded on September 24,2025 at 13:07:24 UTC from IEEE Xplore. Restrictions apply.

Algorithm 1 Pseudo Code for RoBW Partitioning for CSR A.

Input: A^{rp}, A^{col}, A^{v}: CSR row pointers, column indices, and values for matrix A. N: Total number of row pointers in matrix A. M_A: Available GPU memory for CSR A.
Output: RoBW segment data $RoBW_p^S = (A_p^{rp}, A_p^{col}, A_p^{v})$ $(p = 1, 2, ..., n)$.

$startIdx = 0, n = 0$
while $startIdx < N$ **do**
  $endIdx = startIdx, z = 0$ ▷ Initialize non-zero count in the block
  $q = A^{rp}[endIdx + 1] - A^{rp}[endIdx]$ ▷ Non-zero count for the current row
  $k = endIdx - startIdx + 1$ ▷ Number of rows currently in the block
  while $endIdx < N$ **and** `calcMem`$(k, q) \leq M_A$ **do** ▷ Add rows to the block until memory exceeds limit
    $z \mathrel{+}= A^{rp}[endIdx + 1] - A^{rp}[endIdx], endIdx \mathrel{+}= 1$ ▷ Update non-zero count & expand block
    $q = z + A^{rp}[endIdx + 1] - A^{rp}[endIdx], k = endIdx - startIdx + 1$
  end while
  `malloc`: $A_p^{rp}[k], A_p^{col}[z], A_p^{v}[z]$ ▷ Allocate CPU memory
  $A_p^{rp}[0] = 0, z_p = 0$ ▷ Initialize row pointer and non-zero count for the block
  for $i \leftarrow startIdx$ **to** $endIdx - 1$ **do** ▷ Iterate over rows in block
    $rowstartIdx, rowEndIdx = A^{rp}[i], A^{rp}[i + 1]$
    for $j \leftarrow rowstartIdx$ **to** $rowEndIdx - 1$ **do** ▷ Copy column indices and values
      $A_p^{col}[z_p], A_p^{v}[z_p] = A^{col}[j], A^{v}[j]$
      $z_p \mathrel{+}= 1$
    end for
    $A_p^{rp}[i - startIdx + 1] = z_p$ ▷ Update row pointer for the block
  end for
  $n \mathrel{+}= 1, startIdx = endIdx$ ▷ Update block count & move to the next block
end while

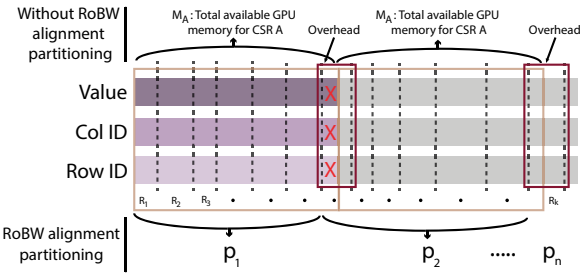

Fig. 4: Illustration of row block-wise (RoBW) partitioning for CSR A. The top is without RoBW alignment partitioning and the bottom is RoBW alignment partitioning. R_1 to R_k is the matrix rows. p is the RoBW alignment partitioning size, n is the number of RoBW.

B, s_A is the sparsity percentage for CSR A, and s_B is the sparsity percentage for CSC B. The memory for CSC B can be calculated as:

$$M_B = (\alpha_B + \beta_B + \theta_B) \tag{6}$$

where $\alpha_B, \beta_B, \theta_B$ are the value size, column id, and row id for CSC B, respectively. Therefore, the optimal block size can be estimated as follows:

$$p = \left(\frac{M - M_C - M_B}{3} \right) \tag{7}$$

Algorithm 1 presents details of dynamically adjusting the block size based on the available memory and processing requirements, where we ensure the optimal utilization of GPU resources.

B. System Design

In dense matrix multiplication, the amount of memory needed to store the output matrix can be easily determined by examining the row or column size directly. In the case of sparse formats, the memory allocation varies, as it also depends on the matching process of the first matrix row and the second matrix column. Therefore, simply reserving the memory based on the matrix size would often lead to resource under-utilization, thus inducing potential latency overheads. Prior work methods, UCG [22] directly utilize row or column sizes and ETC [33] allocate memory equivalent to the larger compressed format, frequently resulting in GPU memory inefficiencies. To combat this issue, we propose a three-phase dynamic scheduling strategy to allocate GPU memory adaptively. Specifically, our design features a dual-way data transfer protocol that leverages three tiers of the memory hierarchy, namely GPU memory, host memory, and NVMe secondary storage.

We illustrate our scheduling with an illustrative example in Figure 5 and a formulation in Algorithm 2. It employs a dual-path strategy where CSC B data is transferred via GPU Direct Storage (GDS) from NVMe directly to GPU memory, while CSR A data is initially transferred to CPU memory where it undergoes preprocessing before being forwarded to the GPU memory. This method effectively harnesses the strengths of both GDS and conventional memory pathways, optimizing

Authorized licensed use limited to: University of Central Florida. Downloaded on September 24,2025 at 13:07:24 UTC from IEEE Xplore. Restrictions apply.

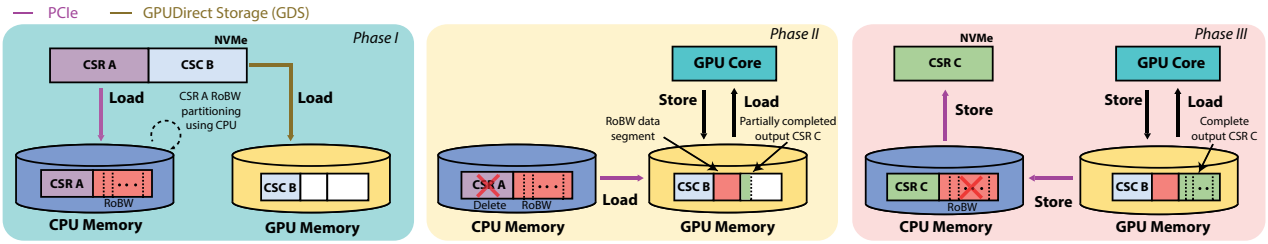

Fig. 5: AIRES memory allocation strategy. In Phase I, CSA A and CSC B dual-way data transfer to CPU memory and GPU memory, respectively, and CSR A RoBW partitioning using the CPU. In Phase II, free memory for CSR A, transfer the RoBW data segment to GPU memory, and compute partial CSR C results using the transferred RoBW data segment. From Phase II to Phase III, free the current RoBW data segment in GPU memory and update it with the next RoBW data segment from CPU to GPU memory, and compute the next partial CSR C; this will keep repeating. In Phase III, complete the output CSR C, free memory for RoBW in CPU memory, transfer the CSR C result to CPU memory, and store the final copy in NVMe.

Algorithm 2 AIRES Dynamic Scheduling.

Initialization: GPU core GC, CPU core CC, GPU memory GM, CPU memory CM, NVMe memory NM, $CSRA$ matrix , $CSCB$ matrix, RoBW segment data $RoBW_p^S$ $(p = 1, 2, ..., n)$.

```
# Load
NM.CSCB → GM.load(∗)                                   ▷ Phase I
NM.CSRA → CM.load(∗)
# Compute
CC.CSRA → CC.RoBW.Process(∗)
for all p < n do                                       ▷ Phase II
    # Load
    if GC.CSRA ≥ RoBW_p^S then
        CM.RoBW_p^S → GM.CSRA.load(∗)
    end if
    GM → GC.load(∗)
    # Compute
    GC.Compute(∗)
    # Store
    GC.CSRC_p → GM.CSRC_p.store(∗)
end for
# Store
GM.CSRC → CM.CSRC.store(∗)                             ▷ Phase III
```

throughput and reducing delays. Our scheduling is described as below:

- **Phase I**: All CSC B data will be loaded into the GPU memory, while the CSR A data will be partitioned and stored in the CPU memory using the RoBW strategy.
- **Phase II**: A batch of data segments from the RoBW partition is transferred from the CPU to the GPU memory. Once in the GPU, partial computations are performed on the available data to produce the output (CSR C), which is stored in the GPU memory.
- **Phase III**: The final output is stored directly in GPU memory, ensuring that the data remains within the GPU for immediate access in subsequent SpGEMM cycles.

As for GCN systems, in the aggregation step, we first initiate with an analysis of the adjacency matrix's byte size in CSR format and the feature matrix in CSC format, and assess the data compatibility with available GPU and CPU memory capacities, avoiding actual data loading to minimize initial overhead.

IV. Implementation

AIRES is an end-to-end system implemented on top of CUDA v12.2 with 1.5K lines of C++ code. In our development of AIRES, we implemented block-wise alignment processing on the CPU using C++. We employ the NVIDIA GPU Direct Storage (GDS) with the cuFile library v1.7 [10] for dynamic scheduling, facilitating rapid data exchanges between storage devices and GPU memory. Additionally, dynamic memory allocation for DMA transfers was managed using *cudaMalloc*, guided by an analytical model to optimize memory usage. This comprehensive approach ensures efficient data handling and computation, leveraging the strengths of both CPU and GPU architectures for out-of-core graph processing. On top of tiling optimization, we implement a specialized compressed sparse matrix multiplication using CUDA kernels to harness GPU capabilities efficiently.

V. Evaluation

A. Experimental Setup

Hardware. We conduct our experiments on a single machine, configured with a single NVIDIA RTX 4090 GPU with 24 GB VRAM, and a 13th Gen Intel(R) Core(TM) i9-13900KF CPU with 128 GB DDR5 memory and 2TB M.2 NVMe SSD. We model the I/O transfer operations and kernel-level computation latency with simulations, which are profiled using the NVIDIA Nsight system.

Model configuration. We train a GCN network with a feature matrix dimension of 256 with 99% uniform sparsity ratio, where one training epoch entails multiple cycles of SpGEMM, activation, and backward gradient descent operations over the entire dataset.

Datasets. For evaluation, we adopt a suite of graph datasets from the SuiteSparse matrix collection [11], namely road_usa

Authorized licensed use limited to: University of Central Florida. Downloaded on September 24,2025 at 13:07:24 UTC from IEEE Xplore. Restrictions apply.

TABLE II: Characteristics of seven diverse graph datasets from SuiteSparse [11], including the number of vertices, edges, the GPU memory requirements, and constraints.

Dataset	No. Vertices (M)	No. Edges (M)	Memory Req. (GB)	Memory Constraint. (GB)
rUSA	23.94	57.70	3.31	3
kV2a	55.04	117.21	6.87	6
kU1a	67.71	138.77	8.2	8
socLJ1	4.84	68.99	12.14	11
kP1a	139.35	297.82	17.45	16
kA2a	170.72	360.58	21.18	18
kV1r	214.00	465.41	27.18	23

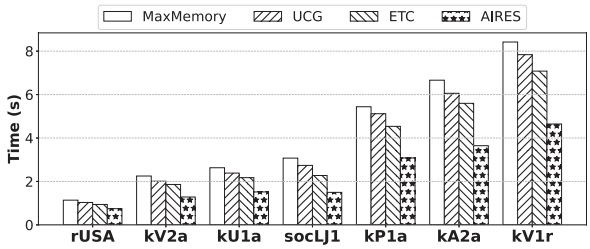

Fig. 6: The AIRES speed up over baseline and state-of-the-art for per-epoch execution time for sub-graph.

(rUSA) from street-network datasets, soc-LiveJournal1 (socLJ1) from social-network datasets [20], and the rest are from the biomedical GenBank datasets, such as kmer_V2a (kv2a). To validate the efficacy of AIRES, we set the GPU memory accordingly to simulate out-of-core scenarios. Memory requirement refers to the combined size of A, B, and C, while memory constraint refers to the total available GPU memory. Details of the datasets are shown in Table II.

Baselines. We compare AIRES with three previous baseline methods and summarized table shown in Table I:

- **MaxMemory.** MaxMemory represents a naive static method that stores a maximum equal amount of both the adjacency matrix and the feature matrix data in GPU memory, with the remainder stored in CPU memory.
- **UCG [22].** UCG [22] designs a unified CPU-GPU protocol designed to train GCNs by utilizing both CPUs and GPUs collaboratively in multi-GPU settings and dynamically balancing the workload between CPU and GPU.
- **ETC [16].** ETC [16] is the state-of-the-art method batching scheme to allow larger batches while minimizing information loss. It employs a three-step data access policy and an inter-batch pipeline mechanism to reduce redundant data access and minimize CPU-to-GPU data transfer.

B. Performance Analysis

End-to-End Results. Figure 6 demonstrates the overall performance comparison across five graph datasets. Due to the limit of computing power, we measure the end-to-end latency every epoch for fair comparison. There are three key observations. First, AIRES achieves consistent speedup overall baseline methods, showing $1.5 \sim 1.8\times$ over ETC. Prior works such as UCG are not optimized for out-of-core resource-constrained systems, therefore inducing higher latency and lower throughput. Second, AIRES is more scalable than previous works. As the dataset size grows, the speedup of AIRES over MaxMemory and other methods increases. Third, AIRES provides consistent latency improvement across datasets of varying sizes. Specifically, AIRES achieves an average total speedup of $1.8\times$, $1.7\times$, and $1.5\times$ over MaxMemory, UCG, ETC, respectively. Such performance gain is attributed to two reasons: 1) AIRES utilizes a block-wise partitioning strategy to effectively minimize the time spent on merging and staging data; 2) the dynamic scheduling in AIRES features an efficient dual-way data transfer mechanism that leverages both DMA and GDS technologies to further reduce I/O latency.

I/O Performance Breakdown of GPU-CPU. To better understand the impact of AIRES's block-wise partitioning, we profile the data transfer time and size between GPU and CPU, with results given in Figure 7. The GPU I/O operations include CUDA memcpy for host-to-device (HtoD) and device-to-host (DtoH), as well as CUDA unified memory (UM) memcpy operations. There are three key observations here in Figure 7. First, AIRES significantly reduces the amount of data transferred between GPU and CPU. For instance, in kA2a dataset, AIRES decreases the amount of data transferred by 84.2% against the MaxMemory method from 30.4 GB to 4.83 GB. Such improvement is due to two reasons: 1) AIRES removes the repetitive merging overheads in the CPU core by transferring sparse data in a block-wise fashion; 2) AIRES optimizes the memory allocation for the output compress matrix, ensuring that memory usage is minimized while maintaining performance, thereby reducing the overall data that needs to be transferred; Second, on top of the reduced data transferred, AIRES also provides consistent latency improvement over baseline methods. For instance, on the kV1r, AIRES reduces the data transfer size and latency by 70% and 75%, respectively, compared to the state-of-the-art ETC method at the same time. Third, AIRES significantly outperforms all other methods across different datasets. The bigger the dataset size, the larger the latency improvement.

I/O Performance Breakdown of GPU/CPU-SSD. To better understand the impact of the dual-way data transfer strategy in AIRES, we further perform bandwidth profiling for data movement between GPU/CPU and secondary storage (SSD). Figure 8, demonstrates that AIRES significantly enhances bandwidth across all datasets when compared to the baseline configuration, thanks to the dual-way path strategy. Specifically, the use of GPU direct storage allows us to transfer more data directly from GPU to SSD instead of the slow PCIe lane, therefore reducing the end-to-end latency.

Ablation Studies. We further conduct ablation experiments to show the impact of GCN model feature size and GPU mem-

Authorized licensed use limited to: University of Central Florida. Downloaded on September 24,2025 at 13:07:24 UTC from IEEE Xplore. Restrictions apply.

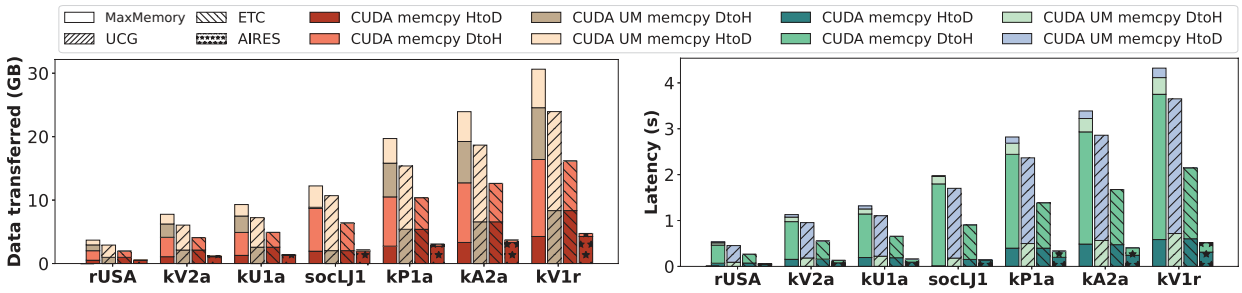

Fig. 7: GPU-CPU I/O performance breakdown for different methods. On the left is data transferred of CUDA memory operations by dataset and on the right is the average latency measured for each operation measured for per-epoch operation. UM stands for Unified Memory, HtoD, and DtoH denote host-to-device and device-to-host, respectively.

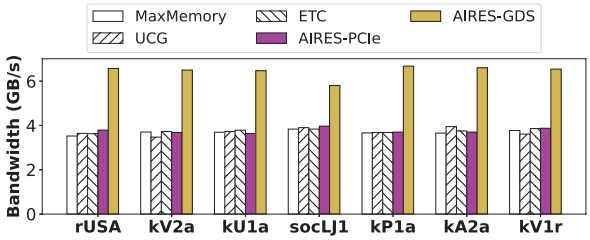

Fig. 8: Bandwidth of GPU/CPU-SSD. GPU-SSD goes through GDS while CPU-SSD is through the PCIe bus.

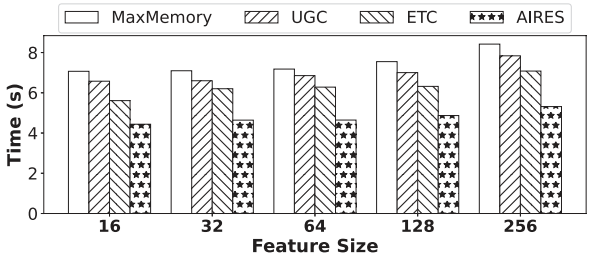

Fig. 9: Per-Epoch execution time evaluation for different GCN feature sizes in AIRES and baselines.

TABLE III: Impact of GPU memory constraints on the per-epoch execution time of AIRES and baseline methods. '-' means out-of-memory error.

Dataset	Mem. constraint (GB)	MaxMemory	UCG	ETC	AIRES
kV1r	24	8.47 s	7.74 s	6.95 s	**4.95 s**
	21	-	-	7.02 s	**5.01 s**
	19	-	-	-	**5.05 s**
kP1a	16	5.75 s	5.07 s	4.49 s	**3.24 s**
	14	-	-	4.63 s	**3.26 s**
	12	-	-	-	**3.37 s**
socLJ1	11	3.18 s	2.80 s	2.29 s	**1.67 s**
	10	-	-	2.34 s	**1.71 s**
	8	-	-	-	**1.76 s**

ory constraints on the end-to-end per-epoch training latency. Figure 9 compares the execution time of different methods under varying feature sizes, ranging from 16 to 256. We can see that AIRES provides consistent speedup across different model configurations, making it a more efficient choice for handling large-scale GCN data. Table III further demonstrates how different methods perform under varying GPU memory constraints for three datasets.

There are two key observations here. First, as the memory constraints decrease, baseline models often run into out-of-core memory (OOM) errors due to the minimum data not available in the GPU memory to complete the computation. In contrast, AIRES demonstrates a robust capability to operate effectively with low memory constraints, showcasing its adaptability to stringent memory constraints. Second, lower memory constraints correlate with increased computation times, as frequent I/O transfers between CPU memory and GPU HBM become necessary to manage data flow and processing continuity. This interaction underscores the importance of dynamic scheduling implemented in AIRES to minimize performance degradation in constrained environments.

VI. Related Work

Sparse General Matrix Multiplication. SpGEMM is a fundamental operation in high-performance computing and scientific applications, particularly in solving large sparse systems and graph algorithms [15]. Existing works on SpGEMM have explored various approaches to optimize performance, such as exploiting parallelism [4], [26], minimizing memory usage [9], [13], improving load balancing [2], and specialized accelerator design [28]. Orthogonal to previous works, AIRES aims to address the SpGEMM operations in out-of-core scenarios.

Memory Optimization. Numerous works have attempted to optimize memory allocations for big data applications [8], [22]. Notably, GMT bridges this gap by adopting a customized approach that maximizes the unique properties of GPU memory to ensure that essential data is stored in the fastest accessible memory [8]. UCG leverages unified memory to distribute workloads between CPUs and GPUs, instead of offloading the majority of tasks to the GPU as traditional methods do [22]. In a similar vein, AIRES introduces a dynamic scheduling strategy for memory optimization to perform memory allocations based on computational needs and data

Authorized licensed use limited to: University of Central Florida. Downloaded on September 24,2025 at 13:07:24 UTC from IEEE Xplore. Restrictions apply.

characteristics, thereby ensuring optimal performance across various out-of-core computations.

VII. Conclusion

In this work, we present AIRES, a novel algorithm-system co-design approach dedicated to enhancing the performance of GCNs in resource-constrained systems, such as single GPU-CPU systems. On the algorithm level, AIRES adopts a row block-wise (RoBW) alignment partitioning method for matrices in compressed format and develops a tiling algorithm for compressed matrix multiplication to facilitate the RoBW. On the system level, AIRES employs a three-phase dynamic scheduling protocol that features a dual-way data transfer strategy within a tiered memory system to maximize GPU utilization. Experiments show that AIRES sustains up to 1.8× latency improvement upon baseline methods.

VIII. Acknowledgments

This work was sponsored in part by the U.S. National Science Foundation (NSF) under Grants 1907765, 2028481, and 2426368. The authors would like to thank the anonymous ASAP reviewers for their constructive feedback to improve this work.

References

[1] S. AlAhmadi, T. Muhammed, R. Mehmood, and A. Albeshri, "Performance characteristics for sparse matrix-vector multiplication on gpus," *Smart Infrastructure and Applications: Foundations for Smarter Cities and Societies*, pp. 409–426, 2020.

[2] P. N. Q. Anh, R. Fan, and Y. Wen, "Balanced hashing and efficient gpu sparse general matrix-matrix multiplication," in *Proceedings of the 2016 International Conference on Supercomputing*, 2016, pp. 1–12.

[3] A. Ashari, N. Sedaghati, J. Eisenlohr, S. Parthasarath, and P. Sadayappan, "Fast sparse matrix-vector multiplication on gpus for graph applications," in *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2014, pp. 781–792.

[4] A. Azad, G. Ballard, A. Buluc, J. Demmel, L. Grigori, O. Schwartz, S. Toledo, and S. Williams, "Exploiting multiple levels of parallelism in sparse matrix-matrix multiplication," *SIAM Journal on Scientific Computing*, vol. 38, no. 6, pp. C624–C651, 2016.

[5] M. M. Baskaran and R. Bordawekar, "Optimizing sparse matrix-vector multiplication on gpus," *IBM research report RC24704*, no. W0812–047, 2009.

[6] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proceedings of the conference on high performance computing networking, storage and analysis*, 2009, pp. 1–11.

[7] P. Bongini, M. Bianchini, and F. Scarselli, "Molecular generative graph neural networks for drug discovery," *Neurocomputing*, vol. 450, pp. 242–252, 2021.

[8] C.-H. Chang, J. Han, A. Sivasubramaniam, V. Sharma Mailthody, Z. Qureshi, and W.-M. Hwu, "Gmt: Gpu orchestrated memory tiering for the big data era," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2024, pp. 464–478.

[9] Y. Chen, G. Xiao, and W. Yang, "Optimizing partitioned csr-based spgemm on the sunway taihulight," *Neural Computing and Applications*, vol. 32, no. 10, pp. 5571–5582, 2020.

[10] N. Corporation, "Nvidia gpu direct storage overview guide," NVIDIA Corporation, Technical Report, 2024. [Online]. Available: https://docs.nvidia.com/gpudirect-storage/index.html

[11] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, pp. 1–25, 2011.

[12] M. Defferrard, X. Bresson, and P. Vandergheynst, "Convolutional neural networks on graphs with fast localized spectral filtering," *Advances in neural information processing systems*, vol. 29, 2016.

[13] Z. Du, Y. Guan, T. Guan, D. Niu, H. Zheng, and Y. Xie, "Accelerating cpu-based sparse general matrix multiplication with binary row merging," *IEEE Access*, vol. 10, pp. 79 237–79 248, 2022.

[14] S. Filippone, V. Cardellini, D. Barbieri, and A. Fanfarillo, "Sparse matrix-vector multiplication on gpgpus," *ACM Transactions on Mathematical Software (TOMS)*, vol. 43, no. 4, pp. 1–49, 2017.

[15] J. Gao, W. Ji, F. Chang, S. Han, B. Wei, Z. Liu, and Y. Wang, "A systematic survey of general sparse matrix-matrix multiplication," *ACM Computing Surveys*, vol. 55, no. 12, pp. 1–36, 2023.

[16] S. Gao, Y. Li, Y. Shen, Y. Shao, and L. Chen, "Etc: Efficient training of temporal graph neural networks over large-scale dynamic graphs," *Proceedings of the VLDB Endowment*, vol. 17, no. 5, pp. 1060–1072, 2024.

[17] J. L. Greathouse and M. Daga, "Efficient sparse matrix-vector multiplication on gpus using the csr storage format," in *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2014, pp. 769–780.

[18] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," *Advances in neural information processing systems*, vol. 30, 2017.

[19] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[20] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[21] C. Li, X. Qin, X. Xu, D. Yang, and G. Wei, "Scalable graph convolutional networks with fast localized spectral filter for directed graphs," *IEEE Access*, vol. 8, pp. 105 634–105 644, 2020.

[22] Y.-C. Lin, G. Deng, and V. Prasanna, "A unified cpu-gpu protocol for gnn training," in *Proceedings of the 21st ACM International Conference on Computing Frontiers*, 2024, pp. 155–163.

[23] D. Luebke, "Cuda: Scalable parallel programming for high-performance scientific computing," in *2008 5th IEEE international symposium on biomedical imaging: from nano to macro*. IEEE, 2008, pp. 836–838.

[24] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda: Is cuda the parallel programming model that application developers have been waiting for?" *Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[25] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "Gpu computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.

[26] M. M. A. Patwary, N. R. Satish, N. Sundaram, J. Park, M. J. Anderson, S. G. Vadlamudi, D. Das, S. G. Pudov, V. O. Pirogov, and P. Dubey, "Parallel efficient sparse matrix-matrix multiplication on multicore platforms," in *International Conference on High Performance Computing*. Springer, 2015, pp. 48–57.

[27] Y. Saad, *Iterative Methods for Sparse Linear Systems*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2003.

[28] L. Song, Y. Chi, A. Sohrabizadeh, Y.-k. Choi, J. Lau, and J. Cong, "Sextans: A streaming accelerator for general-purpose sparse-matrix dense-matrix multiplication," in *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2022, pp. 65–77.

[29] J. Sun, M. Sun, Z. Zhang, J. Xie, Z. Shi, Z. Yang, J. Zhang, F. Wu, and Z. Wang, "Helios: An efficient out-of-core gnn training system on terabyte-scale graphs with in-memory performance," *arXiv preprint arXiv:2310.00837*, 2023.

[30] W. Torng and R. B. Altman, "Graph convolutional neural networks for predicting drug-target interactions," *Journal of chemical information and modeling*, vol. 59, no. 10, pp. 4131–4149, 2019.

[31] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," *arXiv preprint arXiv:1710.10903*, 2017.

[32] R. Waleffe, J. Mohoney, T. Rekatsinas, and S. Venkataraman, "Mariusgnn: Resource-efficient out-of-core training of graph neural networks," in *Proceedings of the Eighteenth European Conference on Computer Systems*, 2023, pp. 144–161.

[33] Q. Wang, Y. Chen, W.-F. Wong, and B. He, "Hongtu: Scalable full-graph gnn training on multiple gpus," *Proceedings of the ACM on Management of Data*, vol. 1, no. 4, pp. 1–27, 2023.

[34] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Y. Philip, "A comprehensive survey on graph neural networks," *IEEE transactions on neural networks and learning systems*, vol. 32, no. 1, pp. 4–24, 2020.

Authorized licensed use limited to: University of Central Florida. Downloaded on September 24,2025 at 13:07:24 UTC from IEEE Xplore. Restrictions apply.