IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 29 May 2026, accepted 12 June 2026, date of publication 22 June 2026, date of current version 30 June 2026.

Digital Object Identifier 10.1109/ACCESS.2026.3705782

RESEARCH ARTICLE

BiKD: Bidirectional Kernel Decomposition for Large-Scale GCNs on GPU

INJE KIM[1,2], JIHUN LEE[2], JONG HYUN JEONG[2], (Graduate Student Member, IEEE), GEONWOO CHOI[2], MYUNG KUK YOON[3], (Member, IEEE), YUNHO OH[4], (Senior Member, IEEE), AND GUNJAE KOO[2], (Member, IEEE)

[1]Korea Electronics Technology Institute, Seongnam-si, Gyeonggi-do 13509, South Korea
[2]Department of Computer Science and Engineering, Korea University, Seoul 02841, South Korea
[3]Division of Artificial Intelligence and Software, Ewha Womans University, Seoul 03860, South Korea
[4]School of Electrical Engineering, Korea University, Seoul 02841, South Korea

Corresponding author: Gunjae Koo (gunjaekoo@korea.ac.kr)

This work was supported in part by the National Research Foundation of Korea (NRF) funded by Korea government (MSIT) (NRF-2021R1C1C1012172, RS-2025-02214322, BK21 FOUR), and in part by the Institute of Information and Communications Technology Planning and Evaluation (IITP) grant funded by the Korea government (MSIT) (IITP-2026-RS-2020-II201819 ICT Creative Consilience Program).

ABSTRACT Graph convolutional neural networks (GCNs) are representative graph neural network (GNN) models that can be used for analyzing and classifying nodes in graph structures. Since graph structures exhibit extremely sparse and irregular connections among many vertices, graph processing is not efficient on GPUs as GPUs are designed for accelerating regularly structured datasets. Since GCN kernels handle large feature data associated with vertices, GCN kernels exhibit extremely low efficiency on GPU. We analyze the behavior of the GCN aggregation kernel on GPU to reveal the performance hurdles of GCN kernels. In this paper, we propose an efficient GCN kernel design approach, called BiKD. We first propose a CTA-level vertex mapping approach to tackle the lower resource utilization in GPU. We reveal that the GPU's kernel execution model is not efficient for handling unbalanced graph structures. By mapping vertices to multiple CTAs in GCN aggregation kernels, GPUs can instantly assign available resources to new CTAs without synchronization among multiple vertices. Then we propose kernel decomposition approaches to reduce the data structure size handled by a single GCN aggregation kernel. We observe GPU's L2 cache cannot exploit inter-vertex locality since the size of feature data associated with vertices is significantly large compared to the L2 cache space. Our evaluation shows that the proposed kernel design improves GCN aggregation performance by 1.31× over GE-SpMM, 1.39× over DGL with METIS reordering (DGL-METIS), 2.11× over GNNAdvisor, and 2.22× over the baseline DGL implementation. We also observe that the proposed kernel design can improve the utilization of the GPU's cache hierarchy dramatically.

INDEX TERMS Kernel design, kernel optimization, GPU, graph neural networks, preprocessing.

I. INTRODUCTION

Graph neural networks (GNNs) are a class of neural network applications widely used for analyzing and classifying graph-structured data. Unlike popular neural network architectures such as convolutional neural networks (CNNs) that operate on regularly structured data, GNNs handle graph data structures that include many vertices connected in irregular patterns. Graph structures are widely used for representing relations among data nodes, which are frequently observed in scientific and social data sets [1], [2], [3], [4]. In order to boost the performance of neural network applications, we frequently rely on GPU's massive thread-level parallelism (TLP) capability since GPUs are designed for executing many simple threads in parallel. However, GNN applications exhibit extremely low efficiency on GPUs due to the inherent characteristics of graph structures. Namely, the adjacency matrix used for representing edges between two vertices

The associate editor coordinating the review of this manuscript and approving it for publication was Wei Wang.

© 2026 The Authors. This work is licensed under a Creative Commons Attribution 4.0 License.
For more information, see https://creativecommons.org/licenses/by/4.0/

is extremely sparse thus such sparse matrix operations incur very low efficiency in storage space and processing units. Even though graph structures can be formatted with compressed data structures that can eliminate zero data, essential graph analytics operations such as traversing neighbor vertices create many irregular memory transactions in the GPU memory hierarchy. Note that many studies report GPU performance can significantly suffer from irregular memory transactions created from graph analytics kernels [5], [6], [7]. Furthermore, GNN kernels typically handle large-scale graph structures where each vertex includes a high-dimensional feature vector, thus the GPU's cache hierarchy is poorly utilized since the feature vectors once allocated in a cache are frequently evicted before re-reference. In the previous articles, researchers presented severe performance hurdles when GNN applications run on GPUs [8], [9].

A single layer of GNN applications typically consists of two phases: *aggregation* and *combination*. GNN applications execute multiple such layers repeatedly to produce final outputs. Prior research discloses that the aggregation phase accounts for a significant portion of overall execution time when running GNNs on GPUs since aggregating the features of a single vertex requires accessing feature vectors from multiple neighbor vertices [9]. These operations are heavy performance burdens in GPUs since the multiple feature data demanded by a single vertex are irregularly allocated in the GPU memory space. On the other hand, the combination phase involves dense matrix-matrix multiplications that fit well with GPU architectures. Consequently, improving the efficiency of aggregation kernels is critical for enhancing the performance of GNN applications on GPUs.

Traditional graph processing kernels associate a single thread of a GPU with the computations of a single vertex (or edge) in order to maximize GPU's thread-level parallelism [5], [10]. However, this approach is significantly inefficient for GNN kernels since each vertex demands multiple large feature vectors to create severe congestion in the GPU memory hierarchy. Furthermore, the thread-level mapping approach can result in low GPU occupancy since the execution of the threads within a warp is explicitly synchronized. Considering the number of neighbor nodes varies across vertices, a warp can be occupied for many cycles by the slowest thread typically corresponding to a vertex with a large number of neighbor nodes. In order to address such inefficiencies, researchers proposed a warp-level mapping approach in which the aggregation operation for a single vertex is assigned to a warp [11]. This method can reduce the amount of data demanded by a single warp since each warp handles only one vertex. However, recent studies reveal GPU's caches are not utilized efficiently even with the warp-level mapping approach [8]. GPU's cache hierarchy cannot exploit the data locality observed in graph structures because the size of GPU caches is small compared to the size of feature vectors in large-scale graphs. Hence, aggregation kernels frequently fetch large amounts of data from GPU's external memory, thus the overall performance of GPU becomes limited by the bandwidth of memory channels. In addition, the low GPU occupancy issue still persists with the warp-level mapping approach since a cooperative thread array (CTA) remains occupied by the slowest warp associated with the vertex that includes many neighbors.

In order to tackle the aforementioned performance issues of the aggregation kernels, we propose an efficient aggregation kernel design approach, called *Bi*directional *K*ernel *D*ecomposition (*BiKD*). BiKD employs the following key features to enable efficient GCN aggregation. First, we propose a CTA-level vertex mapping strategy to mitigate the low occupancy issues observed in the existing aggregation kernels. By assigning vertices to CTAs in the aggregation kernels of BiKD, GPUs can instantly allocate available resources to new CTAs without synchronization across multiple vertices. Second, we propose a kernel decomposition approach to reduce the data structure size handled by a single aggregation kernel. We observe GPU's L2 cache cannot exploit inter-vertex locality since the size of the feature data associated with vertices exceeds the capacity of the L2 cache. To mitigate this limitation, prior work [12], [13], [14] has explored graph partitioning techniques that aim to cluster highly connected vertices and improve data locality in GPU caches. However, such preprocessing-based approaches introduce substantial preprocessing overhead, since graph partitioning is typically performed offline on the CPU before GPU execution. In order to address this issue, BiKD employs a lightweight, cache-aware preprocessing step that partitions the graph and feature matrices based on the GPU's L2 cache size, without relying on complex graph algorithms. By decomposing the aggregation kernels along both feature and graph dimensions, BiKD reduces unnecessary cache line evictions in the GPU's L2 cache. In this paper, we select graph convolutional networks (GCNs) as the primary target GNN application for BiKD since GCNs are among the most popular GNN models. Note that the aggregation operations of GCNs can be regarded as convolutional operations applied to graph-structured data [15]. We mainly describe BiKD's aggregation kernel design using GCN models running on GPUs. However, our proposed approach can also be applied to other GNN models that incorporate similar aggregation operations.

We evaluate the performance of the proposed BiKD kernel design approach using the real GPU platform and profiler. Our evaluation results reveal that BiKD improves the performance of GCN aggregation kernels by 1.31× over GE-SpMM [16], a warp-level mapping–based SpMM kernel optimization, and by 1.39× over the METIS-reordered version of DGL (DGL-METIS), as well as by up to 2.11× over preprocessing-based graph partitioning approaches such as GNNAdvisor. Against the baseline DGL implementation [17], BiKD achieves an average 2.22× speedup across various large-scale graph datasets. We also observe that the proposed kernel design significantly enhances the utilization of the GPU cache hierarchy. Our evaluation also exhibits that

BiKD is an effective kernel design approach to other GNN models such as GIN.

Our contributions are as follows:

- We conduct an in-depth analysis of the performance factors of existing GCN aggregation kernels, taking into account the detailed GPU hardware architecture.
- Based on this analysis, we propose two optimization approaches for GCN aggregation kernels: CTA-level vertex mapping and kernel decomposition.
- We propose BiKD, a bidirectional kernel decomposition approach that applies lightweight, cache-aware graph and feature partitioning based on the GPU's L2 cache capacity, exploiting inter-vertex locality while significantly reducing preprocessing overhead.
- We evaluate the performance of BiKD on large-scale graph datasets, demonstrating that BiKD outperforms both warp-level kernel optimizations and preprocessing-based graph partitioning frameworks, while substantially improving GPU cache utilization.

The remainder of this paper is organized as follows. Section II describes conventional GCN kernel models on GPU and GPU architectures. We propose a simple kernel optimization based on a CTA-level vertex mapping in Section III. We propose kernel partitioning approaches in Section IV. We exhibit the evaluation results in Section V. Section VI includes related work. We conclude in Section VII.

II. BACKGROUND

In this section, we briefly explain the operation of the basic GNN algorithm, followed by a description of the characteristics of GCN models. Then we discuss the architectural features and limitations of GPUs which are commonly deployed to run large-scale GCN kernels. We also explain how GPU's massive thread-level parallelism can be leveraged to run the sparse matrix kernels involved in GCN models.

A. GRAPH NEURAL NETWORKS

GNNs apply neural network models to graph structures where vertices are irregularly connected. Unlike conventional neural network models such as CNNs that extract features from regularly structured data using convolution filters, GNNs perform feature extraction by aggregating feature data embedded in neighboring vertices (i.e. *aggregation*) and updating each vertex's features by combining the aggregated features and weight parameters (i.e. *combination* or *message passing*) [18], [19]. Figure 1 depicts the operation of a GNN. The operation of a single-level GNN consists of *aggregation* and *combination* steps followed by an activation function. A GNN repeats these layer operations.

$$\begin{aligned} a_v^{l+1} &= \textbf{Aggregate}(\{h_u^l : u \in N(v)\} \cup h_v^l) \\ h_v^{l+1} &= \textbf{Combine}(a_v^{l+1}) \end{aligned} \tag{1}$$

Equation 1 presents a general formulation of a single GNN layer. In layer l, the aggregation kernel updates the features of each vertex v by collecting the feature data from its neighbor vertices (i.e. $N(v)$). Then, the combination kernel performs the *Combine* function to generate the updated feature data for each vertex. Variants of GNNs adopt different types of *Aggregate* functions. For instance, graph isomorphism networks (GINs) use summation as an aggregate function and GraphSAGE deploys mean or max pooling.

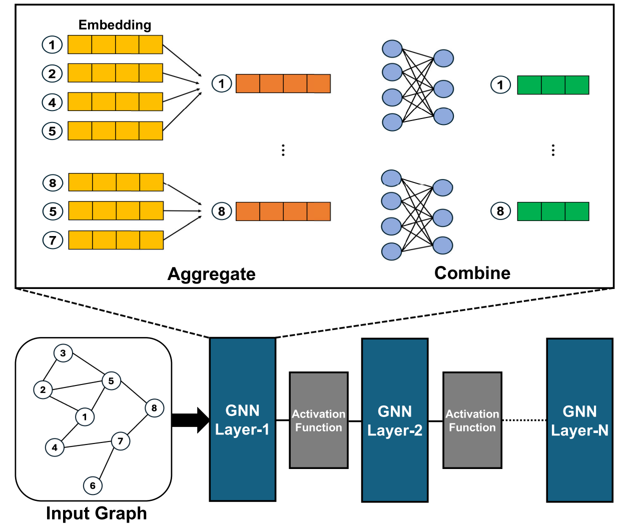


FIGURE 1. GNN operation.

Among GNN variants, GCNs are one of the most popular and foundational GNN models [15]. GCNs employ a weighted sum as an aggregation function and typically apply a normalization function to features collected from each target vertex and its neighboring vertices. Hence, the operation of a single GCN layer can be represented as a series of matrix multiplications as shown in Equation 2 [20]

$$X^{(l+1)} = \sigma(\tilde{A}X^{(l)}W^{(l)}) \tag{2}$$

In the equation, $\tilde{A}$ represents a normalized adjacency matrix of a graph structure. A feature matrix of layer l is represented as $X^{(l)}$, and $W^{(l)}$ is a weight matrix of layer l. A non-linear activation function (σ in the equation) such as rectifier linear unit (ReLU) is applied to the output features of layer l. Note that a single GCN layer performs two consecutive matrix multiplications to generate updated features. The first stage of the matrix multiplications uses the adjacency matrix of a graph structure, thus it represents the *aggregation operation*. The second stage, called *combination*, performs a general matrix multiplication (GeMM) between the weight parameters and the features associated with vertices.

Although GCN kernels rely on the massive parallel computations offered by GPUs, the GCN aggregation kernels exhibit extremely low efficiency. Note that GPU's single-instruction multiple-thread (SIMT) architecture is efficient for computing regularly organized data structures. However, the adjacency matrix of a graph structure used in the

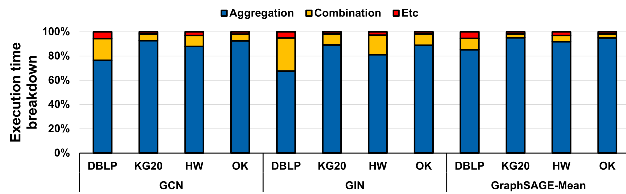


FIGURE 2. Execution time breakdown of a single GNN layer on GPU.

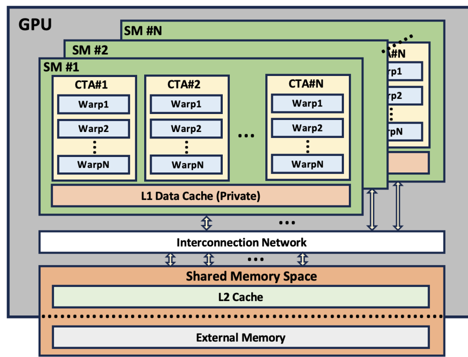


FIGURE 3. GPU memory hierarchy.

aggregation operations is extremely sparse, thus the aggregation kernels create many irregular memory transactions. GPU's memory hierarchy resources are inefficiently utilized for handling such heavy irregular memory traffic, thus the aggregation step poses the primary performance bottleneck in GCNs.

To identify performance bottlenecks shared across SpMM-like GNN variants, we profile the execution time breakdown of a single GNN layer using the DGL framework for representative models, including GCN, GIN, and GraphSAGE with a mean aggregator. [17] We conduct this measurement using representative large-scale graph datasets and feature dimensions listed in Table 1 on an NVIDIA RTX A6000 GPU (see Table 2), and report the per-layer execution time breakdown into aggregation and combination for each model. Our results show that, despite architectural differences among these GNNs, aggregation consistently dominates the per-layer execution time compared to the GeMM-based combination, as shown in Figure 2.

B. GPU ARCHITECTURE AND EXECUTION MODEL

GPUs are widely deployed to run neural network applications since GPUs exploit massive thread-level parallelism (TLP) to boost the performance of parallel computations. GPU architecture is designed for running many concurrent threads generated from a single computation kernel. In order to support massive TLP, GPU equips thousands of compute cores and memory subsystems that can provide high bandwidth. In this paper, we describe GPU architecture using NVIDIA's terminology [21], [22], [23], [24]. A single GPU includes tens of streaming multiprocessors (SMs). Each SM includes multiple CUDA cores (or called streaming processors) to execute tens of threads concurrently. Modern GPUs include specific computation units such as Tensor cores to accelerate matrix operations frequently observed in neural network applications [25], [26]. An SM executes multiple threads in a SIMD-like manner by grouping dozens of threads into a *warp*. Namely, the threads within a warp share the same program counter thus a warp executes the same instructions using different data simultaneously. As shown in Figure 3, an SM includes a small private data cache (typically 64–128 KB per SM) shared by tens of concurrent warps. A GPU also includes a larger L2 cache shared by all SMs to exploit data locality. An SM can prevent pipeline stalls caused by several cycles of instructions by using a quick context switch among concurrent warps. However, researchers have revealed the long latency of demand fetch from external memory cannot be hidden by GPU's fine-grained multithreading execution among warps [27], [28]. Hence, the performance of GPU is significantly influenced by the utilization of L1 and L2 caches [7], [29], [30], [31].

Now we describe GPU's kernel execution model. When a kernel is launched on a GPU, the threads created from the kernel are grouped into multiple cooperative thread arrays (CTAs). Note that the dimension of CTAs can be adjusted by setting *dim_grid* of a CUDA kernel. CTAs are assigned to SMs that have available hardware resources when a kernel is launched, then the threads in a CTA are grouped into multiple warps in an SM. Note that the threads within a warp share the same program counter thus the execution of threads in a warp is explicitly synchronized. Warp execution is terminated when all threads within a warp complete kernel instructions. Since the threads in a warp can go through diverged execution paths, the slowest thread determines the complete time of the warp. Likewise, the execution of an active CTA is terminated when all warps created from the CTA are complete, thus CTA's completion time is determined by the slowest warp. Note that an active CTA holds hardware resources in an SM until the CTA is deallocated from the SM. Hence, GPU's resources such as warps and register files can be wasted if warps in a CTA exhibit diverged execution time.

C. SpMM ON GPU

As described in Section II-A, GCN kernels include the aggregation phase that collects features from neighbor vertices. To demand the feature vectors associated with the neighbor vertices, the aggregation kernel uses the adjacency matrix that represents the connections among vertices. This operation can be represented as a sparse matrix-matrix multiplication (SpMM) between the adjacency and feature matrices.

Algorithm 1 represents the algorithmic flow of the SpMM kernel that employs warp-level vertex mapping [11]. In this kernel model, each vertex is assigned to a single warp and each thread in the warp demands a feature element from neighbor vertices to aggregate the feature values. Since the threads in a warp can access multiple feature elements concurrently, the warp-level mapping approach is more efficient

Algorithm 1 Parallelized SpMM Kernel [11]

1: *adj* $\leftarrow$ adjacency matrix
2: *cta* $\leftarrow$ *adj* split by rows
3: *warp* $\leftarrow$ a row in the split *adj*
4: **for** each *warp* **in parallel do**
5: $N(v) \leftarrow$ number of neighbor vertices
6: **for** $n \leftarrow N(v)$ **do**
7: **for** $f \leftarrow 0$ to f_len **do**
8: $tmp += \text{AGG}(g_val[n], f_val[(n \times f_len) + f])$
9: **end for**
10: **end for**
11: $out[\text{v}] += tmp$
12: **end for**

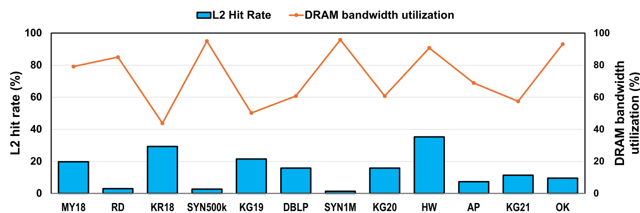


FIGURE 4. L2 hit rate and DRAM bandwidth utilization of SpMM kernels.

for GNN models compared to the thread-level mapping typically employed by graph processing kernels [16], [32], [33]. In Algorithm 1, the adjacent matrix of a graph structure is split by rows into multiple partitions. Each partition of the adjacent matrix is assigned to a CTA of the aggregation kernel. Then, each row in a partition (associated with a CTA) is assigned to a warp. Note that an SM executes multiple warps concurrently by issuing available warps from all concurrent warps. A single warp includes dozens of threads (32 threads for NVIDIA GPUs), thus each thread execution is associated with the operations mapped to each feature element (see line 7). An aggregation function (e.g. arithmetic mean, normalization, and non-linear function) is applied to the feature element demanded by each thread. Then the result of the aggregation function is accumulated in the temporary register (*tmp* in Algorithm 1). Note that the feature read operation associated with each thread can be well-coalesced since feature vectors are usually represented in a dense data format thus the number of memory requests can be reduced by employing the warp-level mapping. This aggregation operation is repeated for the vertices in the neighbor vertex list (see line 6). Since vertices are irregularly connected in a typical graph structure, the feature data of the neighbor vertices can be usually fetched from the shared memory hierarchy levels (i.e. L2 cache or external memory) rather than private caches in each SM. For a large-scale graph structure, the entire feature data cannot be accommodated in the GPU's small L2 cache.

In order to investigate the performance bottleneck in GPU's L2 cache for this SpMM kernel, we measure L2 cache hit rates (represented as bars) and DRAM channel bandwidth utilization (indicated by red-line markers) as shown in Figure 4. We use the graph datasets listed in Table 1 in Section V. Our analysis reveals that the SpMM kernels on GPUs exhibit extremely low L2 cache hit rates, typically lower than 20%, for most graph datasets. This indicates that GPU's shared L2 cache is not utilized efficiently in capturing the inter-vertex locality in graph structures. Furthermore, for some datasets DRAM bandwidth utilization reaches nearly 100%, which means the SpMM kernel creates frequent off-chip memory transactions. Note that fetching data from external DRAM takes significantly more cycles as off-chip memory traffic increases. Overall, our analysis demonstrates that the warp-level SpMM kernels suffer from substantial performance degradation due to poor utilization of the GPU's shared L2 cache. In this work, we use this warp-level mapping approach, say *RowSplit*, as the baseline aggregation kernel design.

D. GRAPH PARTITIONING

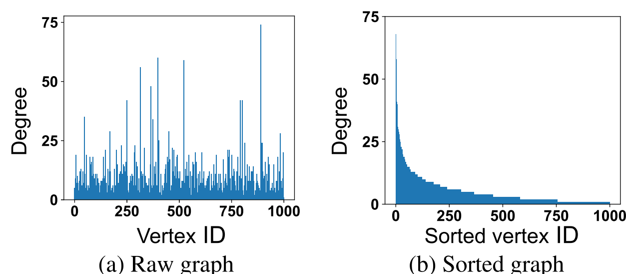
Graph partitioning is a technique for improving data locality in GPU-based graph workloads. It divides a large graph into multiple subgraphs while preserving the original adjacency structure, reordering vertex indices so that densely connected vertices are placed close together. By grouping strongly connected vertices, the feature data required during the aggregation phase are located in nearby memory regions, allowing GPU kernels to exploit spatial locality at the L2-cache level. Several prior studies have proposed graph partitioning algorithms based on this idea. Kumar et al. [12] proposed a multilevel partitioning algorithm, called MiniMax, for irregular graphs that considers grid-computing constraints. LaSalle et al. [13] presented a multi-level graph partitioning algorithm (METIS) that minimizes the number of cross-partition edges to reduce data access across partitions. Rabbit Order [14] was proposed as a parallel graph partitioning method that assigns vertices with strong connectivity into the same partition to improve locality. However, although these graph partitioning approaches improve structural locality, they do not explicitly account for the capacity of the GPU L2 cache. In practice, the feature working set accessed during aggregation often exceeds the L2 cache capacity due to high-dimensional feature vectors, leading to frequent cache evictions and residual accesses to external memory. Moreover, for large-scale graphs, graph partitioning methods based on complex algorithms incur substantial preprocessing overhead, as they are typically executed offline on the CPU. While the aggregation phase of a GCN layer on the GPU completes within milliseconds, such preprocessing steps often require seconds to process large graphs, resulting in a significant imbalance between preprocessing cost and kernel execution time. To address these limitations, we aim to improve aggregation kernel performance without relying on costly preprocessing-based partitioning. Instead, we exploit the GPU memory hierarchy by applying lightweight graph-level and feature-level partitioning, where the partition size is determined based on the GPU's L2 cache capacity. This approach enables

locality-aware aggregation while significantly reducing preprocessing overhead.

III. VERTEX MAPPING

As described in the previous section, the baseline aggregation phase can be implemented using the SpMM kernel that maps each vertex operation to warp-level executions. In this section, we analyze the performance hurdles of the baseline SpMM operations to discuss further optimization approaches for GCN aggregation kernels. First, we propose a CTA-level vertex mapping approach to tackle the low warp occupancy issue of the aggregation kernel.

For the SpMM operations of the aggregation phases, vertices in a graph structure can be mapped into different levels of the GPU's execution model. Namely, the operations of a single vertex can be assigned to *thread-level*, *warp-level*, and *CTA-level* executions. The vertex-centric graph processing kernels frequently employ thread-level mapping approaches that assign each vertex processing to a single thread execution [5], [34]. Even though this approach can process many vertices in parallel by accommodating as many vertices as the maximum concurrent thread count of a GPU, such kernels exhibit significant performance drops since irregular accesses from vertices cause many uncoalesced memory transactions from a warp [6], [35]. For GCN aggregations, the performance drop is more significant since each vertex accesses large feature vectors associated with neighbor vertices.



(a) Raw graph (b) Sorted graph

FIGURE 5. Degree distribution of graphs.

In order to tackle such issues, researchers have proposed warp-level approaches as described in Algorithm 1. In a warp-level kernel, each vertex processing is mapped to a warp and threads in the warp handle the processing of feature elements. The warp-level mapping approaches can utilize the GPU memory subsystem more efficiently since the threads within a warp create well-coalesced memory requests that access feature data allocated regularly. However, the warp-level mapping may not utilize GPU resources efficiently since the imbalanced execution times among warps within a CTA provoke low warp occupancy. Note that the execution time of a CTA is decided by the slowest warp in the CTA as explained in Section II-B. For GCN aggregations, the warp execution times within a CTA are unbalanced since the degrees of vertices are extremely irregular. Figure 5a exhibits that the degrees of individual vertices are distributed irregularly by vertex IDs. Furthermore, the difference in degrees between densely and sparsely connected vertices is extremely large as shown in Figure 5b, which exhibits the power-law distribution of the sorted degrees. We evaluate the warp occupancy (i.e. the number of active warps / the number of maximum concurrent warps supported by a GPU) of the warp-level aggregation kernel presented in Algorithm 1 using the testbed and the workloads listed in Tables 2 and 1 respectively. The test results reveal the average warp occupancy of the warp-level kernel is only 34.8%.

CTA-level mapping approaches can tackle such underutilization issues of the warp-level kernels since CTA executions are synchronized only within a kernel execution. As described in Section II-B, CTAs created from a kernel are assigned to available SMs. Then, when CTA's execution is complete, SM's resources occupied by the CTA are released and a new CTA can be allocated instantly. Hence, if there are enough pending CTAs created from a kernel, GPU's resources can be fully assigned until the number of running CTAs reaches the maximum concurrent CTA count. Notably, the CTA-level mapping approach can mitigate the low warp occupancy issues caused by the disparity in the number of neighbor vertices associated with each vertex. The threads within a CTA are allocated to the feature elements of a vertex in the CTA-level mapping kernel, thus the thread execution times are well-balanced as the feature elements are regularly organized. The warp execution times within a CTA are obviously balanced, thus the long-tail of warp execution times can be alleviated. The CTA-level mapping approach may sacrifice thread-level parallelism (TLP) since a CTA is a coarse-grained block that includes multiple threads (and multiple warps). Namely, fewer vertices can be processed concurrently since each vertex processing task is tied to a CTA. However, modern GPU architectures support a higher number of concurrent CTAs per SM. For instance, NVIDIA GPUs since Maxwell can schedule concurrent CTAs up to half of the maximum concurrent warp count per SM [21], [22], [23], [24]. Considering that the warp occupancy of the warp-level kernels is below 50%, the CTA-level mapping can improve the performance of GCN aggregation kernels by mitigating warp-level load imbalance and achieving higher overall GPU resource utilization.

IV. KERNEL DECOMPOSITION

We disclose the utilization of GPU resources can be improved by employing the CTA-level mapping approach in the previous section. However, higher TLP achieved by allocating more active warps may not always guarantee performance improvement of GPU due to congestion in the GPU memory hierarchy [30], [31], [36]. Thus, to fully benefit from the improved warp occupancy, it is crucial to utilize the GPU memory subsystem more efficiently.

In this section, we propose kernel decomposition approaches to exploit inter-vertex locality more efficiently in GPU caches. Note that GPU's cache hierarchy is not utilized efficiently for the large-scale GCN kernels since the total

size of feature data is extremely large. Even though we can observe strong data locality for high-degree vertices, this inter-vertex locality is not utilized well in GPU caches since many cache lines are evicted before re-reference due to severe cache pollution [8], [9]. Hence, GCN kernels inevitably fetch large feature data from external memory, thus the performance of GCN kernels is restricted by the high latency and low bandwidth of GPU's external memory channels. In order to tackle such performance hurdles, we decompose the aggregation kernels to make each partition handle a smaller feature data chunk more efficiently using the GPU's shared L2 cache. Note that CTAs assigned to multiple SMs can share data in L2 partitions via interconnection network as explained in Section II-B. If the L2 cache can accommodate the feature data associated with high-degree vertices for longer cycles, the feature data can be quickly provided to multiple SMs. We investigate three kernel partitioning approaches: feature partitioning, graph partitioning, and bidirectional kernel decomposition.

A. FEATURE PARTITIONING

Feature partitioning (FP) decomposes an aggregation kernel into multiple smaller kernels that handle a part of feature vectors associated with vertices. As described in Section III, the concurrent threads in a CTA access feature elements individually based on our CTA-level mapping approach. A partitioned FP kernel can handle a part of feature elements since aggregations are performed element-by-element in feature vectors. Assuming a feature matrix where each row is associated with a vertex, an FP kernel processes the feature elements vertically partitioned in the feature matrix. Since the dataset size is reduced with the feature partitioning, the feature elements of high-degree vertices may stay longer in the L2 cache.

Algorithm 2 Host-Side Code for Feature Partitioning

1: $feat_div \leftarrow (feat_len - 1)/cta_size + 1$
2: **for** $i \leftarrow 0$ to $feat_div$ **do**
3: $\quad cuda_fp <<< n_vertex,\ cta_size >>> (i)$
4: **end for**

Algorithm 2 represents how a host code launches the FP kernels. Each FP kernel is associated with the computations of the corresponding partitioned feature data. The feature matrix is split into *feat_div* partitions, then the host code launches the FP kernels iteratively. The feature matrix columns handled by each kernel are computed using the partition index *i*. *cta_size* represents the number of threads per CTA. It should be a multiple of the warp size (32 for NVIDIA GPUs) to guarantee efficient kernel executions.

Even though we can reduce the size of the feature data accessed by individual FP kernels, GPU's L2 cache may not be large enough to exploit the inter-vertex locality of the partitioned features. Note that each CTA of a single FP kernel needs to include a sufficient number of threads to exploit TLP. Thus the number of elements in a partitioned feature vector is restricted by CTA size (i.e. the number of threads in a CTA) while the row size of the partitioned features is equivalent to the number of vertices. Hence, for large-scale graph structures, the size of the partitioned features can exceed the size of the L2 cache.

B. GRAPH PARTITIONING

Graph partitioning (GP) splits a feature matrix by vertex IDs to reduce feature size. Namely an individual GP kernel handles the horizontally partitioned feature matrix associated with the corresponding vertices. Note that an aggregation kernel demands feature vectors of a certain row (i.e. vertex ID) when the kernel collects features from the target vertex and the neighbor vertices. Hence, if the feature vector of a certain vertex is repeatedly demanded within a short window of cycles, the feature elements can be found in the GPU's cache hierarchy. In order to exploit such features, we design a GP kernel where vertices update output features only using a part of feature matrix rows (i.e. feature vectors of the corresponding partitioned graph). For implementing GP kernels, a feature matrix is partitioned horizontally based on selected vertex IDs and adjacency matrices of partitioned subgraphs are reproduced from the original adjacency matrix, usually represented using a CSR (compressed sparse row) format.

Figure 6 depicts how adjacency matrices of subgraphs are generated from an original graph structure when the graph partitioning is applied. In the figure, the original graph including eight vertices is split into four subgraphs that include two vertices per partition. Note that GP kernels require the edge (i.e. connection) information to the destination vertices associated with the partitioned features. While reading the original adjacency matrix formatted with CSR, the adjacency matrices of the target subgraphs are generated. For instance, *ptr*, *vid*, and *idx* of subgraphs 0, 2, and 3 are updated while the destination vertices (i.e, 2, 5, and 3) of *vertex 1* are read. This process is performed simply by adding a new data pair to the CSR-formatted data structure of the target subgraphs.

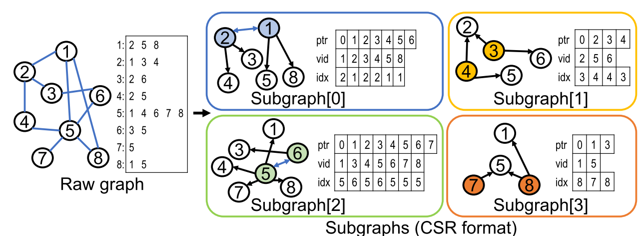


FIGURE 6. Graph partitioning.

The graph partitioning approach can increase the size of the adjacency matrix of the target graph structure since the edge data is changed when subgraphs are generated. For the original graph structure, the size of the adjacency matrix compressed with a CSR format is $O(V + 2E)$, where V and E represent the total numbers of vertices and edges

respectively. Note that an index and a value are associated with each edge in a CSR format. When the graph partitioning is applied, the total size of the adjacency matrices of the generated subgraphs becomes $O((V' \times grp_div)+2E)$, where V' represents the average number of vertices stored in subgraphs and grp_div is the number of subgraphs.

C. BIDIRECTIONAL KERNEL DECOMPOSITION

Since the aggregation kernels demand a large feature matrix data associated with many vertices, it is crucial to accommodate feature vectors of high-degree vertices in GPU's cache hierarchy. Even though FP and GP kernels can split the large feature data into smaller partitions, both partitioning approaches still exhibit the following limitations. For implementing efficient FP kernels, we need to set the *cta_size* of the FP kernels as the multiple of *warp_size* considering the GPU's warp-based execution model. Hence, we cannot make the feature size of the FP kernels smaller than $V \times warp_size$, where V is the number of total vertices in a graph structure and *warp_size* is 32 for NVIDIA GPUs. Note that V is extremely large for large-scale graph structures thus we cannot make small feature partitions using FP. The GP kernels can also exhibit significant performance drops for large-scale graph structures since each partition can include the feature data of a small number of vertices if the feature size of a single vertex is large. Note that we can exploit inter-vertex locality more efficiently if each subgraph partition includes more vertices and output feature vectors are less demanded. For the GP kernels, the performance overhead of updating output features increases if a single subgraph partition includes the features of fewer vertices.

In order to tackle such limitations of FP and GP kernels, we propose bidirectional kernel decomposition, called BiKD. BiKD can reduce the feature size of a single kernel further by partitioning feature data both vertically and horizontally. By applying the BiKD approach for the large-scale aggregation kernels, we can reduce the feature size to fit in GPU's L2 cache, thereby exploiting inter-vertex locality more efficiently while minimizing the performance overhead of output feature updates. Furthermore, BiKD's kernel decomposition approaches can be applied to reduce feature data size once loaded to GPU's device memory. Namely, BiKD can schedule the partitioned features to be loaded to GPU's device memory before the corresponding kernel executions, thus the large-scale graph datasets that cannot fully fit into the device memory can be operated using BiKD's kernel decomposition approach. Now, we describe how to implement the BiKD kernels.

Figure 7 depicts how the BiKD kernels are configured. To simplify the explanation, we assume the GPU L2 cache is extremely small thus the L2 cache can accommodate only four feature elements. We configure that a single subgraph of the BiKD kernel includes two vertices. In order to make feature data fit in the L2 cache, the *cta_size* of the vertical partitioning (i.e. feature partitioning) is set as two. Then, when the first BiKD kernel (the top-left kernel) is launched

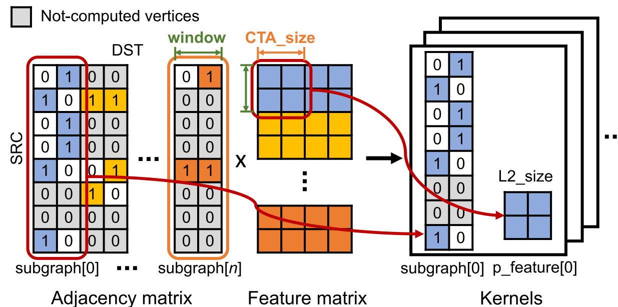


FIGURE 7. Processing an indirect graph with BiKD.

in the GPU, the BiKD kernel will demand the two feature elements of the two vertices, which can be accommodated in the L2 cache. The first BiKD kernel will update four output feature vectors of the vertices connected to the vertices 1 and 2 in the subgraph. Note that the performance overhead for updating output features can be reduced with BiKD since only partial feature vectors are demanded. Since the BiKD kernels can handle smaller feature data that can be accommodated in the L2 cache, BiKD can significantly reduce the data fetch from the external device memory. Once the BiKD completes the kernel execution, then the BiKD launches the next kernel (i.e. the next two feature elements of the first two vertices in the example). Note that this example exhibits BiKD's conceptual kernel partitioning approach and BiKD kernels are actually designed to maximize the parallelism of GPU architecture.

Algorithm 3 Host-Side Code for Launching BiKD Kernels

```
1:  for g ← 0 to G do
2:      v ← subgraph[g].size()
3:      blk ← (v, 1, 1)
4:      for f ← 0 to F step cta_size do
5:          bikd <<< blk, cta_size >>> (subgraph[g], f)
6:      end for
7:      if f < F then
8:          bikd_tail <<< blk, cta_size >>> (subgraph[g], f)
9:      end if
10: end for
```

Algorithm 3 presents the execution flow of the host-side code for launching the partitioned BiKD aggregation kernels. Note that BiKD partitions the adjacency matrix of a graph structure into multiple subgraphs. The feature matrix associated with vertices is vertically partitioned based on the *cta_size* of a single kernel. Namely, a single partition of BiKD includes v vertices and *cta_size* feature elements. BiKD iteratively launches the aggregation kernel for each subgraph g and multiple feature partitions. Even though BiKD launches smaller aggregation kernels multiple times, the kernel launch overhead is negligible compared to the performance benefits of kernel partitioning. Note that recent articles reveal the kernel launch overhead is significantly low

for modern GPUs [37], [38], [39], [40]. In order to handle the boundary condition of the feature matrix, the host code launches *BIKD_tail* if the size of a feature vector is not divided by *cta_size*.

Algorithm 4 Aggregation Kernel Implementation Using BiKD

1: $vid \leftarrow vertex_id[blkIdx.x]$
2: $offset \leftarrow f + \text{threadIdx.x}$
3: $src_addr \leftarrow vid \times len + offset$
4: $preload \leftarrow out[src_addr]$
5: **for** $i \leftarrow g_ptr[vid]$ to $g_ptr[vid + 1]$ **do**
6: $dst_addr \leftarrow g_idx[i] \times len + offset$
7: $dst_val \leftarrow g_val[i]$
8: $tmp = AGG(tmp, f_val[dst_addr], dst_val)$
9: **end for**
10: $out[src_addr] = AGG(tmp, preload)$

Algorithm 4 represents the BiKD aggregation kernel that handles the bidirectionally partitioned feature data. The BiKD kernel gets the vertex id from the precomputed array index by $blkIdx.x$. The *offset* value can be calculated from $threadIdx.x$ and f given as an argument from the host code. Compared to the baseline RowSplit kernel introduced in Section II-C, BiKD repeatedly demands output features to update partial outputs. In order to hide the long latency of fetching output features, we add a temporary feature value (i.e. *preload*) in the BiKD kernel. By excluding output feature updates from the iterative loop, the BiKD kernel minimizes the performance overhead for updating output features.

V. EVALUATION

A. EXPERIMENT SETUP

We evaluate the proposed kernel design approaches using large-scale graph datasets on a commodity GPU platform. For the evaluation, we study ten large-scale real-world graph datasets [17], [41], [42], [43] and two synthetic graph structures generated by the continuous network expansion algorithm [44]. Table 1 lists the graph datasets examined in this study, arranged in ascending order based on the number of vertices. All experiments are conducted on an NVIDIA RTX A6000 GPU, as summarized in Table 2. We implement the proposed GCN kernels using CUDA C/C++ and compile them with CUDA toolkit 11.7 [45]. Performance metrics are collected using NVIDIA Nsight [46].

We implement several GCN aggregation kernel variants to evaluate different kernel design choices. We implement the GCN kernels that employ CTA-level vertex mapping and kernel decomposition approaches (i.e., *FP*, *GP*, and *BiKD*). We also implement a *Naïve* kernel, where only CTA-level vertex mapping is applied without kernel partitioning.

We compare the proposed kernels against representative state-of-the-art GCN aggregation baselines. *RowSplit* is a warp-based GCN kernel design presented in [11], [32], [33]. *Ge-SpMM* is a highly optimized warp-based GCN kernel design approach proposed in [16]. We also study the popular GNN framework, the Deep Graph Library (*DGL*) v0.9.1 [17] as a framework-level baseline. To evaluate the effectiveness of preprocessing-based graph partitioning for improving data locality, we further compare BiKD against graph partitioning–based approaches. DGL-METIS applies METIS-based graph reordering as an offline preprocessing step to improve vertex locality. [13] GNNAdvisor is a state-of-the-art GNN framework that adopts lightweight Rabbit ordering to enhance data locality with low preprocessing overhead. [49]

TABLE 1. Graph datasets.

Dataset	No. of vertices	Average degree	Feature length
MC18 [42]	196607	1530.64	384
RD [17]	232965	489.23	602
KG18 [43]	262144	80.74	384
SYN500k [44]	500000	200	384
KG19 [43]	524288	83.09	384
DBLP [47]	540486	56.41	384
SYN1M [44]	1000000	400	384
KG20 [43]	1048576	85.11	384
HW [47]	1139905	99.91	384
AP [41]	1569960	168.37	200
KG21 [43]	2097152	86.82	384
OK [48]	3072441	41.68	384

TABLE 2. Evaluation platform.

Host server	
CPU	13th Gen Intel i9-13900k @ 5.5GHz
Host memory	128GB, DDR5
OS	Ubuntu 22.04.3 (kernel ver. 5.15.0-92)
CUDA	CUDA toolkit 11.7
GPU	
Model	NVIDIA RTX A6000
SM	84 SMs @ 1.410GHz
Device memory	48GB, GDDR6 @ 2GHz
L2 cache	6MB
L1 cache / SM	128KB

B. PERFORMANCE

Figures 8 and 9 exhibit the performance of GCN aggregation kernels and GCN full-layer operations, respectively. We use the DGL kernels as a baseline, and the performance of all kernels is normalized to the baseline. The evaluation results show that our design approaches can effectively improve the performance of GCN kernels. Namely, the Naïve kernel design improves performance by applying CTA-level vertex mapping, and GCN kernels can be further enhanced by applying kernel partitioning. Compared to the baseline, Naïve improves the performance of GCN kernels by 16.1%. FP, GP, and BiKD exhibit 1.22×, 1.53×, and 2.22× speedups, respectively, compared to DGL (i.e., the baseline). BiKD also improves the performance of GCN aggregation kernels by 82.6% and 45.1% compared to FP and GP, respectively.

We further compare BiKD with graph partitioning–based approaches, including DGL-METIS and GNNAdvisor, which aim to improve performance through graph partitioning.

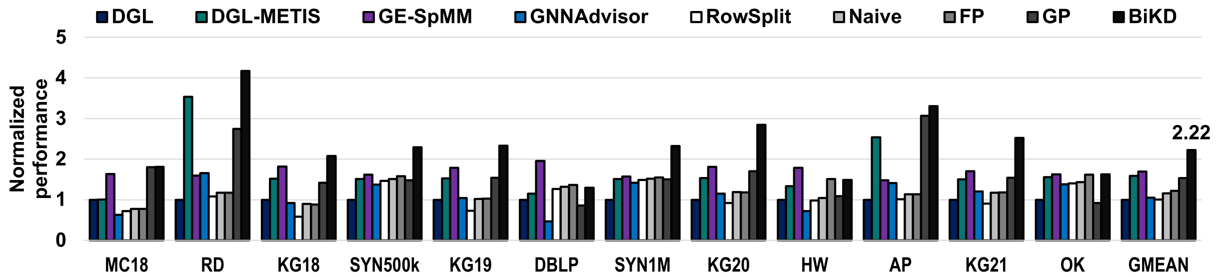


FIGURE 8. Performance of GCN aggregations.

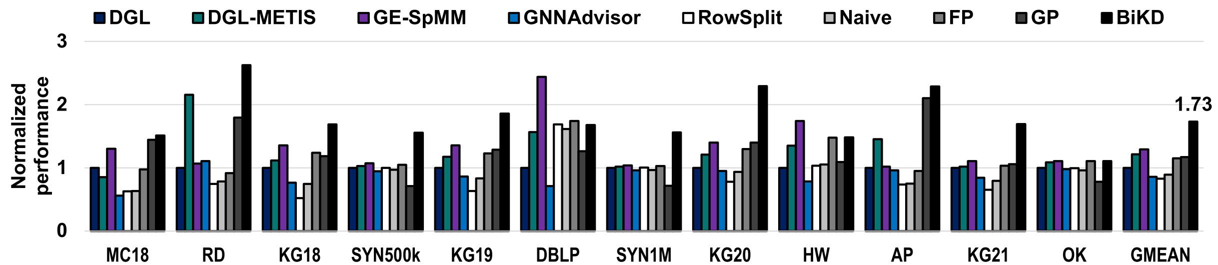


FIGURE 9. Performance of GCN full layers.

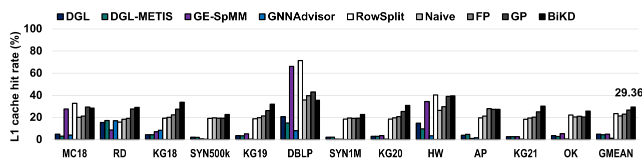


FIGURE 10. L1 cache hit rate.

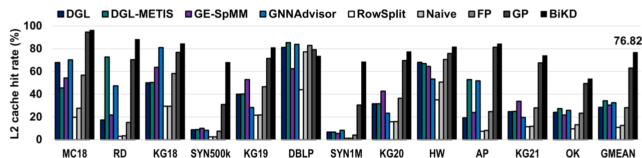


FIGURE 11. L2 cache hit rate.

Compared to these approaches, BiKD achieves 1.39× and 2.11× higher performance for GCN aggregation kernels than DGL-METIS and GNNAdvisor, respectively. The evaluation results reveal that the bidirectional kernel decomposition employed by BiKD can significantly improve the performance of GCN aggregation kernels by exploiting inter-vertex locality in the GPU cache hierarchy.

As shown in Figure 9, BiKD achieves the speedup of 1.73×, 1.21×, 1.34×, and 2.02× for full-layer GCN kernels compared to DGL (i.e., baseline), DGL-METIS, Ge-SpMM, and GNNAdvisor, respectively. BiKD also exhibits effective performance uplifts for full-layer GCN operations since aggregation phases are critical parts of GCN kernels on GPU [8], [9]. Consequently, overall GCN performance can be significantly improved by BiKD since BiKD can optimize the performance of GCN aggregations effectively.

C. MEMORY SYSTEM ANALYSIS

Figure 10 and 11 depict the aggregation kernel's L1 and L2 cache hit rates using the kernel design approaches for each dataset. Note that in the aggregation phase each vertex accesses the neighbor vertices only once and we cannot estimate when the neighbor vertices are re-referenced by other vertices later. Since GPU's small L1 cache cannot hold the neighbor vertex data until they are re-referenced, our BiKD kernel design focuses on utilizing the shared L2 cache that can accommodate more features of vertices. By partitioning features associated with vertices to exploit inter-vertex locality, BiKD achieves 29.4% hit rates for the L1 cache, which improves hit rates by 6.07× and 6.02× compared to DGL and GE-SpMM respectively. Naïve design underperforms compared to RowSplit as the Naïve kernels create excessive in-flight memory requests to provoke increased data fetch latency. On the other hand, our decomposition kernel design approaches can mitigate the performance hurdles in the GPU memory hierarchy. GE-SpMM exhibits lower warp occupancy and exploits shared memory to preload graph indices and values, thus GE-SpMM can utilize the L1 and L2 caches only for memory transactions of the feature matrix. However, in the case of graphs *MC18* and *HW*, RowSplit shows the highest L1 cache hit rate, which means that the graph is well clustered so that vertices can exploit inter-vertex locality at the L1 cache level. Furthermore, *DBLP* is highly clustered, so RowSplit-based kernels such as RowSplit and GE-SpMM can achieve a high hit rate at the L1 cache. Otherwise, BiKD does not work well compared to generic kernels if graphs are clustered.

Since BiKD aims to maximize the utilization of the shared L2 cache, BiKD achieves 76.8% of hit rates at the L2 cache on average, which is 2.70×, 2.24×, 2.52× and 2.35× higher compared to DGL, DGL-METIS, Ge-SpMM, and GNNAdvisor, respectively. Our proposed kernel partitioning approaches (i.e. FP, GP, and BiKD) exhibit significantly higher hit rates at the L2 cache, whereas the

Naïve and RowSplit kernels show very low hit rates. Such evaluation results reveal that the proposed kernel decomposition approaches can effectively utilize GPU's cache hierarchy compared to the prior aggregation kernel design methods. FP exhibits higher L2 hit rates compared to Naïve and RowSplit but FP utilizes the L2 cache less efficiently compared to GP and BiKD. It is because FP partitions feature vectors in a warp granularity (i.e. 32 threads) for all vertices thus the partitioned kernels may not fit well within the L2 cache. As mentioned in the previous analysis, *DBLP* is highly clustered thus Ge-SpMM and RowSplit can exploit inter-vertex locality at the L1 cache rather than L2 cache, thus L2 cache hit rates are rather high for all kernel design approaches. On the other hand, BiKD exhibits significantly high L2 hit rates for unclustered and high-degree graph datasets. To further evaluate whether graph partitioning alone can effectively improve cache utilization, we compare BiKD with preprocessing-based graph partitioning approaches, including DGL-METIS and GNNAdvisor. Although DGL-METIS and GNNAdvisor apply graph partitioning to improve inter-vertex locality, their L2 cache hit rates remain lower than that of BiKD. This is because such preprocessing-based approaches primarily cluster vertices based on graph connectivity, without considering the limited capacity of the GPU L2 cache. Moreover, graph partitioning can even degrade cache locality in certain cases. As shown in Figure 11, GNNAdvisor exhibits an approximately 1.3× lower L2 cache hit rate on the KG20 dataset compared to the baseline without graph partitioning, indicating that partitioning based solely on graph structure does not always guarantee improved cache utilization on GPUs. In contrast, BiKD explicitly incorporates the GPU L2 cache capacity into kernel partitioning, enabling partitions to remain resident in the cache and thereby achieving consistently higher L2 cache hit rates across diverse graph datasets.

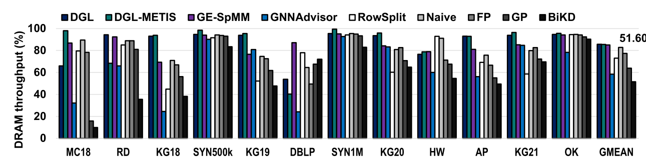


FIGURE 12. DRAM throughput rate.

In order to evaluate the amount of external memory accesses by the kernel design approaches, we measure the DRAM throughput rate of the aggregation kernels as shown in Figure 12. We compute the DRAM throughput rate in the figure as the total amount of data transactions over the theoretical DRAM channel bandwidth of the GPU testbed. Hence, a higher throughput rate indicates that an aggregation kernel creates more memory transactions that fetch data from the external memory. As the data fetch latency from the external memory is significantly higher compared to on-chip caches, frequent external memory transactions can become a critical performance burden for aggregation kernels.

Our evaluation exhibits that kernel design approaches without kernel partitioning (i.e., DGL, GE-SpMM, RowSplit, and Naïve) rely heavily on data fetches from external memory, resulting in high DRAM throughput rates. Note that such kernel designs cannot efficiently exploit inter-vertex locality at the L2 cache, leading to frequent cache misses and excessive DRAM accesses.

From the perspective of external memory traffic, preprocessing-based graph partitioning approaches such as DGL-METIS and GNNAdvisor exhibit DRAM throughput characteristics that lie between kernel designs without partitioning and BiKD. While graph partitioning can reduce redundant memory accesses by improving structural locality, its impact on DRAM traffic becomes limited when the working set of features exceeds the capacity of on-chip caches. As a result, these approaches still generate a considerable amount of external memory transactions compared to BiKD.

In contrast, kernel partitioning approaches (i.e., FP, GP, and BiKD) exhibit substantially lower DRAM throughput rates. These kernels utilize the L2 cache more effectively, allowing a larger fraction of demand data to be served from on-chip memory. Among them, FP shows higher DRAM throughput than GP and BiKD, as FP utilizes the L2 cache less efficiently due to its coarse-grained thread allocation, as discussed in the previous paragraph. GP enables more flexible feature partitioning; however, it still incurs frequent cold misses for partial output data, resulting in additional external memory transactions. BiKD utilizes the L2 cache most effectively, reducing external DRAM accesses by 34.1% and 21.4% on average compared to Naïve and RowSplit, respectively.

D. MEMORY FOOTPRINT COMPARISON

In order to evaluate whether the proposed decomposition increases memory consumption, we measure the GPU memory footprint by BiKD as shown in Figure 13. We use

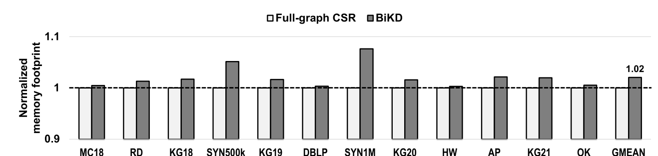


FIGURE 13. Memory footprint comparison.

the full-graph CSR representation as the baseline, which consists of row pointers, column indices, and value arrays, as well as input and output feature matrices. This baseline represents DGL, DGL-METIS, GE-SpMM, RowSplit, and GNNAdvisor, as these approaches operate on the original graphs formatted in CSR. BiKD includes partitioned CSR arrays, vertex ID arrays, and partition metadata such as partition offsets, in addition to the input and output feature matrices. All CSR values and feature tensors are stored in FP32, and the input feature dimension is fixed to 384. Note that FP and GP are not separately reported in Figure 13.

FP simply partitions the feature dimension during kernel execution without altering the graph representations or duplicating feature matrices, thus the memory footprint of FP is identical to that of the original CSR-formatted graphs. GP adopts the same graph binning scheme and L2-size graph partitions as BiKD. BiKD further applies feature-direction decomposition. Hence, the graph structures of GP and BiKD are identical. The evaluation results show that BiKD incurs only marginal memory overhead compared to the full-graph CSR baseline. Although BiKD stores additional partition metadata and vertex ID arrays, the edge data are reorganized into partitioned CSR structures rather than duplicated. As a result, BiKD requires only 1.02× memory footprint on average compared to the full-graph CSR representation.

E. TLP TRADE-OFF

As shown in Figure 14, we evaluate the warp occupancy of aggregation kernels to compare the thread-level parallelism (TLP) achieved by different vertex mapping approaches. Our evaluation reveals that the average warp occupancy of BiKD is significantly higher than that of RowSplit and Ge-SpMM, which employ warp-level vertex mapping. GNNAdvisor, which also adopts warp-level neighbor-group mapping to improve workload balance, achieves higher warp occupancy than conventional warp-level baselines. However, our evaluation results show that the proposed CTA-level vertex mapping in BiKD achieves even higher GPU resource utilization by exposing more parallelism across warps within a CTA. Namely, the proposed CTA-level vertex mapping approaches can potentially improve the performance of aggregation kernels by achieving higher TLP. Since higher TLP may provoke severe congestion in the GPU memory hierarchy, aggregation kernel optimizations need to be accompanied by enhancements in memory systems. For instance, Naïve and GP kernels exhibit lower performance than BiKD even though they achieve higher warp occupancy. Consequently, BiKD achieves significant performance improvement by jointly optimizing GPU core utilization through CTA-level vertex mapping and memory efficiency through cache-aware kernel partitioning.

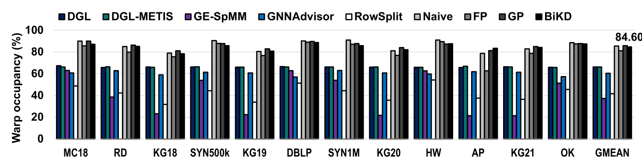


FIGURE 14. Average warp occupancy within a CTA.

F. PREPROCESSING OVERHEAD

Figure 15 exhibits the preprocessing time of BiKD normalized to the graph partitioning time by the METIS algorithm when partitioning a graph with the same number of partitions as BiKD. METIS is a representative graph partitioning algorithm used in various GNN optimization techniques

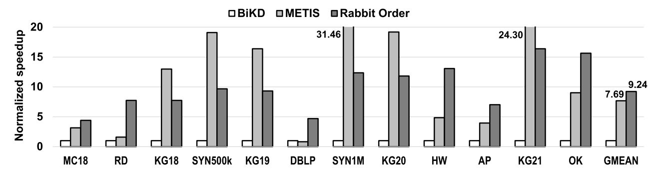


FIGURE 15. Preprocessing overhead comparison.

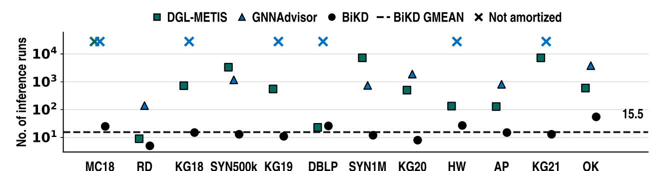


FIGURE 16. Preprocessing amortization (lower is better).

employing graph partitioning [13]. We additionally include Rabbit Order, which is used in GNNAdvisor, as another graph reordering baseline for comparison. GP and BiKD partition the graph through the graph binning technique. Graph binning simply partitions the graph row-wise based on neighbor vertex IDs and regenerates the adjacency matrix of each subgraph from the original adjacency matrix. Thus, unlike METIS and Rabbit Order, graph binning does not require graph-level reordering or clustering during preprocessing. As a result, there is a large gap between the preprocessing time of graph binning and METIS for datasets that require a large number of graph partitions. In particular, for the *SYN1M* dataset, which requires 62 graph partitions, graph binning takes 31.46× less preprocessing time than METIS. On the other hand, the *DBLP* dataset is highly clustered, so the preprocessing overhead is relatively small even when METIS or Rabbit Order is applied. Moreover, graph partitioning is required only once for *DBLP*, which limits the benefit of graph binning. Even for well-clustered graphs such as *MC18* and *HW*, graph binning can provide preprocessing benefits when the graph is partitioned multiple times. Graph binning achieves 3.17× and 3.96× faster preprocessing performance than METIS for *MC18* and *HW*, respectively. Across all datasets, BiKD consistently shows lower preprocessing overhead than both METIS and Rabbit Order. As a result, BiKD achieves 7.69× faster preprocessing time on average than the graph partitioning method based on the METIS algorithm, and 9.24× faster preprocessing time compared to Rabbit Order.

As shown in Figure 16, we further evaluate the preprocessing overhead of BiKD by measuring the number of full-layer GNN inferences required to compensate for the preprocessing time. Note that researchers have proposed preprocessing approaches for large-scale graph structures to reduce the GNN inference time. Hence, we compute the number of GNN inference runs that can amortize the preprocessing overhead, as represented in the following equation.

$$N_{\mathrm{amort}} = \left\lceil \frac{T_{\mathrm{Pre}}}{T_{\mathrm{DGL}} - T_{\mathrm{GNN}}} \right\rceil \tag{3}$$

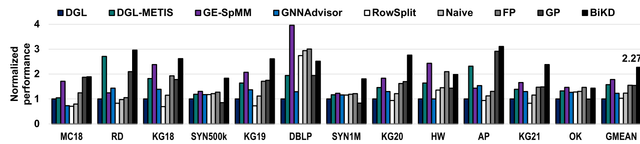


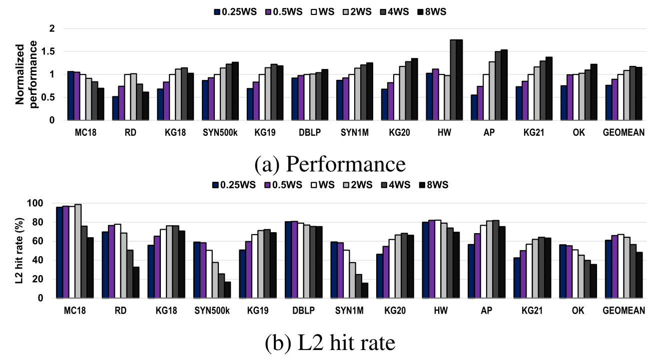
FIGURE 17. Full layers of GIN performance (normalized to the baseline).

We use DGL as the baseline approach since DGL directly performs GCN inferences using the CSR-formatted graphs without preprocessing thereby incurring zero preprocessing overhead. We assume that the GNN inference time can decrease due to preprocessing, thus the denominator of Equation 3 represents the performance gain (i.e. the reduced execution time of GNN inference) of preprocessing. We divide the preprocessing overhead with the reduced GNN inference time to measure the number of GNN full-layer inferences for amortizing the preprocessing overhead. Note that the preprocessing overhead cannot be amortized (i.e. *Not amortized* in Figure 16) if GNN inference time increases despite adopting preprocessing.

Our evaluation reveals that BiKD can amortize the preprocessing overhead with a small number of GNN inference runs since BiKD achieves high performance gain with the simple preprocessing approach. On average, BiKD can amortize the preprocessing overhead with 15.5 full-layer GNN inference runs. On the other hand, METIS exhibits heavy preprocessing overhead thus DGL-METIS requires a large number of inference runs to compensate for the preprocessing overhead of METIS. GNNAdvisor also requires many inference runs to compensate for its preprocessing time.

G. PERFORMANCE OF OTHER GNN APPLICATIONS

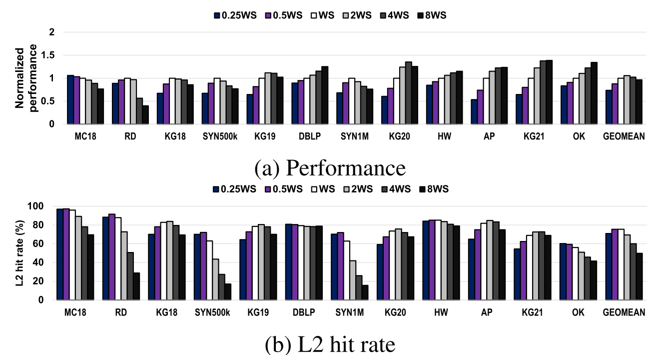
Our proposed BiKD design approach can be extended to other GNN architectures that follow a SpMM-like aggregation pattern, thus BiKD is not limited to GCN aggregation kernels. BiKD can be applied to various GNN models such as GraphSAGE (mean or max aggregator), graph isomorphism network (GIN), and attention-based models such as GAT, where aggregation can be expressed as sparse-dense or masked SpMM operations. To demonstrate this generality, we extend BiKD to the GIN model and evaluate its performance under the same experimental setting. Figure 17 shows full layers of GIN performance across datasets normalized to the baseline. GIN's aggregation is similar to GCN models, but its aggregator updates the maximum value only from neighbor vertices. After aggregation, GIN adjusts the source vertex with the learnable parameter and then adds the aggregated vector. Since GIN's aggregation has simpler operations than GCN's, the memory latency hiding becomes harder because there is less chance of intervening independent instructions between memory operations. As a result, BiKD achieves 2.27× speedup over DGL, 1.33× over GE-SpMM, 1.44× over DGL-METIS, and 1.85× over GNNAdvisor.



(a) Performance

(b) L2 hit rate

FIGURE 18. Performance and L2 hit rate by the partition size of GP.



(a) Performance

(b) L2 hit rate

FIGURE 19. Performance and L2 hit rate by the partition size of BiKD.

H. DESIGN SPACE EXPLORATION: PARTITIONING SIZE

The main purpose of the kernel partitioning approaches proposed in this work is to exploit the inter-vertex locality of the feature matrix in the GPU's cache hierarchy. In order to explore the performance impact of the kernel partitioning strategies, we conduct a design space exploration by varying the partition size (i.e. window size) of GPU and BiKD. Note that FP should always partition the feature matrix in a warp granularity (i.e. 32 threads) thus FP cannot adjust the partition size flexibly. Figures 18 and 19 exhibit the performance and L2 hit rate of GP and BiKD respectively by various kernel partition sizes. We vary the partition size from 0.25× to 8× of the GPU's L2 cache capacity. The performance by partition sizes is normalized to the performance results where the partition size is equal to the L2 cache capacity (i.e. WS).

If the partition size is smaller than the L2 cache capacity, the feature data can be fully accommodated in the L2 cache thus L2 hit rates can become relatively high. However, as the partition size becomes smaller, GP and BiKD create more kernels thus kernel launch overhead can be increased. Furthermore, the smaller kernels can provoke more frequent partial data updates since the L2 cache includes less feature vectors associated with neighbor vertices. Note that the features of a neighbor vertex exist in the partitioned working set, the partly updated features of the origin vertex are demanded again thus the aggregation kernel creates memory transactions. For GP, since each vertex includes all feature

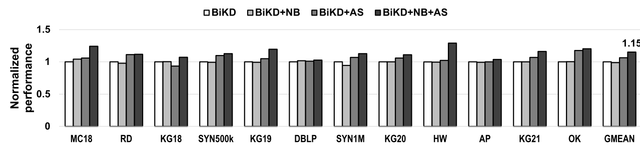


FIGURE 20. Performance impact by scheduling strategies.

elements of the corresponding feature vector, the partitioned kernel includes fewer vertices thus more frequent partial updates can be created. BiKD kernels can include more vertices within a partitioned kernel since BiKD partitions the feature matrix in both row and column dimensions. Thus, BiKD can reduce the number of partial updates compared to GP. On the other hand, when the partition size is larger than the L2 cache capacity, the L2 cache cannot be utilized efficiently. Our evaluation results reveal that the L2 hit rates usually drop as the partition size increases over the L2 cache capacity.

Consequently, the partition sizes of GP and BiKD trade off between the L2 cache utilization and performance overhead by partial feature updates. Our evaluation with the GPU testbed reveals BiKD exhibits optimal performance when the partition size is in the range of the 1× to 2× of the L2 cache capacity. For GP, the performance overhead by partial feature updates is rather high, thus performance can be improved even though the partition size is larger (i.e. 4× to 8× of the L2 cache size).

I. LOAD BALANCING

In order to investigate the performance impact of scheduling strategies for load balancing, we conduct an ablation study that combines the BiKD kernel decomposition approach with CTA and warp scheduling algorithms. Figure 20 exhibits the performance impact of the scheduling strategies applied to BiKD. While BiKD can exploit the inter-vertex locality of graph structures in the GPU's cache hierarchy, the performance of the BiKD kernels can suffer from load imbalance due to the non-uniform distributions of non-zero elements across row blocks. Note that the vertices are connected irregularly in graph structures thus the number of non-zero elements in an adjacency matrix can be highly skewed. As a result, the BiKD's CTA executions can be imbalanced due to the uneven distributions of non-zero elements in row-block partitions. Furthermore, the warp execution times within a CTA can be imbalanced due to highly skewed vertex degrees. To address such load imbalance issues, we apply two scheduling strategies to BiKD, neighborhood-aware scheduling (*NB*) and adaptive warp scheduling (*AS*).

When the *NB* scheduling strategy is applied to BiKD (i.e. *BiKD+NB*), row blocks are partitioned to include a similar number of non-zero elements. By balancing the number of non-zero elements across partitioned row blocks, CTAs in a BiKD kernel can exhibit similar execution times thus we can mitigate the underutilization of GPU's SMs. On the other hand, when BiKD employs the *AS* scheduling strategy (i.e. *BiKD+AS*), the number of warps assigned to each row is adjusted according to the number of non-zero elements in the row. Hence, *AS* can exploit higher TLP for high-degree vertices to mitigate imbalanced warp-level execution times and improve warp occupancy. When both scheduling strategies are applied to BiKD (i.e. *BiKD+NB+AS*), both inter-CTA and intra-CTA load imbalance issues can be alleviated. Our evaluation results shown in Figure 20 reveal that *BiKD+NB+AS* can improve the performance of BiKD by up to 1.29× with a geometric mean of 1.14×.

VI. RELATED WORK

A. CHARACTERISTICS OF GNN KERNELS

Yan et al. [9] analyzed the execution patterns of GCN workloads on real GPUs using hardware performance profilers. The authors compared the performance of GCN kernels with MLP-based neural network kernels and typical graph processing applications. They revealed that the aggregation and combination phases of GCN workloads have diverse characteristics and the aggregation step is critical for the performance of GCNs since the performance counts of GPU memory hierarchy exhibit very low cache hit rates and heavy data traffic to external DRAM during the aggregation phase. Baruah et al. [50] presented a benchmark suite, called GNNMark, for characterizing GNN training workloads on GPUs. The authors analyzed the various aspects of GNN training kernels using a GPU performance profiler to reveal system-level performance bottlenecks. In this paper, we intensively analyze the characteristics of graph structures and associated feature data to reveal the inefficient use of GPU's cache hierarchy and performance overhead by frequent external memory accesses. To address the performance issues of aggregation kernels, we present kernel decomposition approaches that can exploit inter-vertex locality in GPU's L2 cache.

B. GNN OPTIMIZATION ON GPU

Huang et al. [16] proposed a general-purpose sparse matrix multiplication kernel, called GE-SpMM. The authors presented the optimized kernel design that exploits GPU's shared memory and register file to perform multiplications between CSR-formatted and general matrices. The authors exhibited that GNN performance can be improved when GE-SpMM replaces the existing SpMM kernels.

Wang et al. [49] proposed GNNAdvisor, an adaptive runtime system for efficient GNN acceleration on GPUs. The authors designed a 2D workload management strategy that partitions computations along both neighbor and feature dimensions to balance irregular workloads and improve GPU utilization. In addition, they adopted Rabbit Reordering to rearrange node IDs based on community structures, thereby enhancing spatial and temporal locality, improving cache efficiency, and reducing redundant memory accesses during aggregation.

Tian et al. [32] proposed PCGCN that partitions a large graph dataset into smaller subgraphs to exploit possible inter-vertex locality. PCGCN generates subgraphs considering the connectivity among vertices to maximize data locality.

Xie et al. [51] proposed Accel-GCN, a high-performance GPU kernel that mitigates workload imbalance and memory access irregularity during SpMM operations. It employs a lightweight degree sorting to group nodes for better data locality, a block-level partition strategy for workload balance, and a combined warp strategy to maximize memory coalescing. This co-optimization of techniques was shown to significantly improve overall memory bandwidth and efficiency for GCNs.

Recent studies on GCN acceleration have explored leveraging GPU Tensor Cores to improve computational throughput for GNN workloads, particularly by transforming irregular graph computations into forms amenable to dense matrix operations. Wang et al. [52] proposed TC-GNN, a GPU acceleration framework that bridges sparse GNN computations and dense tensor-core operations. It introduces a Sparse Graph Translation (SGT) technique that reorganizes irregular sparse graphs into dense tiles suitable for Tensor Core Units (TCUs), and employs a collaborative execution model that coordinates CUDA cores and TCUs to improve computational efficiency for sparse–dense hybrid workloads. Wang et al. [53] proposed QGTC, a GPU acceleration framework that leverages Tensor Cores for efficient GNN aggregation. QGTC reformulates graph aggregation into a sequence of quantized sparse–dense matrix multiplications, enabling the use of low-precision Tensor Core operations. By applying quantization and block-based aggregation, QGTC improves computational throughput while reducing memory bandwidth requirements for GNN workloads.

We propose GCN kernel design approaches that employ feature-level and graph-level partitioning to exploit inter-vertex locality of graph structures within GPU's L2 cache. Our BiKD kernel design approach can utilize GPU's cache hierarchy more efficiently to improve the performance of GCN aggregation kernels dramatically. We also propose a CTA-level vertex mapping approach to improve the resource utilization of GPU.

C. GRAPH LEARNING

Bai et al. [54] presented hierarchical-aligned quantum Jensen-Shannon kernels (HAQJSKs), a quantum kernel method for graph classification that leverages continuous-time quantum walks. By aligning variable-sized graphs into fixed-size hierarchical structures, HAQJSK captures both local and global graph properties while ensuring permutation invariance and positive definiteness. Li et al. [55] proposed a permutation-equivariant graph framelet attention network (PEGFAN), which incorporates permutation-equivariant graph framelets into GNNs to enhance performance on heterophilous graphs. By using Haar-type framelets for multi-scale feature extraction, PEGFAN achieves strong performance across several challenging heterophilous benchmarks. Li et al. [56] published an article that surveys recent advances in graph neural networks. The article outlines progress in theoretical understanding, novel architectures, and applications such as node classification, link prediction, and graph generation, while highlighting open challenges in scalability, trustworthiness, and dynamic graph learning. Ouyang et al. [57] proposed Graph Neural Evolution (GNE), which applies spectral GNNs to evolutionary optimization by representing individuals as graph nodes and modeling global correlations through frequency-domain filtering. This direction indicates that GNN-based operators are becoming important computational primitives beyond traditional graph mining workloads, extending their impact to broader computational intelligence applications. As these applications scale to larger graphs, populations, and feature dimensions, the efficiency of underlying graph propagation, spectral filtering, and sparse-dense aggregation operations becomes increasingly important. BiKD is complementary to such algorithmic advances because it improves the GPU efficiency of aggregation kernels that are widely used in GNN models.

VII. CONCLUSION

In this paper, we present BiKD, a bidirectional kernel decomposition for optimizing the access pattern of large-scale GCN on GPU. First, we find several observations; compared to graph processing GCN has a huge size feature matrix which incurs severe memory congestion in the cache hierarchy and the access pattern of the feature matrix becomes a critical issue of the performance; prior works that implement kernels exploiting warp-level parallelism are suffered from low occupancy which comes from graph's irregularity. To solve these problems, we design kernels exploiting CTA-level parallelism to alleviate load imbalance between vertices and apply two-way partitioning techniques: feature partitioning makes each kernel consume the feature matrix column-wise; graph partitioning makes each kernel consume row-wise. In addition, to orchestrate two techniques, we propose a bi-directional kernel that combines two partitioning techniques efficiently and a dedicated modified CSR format. The experimental results show that BiKD outperforms for processing high-degree graphs and improves the memory system from the L1 cache to the DRAM.

ACKNOWLEDGMENT

(Inje Kim and Jihun Lee contributed equally to this work.)

REFERENCES

[1] J. Zhang, X. Hu, Z. Jiang, B. Song, W. Quan, and Z. Chen, "Predicting disease-related RNA associations based on graph convolutional attention network," in *Proc. IEEE Int. Conf. Bioinf. Biomed. (BIBM)*, Nov. 2019, pp. 177–182.

[2] J. Yu, H. Yin, J. Li, M. Gao, Z. Huang, and L. Cui, "Enhance social recommendation with adversarial graph convolutional networks," *IEEE Trans. Knowl. Data Eng.*, vol. 34, no. 8, pp. 3727–3739, Aug. 2022.

[3] F. Xu, J. Lian, Z. Han, Y. Li, Y. Xu, and X. Xie, "Relation-aware graph convolutional networks for agent-initiated social e-commerce recommendation," in *Proc. 28th ACM Int. Conf. Inf. Knowl. Manage.* New York, NY, USA: Association for Computing Machinery, Nov. 2019, pp. 529–538, doi: 10.1145/3357384.3357924.

[4] Y. Zheng, C. Gao, X. He, Y. Li, and D. Jin, "Price-aware recommendation with graph convolutional networks," in *Proc. IEEE 36th Int. Conf. Data Eng. (ICDE)*, Apr. 2020, pp. 133–144.

[5] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron, "Pannotia: Understanding irregular GPGPU graph applications," in *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*, Sep. 2013, pp. 185–195.

[6] G. Koo, H. Jeon, and M. Annavaram, "Revealing critical loads and hidden data locality in GPGPU applications," in *Proc. IEEE Int. Symp. Workload Characterization*, Oct. 2015, pp. 120–129.

[7] G. Koo, Y. Oh, W. W. Ro, and M. Annavaram, "Access pattern-aware cache management for improving data utilization in GPU," *ACM SIGARCH Comput. Archit. News*, vol. 45, no. 2, pp. 307–319, Jun. 2017, doi: 10.1145/3140659.3080239.

[8] I. Kim, J. Jeong, Y. Oh, M. K. Yoon, and G. Koo, "Analyzing GCN aggregation on GPU," *IEEE Access*, vol. 10, pp. 113046–113060, 2022.

[9] M. Yan, Z. Chen, L. Deng, X. Ye, Z. Zhang, D. Fan, and Y. Xie, "Characterizing and understanding GCNs on GPU," *IEEE Comput. Archit. Lett.*, vol. 19, no. 1, pp. 22–25, Jan. 2020.

[10] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*, Oct. 2009, pp. 44–54.

[11] C. Yang, A. Buluç, and J. D. Owens, "Design principles for sparse matrix multiplication on the GPU," in *Proc. Euro-Par : Parallel Process., 24th Int. Conf. Parallel Distrib. Comput.*, Turin, Italy, Aug. 2018, pp. 672–687, doi: 10.1007/978-3-319-96983-1_48.

[12] S. Kumar, S. K. Das, and R. Biswas, "Graph partitioning for parallel applications in heterogeneous grid environments," in *Proc. 16th Int. Parallel Distrib. Process. Symp.*, Apr 2002, p. 7.

[13] D. LaSalle and G. Karypis, "A parallel hill-climbing refinement algorithm for graph partitioning," in *Proc. 45th Int. Conf. Parallel Process. (ICPP)*, Aug. 2016, pp. 236–241.

[14] J. Arai, H. Shiokawa, T. Yamamuro, M. Onizuka, and S. Iwamura, "Rabbit order: Just-in-time parallel reordering for fast graph analysis," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, May 2016, pp. 22–31.

[15] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2017.

[16] G. Huang, G. Dai, Y. Wang, and H. Yang, "GE-SpMM: General-purpose sparse matrix-matrix multiplication on GPUs for graph neural networks," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2020, pp. 1–12.

[17] M. Wang, L. Yu, D. Zheng, Q. Gan, Y. Gai, Z. Ye, M. Li, J. Zhou, Q. Huang, C. Ma, Z. Huang, Q. Guo, H. Zhang, H. Lin, J. Zhao, J. Li, A. J. Smola, and Z. Zhang, "Deep graph library: Towards efficient and scalable deep learning on graphs," 2019, *arXiv:1909.01315*.

[18] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Trans. Neural Netw.*, vol. 20, no. 1, pp. 61–80, Jan 2009.

[19] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, "A comprehensive survey on graph neural networks," 2019, *arXiv:1901.00596*.

[20] J. Chen, T. Ma, and C. Xiao, "FastGCN: Fast learning with graph convolutional networks via importance sampling," 2018, *arXiv:1801.10247*.

[21] NVIDIA. (2014). *Nvidia NVIDIA GeForce GTX 980 Featuring Maxwell, The Most Advanced GPU Ever Made*. [Online]. Available: https://www.microway.com/download/whitepaper/NVIDIAMaxwellGM204ArchitectureWhitepaper.pdf

[22] NVIDIA. (2019). *NVIDIA Turing GPU Architecture, Graphics Reinvented*. [Online]. Available: https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf

[23] NVIDIA. (2017). *NVIDIA Tesla V100 GPU Architecture, The World's Most Advanced Data Center GPU*. [Online]. Available: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[24] NVIDIA. (2020). *NVIDIA AMPERE GA102 GPU ARCHITECTURE, Second-Generation RTX*. [Online]. Available:https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf

[25] D. Yan, W. Wang, and X. Chu, "Demystifying tensor cores to optimize half-precision matrix multiply," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, May 2020, pp. 634–643.

[26] S. Markidis, S. W. D. Chien, E. Laure, I. B. Peng, and J. S. Vetter, "NVIDIA tensor core programmability, performance & precision," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops (IPDPSW)*, May 2018, pp. 522–531.

[27] K. Kim, S. Lee, M. K. Yoon, G. Koo, W. W. Ro, and M. Annavaram, "Warped-preexecution: A GPU pre-execution approach for improving latency hiding," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Mar. 2016, pp. 163–175.

[28] G. Koo, H. Jeon, Z. Liu, N. S. Kim, and M. Annavaram, "CTA-aware prefetching and scheduling for GPU," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, May 2018, pp. 137–148.

[29] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-conscious wavefront scheduling," in *Proc. 45th Annu. IEEE/ACM Int. Symp. Microarchitecture*, Dec. 2012, pp. 72–83.

[30] D. Li, M. Rhu, D. R. Johnson, M. O'Connor, M. Erez, D. Burger, D. S. Fussell, and S. W. Redder, "Priority-based cache allocation in throughput processors," in *Proc. IEEE 21st Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2015, pp. 89–100.

[31] Y. Oh, G. Koo, M. Annavaram, and W. W. Ro, "Linebacker: Preserving victim cache lines in idle register files of GPUs," in *Proc. ACM/IEEE 46th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2019, pp. 183–196.

[32] C. Tian, L. Ma, Z. Yang, and Y. Dai, "PCGCN: Partition-centric processing for accelerating graph convolutional network," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, May 2020, pp. 936–945.

[33] Q. Fu, Y. Ji, and H. H. Huang, "TLPGNN: A lightweight two-level parallelism paradigm for graph neural network computation on GPU," in *Proc. 31st Int. Symp. High-Performance Parallel Distrib. Comput.* New York, NY, USA: Association for Computing Machinery, Jun. 2022, pp. 122–134, doi: 10.1145/3502181.3531467.

[34] M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on GPUs," in *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*, Nov. 2012, pp. 141–151.

[35] Q. Xu, H. Jeon, and M. Annavaram, "Graph processing on GPUs: Where are the bottlenecks?" in *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*, Oct. 2014, pp. 140–149.

[36] O. Kayiran, A. Jog, M. T. Kandemir, and C. R. Das, "Neither more nor less: Optimizing thread-level parallelism for GPGPUs," in *Proc. 22nd Int. Conf. Parallel Architectures Compilation Techn.*, Sep. 2013, pp. 157–166.

[37] X. Tang, A. Pattnaik, H. Jiang, O. Kayiran, A. Jog, S. Pai, M. Ibrahim, M. T. Kandemir, and C. R. Das, "Controlled kernel launch for dynamic parallelism in GPUs," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2017, pp. 649–660.

[38] J. Wang and S. Yalamanchili, "Characterization and analysis of dynamic parallelism in unstructured GPU applications," in *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*, Oct. 2014, pp. 51–60.

[39] J. Wang, N. Rubin, A. Sidelnik, and S. Yalamanchili, "Dynamic thread block launch: A lightweight execution mechanism to support irregular applications on GPUs," in *Proc. ACM/IEEE 42nd Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2015, pp. 528–540.

[40] NVIDIA. (2022). *DYNAMIC PARALLELISM IN CUDA*. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#cuda-dynamic-parallelism

[41] O. Shchur, M. Mumme, A. Bojchevski, and S. Günnemann, "Pitfalls of graph neural network evaluation," 2018, *arXiv:1811.05868*.

[42] J. Mycielski, "Sur le coloriage Des.graphs," *Colloq. Mathematicum*, vol. 3, no. 2, pp. 161–162, 1955. [Online]. Available: https://eudml.org/doc/210000

[43] J. Ang, B. Barrett, K. Wheeler, and R. Murphy, "Introducing the graph 500," *Cray Users Group*, vol. 19, p. 22, Jan. 2010.

[44] A.-L. Barabási and R. Albert, "Emergence of scaling in random networks," *Science*, vol. 286, no. 5439, pp. 509–512, Oct. 1999, doi: 10.1126/science.286.5439.509.

[45] NVIDIA. (2022). *Cuda Toolkit 11.7 Document*. [Online]. Available: https://docs.nvidia.com/cuda/archive/11.7.0/

[46] NVIDIA. (2015). *NVIDIA Nsight Compute Document*. [Online]. Available: https://docs.nvidia.com/nsight-compute/NsightCompute/index.html

[47] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *Proc. AAAI*, 2015, pp. 1–2. [Online]. Available: https://networkrepository.com

[48] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," in *Proc. IEEE 12th Int. Conf. Data Mining*, Dec. 2012, pp. 745–754.

[49] Y. Wang, B. Feng, G. Li, S. Li, L. Deng, Y. Xie, and Y. Ding, "GNNAdvisor: An adaptive and efficient runtime system for GNN acceleration on GPUs," in *Proc. 15th USENIX Symp. Operating Syst. Design Implement. (OSDI)*, Jul. 2020, pp. 515–531. [Online]. Available: https://www.usenix.org/conference/osdi21/presentation/wang-yuke

[50] T. Baruah, K. Shivdikar, S. Dong, Y. Sun, S. A. Mojumder, K. Jung, J. L. Abellán, Y. Ukidave, A. Joshi, J. Kim, and D. Kaeli, "GNNMark: A benchmark suite to characterize graph neural network training on GPUs," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, Mar. 2021, pp. 13–23.

[51] X. Xie, H. Peng, A. Hasan, S. Huang, J. Zhao, H. Fang, W. Zhang, T. Geng, O. Khan, and C. Ding, "Accel-GCN: High-performance GPU accelerator design for graph convolution networks," in *Proc. IEEE/ACM Int. Conf. Comput. Aided Design (ICCAD)*, Oct. 2023, pp. 1–9.

[52] Y. Wang, B. Feng, Z. Wang, G. Huang, and Y. Ding, "TC-GNN: Bridging sparse GNN computation and dense tensor cores on GPUs," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, Jul. 2021, pp. 149–164. [Online]. Available: https://www.usenix.org/conference/atc23/presentation/wang-yuke

[53] Y. Wang, B. Feng, and Y. Ding, "QGTC: Accelerating quantized graph neural networks via GPU tensor core," in *Proc. 27th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.* New York, NY, USA: Association for Computing Machinery, Apr. 2022, pp. 107–119, doi: 10.1145/3503221.3508408.

[54] L. Bai, L. Cui, Y. Wang, M. Li, J. Li, P. S. Yu, and E. R. Hancock, "HAQJSK: Hierarchical-aligned quantum Jensen–Shannon kernels for graph classification," *IEEE Trans. Knowl. Data Eng.*, vol. 36, no. 11, pp. 6370–6384, Nov. 2024.

[55] J. Li, R. Zheng, H. Feng, M. Li, and X. Zhuang, "Permutation equivariant graph framelets for heterophilous graph learning," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 35, no. 9, pp. 11634–11648, Sep. 2024.

[56] M. Li, A. Micheli, Y. G. Wang, S. Pan, P. Lió, G. S. Gnecco, and M. Sanguineti, "Guest editorial: Deep neural networks for graphs: Theory, models, algorithms, and applications," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 35, no. 4, pp. 4367–4372, Apr. 2024.

[57] K. Ouyang, Z. Ke, S. Fu, L. Liu, P. Zhao, and D. Hu, "Learn from global correlations: Enhancing evolutionary algorithm via spectral GNN," in *Proc. AAAI Conf. Artif. Intell.*, 2026, vol. 40, no. 29, pp. 24665–24673.

INJE KIM received the B.S. degree from the School of Electronic and Electrical Engineering, Hongik University, in 2021, and the M.S. degree in computer science and engineering from Korea University, in 2023. From April 2023 to July 2023, he interned with the Software Engineering Team, Neubla, South Korea. From 2023 to 2026, he worked as a Ph.D. Research Assistant and a Teaching Assistant with the Litz Laboratory, University of California, Santa Cruz, USA. He is currently a Researcher with the Software Platform Team, Korea Electronics Technology Institute (KETI). His research interests include accelerator architectures, processing-in-memory, general-purpose graphics processing units (GPGPUs), and heterogeneous systems.

JIHUN LEE received the B.S. degree from the School of Electrical and Electrical Engineering, Konkuk University, Seoul, South Korea, in 2024. He is currently pursuing the Ph.D. degree with the Department of Computer Science and Engineering, Korea University. His research interests include GPU and accelerator architectures.

JONG HYUN JEONG (Graduate Student Member, IEEE) received the B.S. degree from the School of Electronic and Electrical Engineering, Hongik University, in 2021, and the M.S. degree in computer science and engineering from Korea University, in 2023, where he is currently pursuing the Ph.D. degree with the Department of Computer Science and Engineering. His research interests include GPU architecture, memory systems, and heterogeneous systems incorporating processing-in-memory (PIM).

GEONWOO CHOI received the B.S. degree from the School of Electronic and Electrical Engineering, Chung-Ang University, in 2023, and the M.S. degree in computer science and engineering from Korea University, in 2025. He is currently working as a GPU Engineer with Moreh Inc. His research interests include GPU architecture and parallel programming.

MYUNG KUK YOON (Member, IEEE) received the B.S. degrees in computer engineering and computational mathematics from Washington State University (WSU), Pullman, WA, USA, in 2011, and the Ph.D. degree in electrical and electronic engineering from Yonsei University, Seoul, South Korea, in 2018. He is currently working as an Associate Professor with the Division of Artificial Intelligence and Software, Ewha Womans University. Prior to joining Ewha Womans University, he worked as a Software Developer with Samsung Inc. His research interests include GPU micro-architecture, machine learning accelerators, and parallel programming.

YUNHO OH (Senior Member, IEEE) received the B.S., M.S., and Ph.D. degrees from the School of Electrical and Electronic Engineering, Yonsei University, Seoul, South Korea, in 2009, 2011, and 2018, respectively. Currently, he is working as an Associate Professor with the School of Electrical Engineering, Korea University. Prior to joining Korea University, he worked as an Assistant Professor with Sungkyunkwan University. From 2019 to 2021, he worked as a Postdoctoral Researcher with the Parallel Systems Architecture Laboratory (PARSA), EPFL, Switzerland. From 2011 to 2014, he worked as a Software Engineer with Mobile Communications Business, Samsung Electronics. His research interests include hardware and software architectures for energy-efficient datacenters, processor architectures (CPUs, GPUs, and neural network accelerators), in-storage processing, memory systems, and high-performance computing.

GUNJAE KOO (Member, IEEE) received the B.S. and M.S. degrees in electrical and computer engineering from Seoul National University, in 2001 and 2003, respectively, and the Ph.D. degree in electrical engineering from the University of Southern California, in 2018. He is currently an Associate Professor with the Department of Computer Science and Engineering, Korea University. His research interests include computer system architecture and span parallel processor architecture, storage and memory systems, accelerators, and secure processor architecture. Prior to joining Korea University, he was an Assistant Professor with Hongik University. His industry experience includes the position of a Senior Research Engineer with LG Electronics and also a Research Intern with Intel.