

# Accel-GCN: High-Performance GPU Accelerator Design for Graph Convolution Networks

\*Xi Xie<sup>[1]</sup>, \*Hongwu Peng<sup>[1]</sup>, Amit Hasan<sup>[1]</sup>, Shaoyi Huang<sup>[1]</sup>, Jiahui Zhao<sup>[1]</sup>, †Haowen Fang, Wei Zhang<sup>[1]</sup>, Tong Geng<sup>[2]</sup>, Omer Khan<sup>[1]</sup>, and Caiwen Ding<sup>[1]</sup>

\*These authors contributed equally.

<sup>[1]</sup>University of Connecticut, <sup>[2]</sup>University of Rochester

<sup>[1]</sup>{xi.xie, hongwu.peng, amit.hasan, shaoyi.huang, jiahui.zhao, wei.13.zhang, khan, caiwen.ding}@uconn.edu,

<sup>[2]</sup> haowfang@gmail.com, <sup>[3]</sup>tgeng@ur.rochester.edu

**Abstract**—Graph Convolutional Networks (GCNs) are pivotal in extracting latent information from graph data across various domains, yet their acceleration on mainstream GPUs is challenged by workload imbalance and memory access irregularity. To address these challenges, we present Accel-GCN, a GPU accelerator architecture for GCNs. The design of Accel-GCN encompasses: (i) a lightweight degree sorting stage to group nodes with similar degree; (ii) a block-level partition strategy that dynamically adjusts warp workload sizes, enhancing shared memory locality and workload balance, and reducing metadata overhead compared to designs like GNNAdvisor; (iii) a combined warp strategy that improves memory coalescing and computational parallelism in the column dimension of dense matrices.

Utilizing these principles, we formulate a kernel for SpMM in GCNs that employs block-level partitioning and combined warp strategy. This approach augments performance and multi-level memory efficiency and optimizes memory bandwidth by exploiting memory coalescing and alignment. Evaluation of Accel-GCN across 18 benchmark graphs reveals that it outperforms cuSPARSE, GNNAdvisor, and graph-BLAST by factors of 1.17×, 1.86×, and 2.94× respectively. The results underscore Accel-GCN as an effective solution for enhancing GCN computational efficiency. The implementation can be found on [Github](#)\*

**Index Terms**—Graph Convolution Network, sparse matrix multiplication (SpMM), parallel computing, GPUs

## I. INTRODUCTION

Graph Convolutional Networks (GCNs) [1], [2] are a type of Graph Neural Networks (GNNs) that has drawn tremendous attention in the past years due to their unique ability to extract latent information from graph data [3]. Practical applications of GCNs include prediction of cascading power-grid failure [4], traffic forecasting [5], recommendation systems [6], and drug discovery [7]. The deployment of GCNs in these applications typically poses strict constraints on latency and throughput.

When designing GNN accelerators, GPU platforms have emerged as the mainstream choices. Existing GCN acceleration design mainly process a moderately sparse graph feature matrix ( $X$ ) multiplication with a dense and small weight matrix ( $W$ ), and then multiply the output with the highly sparse and irregular adjacency matrix ( $A$ ). They exhibit two main

challenges: workload imbalance and data locality. The power-law distribution prevalent in the  $A$  matrix of a graph often leads to significant sparsity and irregularity [8], which brings challenges to workload mapping for existing hardware platforms. Conventional workload partition methods [9] for  $A \cdot X$  operation may result in workload imbalances across various warps. Consequently, simple row-wise workload allocation of  $A$  workload could trigger idleness in certain threads, inhibiting overall performance. Efficient SpMM algorithms for  $A \cdot X$  necessitate the effective management of parallelism across both the sparse matrix rows or columns and the resulting dense matrix accumulation to optimally distribute workload, ideally at the warp level.

Contemporary GPUs display a multi-level memory hierarchy [10], and SpMM operations employ memory-efficient formats like CSR or CSC, resulting in irregular data structures and non-coalesced memory accesses. State-of-the-art (SOTA) approaches, including GNNAdvisor [9], Graph-BLAST [11], and cuSPARSE [12], have sought to optimize SpMM performance but exhibit limitations. GNNAdvisor's use of non-zero groups (NG) [9] enhances workload mapping but can result in warp-level workload imbalance and resource underutilization on graphs with power-law non-zero distribution. Graph-BLAST [11], though supporting diverse graph operations and improving memory coalescing, lacks efficiency in SpMM, particularly in dense matrix column dimension traversal. CuSPARSE [12], a strong baseline for SpMM kernels, restricts further insight due to its closed-source nature. However, these approaches encounter performance bottlenecks due to workload balance [9], efficiency in dense dimension traversal [11], and limitations in extensibility or insight due to closed-source development [12].

In this research, we introduce Accel-GCN, an open-source GPU kernel design for GCNs that outperforms SOTA methods and libraries, including GNNAdvisor [9], graph-BLAST [11], and cuSPARSE [12]. Accel-GCN aims to enhance various computational aspects such as data locality, multi-level memory efficiency, workload assignment, and memory access coalescing through the integration of degree sorting, block-level partition, and combined warp techniques. The core contributions of this work are encapsulated in the design of the CUDA kernel that leverages the aforementioned techniques and are

\*<https://github.com/xiexi1990/ICCAD-Accel-GNN>

†H. Fang is now affiliated with Synopsys, Mountain View, CA.

outlined as follows:

- A degree sorting based preprocessing step with  $\mathcal{O}(n)$  complexity is proposed. This lightweight step aims to enhance data locality and facilitate workload mapping by grouping rows with identical degrees together.
- A block-level partition scheme is developed to dynamically adjust warp workload sizes across different blocks. Compared to SOTA designs like GNNAdvisor [9] which use a fixed workload size assignment, the dynamical allocation creates a more balanced workload among warps. We further customize a metadata format for block-level partition, enabling all warps within a block to share a single metadata for workload mapping. As such, this strategy enhances workload balance and shared memory reuse efficiency.
- A combined warp strategy is formulated to maximize thread-address continuity in the traversal and processing of dense matrix's column dimension. This approach furthers the cause of column dimension memory coalescing and computational parallelism, leading to a more efficient execution.

We conduct evaluation of Accel-GCN on 18 benchmark graphs and observe a significant performance boost. The average improvements are  $1.17\times$  over cuSPARSE [12],  $1.86\times$  over GNNAdvisor [9], and  $2.94\times$  over graph-BLAST [11].

## II. PRELIMINARY AND RELATED WORK

### A. Graph Convolution Network

Graph Convolutional Networks (GCNs) [1], comprised of GCNConv layers, undergo two primary stages: linear transformation and feature aggregation, as illustrated in Fig. 1. Given a graph  $G = (\mathcal{V}, \mathcal{E}, A)$ , where the adjacency matrix  $A$  represents the existence of edges between nodes, the forward propagation in the  $l$ -th GCNConv layer can be decoupled into: (1) linear transformation,  $Y^l = X^l W^l$ , and (2) feature aggregation,  $X^{l+1} = \sigma(A' Y^l)$ . Here,  $X^l$  is the feature embedding matrix,  $W^l$  denotes the weight matrix, and  $A'$  is the normalized adjacency matrix. The activation function, typically an element-wise ReLU, calculates the feature embedding matrix output.

GCN variants like GraphSAGE [13] and Graph Isomorphism Network (GIN) [14] maintain similar structures but with distinct aggregation functions, upholding the same forward propagation model as traditional GCNs. The efficiency of GCNs is contingent on the feature aggregation stage [8], [15], chiefly executed as sparse matrix multiplication (SpMM) between the adjacency list  $A'$  and embeddings  $Y^l$ . This ultra-irregular operation is typical in GCNConv layers, and given the significant role of SpMM in GCN, its acceleration is essential for boosting GCN performance.

### B. SpMM Acceleration

GraphBLAST [11], [16], GE-SpMM [17], and Jiang et al. [18] present distinct methods for enhancing sparse matrix-matrix (SpMM) multiplication on GPUs. GraphBLAST adopts

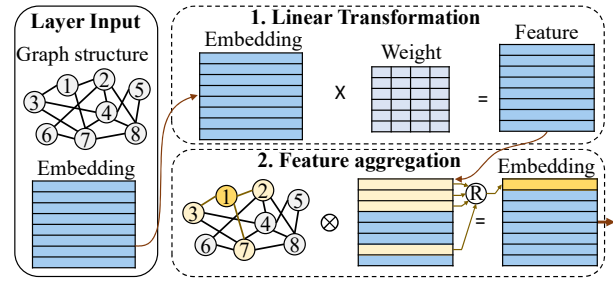


Fig. 1. Computational workflow of single GCNConv layer.

a block-based SpMM algorithm utilizing instruction-level parallelism to minimize latency and combines it with a "row-splitting" memory access pattern and "static scheduling" load balancing for efficient matrix access and adaptive thread allocation. Conversely, GE-SpMM offers an effective CSR-based SpMM kernel for generalized integration with Graph Neural Network (GNN) algorithms, eliminating data conversion overhead. Jiang et al. introduce a row-reordering technique to improve SpMM performance across hardware platforms and investigate its applicability for diverse parameter settings.

GNNAdvisor [9] introduces an adaptive runtime system designed for GNN acceleration on GPUs, incorporating specialized memory optimizations, community-aware node renumbering, and warp-aware memory customization. Notably, SpMM is utilized in Deep Graph Library (DGL) for sum-reduced aggregation. While these methods represent significant advancements, limitations persist in practical applications, primarily due to right-multiply matrix dimensional constraints [11], [16]–[18]. Jiang et al.'s row-reordering method is challenging to implement on-the-fly, and GNNAdvisor relies on shared memory caching for performance improvements, which does not entirely benefit from memory alignment and varies with the right-hand matrix dimensions.

In summary, two primary challenges within the SpMM domain require optimization: computational workload unbalance and memory access irregularity.

## III. ACCEL-GCN FRAMEWORK

### A. Motivation

1) **Workload Allocation Matters**: The adjacency matrix of a graph, often exhibiting a power-law distribution and extreme sparsity [8], can lead to load imbalance across warps and blocks with naive workload partitioning [9]. As shown in the Collab graph degree distribution [19], nodes can have degrees up to 66 times greater than the average, as presented in Fig. 2. This discrepancy may cause uneven workload allocation, resulting in idle threads and worse performance.

Moreover, efficient utilization of GPU resources in SpMM requires exploiting multiple levels of parallelism, encompassing parallelism across rows or columns of the sparse matrix and the computation of resulting dense matrix elements. Designing an algorithm that effectively manages this parallelism can be challenging. Therefore, an SpMM algorithm must distribute the workload judiciously and efficiently, ideally down to the level of individual warps.

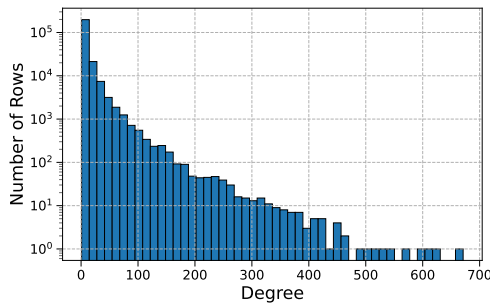


Fig. 2. A histogram for the row degree distribution of Collab.

2) **Memory Access Patterns Matters:** Modern GPUs exhibit a three-level memory hierarchy [10]: global DDR memory, cache (L1 and L2), and shared memory. In SpMM operations, sparse matrices use efficient representations such as CSR or CSC, leading to irregular data structures and non-coalesced memory accesses. These access patterns can result in increased latency [8], [18], memory bank conflicts, and reduced memory bandwidth utilization. To optimize SpMM performance, all levels of the GPU memory hierarchy should be considered. We summarize two challenges as follows:

a) **Shared Memory Mapping Challenge.** Efficient use of shared memory can help minimize global memory access latency [10]. However, due to the irregular structure of sparse matrices, effectively using shared memory for SpMM can be challenging. Loading data into shared memory might involve complex indexing and synchronization, which can increase overhead and diminish performance benefits.

b) **Cache Locality Challenge.** The irregular memory access patterns in SpMM can lead to inefficient cache utilization [20], causing cache thrashing and increased memory access latency. Designing algorithms that maximize cache utilization can be challenging due to the non-uniform distribution of non-zero elements in the sparse matrix.

To address these challenges, various optimization techniques can be applied, including selecting appropriate sparse matrix formats [18], [21], exploiting multiple levels of parallelism [16], implementing coalesced memory access patterns [17], efficiently using shared memory [9], and employing dynamic load balancing techniques [8]. However, in spite of these optimizations, SpMM often remains a complex task in terms of both computational and memory aspects.

### B. *Accel-GCN Preliminary*

Sparse Matrix Representation and Reordering significantly influence SpMM algorithm performance, with proper techniques enhancing memory access, minimizing complexity, and optimizing performance. Research has developed reordering input data and new sparse matrix representations, such as ELLPACK-R in FastSpMM [21], SELLP in MAGMA [22], register blocking in OSKI [23], and Compressed Sparse Blocks (CSB) [24]. These implementations have yielded performance gains. RS-SpMM [25] introduced a format for better SpMM data locality, albeit with limitations.

I-GCN [15] evaluated graph reordering techniques including HATS [26], SlashBurn [27], and Rabbit [28], each with their own strengths and limitations. HATS [26] enhances cache hierarchy efficiency, whereas SlashBurn [27] clusters non-zeros effectively but requires complex, non-parallelizable logic. Rabbit reordering [28] outperforms other approaches in terms of data locality, parallelization ease, and performance but is unsuitable for *Accel-GCN* due to its processing overhead compared to GCN inference.

**Shared Memory Utilization and Alignment** Efficient alignment and utilization of shared memory are vital for optimizing GPU performance. Coalesced memory access [29], [30] enhances this efficiency but aligning access when writing to global memory can be challenging. Padding the shared memory array to the nearest multiple of 32 optimizes alignment when handling intermediate SpMM results [11], [16], [17]. Though optimizing global memory alignment is complex, shared memory offers opportunities for alignment improvement, thus potentially enhancing SpMM performance, given proper padding and indexing management. Some previous works, such as GraphBLAST [11], [16], GE-SpMM [17], Jiang et al. [18], and GNNAdvisor [9], have faced alignment inefficiencies in corner cases, leading to suboptimal performance.

Although optimizing memory access alignment for global memory is challenging, using shared memory for intermediate results offers opportunities for alignment optimization, potentially improving SpMM performance. However, proper management of padding and indexing is necessary to ensure accurate results and avoid memory access conflicts.

Certain prior works, including GraphBLAST [11], [16], GE-SpMM [17], Jiang et al. [18], GNNAdvisor [9], and MergePath-SpMM [31], have encountered inefficient alignment in corner cases, resulting in suboptimal memory system performance.

### C. *Accel-GCN Preprocessing*

Our *Accel-GCN* design incorporates two key preprocessing steps—degree sorting and block-level partitioning—to enable efficient parallel processing of sparse matrices.

**Degree Sorting.** Degree sorting serves as a preliminary step for block-level partitioning. Sorting sparse matrix degree in a CSR-formatted sparse matrix requires the following steps: (1) computing each row's degree using the row pointer array, which has a time complexity of  $\mathcal{O}(n)$  when employing count sort [32] or radix sort [33], with  $n$  indicating the number of rows; (2) applying a stable sorting algorithm to sort rows based on the degrees; and (3) updating the row pointer array to reflect the new row order, with a time complexity of  $\mathcal{O}(n)$ . The dominant time complexity of this operation arises from applying the stable sorting algorithm. Nevertheless, employing count sort, a linear time-complexity algorithm, can optimize the overall time complexity to  $\mathcal{O}(n)$ . This lower time complexity enhances efficiency compared to alternative algorithms and allows on-the-fly execution.

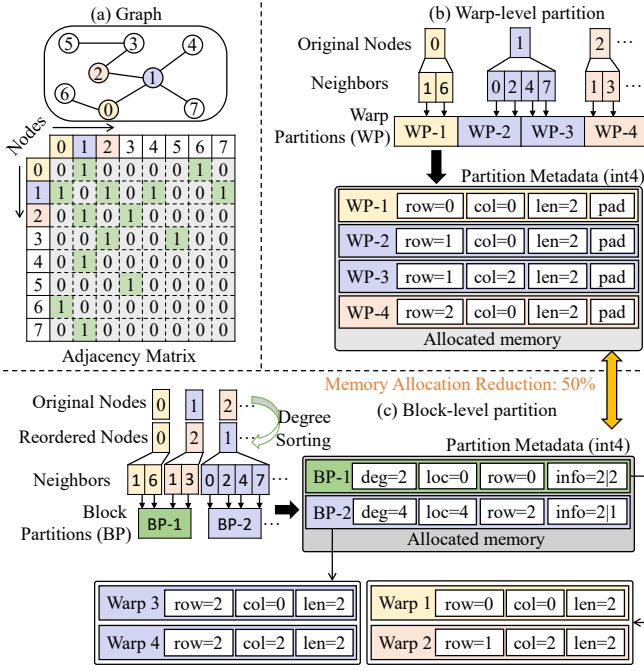


Fig. 3. Metadata generation process: (a) original graph structure. (b) Metadata of warp-level partition. (c) Metadata of block-level partition.

#### Algorithm 1 Get partition patterns

```

1:  $deg\_bound \leftarrow max\_block\_warps \times max\_warp\_nzs$ ;
2:  $factors \leftarrow$  all factors of  $max\_block\_warps$ ;
3:  $i \leftarrow 0$ ;  $deg \leftarrow 1$ ;
4: while  $deg < deg\_bound$  do
5:   if  $factors[i] \times warp\_max\_nz \geq deg$  then
6:     assign  $block\_rows \leftarrow \frac{max\_block\_warps}{factors[i]}$  to current
       block-partition pattern;
7:     assign  $warp\_nzs \leftarrow \lceil \frac{deg}{factor[i]} \rceil$  to current block-
       partition pattern;
8:      $deg ++$ ;
9:   else
10:     $i ++$ ;
11:  end if
12: end while

```

**Block-level Partition.** Block-level partition serves as a highly effective method for optimizing workload distribution among warps, which is also within a time complexity of  $\mathcal{O}(n)$ . This approach not only enhances computational resource allocation but also notably reduces the meta-data size necessitated for the partitioning process.

The block-level partition algorithm consists of two parts. Algorithm 1 presents the first part, get partition patterns, which entails determining the maximum number of warps per block and the maximum number of non-zero elements that each warp can handle. Their product, referred to as  $deg\_bound$ , signifies the maximum number of non-zero elements manageable by a single block.

For rows with a degree (number of non-zero elements)

#### Algorithm 2 Block-level partitioning

```

1: for each  $deg$  do
2:   if  $deg \leq deg\_bound$  then
3:      $row\_remaining \leftarrow$  total number of rows of  $deg$ 
4:     while  $rows\_remaining \geq pattern[deg].block\_rows$  do
5:        $\{row, loc, deg, warp\_nzs | block\_rows\} \rightarrow$  current
       metadata;
6:        $rows\_remaining -= pattern[deg].block\_rows$ ;
7:     end while
8:      $\{row, loc, deg, warp\_nzs | rows\_remaining\} \rightarrow$  cur-
       rent metadata;
9:   else
10:     $deg\_remaining \leftarrow deg$ 
11:    while  $deg\_remaining \geq deg\_bound$  do
12:       $\{row, loc, deg, deg\_bound\} \rightarrow$  current metadata;
13:       $deg\_remaining -= deg\_bound$ ;
14:    end while
15:     $\{row, loc, deg, deg\_remaining\} \rightarrow$  current metadata;
16:  end if
17: end for

```

less than or equal to  $deg\_bound$ , each block processes one or more rows. By enumerating every factor  $factor_i$  from 1 to  $max\_block\_warps$ ,  $factor_i$  warps process rows with degrees not exceeding  $factor_i \cdot max\_warp\_nzs$ , while  $\frac{max\_block\_warps}{factor_i}$  rows are allocated to one block. When a row's degree surpasses  $deg\_bound$ , non-zero elements are assigned across multiple blocks to maximize loading.

The second part is described by Algorithm 2. After traversing through all rows of the graph once, the block-level partition algorithm generates meta-data for each block, shared by all warps within the same block. To fully exploit modern GPU read and write bandwidth, which permits reading and writing 128 bits simultaneously, the meta-data consists of an array of int4 data structures with a length equal to the number of blocks. The meta-data encompasses four elements: the block's degree, starting row number, starting address, and additional 32 bits of information. When the block's degree does not exceed  $deg\_bound$ , the additional information is split into two 16-bit segments, one for the number of non-zero elements handled by each warp and the other for the number of rows handled by the block. If the block's degree is greater than  $deg\_bound$ , the additional information stores the number of non-zero elements assigned to the block. Since the block-level partitioning algorithm can be completed with a single pass through the rows of the graph, its time complexity is also within  $\mathcal{O}(n)$ . Therefore, the combined time complexity of degree sorting and block-level partitioning is also within  $\mathcal{O}(n)$ . Moreover, both algorithms are straightforward and suitable for on-the-fly execution.

**Metadata Format for Block-level Partition.** Figure 3 illustrates a representative example contrasting the metadata formats of block-level partition and warp-level partition.

In the warp-level partition depicted in Fig. 3(b), each warp

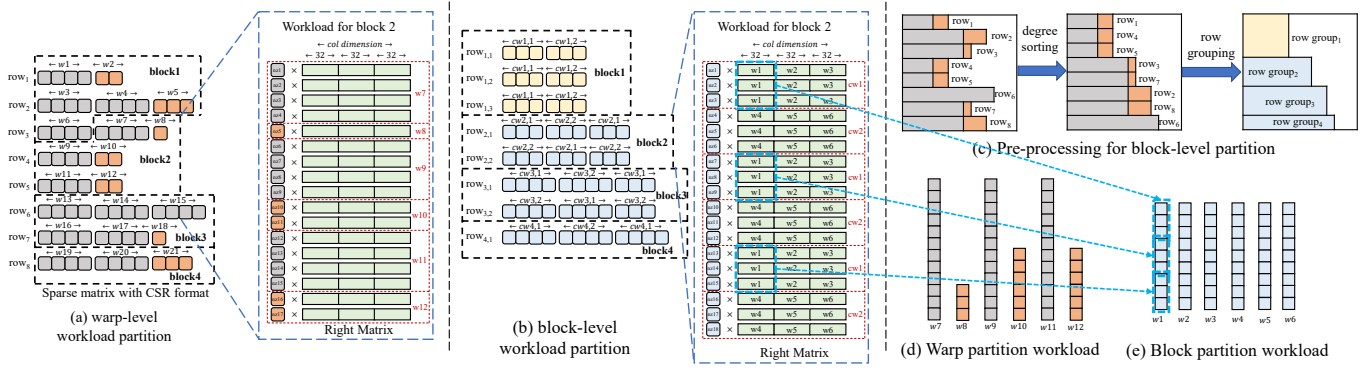


Fig. 4. Illustration of hardware mapping strategy: (a) the warp-level workload partitioning with a single warp traversing the column dimension of the dense matrix. (b) The block-level workload partitioning (right) with the combined warp strategy handling the column dimension of the dense matrix. (c) Preprocessing steps for block-level partitioning strategy. (d) Workload distribution of warp-level partitioning. (e) Workload distribution of block-level partitioning.

manages at most 2 non-zero elements. For instance,  $WP-1$  oversees all non-zero elements of  $row_0$  starting at  $col_0$ , and its corresponding metadata is  $row = 0, col = 0, len = 2$ . Metadata for  $WP-2$ ,  $WP-3$ , and  $WP-4$  are constructed analogously. The cumulative warp metadata amount to 96 bits, necessitating 32 bits of padding to align with the 128-bit memory bus size.

Fig. 3(c) explicates the block-level partitioning process. Initially, degree sorting is applied to the graph nodes, resulting in an ordered sequence of  $row_0, row_2$ , and  $row_1$  for  $row_0, row_1$ , and  $row_2$ , respectively. Subsequently, block-level partitioning is executed such that each warp manages at most two non-zero elements and each block encompasses two warps. Consequently, both  $row_0$  and  $row_2$  (each with a degree of 2) encompassing a total of 4 non-zero elements are governed by  $BP-1$ , while  $row_1$  falls under the jurisdiction of  $BP-2$ .

The metadata for block-level partitioning is encapsulated within an int4 array. It comprises the degree of the rows overseen by the block, the position of the first non-zero element, the row number of the initial row, and ancillary details, encompassing the quantity of non-zero elements handled by each warp and the number of rows managed by the block (each represented by 16 bits). Therefore, the metadata for  $BP-1$  is  $deg = 2, loc = 0, row = 0, info = 2|2$ , and the metadata for  $BP-2$  is  $deg = 4, loc = 4, row = 2, info = 2|1$ . The metadata storage of block-level partitioning normalized to warp-level partitioning is:

$$\frac{S_B}{S_W} \approx \frac{1}{\text{Avg. Warps per Block}} \quad (1)$$

Block-level partitioning, in comparison to warp-level partitioning, exhibits significant storage efficiency, typically requiring less than 10% of the storage space. For instance, with a parameter of  $max\_block\_warps$  set to 12, the block-level partitioning strategy necessitates a mere 8% of the metadata storage relative to the warp-level approach.

One salient advantage of this approach is that the workload allocation for each warp within a block can be directly deduced from the block-level partition's metadata. Consider  $BP-1$ , encompassing  $Warp-1$  and  $Warp-2$ . Given that the starting

row of  $BP-1$  is 0, the degree ( $deg$ ) is 2, the number of accountable rows is 2, and each warp manages 2 non-zero elements, the responsibility for  $row_0$  and  $row_1$  is assigned to  $Warp-1$  and  $Warp-2$ , with corresponding column values of 0 and 2, respectively. This logic extends to other warps, such as  $Warp-3$  and  $Warp-4$  within  $BP-2$ , enabling a consistent and systematic workload allocation.

The efficacy of block-level partitioning is further elucidated in Fig. 4(e). For all rows with a degree less than or equal to  $deg\_bound$ , the block-level partitioning patterns ensure a uniform workload distribution within each block. This stands in stark contrast to the warp-level partitioning, which exhibits a differentiated and uneven workload distribution. Consequently, the block-level partition patterns mitigate the decreased utilization of issue slots often associated with higher warp inactivity rates when handling residual workloads. This efficient alignment with the underlying computational architecture enhances parallelism and, ultimately, execution efficiency.

#### D. Accel-GCN Mapping

**Combined Warp for Block-Warp Mapping.** The combined warp approach represents an exceptionally efficient organizational strategy for addressing dense matrix dimensions. By leveraging memory coalescing and alignment, this method effectively optimizes memory bandwidth utilization, contributing to enhanced performance in the context of GPU acceleration.

SPMM is distinguished from Sparse Matrix-Vector Multiplication (SpMV) by its essential aspect of column dimension traversal of the right matrix. The traversal method for the right matrix significantly impacts performance, as SpMM is typically memory-bound.

When the column dimension of the right matrix surpasses the number of threads within a warp (usually 32), a single warp cannot accommodate the column dimension workload, necessitating traversal. Previous works, such as GNNAdvisor [9], adopt the natural method of introducing an inner loop for this warp. However, this approach introduces instruction-level branching and jumps, which, combined with memory

TABLE I  
GRAPH DATASETS DETAILS

Graph Name	# Nodes	# Edges	Graph Name	# Nodes	# Edges	Graph Name	# Nodes	# Edges
am	881,680	5,668,682	amazon0601	403,394	5,478,357	Artist	50,515	1,638,396
Arxiv	169,343	1,166,243	Citation	2,927,963	30,387,995	Collab	235,868	2,358,104
com-amazon	334,863	1,851,744	OVCAR-8H	1,889,542	3,946,402	PRODUCTS	2,449,029	123,718,280
Pubmed	19,717	99,203	PPA	576,289	42,463,862	Reddit	232,965	114,615,891
SW-620H	1,888,584	3,944,206	TWITTER-Partial	580,768	1,435,116	wikikg2	2,500,604	16,109,182
Yelp	716,847	13,954,819	Yeast	1,710,902	3,636,546	youtube	1,138,499	5,980,886

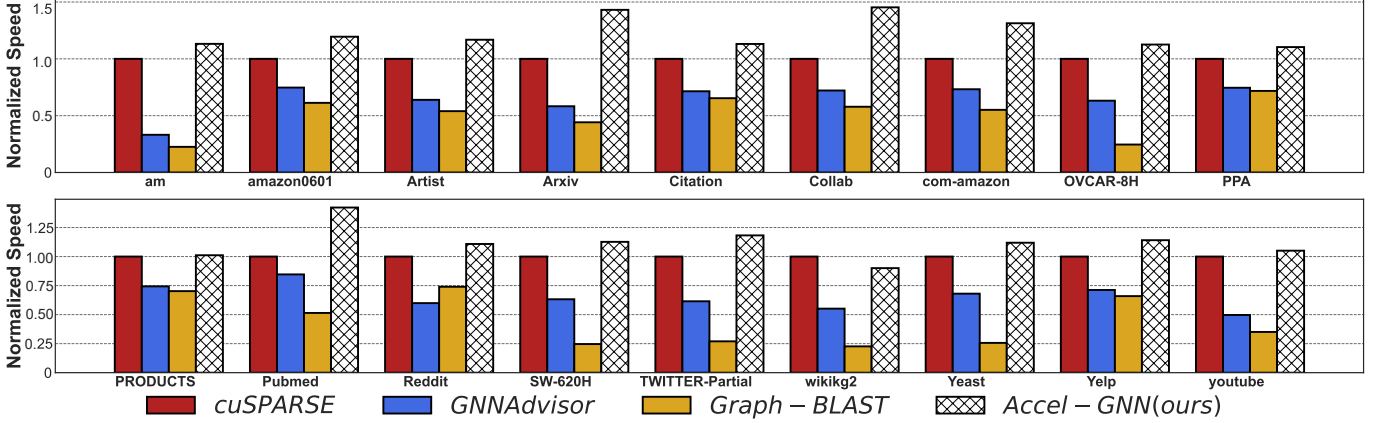


Fig. 5. Overall kernel performance comparison for cuSPARSE, GNNAdvisor, Graph-BLAST, and Accel-GCN (ours) on each graph. The speedup is normalized to cuSPARSE.

positions mapped according to non-zero element positions, may fragment memory coalescing in the column dimension and lead to reduced efficiency.

To address these issues, we propose a combined warp execution approach that combines several consecutive warps, treating them as a single combined warp tasked with handling the entire column dimension workload. This method results in continuous memory position access within a combined warp’s column dimension, improving cache hit rate and memory coalescing.

The implementation strategy for the combined warp unfolds as follows: First, compute the column dimension divided by 32 and round up. This produces the number of warps  $c$  within the combined warp, which defines  $round\_dim = c \times 32$ . The kernel computation’s outermost loop includes an additional loop ranging from 0 to  $c - 1$ , incrementing the thread id by the block dimension with each iteration to produce a new thread id. The warp id for the combined warp is then determined by dividing this thread id by  $round\_dim$ , and the division’s remainder provides the lane id. Threads featuring a lane id greater than or equal to the right-hand matrix’s column dimension are truncated, and the combined warp supplants the single warp for workload execution.

An illustration of the combined warp strategy is provided in Figure 4(b), elucidating its contrast to the approach presented in Fig.4(a), as adopted in GNNAdvisor [9]. In this example, a dense matrix with a column dimension of 96 is considered.

The combined warp strategy groups warps  $w_1$  to  $w_3$  together to form a combined warp  $cw_1$ , and likewise,  $w_4$  to  $w_6$  are amalgamated to form  $cw_2$ . These combined warps,  $cw_1$  and  $cw_2$ , are delegated with the responsibility of handling all workloads of  $NZ_1$  to  $NZ_3$  and  $NZ_4$  to  $NZ_6$ , respectively. Subsequent workloads for other non-zero groups ( $NZ$ s) are sequentially assigned to  $cw_1$  and  $cw_2$ .

Contrary to the approach in Fig. 4(a), where a single warp loops through the column dimension to process the single workload group, the combined warp strategy endeavors to access memory addresses in one row using continuous thread IDs. This method thereby maximizes memory coalescing and computational parallelism, offering an advantageous approach for optimizing memory access patterns and improving execution efficiency.

**Summary and Further Enhancement:** We present an optimized computational approach that leverages the GPU’s memory hierarchy and CUDA’s *atomicAdd\_block* feature. The proposed hierarchical warp computation strategy systematically accumulates partial results across three cache levels.

In the first cache level, independent parallel threads manage non-zero elements along the column dimension. The second cache level ensures atomicity among all combined warps handling identical row workloads within the same block. Utilizing CUDA’s *atomicAdd\_block* function, atomic operations within shared memory are enabled.

The third cache level addresses rows with degrees exceeding

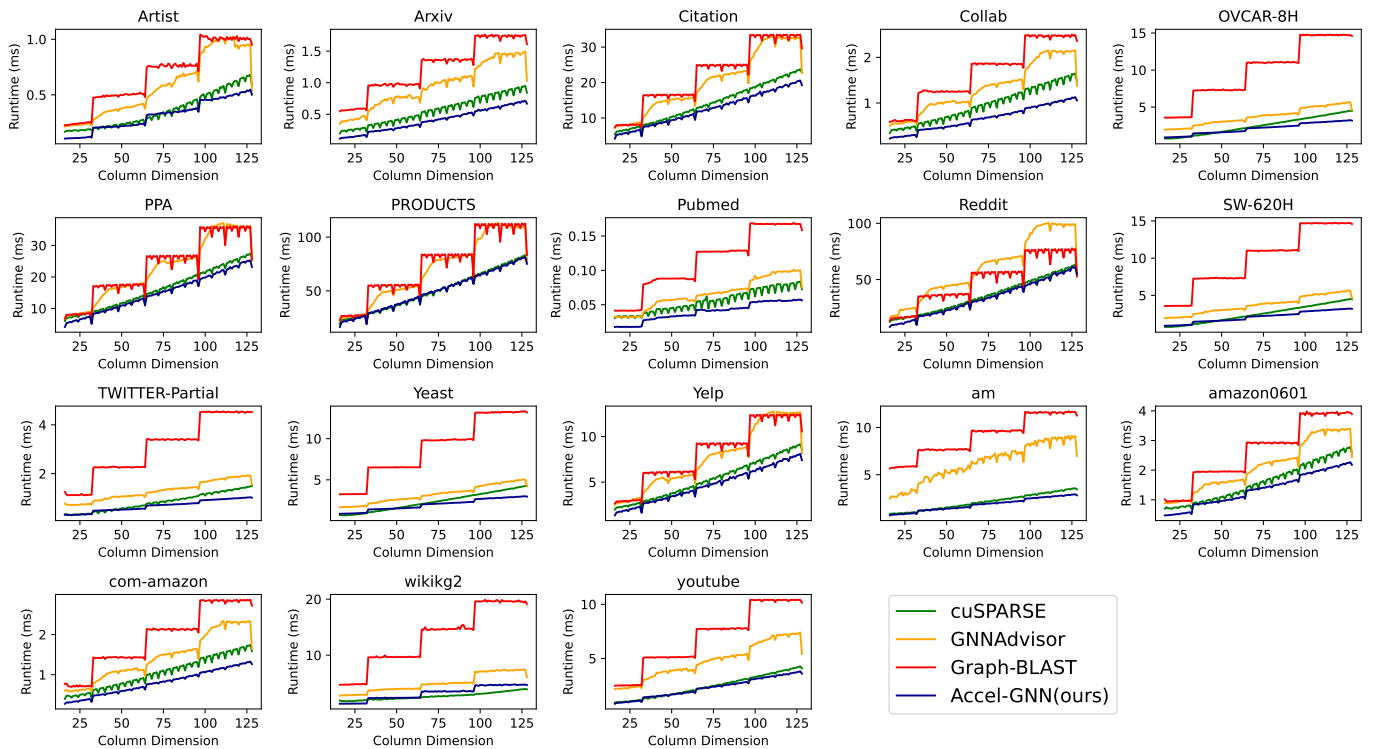


Fig. 6. The average SPMM kernel execution times for Accel-GCN (ours), GNNAdvisor, Graph-BLAST, and cuSPARSE have been tested on each graph, with the right-hand matrix’s column dimensions ranging from 16 to 128.

the *deg\_bound* by concurrently executing partial results in multiple blocks and atomically accumulating them in global memory. The combined warp approach aligns shared memory, with higher kernel performance achieved when the column dimension is an integer multiple of 32.

Overall, this strategy efficiently utilizes memory hierarchy and atomic features, demonstrating significant performance gains and offering a pathway for future enhancements.

#### IV. EXPERIMENTAL ANALYSIS

##### A. Experimental Framework

The CUDA source code used in this study is compiled utilizing NVCC, version 12.0, and the execution is carried out on an Nvidia GeForce RTX 3090 platform equipped with Ubuntu 20.04. The experimental design involves the execution of SPMM with the left-hand sparse matrix from 18 benchmark graphs, specified in Table I. The tests encompass scenarios where the column dimensions of the right-hand matrix vary from 16 to 128.

The performance of our proposed kernel is gauged against benchmark techniques such as GNNAdvisor [9], Graph-BLAST [16], and the recent cuSPARSE, version 12.0. The latency measurements are conducted using the Nsight Compute [34] tool. Note that the comparisons mainly focus on kernel execution time, excluding data transfer and preprocessing durations. Moreover, the impact on performance introduced

by the adoption of block-level partition and combined warp strategy is evaluated.

The selection of graphs for this experiment involves popular benchmark datasets widely used in previous research [9], [13], [35]–[39]. Table I provides detailed parameters for each graph. The range of graph sizes in the tests extends from a node count of 19,717 to 2,927,963, edge numbers ranging from 99,203 to 123,718,280, and densities spanning from  $1.1 \times 10^{-6}$  to  $2.1 \times 10^{-3}$ . This diverse array of sizes and densities facilitates a comprehensive appraisal of the evaluated kernels’ performance, demonstrating their scalability and proficiency under various circumstances.

TABLE II  
IMPACT OF BLOCK-LEVEL PARTITIONING AND COMBINED WARP

Speed Ratio(%) Column Dimension Range	Block-Level Partition			Combined Warp		
	Avg	Max	Min	Avg	Max	Min
[16, 32]	105.2	129.2	92.4	133.4	194.5	104.8
(32, 64]	107.2	130.7	94.1	127.8	174.0	87.3
(64, 96]	106.5	127.7	92.4	105.5	126.5	81.3
(96, 128]	106.8	126.0	92.9	122.9	156.0	86.7

##### B. Performance Evaluation

As illustrated in Fig. 5, our proposed kernel depicts a comprehensive enhancement in performance compared to cuSPARSE in the majority of the cases, and distinctly outperforms

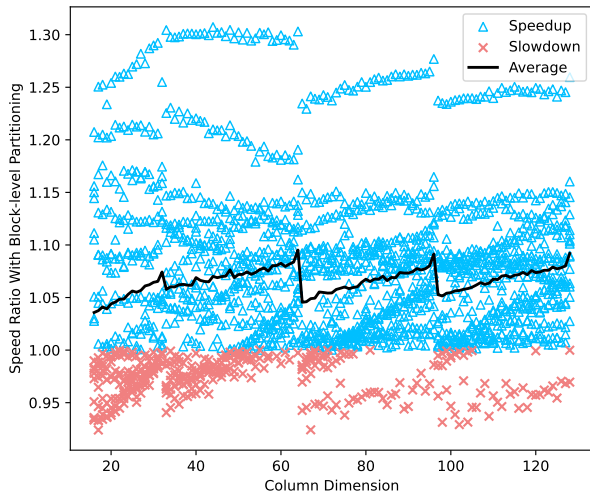


Fig. 7. Speedup of (i). degree sorting & block-level partition over (ii). warp-level partition. Both integrated with combined-warp strategy.

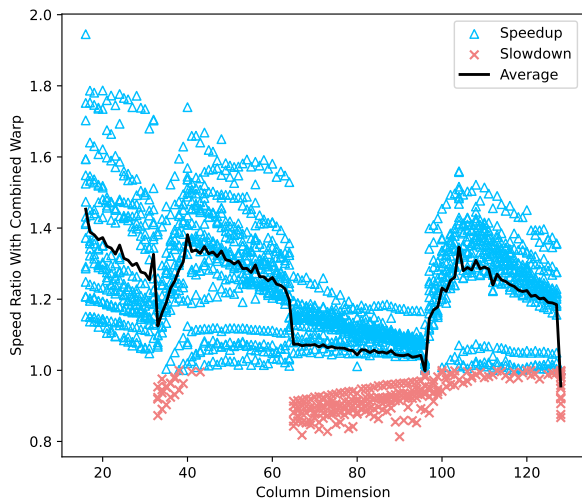


Fig. 8. Speedup of degree sorting & block-level partition (i). with combined-warp strategy over (ii). without combined-warp strategy.

the state-of-the-art predecessors, namely GNNAdvisor and graph-BLAST, in all instances. When considering the average computational performance of kernels across all column dimensions (ranging from 16 to 128) on a variety of graphs, our kernel manifests an average improvement of  $1.17\times$  over cuSPARSE, reaching a maximum increment of  $1.45\times$ . It surpasses a  $1.3\times$  improvement in 22% of the cases, underperforming compared to cuSPARSE in a single instance and approximately equalling cuSPARSE's performance in another. Our kernel significantly supersedes GNNAdvisor and graph-BLAST across all benchmark graphs, yielding an average speedup of  $1.86\times$  and  $2.94\times$ , and a maximum speedup of  $3.41\times$  and  $5.02\times$ , respectively.

Fig. 6 exhibits the runtime of all kernels on each graph for every column dimension of the right-hand matrix. Benefiting from memory coalescing and automatic alignment of

intermediate results proffered by the combined warp strategy, the runtime demonstrates a gradual increase as the column dimension escalates, exhibiting minimal effect when the column dimension deviates from a power of 2.

**Ablation Study 1: Block-level Partition vs. Warp-level Partition.** Figure 7 illustrates the comparative speed ratio achieved by block-level partition as opposed to warp-level partition. The block-level partition, leveraging dynamically varying NZ group sizes according to node degree, manifests superior shared memory reuse efficiency and enhanced locality relative to warp-level partition. As substantiated by Table II, block-level partition has realized an average speedup ranging from  $1.05\times$  to  $1.07\times$  across disparate column dimension intervals, culminating in a peak improvement of  $1.31\times$  and a least effective case of  $0.92\times$ . Importantly, the enhancement in performance facilitated by block-level partitioning is observed to remain consistent across varying column dimensions.

**Ablation Study 2: Combined Warp.** As depicted in Fig. 8, the speedup resulting from block-level partition (i) with and (ii) without combined warp strategies is given. The implementation of combined warp strategy leads to a marked performance improvement specifically within the column dimension intervals  $[0, 32]$ ,  $[32, 64]$ , and  $[96, 128]$ , with an average speed gain recorded between  $1.23\times$  and  $1.33\times$ . Conversely, this enhancement is somewhat diminished within the column dimension range  $[64, 96]$ , a divergence potentially ascribable to unaligned cache line size in the prevailing GPU architecture.

In summation, the ablation studies collectively attest to the vital contributions of both block-level partition and combined warp strategy in accelerating processing speed for most of the graphs. The nuanced differences between these strategies highlight the necessity of targeted optimization based on specific column dimension intervals and underscore the potential for further investigation and refinement in future work.

## V. CONCLUSION

In conclusion, this paper presents Accel-GCN, a GPU accelerator architecture addressing workload imbalance and memory access irregularity in GCNs. Incorporating a lightweight,  $\mathcal{O}(n)$  preprocessing stage with degree sorting and block-level partition, it optimizes memory utilization and workload distribution. The kernel design further leverages a combined warp strategy for dense column dimension processing, enhancing performance and memory efficiency. Accel-GCN further improves memory coalescing and alignment to achieve a better memory bandwidth utilization. Evaluated on 18 benchmark graphs, Accel-GCN outperforms cuSPARSE, GNNAdvisor, and graph-BLAST by  $1.17\times$ ,  $1.86\times$ , and  $2.94\times$  respectively, highlighting its potential in general GCN acceleration applications.

## VI. ACKNOWLEDGEMENT

This work was partially funded by the Semiconductor Research Corporation (SRC) Artificial Intelligence Hardware program, and the UIUC HACC program.



## REFERENCES

- [1] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.
- [2] Jielun Liu, Ghim Ping Ong, and Xiquan Chen. Graphsage-based traffic speed forecasting for segment network with sparse data. *IEEE Transactions on Intelligent Transportation Systems*, 2020.
- [3] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open graph benchmark: Datasets for machine learning on graphs. *Advances in neural information processing systems*, 33:22118–22133, 2020.
- [4] Yuxiao Liu, Ning Zhang, Dan Wu, Audun Botterud, Rui Yao, and Chongqing Kang. Guiding cascading failure search with interpretable graph convolutional network. *Computing Research Repository (CoRR) in arXiv*, abs/2001.11553, 2020.
- [5] Weiwei Jiang and Jiayun Luo. Graph neural network for traffic forecasting: A survey. *Expert Systems with Applications*, page 117921, 2022.
- [6] Shiwen Wu, Fei Sun, Wentao Zhang, Xu Xie, and Bin Cui. Graph neural networks in recommender systems: a survey. *ACM Computing Surveys (CSUR)*, 2020.
- [7] Pietro Bongini et al. Molecular generative graph neural networks for drug discovery. *Neurocomputing*, 450:242–252, 2021.
- [8] Tong Geng, Ang Li, Runbin Shi, Chunshu Wu, Tianqi Wang, Yanfei Li, Pouya Haghi, Antonino Tumeo, Shuai Che, Steve Reinhardt, et al. Awgcn: A graph convolutional network accelerator with runtime workload rebalancing. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 922–936. IEEE, 2020.
- [9] Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding. Gnnadvisor: An efficient runtime system for gnn acceleration on gpus. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'21)*, 2021.
- [10] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P Scarpazza. Dissecting the nvidia volta gpu architecture via microbenchmarking. *arXiv preprint arXiv:1804.06826*, 2018.
- [11] Carl Yang, Aydın Buluç, and John D Owens. Implementing push-pull efficiently in graphblas. In *Proceedings of the 47th International Conference on Parallel Processing*, pages 1–11, 2018.
- [12] M. Naumov, L. Chien, P. Vandermersch, and U. Kapasi. Cusp sparse library. *GPU Technology Conference (GTC)*, 2010.
- [13] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. *Advances in neural information processing systems*, 30, 2017.
- [14] Yifan Hou, Jian Zhang, James Cheng, Kaili Ma, Richard TB Ma, Hongzhi Chen, and Ming-Chang Yang. Measuring and improving the use of graph information in graph neural networks. In *International Conference on Learning Representations*, 2019.
- [15] Tong Geng, Chunshu Wu, Yongan Zhang, Cheng Tan, Chenhao Xie, Haoran You, Martin Herbordt, Yingyan Lin, and Ang Li. I-gcn: A graph convolutional network accelerator with runtime locality enhancement through islandization. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1051–1063, 2021.
- [16] Chao Yang, Aydın Buluç, and John D. Owens. Design principles for sparse matrix multiplication on the gpu. In *European Conference on Parallel Processing*, pages 672–687. Springer, 2018.
- [17] Guyue Huang, Guohao Dai, Yu Wang, and Huazhong Yang. Ge-spmv: General-purpose sparse matrix-matrix multiplication on gpus for graph neural networks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page Article 72, 2020.
- [18] Peng Jiang, Changwan Hong, and Gagan Agrawal. A novel data transformation and execution strategy for accelerating sparse matrix multiplication on gpus. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2020.
- [19] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015.
- [20] Xia Zhao, Almutaz Adileh, Zhibin Yu, Zhiying Wang, Aamer Jaleel, and Lieven Eeckhout. Adaptive memory-side last-level gpu caching. In *Proceedings of the 46th international symposium on computer architecture*, pages 411–423, 2019.
- [21] Gloria Ortega, Francisco Vázquez, Inmaculada García, and Ester M. Garzón. Fastspmv: An efficient library for sparse matrix matrix product on gpus. *Comput. J.*, 57(7):968–979, 2013.
- [22] Hartwig Anzt, Stanimire Tomov, and Jack Dongarra. Accelerating the lobpcg method on gpus using a blocked sparse matrix vector product. In *Proceedings of the Symposium on High Performance Computing (HPC '15)*, pages 75–82, San Diego, CA, USA, 2015. Society for Computer Simulation International.
- [23] Richard Vuduc, James W Demmel, and Katherine A Yelick. Oski: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series*, 16(1):521–530, 1 2005.
- [24] Hasan Metin Aktulga, Aydın Buluç, Samuel Williams, and Chao Yang. Optimizing sparse matrix-multiple vectors multiplication for nuclear configuration interaction calculations. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium (IPDPS'14)*, pages 1213–1222. IEEE Computer Society, 5 2014.
- [25] Changwan Hong, Aravind Sukumaran-Rajam, Bortik Bandyopadhyay, Jinsung Kim, Süreyya Emre Kurt, Israt Nisa, Shivani Sabhlok, Ümit V Çatalyürek, Srinivasan Parthasarathy, and P Sadayappan. Efficient sparse-matrix multi-vector product on gpus. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, pages 66–79. ACM, 2018.
- [26] Anurag Mukkara, Nathan Beckmann, Maleen Abeydeera, Xiaosong Ma, and Daniel Sanchez. Exploiting locality in graph analytics through hardware-accelerated traversal scheduling. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–14, 2018.
- [27] Yongsub Lim, U Kang, and Christos Faloutsos. Slashburn: Graph compression and mining beyond caveman communities. *IEEE Transactions on Knowledge and Data Engineering*, 26(12):3077–3089, 2014.
- [28] Junya Arai, Hiroaki Shiokawa, Takeshi Yamamuro, Makoto Onizuka, and Sotetsu Iwamura. Rabbit order: Just-in-time parallel reordering for fast graph analysis. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 22–31. IEEE, 2016.
- [29] Nvidia cuda toolkit. <https://docs.nvidia.com/cuda/index.html>. Accessed: 2023-05-05.
- [30] David B Kirk and W H Wen-mei. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan kaufmann, 2016.
- [31] Mohsin Shan, Deniz Gurevin, Jared Nye, Caiwen Ding, and Omer Khan. Mergepath-spmv: Parallel sparse matrix-matrix algorithm for graph neural network acceleration. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 145–156. IEEE, 2023.
- [32] Weidong Sun and Zongmin Ma. Count sort for gpu computing. In *2009 15th International Conference on Parallel and Distributed Systems*, pages 919–924. IEEE, 2009.
- [33] Duane Merrill and Andrew Grimshaw. High performance and scalable radix sorting: A case study of implementing dynamic parallelism for gpu computing. *Parallel Processing Letters*, 21(02):245–272, 2011.
- [34] Nvidia nsight compute. <https://developer.nvidia.com/nsight-compute>. Accessed: 2023-08-20.
- [35] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations (ICLR)*, 2017.
- [36] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *International Conference on Learning Representations (ICLR)*, 2019.
- [37] Kristian Kersting, Nils M. Kriege, Christopher Morris, Petra Mutzel, and Marion Neumann. Benchmark data sets for graph kernels. 2016.
- [38] Matthias Fey and Jan Eric Lenssen. Fast graph representation learning with pytorch geometric. *arXiv preprint arXiv:1903.02428*, 2019.
- [39] Jure Leskovec and Andrej Krevl. Snap datasets: Stanford large network dataset collection. <http://snap.stanford.edu/data>, 2014.