IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 1 October 2022, accepted 20 October 2022, date of publication 26 October 2022, date of current version 2 November 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3217222

RESEARCH ARTICLE

Analyzing GCN Aggregation on GPU

INJE KIM[1], JONGHYUN JEONG[1], YUNHO OH[2], (Member, IEEE), MYUNG KUK YOON[3], (Member, IEEE), AND GUNJAE KOO[1], (Member, IEEE)

[1]Department of Computer Science and Engineering, Korea University, Seoul 02841, South Korea
[2]School of Electrical Engineering, Korea University, Seoul 02841, South Korea
[3]Department of Computer Science and Engineering, Ewha Womans University, Seoul 03760, South Korea

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

This work was supported in part by the National Research Foundation of Korea (NRF) grants funded by Korea government (MSIT) (NRF-2021R1C1C1012172, No. 2021R1G1A1092196), in part by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF-2022R1C1C1011021), and in part by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grants [IITP-2022-2020-0-01819 ICT Creative Consilience Program, No. 2022-0-01198 Convergence Security Core Talent Training Business (Korea University)].

ABSTRACT Graph convolutional neural networks (GCNs) are emerging neural networks for graph structures that include large features associated with each vertex. The operations of GCN can be divided into two phases - aggregation and combination. While the combination just performs matrix multiplications using trained weights and aggregated features, the aggregation phase requires graph traversal to collect features from adjacent vertices. Even though neural network applications rely on GPU's massively parallel processing, GCN aggregation kernels exhibit rather low performance since graph processing using compressed graph structures provokes frequent irregular accesses in GPUs. In order to investigate the performance hurdles of GCN aggregation on GPU, we perform an in-depth analysis of the aggregation kernels using real GPU hardware and a cycle-accurate GPU simulator. We first analyze the characteristics of the popular graph datasets used for GCN studies. We reveal the fractions of non-zero elements in feature vectors are diverse among datasets. Based on the observation, we build two types of aggregation kernels that handle uncompressed and compressed feature vectors. Our evaluation exhibits the performance of aggregation can be significantly influenced by kernel design approaches and feature density. We also analyze the individual loads that access the data arrays of the aggregation kernels to specify critical loads. Our analysis reveals the performance of GPU memory hierarchy is influenced by access patterns and feature size of graph datasets. Based on our observations we discuss possible kernel design approaches and architectural ideas that can improve the performance of GCN aggregation.

INDEX TERMS GCN, aggregation kernel, GPU, characteristics.

I. INTRODUCTION

Graph neural networks (GNNs) are emerging neural network models which analyze graph structures that include feature data in each vertex. Since a graph structure is used for representing a data structure where nodes are connected to each other irregularly, many researchers have focused on the applications of GNNs for a wide range of domains such as social network analysis [1], electronic commerce [2], recommendation systems [3], molecular structure analysis [4], and biomedical research [5]. As GNN applications are getting more popular recently, high-performance systems are demanded to process large graph structures. However, the operations in GNN models are not efficient for the existing graphics processing units (GPUs) which are widely deployed for running popular neural network applications. It is because a GNN model includes aggregate operations that traverse adjacent vertices irregularly connected to a target vertex, unlike the popular neural networks that handle regularly allocated data structures. Furthermore, feature data associated with each vertex exhibit a wide range of sparsity levels across graph datasets, thus a single approach to the aggregation

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

operations cannot handle such diverged datasets efficiently. Therefore analyzing performance bottlenecks in GNN operations is critical for designing efficient kernels and processor architectures for GNNs.

In this paper, we fulfill an in-depth analysis of the performance of graph convolutional neural network (GCN) kernels using a GPU hardware performance profiler and a cycle-accurate GPU simulator. GCNs perform convolution operations between trained weights and the aggregated features of a target vertex, thus GCN operations can be split into two phases - aggregation and combination. During the combination phase, GCN just performs simple matrix multiplications of weights and computed feature vectors associated with vertices. GPUs can execute matrix multiplications efficiently thus GCN's combination can be effectively accelerated by GPUs. On the other hand, the aggregation phase requires graph traversal using an adjacency matrix that represents connections between vertices. Since the adjacency matrix of a graph structure is extremely sparse, graph structures are normally compressed to store non-zero elements only. During the aggregation phase, GCN collects the feature vectors associated with adjacent vertices, thus the accesses to the feature data of adjacent vertices create lots of irregular accesses, which cannot be processed efficiently in GPUs. Since a GCN handles large feature data in many vertices, the demand accesses to the feature data in the aggregation phase can lead to significant congestion in the GPU memory subsystem. Furthermore, the large feature data irregularly referenced by the GCN aggregation cause frequent cache thrashing thus GPU cache hierarchy may not exploit inter-vertex locality observed in graph structures. Consequently, in the aspect of processor architecture, the heavy performance burdens by the demand accesses in GCN aggregation kernels can be more critical compared to the general graph processing kernels that handle scalar data associated with vertices and edges.

In order to reveal the performance hurdles of GCN aggregation on GPU, we investigate the detailed characteristics of GCN aggregation kernels using real GPU hardware and a cycle-accurate GPU simulator. Note that the size of the datasets handled by GCN kernels is extremely large since the graph datasets include a massive number of vertices and a large feature vector associated with each vertex. We analyze the statistical properties of popular graph datasets to reveal different datasets exhibit diverse sparsity in feature vectors. Based on our observation, we build two types of aggregation kernels that handle uncompressed and compressed feature data. Our evaluation results exhibit the performance of GCN aggregation can be improved if different kernel design approaches are employed considering the feature density of a graph dataset. We also analyze the behavior of individual loads in the kernels to reveal the significant loads critical to the performance of aggregation. We also explore GPU design space that can influence the performance of GCN aggregation by exploiting the GPU simulator. Based on our observations we discuss several kernel-level and architecture-level approaches that can improve GCN performance on GPU.

We summarize the contributions of this paper as follows.

- In this paper, we analyze the detailed characteristics of GCN aggregation kernels executed on GPUs. For in-depth analysis, we exploit a GPU hardware profiler and a cycle-accurate GPU simulator to reveal the performance hurdles of GCN aggregation in GPU memory hierarchy.
- We analyze the various graph datasets to exhibit the diverged sparsity levels of feature data. We disclose that if the sparsity level of feature data is high the performance of GCN aggregation can benefit from alleviated data movement cost by the compressed feature data despite the irregularity in access patterns.
- We break out the individual loads of the GCN aggregation kernels to highlight the access patterns by different types of loads. By exploiting the cycle-accurate simulator, we collect the detailed performance metrics of the individual loads to reveal the demand accesses critical to the performance of GCN aggregation.
- Based on our observations using the GPU simulator, we explore the GPU design space that can enhance the performance of GCN aggregation. We also discuss several architectural ideas that can accelerate GCN aggregation on GPU.

The remainder of this manuscript is organized as follows. We briefly explain about GCN algorithm and GPU architecture in Section II. In Section III, we analyze the structure of GCN aggregation kernels and categorize individual loads in the kernels based on access patterns. We present the graph datasets and experimental environments in Section IV. The experiment results are exhibited and analyzed in Section V. Based on the evaluation we discuss the possible approaches for GCN kernel design and GPU architecture in Section VI. We present related work in Section VII. We conclude in Section VIII.

II. BACKGROUND

In this section, we briefly explain the basic characteristics of a GCN algorithm. Then we introduce GPU architecture, which is employed for analyzing the performance of GCN kernels.

A. GRAPH CONVOLUTIONAL NETWORKS

GCNs are one of the widely used graph neural network models that can be applied for analyzing graph structures. Graph structures are exploited for describing data elements (represented as *vertices*) related or connected (represented as *edges*) to each other irregularly. Whereas currently popular neural network models such as convolutional neural networks (CNNs) and recurrent neural networks (RNNs) are employed for regularly structured data (i.e. speech, image, and video data) [6], [7], GNNs can handle irregularly structured data represented as graph structures. Thus GNN models can be applied to a wide range of domains that rely on graph structures. In order to focus on specific features of graph structures, researchers have developed various GNN models such as graph convolutional networks (GCNs) [8], graph attention networks (GATs) [9], graph auto-encoders (GAEs) [10], and graph generative networks (GGNs) [11]. Among them, GCNs apply convolution operations to a vertex whose feature data are aggregated with the features of adjacent vertices. Note that the existing neural network models (e.g. CNNs) rely on convolution operations for classification. As such, GCNs can be applied to various applications that rely on graph structures, such as node classification, link prediction, and graph clustering.

GCNs employ *convolution* operations to generate feature data for each vertex using trained weight values. Like CNNs, convolution operations are repeatedly performed for each convolution layer of GCN. A single convolution layer of GCN is composed of two phases - *aggregation* and *combination*. During the aggregation phase, the feature vectors associated with the neighbor vertices are collected for each target node. Then the collected feature vectors are combined with the feature data of the target vertex. GCN variants employ different aggregation functions (e.g. arithmetic mean, normalization, and non-linear functions) in the aggregation phase. In turn, GCN generates an aggregated feature vector associated with each vertex during the aggregation phase. Then GCN performs vector multiplication between the aggregated feature vectors and trained weights during the combination phase to generate output features. This process employs regular vector-matrix multiplications since each vertex contains fixed-sized feature vectors. Then GCN applies non-linear functions to the output feature vectors. Note that CNNs perform similar processes for each convolution layer, namely input features are convoluted with trained weights to generate output features. However, GCNs require traversing adjacent vertices connected irregularly whereas CNNs generate input feature maps using regularly organized data.

The below equation exhibits GCN's single convolution layer operation represented as simple matrix multiplications.

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

In the equation, A represents an adjacency matrix of a graph structure. An element a_{ij} of the adjacency matrix A is an edge weight between vertices i and j. Normally GCN employs a normalized adjacency matrix (represented as $\tilde{A}$) since vertices of a graph structure exhibit various levels of edge connections. A matrix of input features of layer l is represented as $X^{(l)}$, where a row i includes the feature vector of a vertex i. $W^{(l)}$ 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.

GCN's convolution layer operation exhibited in Equation 1 can be split into two phases (*aggregation* and *combination*) as shown in the below formulas [12].

$$\begin{aligned} a_v^l &= \textbf{Aggregate}(h_u^{l-1} : u \in N(v) \cup v) \\ h_v^l &= \textbf{Combine}(a_v^l) \end{aligned} \tag{2}$$

In this equation h^l represents a feature vector of vertex v in a convolution layer l. During the aggregation phase, GCN collects feature vectors from adjacent vertices and then applies *aggregate* function to the collected feature vectors. As mentioned before, GCN variant models employ different aggregate functions. Then GCN performs multiplication between the aggregated feature vector and trained weights. This operation requires general matrix multiplications (GEMM) and non-linear functions. Normally the sparsity of a graph structure is extremely high, thus the *aggregation* and *combination* phases exhibit diverged operational behaviors on general-purpose processors. For instance, the GEMM operations and the parallel non-linear functions demanded by the combination phase fit well to parallel processor architectures such as GPUs. However, many studies present graph processing is inefficient on GPU architecture [13], [14], [15], [16], [17], [18], [19], [20], [21]. Consequently, we focus on the characteristics of GCN's aggregation phase to reveal performance hurdles. In Section III, we will describe the aggregation kernel studied in this manuscript.

B. GPU ARCHITECTURE

Most neural network (NN) applications demand intensive computations using large volumes of feature data, thus such applications rely on GPU's massively parallel processing capability to boost the performance of inferences. Since GNNs handle large graph structures that include a huge number of vertices and edges, the performance of GNN kernels can also benefit from GPU's parallel execution model. As we investigate the detailed performance bottlenecks of GCN aggregation kernels on GPU in this paper, we briefly explain GPU architecture to provide the basic knowledge for understanding GPU's unique execution and performance model. We use NVIDIA's terminology to describe GPU architecture since we study NVIDIA GPU hardware and architectural models to analyze the characteristics of GCN kernels.

To support massive concurrent threads, a GPU equips hundreds of streaming processors (SPs, or called CUDA cores), and these simple cores are organized hierarchically. Namely, a GPU is composed of tens of streaming multiprocessors (SMs) and each SM includes dozens of SPs and other processing units such as load/store units (LSUs), special function units (SFUs), and tensor cores [22]. GPU concurrently runs the multiple threads generated from a single GPU kernel, thus such threads share the same kernel code. GPU assigns multiple threads to SMs in a thread block granularity. Namely, the multiple threads organized by *thread block dimensions* are grouped into a cooperative thread array (CTA), thus a single kernel is split into multiple CTAs. Once a CTA is assigned to an SM, the threads in a CTA are grouped into a warp (or called a wavefront in AMD GPUs), which is a basic execution group in an SM. Namely a single scheduler in an SM (there can be multiple schedulers per SM) can issue one warp every cycle if the warp is ready to be executed. The threads within a warp share the same program counter for the same kernel code, thus a warp executes multiple threads like a single-instruction multiple-data (SIMD) machine if all threads in the warp are activated. If the threads within a warp have diverged execution paths, only part of the threads in the warp is activated thus the performance of the GPU is degraded.

When a warp executes load/store instructions, each thread creates its own memory transaction. For instance, 32 threads in a single warp can create 32 different memory transactions. Unlike an arithmetic/logic execution unit associated with each thread in a warp, memory transactions share datapaths (such as caches and interconnection networks) in the GPU memory subsystem. If these dozens of transactions are serialized in the GPU memory hierarchy, GPU performance is severely degraded due to memory congestion. In order to handle the memory transactions from multiple threads within a warp, GPU merges multiple memory requests that lie in the same cache block space into a single transaction. Using memory coalescing, a warp can generate only one or two memory transactions if threads in the warp access regularly allocated data elements. However, if threads in a warp exhibit irregular memory access patterns, the memory requests generated from the warp are not well-coalesced thus many memory transactions are generated. Prior studies reveal that graph processing applications include indirect memory accesses thus the memory coalescing does not work well for such applications [13], [14], [15]. Hence graph applications cannot achieve high performance on GPUs even though many vertices and edges are processed in parallel. Moreover, the hardware resources in the GPU memory hierarchy are inefficiently occupied and polluted by lots of uncoalesced transactions that fetch large data blocks [23]. In order to mitigate the heavy data traffic by uncoalesced requests, the modern GPU architecture employs sector caches that allow smaller data fetch blocks (32 bytes – 128 bytes) [24]. However, the severe memory congestion by uncoalesced transactions observed in graph processing applications is still one of the critical performance bottlenecks in GPU.

C. GCN vs. GRAPH PROCESSING ON GPU

Now we compare the features of GCN and typical graph processing kernels in the aspect of GPU architecture. Even though both kernels handle graph structures that include many vertices connected irregularly, GCN and graph processing kernels can exhibit disparate characteristics due to the different data types associated with an individual vertex. Note that a typical graph processing kernel such as page rank usually handles graph structures where each vertex includes a single element. On the other hand, GCN kernels handle graph datasets where a single vertex contains a large feature vector that encloses hundreds or thousands of elements. Considering vertices are irregularly connected to each other, the size of data associated with an individual vertex influences the effectiveness of inter-vertex locality in GPU cache hierarchy.

We compare the size of feature data handled by GCN and graph processing kernels in Table 1. The total feature size of a graph dataset processed by GCN kernels is equivalent to a single feature vector size multiplied by the number of total vertices. The feature data size of the GCN datasets is extremely large compared to the total size of the feature data handled by graph processing kernels. It is because GCN kernels handle large feature vectors associated with individual vertices. We also calculate the ratios of GPU L1 and L2 cache sizes to the total feature size of graph datasets in the third and fourth columns of the table. We assume the sizes of L1 and L2 caches are 64 KB and 3 MB respectively as listed in Table 5. For the graph datasets processed by graph processing kernels, the total feature size is smaller than the L2 cache size thus inter-vertex locality can be efficiently exploited in GPU L2 cache. On the other hand, we expect that GCN kernels exhibit severe congestion and cache thrashing in the GPU memory subsystem since GPU caches are too small to hold extremely large feature data. Consequently, GCN kernels can exhibit totally different characteristics in GPU memory hierarchy compared to typical graph processing kernels.

TABLE 1. Comparison between GCN and graph processing.

Dataset		Total feature size	Ratio of L1 cache size	Ratio of L2 cache size
Corafull	GCN	657.6 MB	0.0095%	0.46%
	Graph	77.3 KB	82.8%	> 100.0%
Reddit	GCN	535.0 MB	0.012%	0.56%
	Graph	910.0 KB	7.03%	> 100.0%
Yelp	GCN	820.4 MB	0.0076%	0.37%
	Graph	2,800.2 KB	2.29%	> 100.0%

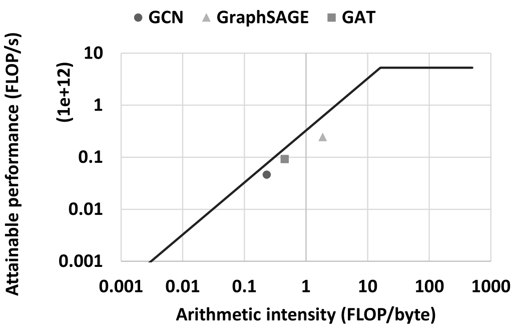


FIGURE 1. Roofline model of NVIDIA RTX 2060.

Another distinctive feature of GCN aggregation on GPU is that GCN allocates multiple threads or warps for a single vertex to exploit GPU's thread-level parallelism for handling hundreds or thousands of feature data elements. Note that typical vertex-centric graph processing kernels usually assign one thread per vertex. Considering the explicit synchronization among threads within a warp, GCN kernels can exhibit different warp-level execution behaviors compared to graph processing kernels. We intensively analyze the design approaches for GCN aggregation kernels in Section III.

D. GCN vs. OTHER GNN AGGREGATION ON GPU

GCN is a good candidate for exploring the performance issues and characteristics of GNN kernels on GPU memory hierarchy. GCN exhibits low arithmetic intensity since GCN aggregation includes frequent demand requests for large feature vectors and relatively simple aggregation functions. Hence, the performance of GCN aggregation can be bounded by limited resources in the GPU memory hierarchy. Figure 1 shows the GPU roofline graph and the arithmetic intensities of GNN aggregation kernels measured using NVIDIA Nsight profiler on NVIDIA RTX 2080. In a roofline model, arithmetic intensity is the ratio of floating-point operations to demanded data movement [25]. The slope of the roofline graph represents the peak performance of GPU is restricted by memory system resources since the arithmetic intensity of an application is low. On the other hand, the peak performance is bounded by GPU's floating-point computation capability if the arithmetic intensity is high (i.e. the horizontal line of the roofline). As shown in the figure the arithmetic intensities of the studied GNN aggregation kernels are low thus we can observe the peak performance of these applications is restricted by GPU memory hierarchy. GCN aggregation exhibits the lowest arithmetic intensity among the tested GNN kernels. That means the performance of GCN aggregation is more influenced by the hardware resources in the GPU memory system compared to other GNN algorithms. Note that GraphSAGE and GAT also rely on similar data demands since those algorithms just employ different aggregate functions [9], [26]. In this paper we investigate the characteristics of GCN aggregation since GCN can exhibit the performance hurdles caused by high data movement cost more clearly on GPU memory hierarchy.

III. ANALYSIS OF AGGREGATION KERNELS

GCN aggregation is a critical part of entire GCN operations on GPU since data accesses during the aggregation phase do not fit well with GPU architecture. In this section, we first describe the GCN kernels studied in this work. Then in order to highlight the *critical* demand accesses, we break out the individual loads in the kernels based on access patterns.

A. KERNEL DESIGN APPROACH

In order to analyze the performance burdens from the aggregation phase, we first study the GCN kernels designed using PyTorch Geometry (PyG) packages. The aggregation step of the PyG GCN kernel consists of three kernels – *IndexSelect*, *ElementWise*, and *ScatterGather*. For each vertex *IndexSelect* collects feature vectors from adjacent vertices to generate a feature matrix. Then *ElementWise* performs multiplications for each element in the feature matrix for normalization. Finally, *ScatterGather* applies an aggregation function to the normalized feature matrix. Note that the graph structures handled by the GCN kernels are extremely sparse thus *IndexSelect* creates many irregular memory accesses for traversing the feature vectors of adjacent vertices. Such operation is inefficient for GPUs since GPU's single-instruction multiple-thread (SIMT) architecture and memory subsystem are suitable for dense linear algebra operations [15]. In order to analyze the performance bottlenecks of the GCN kernels, we need to analyze the instruction-level operations for data transfers. However, the PyTorch framework does not generate the CUDA codes required for detailed architectural analysis. Moreover, since PyG performs kernel-level optimizations only, overall GCN operations require large device memory space thus large graph data sets cannot fit in GPU's device memory.

In order to study the detailed characteristics of GCN's aggregation phase on GPU architecture, we build CUDA-based GCN aggregation kernels that work similarly to the PyTorch-based GCN aggregation operations. Our aggregation kernel handles the graph structures compressed using compressed sparse row (CSR) formats to save GPU's device memory space occupied by large graph structures. We merge the operations of the three PyG kernels (i.e. *IndexSelect*, *ElementWise*, and *ScatterGather*) to minimize the performance overhead by kernel-level synchronizations. The graph traversing of the aggregation kernel is similar to the breadth-first search (BFS) kernel ported to GPU domains [27]. Whereas the BFS kernel assigns the operations of each vertex to a thread execution, our kernel associates the aggregation operations of a single vertex with a cooperative thread array (CTA) execution. Considering each vertex of a GCN dataset includes a large number of feature elements, our design approach can improve the kernel performance since threads within a CTA are explicitly synchronized thus the execution time of a CTA is decided by the high-degree vertices that have many adjacent vertices. Namely, if the BFS-like design approach is applied, CTA execution does not complete until the thread mapped to the high-degree vertex that exhibits the longest execution time completes even though the executions of many other low-degree vertices are already completed. In this case, we cannot exploit the parallelism of GPU efficiently since incomplete CTAs still occupy GPU resources. Our aggregation kernel can achieve higher parallelism because a CTA returns GPU resources immediately when the aggregation for a vertex is complete.

B. AGGREGATION KERNELS BY FEATURE FORMATS

We implement two types of aggregation kernels based on the representations for feature vector elements. Note that GCN kernels process large graph structures that include a long feature vector in each vertex whereas typical graph processing algorithms handle vertices or edges associated with scalar data. GCN kernels demand extremely large data for processing a single vertex, thus such large data may not fit into caches or scratchpad memories in a processor. Moreover, GCN kernels need to exploit parallelism to access multiple elements in a feature vector. Hence, we focus on how to handle large feature vectors for designing GCN aggregation kernels. As shown in Table 4 in Section IV, the feature elements included in each vertex exhibit diverged sparsity levels from 0.39% (Coauthor-phy) to 50.90% (Yelp) for the graph datasets studied in this work. Thus a feature vector in a vertex can be represented as an uncompressed format (*dense* format) and a compressed format (*sparse* format). If the feature data are compressed, we can reduce the memory space

TABLE 2. Variables in the aggregation kernels.

Variable	Description
cta_id	CTA (thread block) ID
cta_dim	Number of threads within a CTA
tid	Thread ID within a CTA
vid	Vertex ID
len	Length of a feature vector in a vertex
iter	Number of iterations for feature aggregation
nnz	Number of non-zero elements in a feature vector
idx	Index for accessing sparse-format feature elements
grp_ptr	Pointer of a CSR-formatted adjacent matrix
grp_idx	Index of a CSR-formatted adjacent matrix
feat_ptr	Pointer of a CSR-formatted feature matrix
feat_idx	Index of a CSR-formatted feature matrix
feat_data	Element of a feature matrix
out	Output matrix of aggregated features

size occupied by large feature vectors, however, it incurs more irregular memory accesses which are inefficient for GPU architecture as mentioned in Section II-B. We exhibit the pseudo-codes of the two types of aggregation kernels in Algorithm 1 and Algorithm 2 respectively. The descriptions for variables used in the kernels are summarized in Table 2.

As mentioned previously each CTA of the aggregation kernel is associated with the target vertex, thus each CTA id (*cta_id*) is mapped to the target vertex id (*vid*). As the aggregation kernel traverses graph structures compressed with the CSR format, the feature vector of an adjacent vertex is indexed by one of the non-zero elements in the target row of the adjacent matrix. Each thread within a CTA accesses a feature vector element (*feat_data*) of an adjacent vertex. Our aggregation kernel exploits GPU's thread-level parallelism (TLP) for computing features since threads in a CTA can access feature vector elements and accumulate output feature elements in parallel. If we consider parallel executions of a single CTA, TLP can be maximized if the number of threads in a CTA is equivalent to the size of a single feature vector. However, it is not a viable option since graph datasets have different feature vector sizes. Furthermore, if the CTA size is too large (i.e. a too large number of threads per CTA), an SM cannot run many CTAs concurrently due to hardware limitations thus the performance of the aggregation kernel can be degraded in this case. In order to figure out the appropriate CTA size, we measure the performance of the aggregation kernels by changing the CTA sizes on NVIDIA RTX 2060. We set the number of threads per CTA (*cta_dim*) as 256 since the aggregation kernels exhibit the best performance on average with this configuration. Consequently, the iteration number of the inner loop (*iter*) is determined considering the number of threads within a CTA and the feature size.

Algorithm 1 represents the algorithmic flow of the aggregation kernel that handles uncompressed feature vectors. Namely, the adjacent matrix of a graph structure is compressed using the CSR format and the feature vector in each vertex is not compressed. Thus once the feature vector of an adjacent vertex is pointed by the indexes of the adjacent vertex, the feature vectors regularly allocated in the *feat_data*

Algorithm 1 Aggregation Kernel Using Dense Feature Format

```
1:  $iter \leftarrow (len - 1)/cta\_dim + 1$
2:  for $i \leftarrow grp\_ptr[vid]$ to $grp\_ptr[vid + 1]$ do
3:      $dst \leftarrow grp\_idx[i]$
4:      for $j \leftarrow 0$ to $iter$ do
5:          $feat\_idx \leftarrow j \times cta\_dim + tid$
6:          if $feat\_idx < len$ then
7:              $out[vid \times len + feat\_idx]$
8:              $+= feat\_data[dst \times len + feat\_idx]$
9:          end if
10:     end for
11: end for
```

array are accessed by each thread. In this paper, we call the aggregation kernel that processes the uncompressed feature data as a *dense-format kernel*. Like the typical BFS kernel, the aggregation kernel gets the ids of the neighbor vertices connected to the target vertex, then the pointers (*dst*) of the feature vectors are read from *grp_idx*. As the dense-format kernel accesses all elements in the feature vector, the threads within the CTA iteratively access the feature vector elements in parallel. As explained, the number of iterations is equal to the roundup of the feature vector size divided by *cta_dim*. The dense-format kernel exhibits regular data access patterns in the inner loop of the kernel.

Algorithm 2 Aggregation Kernel Using Sparse Feature Format

```
1:  for $i \leftarrow grp\_ptr[vid]$ to $grp\_ptr[vid + 1]$ do
2:      $dst \leftarrow grp\_idx[i]$
3:      $nnz \leftarrow feat\_ptr[dst + 1] - feat\_ptr[dst]$
4:      $iter \leftarrow (nnz - 1)/cta\_dim + 1$
5:      for $j \leftarrow 0$ to $iter$ do
6:          $idx \leftarrow feat\_ptr[dst] + cta\_dim \times j + tid$
7:          if $idx < feat\_ptr[dst] + nnz$ then
8:              $out[vid \times len + feat\_idx[idx]]$
9:              $+= feat\_data[idx]$
10:         end if
11:     end for
12: end for
```

GCN aggregation kernels can be designed to handle compressed feature vectors as shown in Algorithm 2. Since some graph datasets exhibit extremely high sparsity in feature data, we can reduce data movement costs on GPUs if such feature data are compressed. In the aggregation kernel that handles the compressed feature data (called *sparse-format kernel* in this paper), the feature vectors are also compressed using CSR format thus only non-zero elements are stored in *feat_data*. The sparse-format kernel also traverses adjacent vertices using *grp_ptr* since the same CSR-formatted adjacent matrix is used for representing the edges. As only non-zero elements of the feature data are used for aggregation, the sparse-format kernel utilizes *feat_ptr* and *feat_idx* like

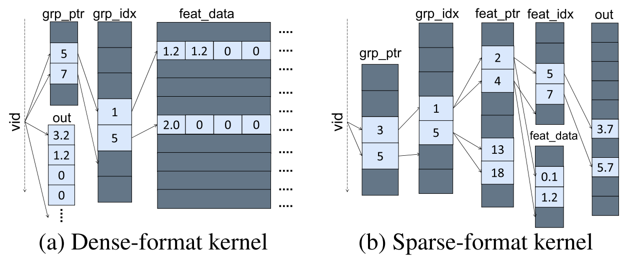


FIGURE 2. Data access patterns of aggregation kernels.

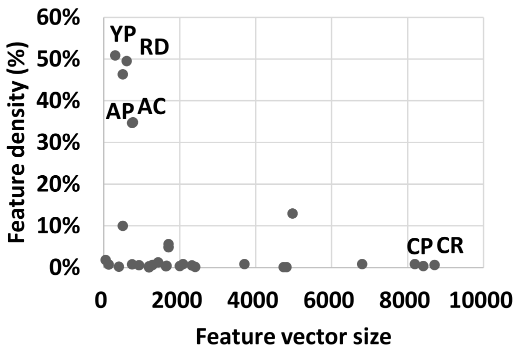


FIGURE 3. Feature density with respect to feature vector size.

TABLE 3. Characteristics of loads.

Load	Access pattern	Iteration	CTA locality
Dense-format kernel			
grp_ptr	REG_CTA	–	–
grp_idx	REG_CTA	*degree*	INTER
feat_data	REG_TH	*degree* × *iter*	INTER
out	REG_TH	*degree* × *iter*	INTRA
Sparse-format kernel			
grp_ptr	REG_CTA	–	–
grp_idx	REG_CTA	*degree*	INTER
feat_ptr	REG_CTA	*degree*	INTER
feat_idx	REG_TH	*degree* × *iter*	INTER
feat_data	REG_TH	*degree* × *iter*	INTER
out	IRR_TH	*degree* × *iter*	INTRA

graph traversal. Hence, the inner loop of the sparse-format kernel incurs indirect memory accesses that cause irregular access patterns. The load warps in the inner loop of the sparse-format kernel can result in severe memory congestion due to uncoalesced memory requests even though the size of feature vectors is decreased.

C. ANALYZING ACCESS PATTERNS OF LOADS

Figure 2 depicts the data access patterns of the aggregation kernels that handle uncompressed (dense-format) and compressed (sparse-format) feature data. Note that data access patterns influence the performance of GPU kernels significantly [15], [28], [29]. Both kernels first access *grp_ptr* using *vid* and then *grp_idx* array is accessed using the data fetched from *grp_ptr*. Since all threads within a CTA share the same *vid*, the accesses to *grp_ptr* and *grp_idx* are well-coalesced. Then the dense-format kernel accesses *feat_data* using the pointer fetched from *grp_idx*. The accesses to *feat_data* are also well-coalesced since the threads of the dense-format kernel access regularly allocated *feat_data*. On the other hand, the threads of the sparse-format kernel access *out* using the *feat_idx* indexed by *idx*, thus the accesses to *out* are irregular.

Table 3 summarizes the characteristics of load instructions that access data arrays of the aggregation kernels. We categorize the characteristics of loads based on access patterns, the number of iterations, and locality. In the *access pattern* column, *REG* and *IRR* represent regular and irregular accesses respectively. For the postfix of the access pattern category, *CTA* means all threads within a CTA generate memory transactions using *cta_id*. On the other hand, the load warps marked with *TH* create memory requests using both *cta_id* and *tid*, thus threads in a warp use different addresses to access a target data array. The *iteration* column represents the iteration counts of loads within a warp. As exhibited in the algorithmic flows of the aggregation kernels, the loads that access *grp_ptr* are executed only once, thus these loads are not critical for the performance of the aggregation kernels. The iteration counts of *grp_idx* and *feat_ptr* are equivalent to the number of adjacent vertices (i.e. *degree*). The loads of *feat_idx*, *feat_data*, and *out* are repeatedly executed within the inner loop of the kernels, thus those loads are executed *degree*×*iter* times. The last column of the table represents CTA-level data locality. Namely, *INTER* means the data accessed by one CTA can be referenced again by other CTAs and *INTRA* represents the data that can be repeatedly referenced inside of a CTA. Vertices that have many connections can be referenced many times, thus the loads that exhibit *INTER* CTA locality can benefit from cache hits in the L2 cache shared across multiple SMs. The loads that access *out* exhibit *INTRA* CTA locality, thus the data fetch latency of such loads can be reduced if the requested data exist in the L1 data cache. Our instruction-level analysis of the loads in the kernels reveals the critical loads that influence the performance of aggregation.

IV. METHODOLOGY

A. GRAPH DATASETS

As described in the previous section, the performance of the GCN aggregation kernels can be influenced by the size of a feature vector per vertex and the fraction of non-zero elements in a feature vector (called *feature density* in this paper). Figure 3 exhibits the feature vector size (X-axis) and the corresponding feature density (Y-axis) of the 32 homogeneous graph datasets used for the prior GCN researches [30], [31], [32], [33], [34], [35], [36], [37], [38], [39], [40], [41]. As shown in the figure, the feature density of a graph dataset is relatively low as the feature vector size is large. The feature density is less than 5% for most graph datasets whose feature vector size is larger than 1000.

Among the tested graph datasets, we select six large graph datasets that exhibit various feature density levels to analyze the performance of the GCN aggregation kernels. Table 4

TABLE 4. Graph datasets.

Dataset	No. of vertices	Graph density	Feature length	Feature density
Corafull (CR)	19793	0.0374%	8710	0.65%
Coauthor-phy (CP)	34493	0.0446%	8415	0.39%
Amazon-photo (AP)	7650	0.4250%	745	34.72%
Amazon-com (AC)	13752	0.2728%	767	34.85%
Reddit (RD)	232965	0.2100%	602	49.55%
Yelp (YP)	716848	0.0028%	300	50.90%

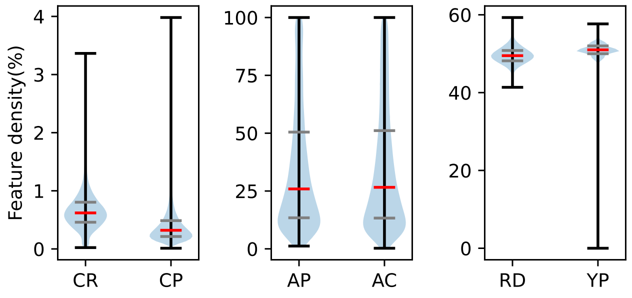


FIGURE 4. Feature density of graph datasets.

summarizes the statistics of the graph datasets studied in this work. All graph structures exhibit extremely high sparsity. The graph density (i.e. the fraction of non-zero elements in an adjacent matrix) of all graph datasets is less than 0.5%. On the other hand, the graph datasets exhibit diverse levels of feature density (0.4% – 50%). Each vertex of Corafull (CR) and Coauthor-phy (CP) datasets includes large sized feature vector, however, less than 1% of elements in the feature vector have non-zero values [32], [35]. The graph size of Amazon-photo (AP) and Amazon-com (AC) is relatively small and the feature density of the graphs is around 35% [35]. Reddit (RD) and Yelp (YP) have very large graph structures where each vertex includes dense feature vectors [26], [31].

In order to visualize the characteristics of feature vectors of each graph dataset, we utilize violin plots as shown in Figure 4. We profile the feature density of each vertex to analyze the distributions of the feature density, and then we apply *scott* estimator to smooth the probability density [42]. In a violin plot, the blue-colored surface represents the distribution of the feature density. The upper/lower black bars represent the maximum/minimum values, and the red bar is a median value. The grey bars of a violin plot represent low/high 25% of a distribution. For CR and CP, the range of feature density is narrow (0% – 4%). These datasets also exhibit very high sparsity in feature vectors. AP and AC exhibit a very wide range of feature densities from 0% to 100%, thus the inner loop of the sparse-format kernel has diverse iteration counts. Overall, the distributions of feature density are similar to asymmetric Gaussian distributions.

B. EXPERIMENT SETUP

In order to analyze the performance metrics of the aggregation kernels, we use both real GPU hardware and an architectural simulator. We design the dense-format and the sparse-format aggregation kernels using CUDA toolkit 10.1. We run the aggregation kernels on NVIDIA RTX 2060 GPU to collect performance counts using NVIDIA Nsight profiler. In order to perform instruction-level analysis, we exploit a cycle-accurate GPU simulator, GPGPU-Sim v4.0 [43]. We also configure GPGPU-Sim using NVIDIA RTX 2060 hardware specifications. The detailed configurations of our experiment setup are listed in Table 5.

TABLE 5. Experiment environments.

GPU	
Model	GeForce RTX 2060 [24]
Core	30 CUDA SMs @ 1.365GHz
Memory	6GB, GDDR6 @ 1.75GHz
Comm.	PCI-E GEN 3.0
Simulator	
Version	GPGPU-Sim v4.0 [44]
Configs	GeForce RTX 2060
Core	30 CUDA SMs @ 1.365GHz
CTAs / SM	Max. 16 CTAs
Warps / SM	Max. 32 warps
Register file / SM	256KB
L1D cache / SM	Sector, 64KB, 128B line, 512-way, 256 MSHRs
L2 cache	Shared, 3MB, 128B line, 16-way, 192 MSHRs
Memory	GDDR6 @ 1.75GHz

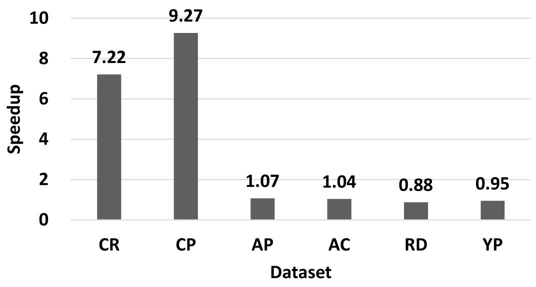


FIGURE 5. Speedup by the sparse-format kernel over the dense-format kernel.

V. EXPERIMENTAL RESULTS

A. PERFORMANCE OF AGGREGATION KERNELS

As described in Section III we design two types of aggregation kernels that handle uncompressed and compressed feature vectors respectively. Note that graph datasets exhibit diverse feature densities whereas the sparsity of graph structures is extremely high (see Table 4). In order to study the impacts on the performance of GCN kernel design approaches, we measure the execution time of each kernel using the real RTX 2060 GPUs. Figure 5 exhibits the speedup by the sparse-format aggregation kernel compared to the dense-format kernel. The experiment results exhibit the sparse-format kernel outperforms the dense-format kernel if the feature density of a graph dataset is very low. For CR and CP, the speedup by the sparse-format kernel is 7.22 and 9.27 respectively. Note that for CR and CP the number of feature elements per vertex is very large but the feature density is extremely low. In this case, the sparse-format kernel can effectively reduce the memory traffic by accessing

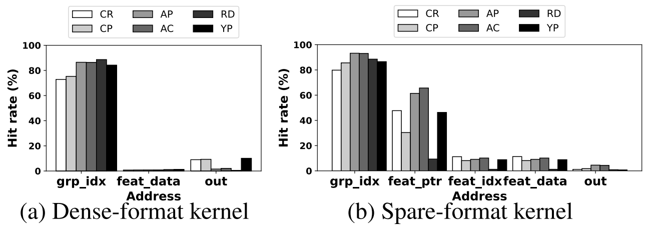


FIGURE 6. **L1 cache hit rates of individual loads.**

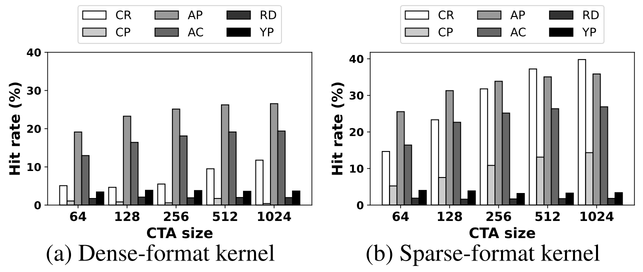


FIGURE 7. **L2 hit rate for *feat_data*.**

feature vectors. The sparse-format kernel is more efficient for AP and AC whose feature density is around 30%, however, the speedup is relatively low. If the feature density is high, the performance of the aggregation kernel is degraded if feature vectors are compressed using the CSR format. Our evaluation results show the aggregation performance drops if the sparse-format kernel is applied to RD and YP. It is because GPU's memory subsystem does not work efficiently for irregular accesses even though the number of memory requests can be decreased by data compression. While the prior GCN kernel researches focus on the high sparsity of graph structures, our experiment results reveal the performance of the GCN aggregation kernel can be influenced by the feature density of graph datasets.

B. CACHE PERFORMANCE

Since GCN kernels handle large graph structures where each vertex includes a long feature vector, the performance of memory hierarchy is critical for the overall performance of GCN applications. Although GPUs employ cache hierarchy to exploit the high data locality observed in GPU applications and datasets, prior studies present that GPU's cache hierarchy is not utilized efficiently if each thread generated from a kernel frequently creates demand requests [23], [44], [45], [46]. In order to investigate the performance bottlenecks of the aggregation kernels in the GPU memory hierarchy, we measure the cache performance of individual load instructions using GPGPU-Sim.

1) L1 CACHE

Figure 6 exhibits the hit rates in GPU L1 cache for individual loads of the dense-format and the sparse-format aggregation kernels. Among the loads in Table 3 we select the loads that access data arrays *repeatedly* since these loads are critical for the performance of the aggregation kernels. In both kernels, the loads that access *grp_idx* exhibit the high L1 cache hit rate since these loads share the same data block fetched by the first warp. Note that *grp_idx* loads are classified into *REG_CTA*. In the sparse-format kernel, the loads that access *feat_ptr* are classified into the same class, thus the L1 cache hit rate by the loads is relatively high. On the other hand, our analysis reveals that the loads repeatedly executed in the inner loop of the kernels exhibit extremely low L1 cache hit rates. It is because these loads fetch a large number of data blocks to access long feature vectors. Especially the loads that access *out* exhibit high temporal locality as described in Section III-C, however, such high locality is not exploited by GPU's L1 cache since the cache lines occupied by *out* are frequently evicted by the memory transactions for large feature data.

2) L2 CACHE

Based on the analysis of the access patterns in Section III-C and the measurement of the L1 cache hit rates for individual loads, we can identify the loads that access *feat_data* and *out* are critical loads that dominate the performance of the aggregation kernels on GPU. Thus we focus on the performance of these loads in the GPU's L2 cache. Note that the L2 cache is shared across multiple SMs thus the feature data of high-degree vertices can be re-referenced by multiple CTAs in the L2 cache. In order to investigate the performance changes by the number of concurrent CTAs in a GPU, we measure the L2 hit rates of loads for *feat_data* and *out* by changing the size of a single CTA. As shown in Table 5, the maximum number of concurrent warps per SM is specified by GPU hardware specifications. Thus the number of concurrent CTAs in a GPU is decreased if the size of a single CTA is increased. For instance, if a single CTA includes 256 threads (i.e. 8 warps), a single SM can run 4 concurrent CTAs. In this case, a single RTX 2060 GPU can run a total of 120 CTAs concurrently since RTX 2060 includes 30 SMs.

Figure 7 shows the L2 cache hit rate for feature inputs (*feat_data*) of the aggregation kernels. Note that the feature data of high-degree vertices are repeatedly referenced by multiple CTAs. For instance, the vertex connected to N other vertices is referenced $N + 1$ times (including its own output feature computation) during an aggregation phase. The requests to the feature vector can be served from the L2 cache rather quickly if the feature data exist in L2.

However, the L2 cache hit rates of CR and CP are extremely low (6.80% and 0.83% respectively) if the dense-format kernel is employed. Note that the feature-length of these data sets is very long thus the feature data of the high-degree vertices are frequently evicted from L2. The L2 hit rates of these data sets increase as the number of concurrent CTAs decrease due to larger CTA sizes. Hence the L2 cache hit rates for CR and CP can be improved if the sparse-format kernel is employed. Our evaluation results reveal that the GCN aggregation can benefit from higher L2

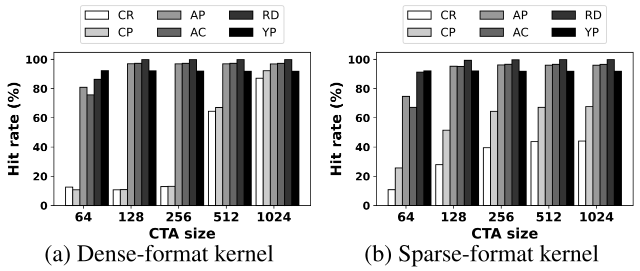


FIGURE 8. L2 hit rate for *out*.

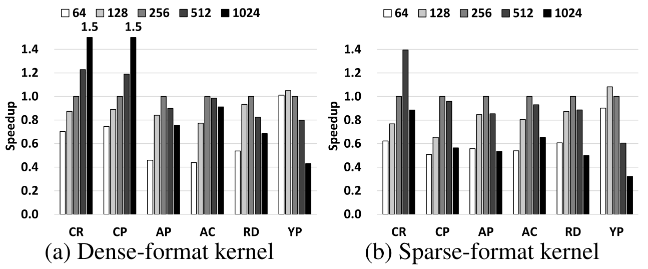


FIGURE 9. Performance by CTA size.

cache hit rates by inter-CTA data locality if the feature data that exhibit high sparsity are compressed. For AP and AC datasets, the L2 cache hit rates are relatively high since the size of these datasets is smaller than other graph datasets thus the feature data can be kept longer in L2. On the other hand, RD and YP datasets include a massive number of vertices, thus the L2 cache is not utilized efficiently due to severe cache thrashing. The GCN aggregation cannot exploit inter-vertex locality as the reuse intervals for a certain vertex is too long for such large graph structures. The sparse-format kernel is not beneficial for these datasets since the compression efficiency is low due to the high feature density and the large graph structure size.

Figure 8 shows the L2 cache hit rates of demand fetch for *out* in the aggregation kernels. As mentioned in Section III-C the accesses to *out* exhibit strong *intra-CTA* locality since the feature output data are repeatedly accumulated by the aggregated feature input data. However, our observations for the L1 cache hit rates disclose such data locality is not exploited in L1. Hence we examine the utilization of the L2 cache for fetching output features. Note that the demand fetches for *feat_data* and *out* are critical loads that dominate the performance of GCN aggregation on GPU. Since the accesses to *out* exhibit *intra-CTA* locality, the reuse intervals for *out* are shorter compared to *feat_data*. Thus the L2 cache may be utilized more efficiently for access to the output features.

As shown in Figure 8 the L2 cache hit rate for the *out* loads is relatively higher compared to the demand fetches for *feat_data*. However, the L2 cache hit rate for CR and CP datasets is very low when the CTA size is small. For these datasets, the L2 hit rate increases dramatically as the CTA size increases. Note that the number of concurrent CTAs decreases if the aggregation kernel includes more threads per CTA. Therefore for CR and CP, the data blocks of *out* are interfered with by large input feature data in the L2 cache. When the sparse-format kernel is employed, the L2 cache hit rates for these datasets are improved compared to the dense-format kernel. Such results reveal the utilization of the L2 cache for the output feature data is highly influenced by the size of input feature data during the aggregation phase. This statement is also supported by the observations that the L2 cache hit rate of *out* is relatively high for other graph datasets that have smaller features per vertex.

C. PERFORMANCE BY CONCURRENT CTA COUNTS

Based on the kernel analysis in Section III-C, if CTA size is decreased the execution time of individual warps increases since the iteration counts of the inner loop get larger. On the other hand, the aggregation kernels may exploit inter-vertex locality since more vertices are concurrently processed by the aggregation kernels. Hence, the aggregation kernels benefit from a stronger inter-CTA locality resulting from higher CTA concurrency if the warp execution time is proportional to the number of concurrent CTAs. However, as presented in Section V-B the GPU cache hierarchy suffers from severe congestion by larger feature data of more vertices if the number of concurrent CTAs increases.

As shown in Figure 9 we measure the performance of both aggregation kernels for each dataset by changing the CTA size. The performance of each dataset is normalized to the performance by the baseline CTA size (i.e. No. of threads per CTA = 256). For CR and CP datasets, the performance of the dense-format kernel is improved when the CTA size is increased. Note that each vertex of CR and CP has a large feature vector thus GPU's L2 cache exhibits lower utilization due to cache thrashing when more CTAs from the dense-format kernel run concurrently (see Section V-B). For other datasets, we can observe the optimal CTA size (around 256 threads per CTA) that can exhibit better performance. For the sparse-format kernel, we can also find the optimal number of concurrent CTAs that can achieve higher performance. Since CR and CP datasets exhibit relatively high L2 cache hit rates when the sparse-format kernel is employed, the optimal CTA sizes for CR and CP are 512 and 256 respectively. Consequently, our observations reveal both inter-vertex locality and congestion in cache hierarchy should be considered to achieve higher performance using the aggregation kernels. We present the detailed analysis results using the sparse-format kernels.

Figure 10 shows the fraction of the issued instructions in each warp of a single CTA from the sparse-format kernel. We set the size of a CTA as 256. Since the sparse-format kernel handles only non-zero elements in a feature vector, each warp exhibits different iteration counts for the inner loop of the kernel. Note that the warp of the longest execution time decides the overall execution time of a CTA since the warp executions are explicitly synchronized within a CTA. Our evaluation results present the executed instruction counts are unbalanced within a CTA due to the mechanism of the

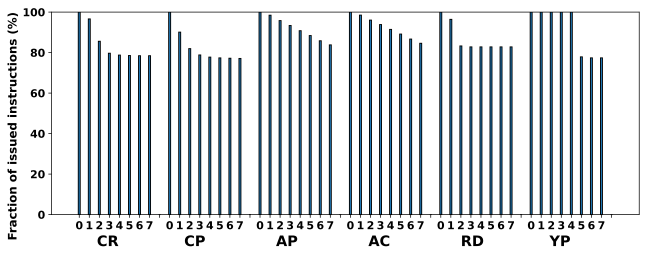


FIGURE 10. **Fraction of issued instructions in the sparse-format kernel.**

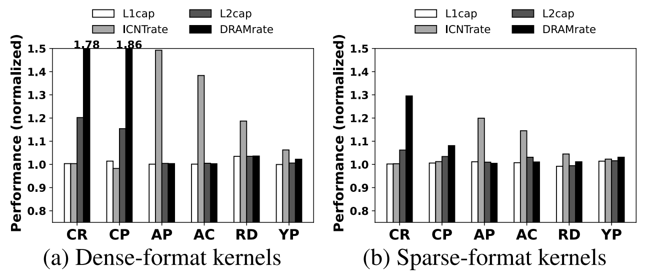


FIGURE 11. **Performance changes by memory subsystem resources.**

sparse-format kernel. Normally the first two or three warps execute more instructions. Hence we can maximize the parallelism and the inter-vertex locality of the sparse-format kernel if we set the CTA size as 64. However, our performance analysis exhibits the optimal CTA size is 256 for the sparse-format kernel. Such results explain the utilization in both caches is one of the critical factors for the performance of the aggregation kernels on GPU.

D. DESIGN SPACE EXPLORATION

Our analysis of the GCN aggregation kernels on GPU reveals that the feature density of a graph dataset influences the performance of kernels significantly thus appropriate feature representations are required when designing GCN aggregation kernels. Furthermore, the performance of GPU is restricted by heavy demand requests to feature data of large graph structures since GPU's memory subsystem resources are not utilized efficiently. In this section, we explore the design approach of GPU architecture to investigate the possible performance uplifts of GCN aggregation kernels by configuration changes in GPU architectures. By exploring possible hardware design space, we can disclose insufficient hardware resources that limit the performance of the GCN aggregation in the current GPU configurations. Additionally, we can reveal hardware components that need architectural solutions to improve the performance of GCN kernels. We explore the design space of GPU for GCN aggregation kernels by modifying the parameters of GPGPU-Sim configures like NVIDIA RTX 2060 as shown in Table 5. The CTA size of the aggregation kernels is set as 256 threads per CTA.

Figure 11 exhibits the performance changed by parameters in GPU memory subsystems. As the performance of the aggregation kernels is restricted by insufficient hardware resources in the GPU memory subsystem, we measure the aggregation performance by increasing the resources in the memory hierarchy. The performance by the parameter changes exhibited in the figure is normalized to the performance of the baseline machine (i.e. RTX 2060 configurations).

1) L1 DATA CACHE CAPACITY

As mentioned in Section V-B GPU's L1 data cache is not utilized well for the aggregation kernels. Especially the critical loads (*feat_data* and *out*) suffer from extremely low hit rates in L1. In order to investigate the performance benefits of a larger L1 data cache, we measure the performance of the aggregation kernels using GPGPU-Sim by doubling the size of the L1 data cache. The performance by the larger L1 data cache is labeled as *L1cap*. As shown in the results, even if the L1 cache size is doubled the performance is not improved for both dense-format and sparse-format kernels since the cache hit rates are still extremely low. The experimental results present increasing L1 cache space simply cannot resolve the interference between input and output feature data.

2) INTERCONNECTION NETWORK BANDWIDTH

As many demand requests cannot be serviced from GPU's L1 data cache, the missed demand transactions are delivered to the L2 cache via an interconnection network. If lots of missed demand requests are hit in the L2 cache and the demanded data are serviced from L2 rather quickly, the resources of the interconnection network may be consumed by heavy traffic from SMs and L2 partitions. Note that data packets encounter severe queuing delays if many previous packets are waiting in the interconnection network queues. Our simulation results show that the performance of the aggregation kernels is improved for AP, AC, RD, and YP datasets by 11.83% on average (labeled as *ICNTrate*) when the bandwidth limit of the interconnection is doubled. Note that for these datasets the aggregation kernels exhibit low L1 hit rates and high L2 hit rates thus the congestion in the interconnection network increases queuing delays.

3) L2 CACHE CAPACITY

Our experimental results exhibit that a large amount of the demanded input and output feature data are serviced from the L2 cache, however, CR and CP datasets exhibit low L2 hit rates since the feature vector size of these datasets is very large. We can expect the aggregation performance for these datasets can benefit from a large L2 cache (labeled as *L2cap*). As shown in Figure 11 the performance of the aggregation kernels is improved significantly for both dense-format and sparse-format kernels when the L2 cache size is doubled. For CR and CP datasets, the performance of the dense-format kernel is improved by 6.48% when 6 MB of the L2 cache is employed. For the sparse-format kernel, the performance uplift is rather lower (average 2.43%) with the large L2 cache compared to the dense-format kernel. It is because the capacity misses in the L2 cache are already alleviated by the compressed feature data.

4) DRAM BANDWIDTH

DRAM bandwidth is critical for GPU's performance especially when the data demanded by many concurrent threads are streamed from GPU DRAM. Namely if the data cache hierarchy (i.e. L1 data cache and L2 cache) of GPU is not utilized efficiently, the performance of GPU kernels is significantly influenced by the bandwidth of GPU device memory. Our evaluation results exhibit the aggregation kernels cannot benefit from GPU caches if the features of a graph dataset are large. Hence the performance of the aggregation kernels that handle such graph datasets can be significantly influenced by DRAM bandwidth. In Figure 11 the bar graphs labeled as *DRAMrate* represented the normalized performance of the aggregation kernels when GPGPU-Sim is configured with 2× DRAM data rate. As expected the performance of the dense-format kernel is significantly improved for CR and CP datasets by 78.25% and 86.40% respectively. It is because the dense-format kernel exhibits very low L2 hit rates when CR and CP datasets are processed as shown in Figures 7 and 8.

VI. DISCUSSION

In this section, we discuss possible approaches for improving the performance of GCN aggregation kernels on GPU based on our experiment results.

A. KERNEL DESIGN

Based on our analysis of graph datasets and evaluation of the two types of aggregation kernels, we disclose the performance of GCN aggregation can be significantly influenced by how to handle sparse feature data while the prior researches focus on only the sparsity of graph structures. we observe feature vectors embedded in vertices exhibit diverse feature densities for different graph datasets. Note that GCN handles large graph structures where each vertex also includes large feature vectors thus sparse feature vectors include a very low fraction of non-zero data. Our evaluation reveals irregular access to compressed feature data can be more effective even though GPU architecture is not efficient to handle irregular data accesses. Hence in order to improve the overall performance of GCN kernels on GPU, the sparsity of feature data should be also considered for aggregation kernel design.

B. CONSIDERATIONS OF ACCESS PATTERNS

Our analysis of the aggregation kernels reveals the loads exhibit diverse data locality patterns, however, the locality of the loaded data is not utilized efficiently in the GPU cache hierarchy. Even though such data cannot be reused in the cache, lots of transactions from the loads occupy cache resources such as cache request queues, cache blocks, and miss status handling registers (MSHRs). Hence diverged handling approaches can be applied to loads that exhibit different access patterns in order to utilize cache resources more efficiently. For instance, for the loads that exhibit extremely low utilization in the L1 cache, cache bypassing can be beneficial to improve the performance of the aggregation kernels. In order to utilize the data locality more efficiently, programmers can exploit the shared memory in an SM to prevent the evictions of frequently reused data. Our evaluation reveals the output feature data are re-referenced within a CTA execution, however, the cache blocks allocated to the output feature data are evicted from caches. Thus programmers can explicitly manage the output feature data in the shared memory to prevent frequent cache evictions.

C. WARP SCHEDULING

As shown in Figure 10, warps within a CTA include different instruction counts for the sparse-format aggregation kernel. Such imbalance among warps can decrease the performance of the aggregation kernels since the warp that exhibits the longest execution time in a CTA decides the execution time of the corresponding CTA. Hence the performance of the aggregation kernel can be improved if a warp scheduler set higher priority for the warps that handle more non-zero feature elements.

VII. RELATED WORK

A. CHARACTERISTICS OF GCN KERNELS

Yan et al. analyzed the execution patterns of GCN workloads on real GPU using hardware performance profiler [47]. The authors compared the performance of GCN kernels with MLP-based neural network kernels and typical graph processing applications. They revealed the aggregation and combination phases of GCN workloads have diverse characteristics and the aggregation step is critical for the performance of GCN 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. In this paper, we perform a detailed analysis of the GCN aggregation kernels to reveal the performance bottlenecks on GPU. We utilize both real GPU hardware and a cycle-accurate GPU simulator to reveal the detailed characteristics of data loads.

Baruah et al. presented a benchmark suite, called GNNMark, for characterizing GNN training workloads on GPUs [48]. The authors analyzed the various aspects of GNN training kernels using a GPU performance profiler to reveal system-level performance bottlenecks. We focus on analyzing the characteristics of GCN aggregation kernels on GPU in this work. We utilize the two types of aggregation kernels that work similarly to the GCN kernels programmed using the popular neural network framework.

B. GRAPH PROCESSING ON GPU

Che et al. presented a collection of graph applications, called Pannotia, to evaluate graph processing on SIMD architecture [14]. To run graph applications on GPUs, the authors rewrote various graph processing kernels using OpenCL. They evaluated the ported graph applications using an AMD

GPU and a hardware profiling tool. Xu et al. revealed the detailed characteristics of graph applications on GPU using a cycle-accurate GPU simulator [13]. They collected performance metrics by running graph processing kernels using the simulator to suggest several hardware-level optimizations that can improve the performance of graph processing on GPUs. Those researches present the distinctive characteristics of graph processing kernels focusing on GPU's microarchitecture and execution models. Koo et al. categorized load instructions observed in GPU kernels of various application domains [15]. The authors revealed the loads that exhibit non-deterministic access patterns are critical for the performance of GPU applications. They revealed the detailed characteristics of these critical loads in GPU architecture.

In this work, we investigate the characteristics of GCN aggregation kernels that handle large graph structures and feature data. We focus on the heavy performance burdens by GCN aggregation on GPU memory hierarchy to reveal the criticality of the loads that access the large feature data.

C. GNN OPTIMIZATION ON GPU

Huang et al. proposed a general-purpose sparse matrix multiplication kernel, called GE-SpMM [49]. The authors designed the optimized kernel that exploits GPU's shared memory and register file to perform multiplication between CSR-formatted and general matrices. The authors exhibited that GNN performance can be improved when GE-SpMM replaces the existing SpMM kernels. In this paper, we build general GCN aggregation kernels to disclose performance burdens on GPU. We also focus on the sparsity in feature vectors to reveal compressed feature data can alleviate congestion in the GPU memory hierarchy.

Tian et al. proposed PCGCN that partitions a large graph dataset into smaller subgraphs to exploit possible intervertex locality [50]. PCGCN generates subgraphs considering the connectivity among vertices to maximize data locality. We perform fine-grained analysis on individual loads in GCN aggregation kernels to reveal inefficient use of the GPU memory subsystem. Based on the evaluation results, we discuss several architectural approaches.

D. GCN ACCELERATION

Several researchers have presented accelerator architectures that can execute GCN operations more efficiently compared to general-purpose processors. The researchers have focused on the specific characteristics of graph structures to tackle the performance hurdles of GCNs. Namely diverged execution time of multiple vertices in an aggregation phase is one of the critical performance bottlenecks since GCN accelerators rely on parallel executions of multiple vertices and synchronizations between aggregation and combination phases. Yan et al. presented HyGCN that can optimize the operations of the aggregation phase by exploiting intra-vertex parallelism [12]. Geng et al. proposed AWB-GCN which employs a hardware-based auto-tuning mechanism to tackle the load imbalances in the aggregation phase [51]. Li et al. proposed GCNAX that applies loop reordering and loop fusion approach to tackle performance bottlenecks in DRAM accesses in GCN accelerator architecture [52].

VIII. CONCLUSION

A graph convolutional neural network is one of the popular graph neural networks that can analyze non-Euclidean graph datasets using neural network models. Even though GCN kernels generated from popular deep learning frameworks rely on GPU's massive thread-level parallelism, many prior studies reveal GPU architecture is not efficient for processing sparse graph structures. Since the aggregation phase of GCN performs graph traversal to aggregate feature data in adjacent vertices, the performance of the aggregation phase is critical for the overall GCN performance on GPUs. In order to reveal the performance hurdles of the aggregation phase of GCN, we investigate the detailed characteristics of GCN aggregation kernels using real GPU hardware and a cycle-accurate GPU simulator. Based on the analysis of graph datasets handled by GCN kernels, we build two different types of GCN aggregation kernels that handle uncompressed and compressed feature data. Our experimental results reveal the sparse-format aggregation kernel that processes sparse feature data can be more efficient despite generating more irregular accesses since the sparse-format kernel can effectively reduce data traffic in GPU. We also analyze the access patterns of individual loads in the aggregation kernels to investigate the utilization of cache hierarchy. Based on our evaluation results, we propose several kernel design approaches and architectural ideas that can improve the performance of GCN aggregation on GPUs.

REFERENCES

[1] J. Yu, H. Yin, J. Li, M. Gao, Z. Huang, and L. Cui, "Enhancing social recommendation with adversarial graph convolutional networks," *IEEE Trans. Knowl. Data Eng.*, vol. 34, no. 8, pp. 3727–3739, Aug. 2022.
[2] 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, Nov. 2019, pp. 529–538, doi: 10.1145/3357384.3357924.
[3] 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.
[4] S. Ryu, J. Lim, S. Hwan Hong, and W. Youn Kim, "Deeply learning molecular structure-property relationships using attention- and gate-augmented graph convolutional network," 2018, *arXiv:1805.10988*.
[5] 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.
[6] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," *Commun. ACM*, vol. 60, no. 6, pp. 84–90, May 2017, doi: 10.1145/3065386.
[7] T. Mikolov, M. Karafiát, L. Burget, J. Cernocký, and S. Khudanpur, "Recurrent neural network based language model," in *Proc. 11th Annu. Conf. Int. Speech Commun. Assoc. (INTERSPEECH)*, vol. 2, 2010, pp. 1045–1048.
[8] J. Chen, T. Ma, and C. Xiao, "FastGCN: Fast learning with graph convolutional networks via importance sampling," 2018, *arXiv:1801.10247*.
[9] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," 2017, *arXiv:1710.10903*.

[10] T. N. Kipf and M. Welling, ''Variational graph auto-encoders,'' 2016, *arXiv:1611.07308*.

[11] X. Guo and L. Zhao, ''A systematic survey on deep generative models for graph generation,'' 2020, *arXiv:2007.06686*.

[12] M. Yan, L. Deng, X. Hu, L. Liang, Y. Feng, X. Ye, Z. Zhang, D. Fan, and Y. Xie, ''HyGCN: A GCN accelerator with hybrid architecture,'' in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2020, pp. 15–29.

[13] 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.

[14] 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.

[15] 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.

[16] X. Shi, Z. Zheng, Y. Zhou, H. Jin, L. He, B. Liu, and Q.-S. Hua, ''Graph processing on GPUs: A survey,'' *ACM Comput. Surv.*, vol. 50, no. 6, pp. 1–35, Nov. 2018, doi: 10.1145/3128571.

[17] Y. Lü, H. Guo, L. Huang, Q. Yu, L. Shen, N. Xiao, and Z. Wang, ''GraphPEG: Accelerating graph processing on GPUs,'' *ACM Trans. Archit. Code Optim.*, vol. 18, no. 3, pp. 1–24, Sep. 2021, doi: 10.1145/3450440.

[18] J. Zhong and B. He, ''Medusa: Simplified graph processing on GPUs,'' *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 6, pp. 1543–1552, Jun. 2014.

[19] L. Ma, Z. Yang, H. Chen, J. Xue, and Y. Dai, ''Garaph: Efficient GPU-accelerated graph processing on a single machine with balanced replication,'' in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, Santa Clara, CA, USA, Jul. 2017, pp. 195–207. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/ma

[20] F. Khorasani, R. Gupta, and L. N. Bhuyan, ''Scalable SIMD-efficient graph processing on GPUs,'' in *Proc. Int. Conf. Parallel Archit. Compilation (PACT)*, Oct. 2015, pp. 39–50.

[21] Y. Guo, M. Biczak, A. L. Varbanescu, A. Iosup, C. Martella, and T. L. Willke, ''How well do graph-processing platforms perform? An empirical performance evaluation and analysis,'' in *Proc. IEEE 28th Int. Parallel Distrib. Process. Symp.*, May 2014, pp. 395–404.

[22] NVIDIA. *Nvidia Tesla V100 GPU Architecture, The World's Most Advanced Data Center GPU*. NVIDIACorporation. Accessed: Jul. 8, 2022. [Online]. Available: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[23] G. Koo, Y. Oh, W. W. Ro, and M. Annavaram, ''Access pattern-aware cache management for improving data utilization in GPU,'' in *Proc. 44th Annu. Int. Symp. Comput. Archit.*, Jun. 2017, pp. 307–319.

[24] NVIDIA. *Nvidia Turing GPU Architecture, Graphics Reinvented*. NVIDIACorporation. Accessed: Jul. 8, 2022. [Online]. Available: https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf

[25] S. Williams, A. Waterman, and D. Patterson, ''Roofline: An insightful visual performance model for multicore architectures,'' *Commun. ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[26] W. L. Hamilton, R. Ying, and J. Leskovec, ''Inductive representation learning on large graphs,'' in *Proc. 31st Int. Conf. Neural Inf. Process. Syst.*, Red Hook, NY, USA, 2017, pp. 1025–1035.

[27] P. Harish and P. J. Narayanan, ''Accelerating large graph algorithms on the GPU using CUDA,'' in *High Performance Computing–(HiPC)*, S. Aluru, M. Parashar, R. Badrinath, and V. K. Prasanna, Eds. Berlin, Germany: Springer, 2007, pp. 197–208.

[28] 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.

[29] M. A. O'Neil and M. Burtscher, ''Microarchitectural performance characterization of irregular GPU kernels,'' in *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*, Oct. 2014, pp. 130–139.

[30] Z. Yang, W. W. Cohen, and R. Salakhutdinov, ''Revisiting semi-supervised learning with graph embeddings,'' in *Proc. 33rd Int. Conf. Mach. Learn.*, vol. 48, 2016, pp. 40–48.

[31] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, and V. Prasanna, ''GraphSAINT: Graph sampling based inductive learning method,'' 2019, *arXiv:1907.04931*.

[32] A. Bojchevski and S. Günnemann, ''Deep Gaussian embedding of graphs: Unsupervised inductive learning via ranking,'' in *Proc. Int. Conf. Learn. Represent.*, 2018.

[33] W. L. Hamilton, R. Ying, and J. Leskovec, ''Inductive representation learning on large graphs,'' 2017, *arXiv:1706.02216*.

[34] O. Shchur, M. Mumme, A. Bojchevski, and S. Günnemann, ''Pitfalls of graph neural network evaluation,'' in *Proc. Relational Represent. Learn. Workshop (NeurIPS)*, 2018, pp. 1–11.

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

[36] R. Yang, J. Shi, X. Xiao, Y. Yang, J. Liu, and S. S. Bhowmick, ''Scaling attributed network embedding to massive graphs,'' *Proc. VLDB Endowment*, vol. 14, no. 1, pp. 37–49, Sep. 2020, doi: 10.14778/3421424.3421430.

[37] D. Lim, F. Hohne, X. Li, S. Linda Huang, V. Gupta, O. Bhalerao, and S.-N. Lim, ''Large scale learning on non-homophilous graphs: New benchmarks and strong simple methods,'' 2021, *arXiv:2110.14446*.

[38] L. F. R. Ribeiro, P. H. P. Saverese, and D. R. Figueiredo, ''Struc2vec: Learning node representations from structural identity,'' in *Proc. 23rd ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, New York, NY, USA, Aug. 2017, pp. 385–394, doi: 10.1145/3097983.3098061.

[39] H. Pei, B. Wei, K. C.-C. Chang, Y. Lei, and B. Yang, ''Geom-GCN: Geometric graph convolutional networks,'' in *Proc. Int. Conf. Learn. Represent.*, 2020, pp. 1–12. [Online]. Available: https://openreview.net/forum?id=S1e2agrFvS

[40] B. Rozemberczki, C. Allen, and R. Sarkar, ''Multi-scale attributed node embedding,'' 2019, *arXiv:1909.13021*.

[41] W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, and J. Leskovec, ''Open graph benchmark: Datasets for machine learning on graphs,'' in *Proc. 34th Conf. Neural Inf. Process. Syst.*, vol. 33, 2020, pp. 22118–22133.

[42] D. W. Scott, ''On optimal and data-based histograms,'' *Biometrika*, vol. 66, no. 3, pp. 605–610, 1979, doi: 10.1093/biomet/66.3.605.

[43] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, ''Accel-sim: An extensible simulation framework for validated GPU modeling,'' in *Proc. ACM/IEEE 47th Annu. Int. Symp. Comput. Archit. (ISCA)*, May 2020, pp. 473–486.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] T. Baruah, K. Shivdikar, S. Dong, Y. Sun, S. A. Mojumder, K. Jung, J. L. Abellan, 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.

[49] 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. SC20: Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2020, pp. 1–12.

[50] 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.

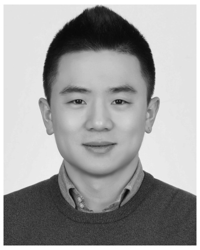
[51] T. Geng, A. Li, R. Shi, C. Wu, T. Wang, Y. Li, P. Haghi, A. Tumeo, S. Che, S. Reinhardt, and M. C. Herbordt, ''AWB-GCN: A graph convolutional network accelerator with runtime workload rebalancing,'' in *Proc. 53rd Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2020, pp. 922–936.

[52] J. Li, A. Louri, A. Karanth, and R. Bunescu, ''GCNAX: A flexible and energy-efficient accelerator for graph convolutional neural networks,'' in *Proc. IEEE Int. Symp. High-Perform. Comput. Archit. (HPCA)*, Feb. 2021, pp. 775–788.

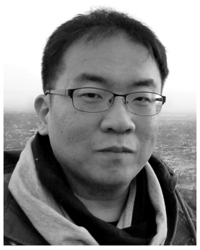
INJE KIM received the B.S. degree from the School of Electronic and Electrical Engineering, Hongik University, in 2021. He is currently pursuing the master's degree with the Department of Computer Science and Engineering, Korea University. Prior to entering Korea University, he worked as an Undergraduate Researcher for two years at Hongik University. His research interests include accelerator architectures, general-purpose graphics processing units (GPGPUs), and the systems for graph neural networks (GNNs).

JONGHYUN JEONG received the B.S. degree from the School of Electronic and Electrical Engineering, Hongik University, in 2021. He is currently pursuing the master's degree with the Department of Computer Science and Engineering, Korea University. His research interests include parallel processor architecture, memory systems, and memory controllers.

YUNHO OH (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. 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 in mobile communications business at Samsung Electronics. Currently, he is working as an Assistant Professor with the School of Electrical Engineering, Korea University. Prior to joining Korea University, he worked as an Assistant Professor at Sungkyunkwan University. 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.



MYUNG KUK YOON (Member, IEEE) received the B.S. degree 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 Assistant Professor with the Department of Computer Science and Engineering, Ewha Womans University. Prior to joining Ewha Womans University, he worked as a Software Developer at Samsung Inc. His research interests include GPU micro-architecture, machine learning accelerators, and parallel programming.



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 Assistant 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.