An Efficient GCNs Accelerator Using 3D-Stacked Processing-in-Memory Architectures

Runze Wang, Ao Hu, Long Zheng, *Member, IEEE*, Qinggang Wang, Jingrui Yuan, Haifeng Liu, Linchen Yu, Xiaofei Liao, *Member, IEEE*, and Hai Jin, *Fellow, IEEE*

***Abstract*—Graph convolutional networks (GCNs) hold great promise in facilitating machine learning on graph-structured data. However, the sparsity of graphs often results in a significant number of irregular memory accesses, leading to inefficient data movement for existing GCNs accelerators. With the advancement of 3D-stacked technology, the processing-in-memory (PIM) architecture has emerged as a promising solution for graph processing. Nevertheless, existing PIM accelerators are confronted with the challenges of irregular remote access in the aggregation phase of GCNs and dynamic workload variations between phases. In this article, we present GCNim, a PIM accelerator based on 3D-stacked memory, which features two key innovations in terms of the computation model and hardware designs. First, we present a PIM-based hybrid computation model, which employs a remote merging strategy to achieve the outer product in aggregation and the row-wise product in combination. Second, GCNim builds a three-stage aggregation and combination pipeline and integrates unified processing elements (PEs) supporting these three stages at the bank level, achieving load balance among PEs through a lightweight data placement algorithm. Compared with the state-of-the-art software frameworks running on CPUs and GPUs, GCNim achieves an average speedup of 3,736.06× and 76.56×, respectively. Moreover, GCNim outperforms the state-of-the-art GCN hardware accelerators, I-GCN, PEDAL, FlowGNN, and GCIM, with average speedups of 3.35×, 8.97×, 2.24×, and 5.58×, respectively.**

***Index Terms*—3D-stacked memory, accelerators, graph convolutional networks (GCNs), processing-in-memory (PIM).**

Manuscript received 31 May 2023; revised 7 October 2023; accepted 15 November 2023. Date of publication 12 December 2023; date of current version 23 April 2024. This work was supported in part by the National Key Research and Development Program of China under Grant 2023YFB4503400; in part by the National Natural Science Foundation of China under Grant 62322205, Grant 62072195, and Grant 61825202; and in part by Zhejiang Laboratory under Grant 2022P10AC02. This article was recommended by Associate Editor V. Pavlidis. *(Corresponding author: Long Zheng.)*

Runze Wang, Ao Hu, Long Zheng, Qinggang Wang, Jingrui Yuan, and Haifeng Liu are with the National Engineering Research Center for Big Data Technology and System, the Service Computing Technology and System Lab, the Cluster and Grid Computing Lab, and the School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China, and also with the Graph Processing Research Center, Zhejiang Laboratory, Hangzhou 311121, China (e-mail: rzwang@hust.edu.cn; ahu@hust.edu.cn; longzh@hust.edu.cn; qgwang@hust.edu.cn; jryuan@hust.edu.cn; hfliu@hust.edu.cn).

Linchen Yu is with the School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan 430074, China (e-mail: linchenyu@hust.edu.cn).

Xiaofei Liao and Hai Jin are with the National Engineering Research Center for Big Data Technology and System, the Service Computing Technology and System Lab, the Cluster and Grid Computing Lab, and the School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China (e-mail: xfliao@hust.edu.cn; hjin@hust.edu.cn).

Digital Object Identifier 10.1109/TCAD.2023.3341753

I. Introduction

BENEFITING from deep neural networks, machine learning has shown remarkable achievements in various domains, including computer vision [1], [2] and natural language processing [3]. However, they are restricted to representing and analyzing Euclidean data, including images, text, and audio [4], [5]. Relational data like social networks [6] and knowledge graphs [7] arising in many applications is also ubiquitous, which is naturally represented by graphs. Hence, graph convolution networks (GCNs) have emerged as an effective model for extracting and analyzing valuable information from relational data. GCNs have exhibited superior performance in a wide range of applications, such as node classification [8], [9], [10], link prediction [11], [12], and graph recommendation [13].

The primary strength of GCNs lies in the two key phases of the convolution layer: 1) *aggregation* and 2) *combination*, which jointly dominate the GCN inference time. Each vertex gathers feature vectors from its neighboring vertices during the aggregation phase, which operates on the graph structure. The combination phase resembles traditional neural networks [14], as it involves performing computation operations on features of vertices using a multilayer perceptron (MLP). This process is often represented by a matrix–vector multiplication (MVM) [5].

In response to the ever-increasing demands for enhanced GCNs inference performance, several dedicated GCNs accelerators have emerged in recent years. These accelerators generally adhere to one of two design philosophies. The first follows a divide-and-conquer design philosophy that utilizes two distinct engines to enhance the efficiency of each phase individually. An example of such an architecture is HyGCN [15]. The second category maps the two phases into a uniform model of sparse–dense matrix multiplications (SpMM) operating upon a unified hardware architecture. For instance, GCNAX [16] adopts an outer product with the two-stage *multiply* and *merge* pipeline. This architecture overcomes the accelerator resource underutilization caused by dynamic workload variations in the separate architecture [17]. However, this procedure generates many partial matrices, leading to repetitive off-chip memory access. Through caching reusable data in the on-chip memory, it can provide some relief by reducing off-chip accesses. However, the cache size required for large graphs can grow exponentially, resulting in substantial area and energy consumption. As a result, GCNs accelerators on conventional architectures often remain bottlenecked by off-chip memory accesses [15], [16], [18].

The processing-in-memory (PIM) architecture presents a promising solution to resolving the memory bottleneck by integrating computational logic directly into memory. In advanced 3D-stacked memory, multiple DRAM dies are stacked on top of a base die, resulting in a cube structure. This cube is partitioned into multiple vaults. The digital logic can be integrated either at the base die or near the memory bank of the DRAM layers [19], [20], [21]. Compared to integrating the digital logic at the base die with sufficient computing ability, the integration of digital logic near the memory banks can provide lower local data access latency [22].

GCIM, a state-of-the-art PIM-based GCNs accelerator [23], integrates computing units at both the DRAM layer and base die for executing the *aggregation* and *combination* phases, respectively. Although GCIM effectively exploits both the computation capability of the base die layer level and the access latency advantage of the bank level, it achieves suboptimal performance for the following reasons. First, there are a large number of time-consuming neighbor data accesses. In 3D-stacked memory, data, such as graph vertices and their feature vectors, are stored in different banks of different DRAM layers. When each processing element (PE) of the DRAM layer performs aggregation operations, the information of adjacent vertices they require may be irregularly distributed across different banks. This leads to dynamic remote random access. Compared to data access from the local bank next to PE, this will cause a significant delay in accessing data from other banks. Second, dynamic load variations may result in an uneven workload distribution among computational units. Specifically, in GCIM, the divide-and-conquer design fits well with the scenario where the execution times for aggregation and combination can be overlapped fully. However, due to the dynamic variation of workloads, fast engines always have to wait for slow engines. Even worse, data transfers between different engines also lead to delays.

Fortunately, we observe that the output features in the preceding layer of GCNs serve as input features for the subsequent layer. Meanwhile, with the outer product method in the aggregation phase, the partially generated matrices in the multiply stage are composed of multiple partial feature vectors from different vertices. As a result, the partial feature vectors generated by local vertices can be directly transmitted to the PEs next to the banks of their corresponding remote neighbors for accumulation and obtaining the final output feature vectors. Direct transmission of partial vectors between PEs avoids frequent data transfer between banks and PEs during the multiply and merge stages. Further, it is also observed that the row-wise product in the combination phase enables the construction of a unified PE that alternates between executing the combination and aggregation operations. This eliminates uneven workload issues between different phases, and also, the delay in feature vector transmission can be hidden behind the computation, avoiding computation stalling caused by obtaining remote data during the vertex aggregation process. Therefore, we are motivated to design a bank-level PIM-based 3D-stacked memory architecture, which is capable of alternately performing two kernels of GCNs by a hybrid matrix computation model with impressive performance and energy gains.

In this article, we present GCNim, a GCNs accelerator that situates PEs near the banks of the 3D-stacked memory and incorporates three significant design aspects. First, GCNim is equipped with a new GCNs computational model, which adopts the execution order of aggregation after combination. The model leverages the row-wise multiplication method in combination and utilizes the outer product in aggregation. The intermediate matrix generated in the multiply stage of the outer product is subdivided into multiple partial vectors, which are then sent to the corresponding PEs for merging. This approach transforms irregular data access in aggregation into directed data transmission, significantly reducing nonlocal DRAM access. Second, GCNim adopts a three-stage pipeline for parallel computations, where each stage is abstracted as an operation of dense vectors. We integrate the identical execution unit (EU), prefetcher, and buffers alongside each memory bank. This allows GCNim to eliminate any potential latency and energy overhead arising from data movement between different engines and avoid the problem of uneven workloads caused by different engines. Finally, we analyze the sources of load on each PE at each stage and propose a lightweight data placement algorithm to improve load balance between PEs.

The contributions of this article are summarized below.

1) We present a hybrid GCNs computation model tailored for 3D-stacked memory, which employs the outer product for the aggregation phase and utilizes the row-wise product for the combination phase.
2) We propose a novel GCNs accelerator, GCNim, integrating well-designed uniform PEs near the memory bank. It enables efficient pipelined execution of both the combination and aggregation kernels, avoiding underutilization of computational units caused by dynamic load imbalance across the kernels.
3) We introduce a lightweight data allocation strategy to attain task distribution equilibrium and support the designed architecture efficiently.
4) We evaluate GCNim with various graph datasets. GCNim demonstrates superior performance compared to the state-of-the-art CPU system, GPU system, GCN accelerators I-GCN, PEDAL, FlowGNN, and GCIM. GCNim achieves a speedup of 3736.06×, 76.56×, 3.35×, 8.97×, 2.24×, and 5.58× while achieving energy savings of 8292.46×, 81.45×, 1.83×, 5.53×, 1.32×, and 2.83× on average.

The remainder of this article is as follows. Section II introduces the background and motivation of this work. Section III presents the novel hybrid execution model. Section IV describes the details of GCNim architecture. Section V introduces the tailored data placement method. GCNim is evaluated in Section VI. Section VII reviews the related works, and finally, Section VIII concludes this article.

II. Background and Motivation

In this section, we first present the fundamental tenets of the underlying GCNs. Subsequently, we comprehensively analyze existing GCNs accelerator architectures and implementation techniques. Afterward, we introduce the PIM approach in

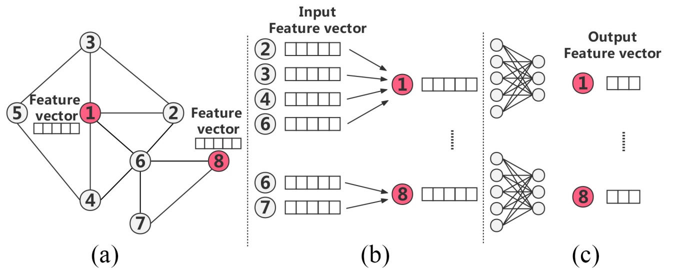


Fig. 1. Illustrative instance showcasing GCN inference, encompassing (a) graph, (b) aggregation kernel, and (c) combination kernel.

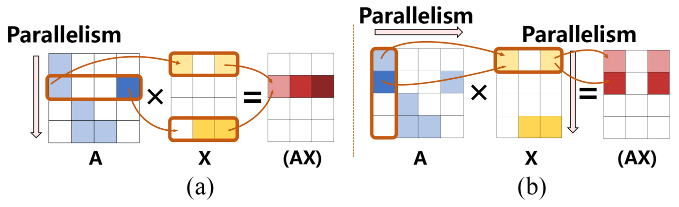


Fig. 2. Comparison of (a) row-wise product and (b) outer product approach in the aggregation. The row-wise product generates a computed row vector in the resulting matrix, while the outer product obtains a partial matrix.

3D-stacked memory and expound on our approach of integrating GCNs with 3D-stacked PIM architectures.

A. Graph Convolutional Networks

GCNs are structured as a sequence of graph convolutional layers, where each layer is mainly composed of the aggregation and combination phases, as depicted in Fig. 1. From a linear algebra perspective, the inference procedure can be represented as follows:

$$X^{(l+1)} = \sigma\left(AX^{(l)}W^{(l)}\right) \tag{1}$$

where A represents the adjacency matrix of the graph. $X^{(l)}$ denotes the input feature matrix of the lth layer. Each row of $X^{(l)}$ represents a feature vector of a vertex. $W^{(l)}$ denotes the weight matrix of the lth layer, and σ denotes a nonlinear activation function, such as ReLU [5]. A is usually normalized to avoid scale changes. The normalization process involves computing $\tilde{A} = D^{-(1/2)} \times (A + I) \times D^{-(1/2)}$, where I is the identity matrix, and D is a diagonal matrix. Note that $\tilde{A}$ remains constant during the GCNs training and inference. Furthermore, as A could be performed offline from $\tilde{A}$, it is used to represent $\tilde{A}$ throughout the rest of this article.

Many variant GCNs models have been devised from the GCN model, such as GraphSage [24] and graph isomorphism network (GIN) [25]. As shown in GCNAX [16], the forward propagation of the majority of GCNs can be generalized and represented by (1). In this context, the efficiency of GCNs' inference predominantly hinges on the performance of aggregation and combination kernels, which constitute the primary focus of this article.

B. Accelerator Design Exploration

Several GCNs-specific architectures [15], [16], [26], [27] have been proposed to accelerate GCNs inference in recent years. These accelerators have implemented specific computation models and are tightly co-designed with microarchitecture. We next introduce the execution order of GCNs and scrutinize the characteristics and issues of diverse approaches used for GCNs inference.

Execution Order: Previous studies have summarized two possible execution orders for graph convolution layers: 1) combination-first (i.e., $A \times (X^{(l)} \times W)$) and 2) aggregation-first (i.e., $(A \times X^{(l)}) \times W$). The execution order does not impact the correctness but influences the computation required [28]. Assuming that $A \in \mathbb{R}^{N\times N}$, $X^{(l)} \in \mathbb{R}^{N\times D}$, $W \in \mathbb{R}^{D\times F}$, and $X^{(l+1)} \in \mathbb{N}^{N\times F}$. The computation amount of the aggregation-first method is $(N \times N) \times (N \times D)$ in the aggregation phase and $(N \times D) \times (D \times F)$ in the combination phase. In comparison, the computation amount of the combination-first method is $(N \times N) \times (N \times F)$ and $(N \times D) \times (D \times F)$, respectively. This suggests that the computation amount depends on the input feature vector dimensionality D and the output feature vector dimensionality F. For most datasets, D is larger than that of F. Consequently, the combination-first method generally has superior performance than the aggregation-first one.

Matrix-Multiplication-Based GCNs Inference: The computation pattern of a GCNs layer is the multilayer matrix multiplication. For the three matrices involved in the GCN inference, A and X are sparse, while W is dense. Therefore, changes in the execution order will correspondingly lead to changes in the calculation kernel. For the aggregation-first method, the aggregation kernel corresponds to sparse–sparse matrix multiplication (SpGEMM), while the combination kernel is a regular dense-dense GEMM operation. For the combination-first method, both aggregation and combination kernels correspond to SpMM. The sparse matrix multiplication techniques used in past accelerators can be categorized into three main types [29]: 1) *row-wise product*; 2) *column-wise product*; and 3) *outer product*.

1) *Row-Wise Product:* It comprises two main steps, as shown in Fig. 2(a). First, the algorithm multiplies each nonzero element in a given row of A with all elements in the corresponding row of X. The rows in X depend on the column index of the nonzero elements in the given row of A. Second, the partial results are accumulated to obtain a row of the final product matrix. Each row of A is computed in parallel, resulting in the corresponding row in the output matrix. The early GCNs accelerator, HyGCN [15], uses this method based on the aggregation-first sequence, which processes SpGEMM and GEMM operations in aggregation and combination through two independent engines. However, the row-wise product is not friendly for the aggregation phase. On the one hand, X cannot be entirely stored on-chip, which may lead to additional off-chip accesses to X that may occur for the feature aggregation of each vertex. On the other hand, the sparsity of features leads to a significant reduction in arithmetic intensity during aggregation.

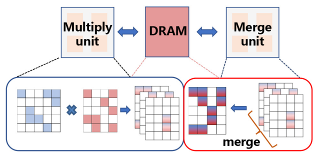


Fig. 3. Partial matrices generated during the multiply stage are often moved frequently between processing units and DRAM.

2) *Column-Wise Product:* In this method, the nonzero elements in a single column of X are multiplied with the corresponding columns of matrix A. The resulting values are accumulated to form a column in the output matrix. This method exhibits similarities to the row-wise product. The AWB-GCN [26] leverages this method to construct an accelerator that performs the two phases alternately. However, it encounters similar issues as the row-wise product method, requiring redundant off-chip access to A and suffering from insufficient utilization of computational resources due to graph sparsity.

3) *Outer Product:* This method involves multiplying a column of A with a row of X, resulting in a partial matrix of the output matrix. The final result is obtained by merging all the partial matrices. Fig. 2(b) illustrates the outer product approach and its parallel process. GCNAX [16] proposes an optimized dataflow using a unified architecture based on the outer product, which allows reusing input matrices efficiently and avoiding zero-valued operations by processing W first. However, the outer product gives rise to a multitude of partial matrices during the process, leading to a significant reduction in output reuse. As these matrices cannot be entirely stored on-chip, merging them creates redundant off-chip accesses. This results in partial matrices undergoing repeated movement between processing units and DRAM, as illustrated in Fig. 3.

In summary, previous works on accelerating GCNs have utilized different matrix multiplication methods to handle GCNs inference. However, due to the large graph scale and the high-dimensional feature vector, A and X cannot be entirely stored on-chip. Also, the sparsity of the graph and feature vectors results in frequent and irregular data movement between DRAM and the processors. All these factors jointly lead to the fact that conventional memory architectures are of great necessity to be innovated for efficient GCNs inference.

C. 3D-Stacked Processing-in-Memory

The 3-D memory architecture [30], [31] is innovative to enable the vertical stacking of memory layers, enhancing memory density and reducing the footprint of the memory module. Through-silicon vias (TSVs) are employed in 3D-stacked memory to establish interconnections between the different layers. Typically, 3-D memory comprises a base logic layer and multiple DRAM layers stacked on it. Each DRAM layer is divided into multiple partitions, separating the entire 3-D stack into several vertical vaults. These vaults possess memory controllers located on the base logic layer, which can simultaneously access multiple partitions on the DRAM layer. This provides highly parallel memory access.

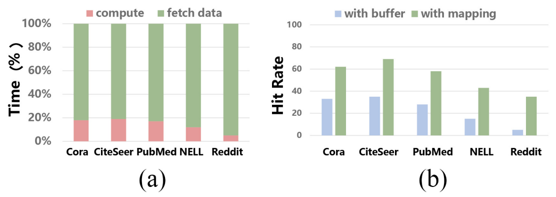


Fig. 4. Effect of remote random access on (a) execution time ratio with a naive setting and (b) hit rate with the buffer and mapping algorithm.

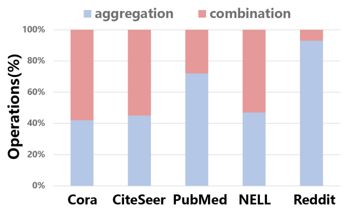


Fig. 5. Percentage distributions of the operations in the combination and aggregation kernels by benchmarking a 2-layer GCN model.

The concept of PIM takes advantage of the proximity between processors and memory cells to accelerate data processing and reduce energy consumption. Situating processing units near the memory subsystem facilitates direct and high-bandwidth communication between these components. This minimizes the data transfers between processors and memory, leading to significant improvements in performance, reduced latency, and increased energy efficiency.

The 3-D memory architecture offers two PIM solutions, with the processor on the logic or DRAM layers. GCIM [23] is the first accelerator that employs the PIM-based 3-D memory architecture to handle GCNs inference. It accommodates the unique characteristics of the two phases of GCN by placing the aggregation engine next to the bank and integrating the combination engine in the logic layer. The aggregation engine uses the traditional pull-based approach of graph processing. In contrast, the combination engine utilizes the systolic array, similar to HyGCN, which follows the aggregation-first computational order. However, this design poses challenges in implementing data locality and load balancing. On the one hand, GCN kernels with random access suffer significantly long latency when accessing data from other vaults or even other cubes during local vertex aggregation. Fig. 4(a) illustrates that under the naive configuration, the computing unit's real-time execution spans no more than 20%. Fig. 4(b) demonstrates that remote random access remains prevalent and substantial despite employing buffer and mapping strategies to optimize locality. Additionally, due to stringent area constraints and the high dimensionality of features, storing replicas results in unacceptable wastage of storage resources.

On the other hand, this separate two-phase design cannot easily handle dynamic workload changes between phases, leading to insufficient utilization of computational resources. Fig. 5 depicts the percentage distributions of these two types of operations by benchmarking a 2-layer GCN model on the five real-world datasets. The results indicate significant variations in the workload distribution between aggregation and combination phases across different datasets. Particularly for the large-scale Reddit dataset, the number of operations in aggregation is significantly higher than in combination.

D. Combining GCNs With 3D-Stacked PIM

To address the challenges mentioned above, we introduce GCNim, which has several innovative features.

First, GCNim employs a new GCN-specific computational model to maximize performance gains on a 3D-stacked memory PIM architecture. Our insight is to use an outer product multiplication method during the aggregation phase, which splits the partial matrix obtained from the multiply stage into vectors sent to other PEs for merging. This converts the irregular remote data access during the aggregation phase into regular data transfers, thus, avoiding the performance impact arising from inactive states by collecting feature information from other vertices.

Second, GCNim exploits the 3D-stacked bank-level PIM to build the unified hardware that supports combination and aggregation. With a specialized model that performs the row-wise product for the combination phase and the outer product for the aggregation phase, the entire inference process can be abstracted into three dense vector operations supported by a set of unified computation units. The unified PE integrated near the DRAM bank allows alternating execution of combination and aggregation, with intermediate results residing in the registers of the computation unit. The GCNs architecture eliminates data transfer overhead between phases and avoids uneven workloads caused by dynamic workload changes between phases.

Third, since aggregation and combination are executed on a unified architecture, GCNim does not require load balancing between phases. Instead, we only need to design a lightweight data placement algorithm for analyzing the workload of each vertex in GCNs, simplifying designs.

III. GCNIM COMPUTATION MODEL

As elucidated in Section II-B, the aggregation and combination phases of the GCNs convolutional layer can be represented as a sequence of successive SpMM computations. However, the two phases exhibit distinct and unique properties. This provides two opportunities for designing a GCNs inference computation model. The first lies within each phase, where the design needs to account for the specific characteristics of both phases to accommodate the varying demands of computational and memory resources. Notably, the execution of the aggregation exhibits significant differences compared to the combination, with the highly sparse kernel normally the bottleneck in GCNs processing. Therefore, GCNim prioritizes the method of aggregation. The second opportunity arises in the interaction design between phases. GCNs allow any phase to precede another phase. Although each phase can adopt any matrix multiplication method, the choice of one phase may impact the next one. This is because the chosen method affects memory access to transfer data to the next phase. Therefore, careful consideration should be given to selecting an appropriate matrix multiplication approach for each phase, considering its impact on the overall pipelining and data reuse.

A. Remote Merging in Aggregation

Compared to storage-and-compute decoupled architectures and designs with PEs located in the logic layer, the most significant advantage of architecture with PEs near banks in the DRAM layer is its proximity to data storage. As a result, PEs can read and write data from local banks with lower latency. However, the graph's vertices and feature vectors are distributed across different banks. In the aggregation phase, if the neighboring information required by a vertex is not stored locally, it results in random remote data access, which leads to the loss of the advantage of processing data close to memory.

To address this issue, we devise a method in the aggregation phase that utilizes the outer product to achieve local multiplication and remote merging. The aggregation phase is subdivided into multiply and merge stages based on the explicit phase change of the outer product.

The multiply stage is carried out thoroughly in parallel within local PEs. Each PE computes a column of A and the corresponding rows of X or (XW) to generate a partial matrix. Instead of immediately merging the partial matrices or storing them in their respective banks, we divide the partial matrix into multiple row vectors. These row vectors correspond to the partial output features. In multilayer GCNs models, the output features of the preceding layer serve as the input for the subsequent layer. Therefore, we eliminate all zero-valued row vectors and send the remaining vectors to the corresponding PE with a matching row index for merging, called remote merging. The complete output feature vector is obtained through iterative accumulating during the merge stage and is stored in nearby banks. In other words, we transformed instruction-driven data retrieval into data-driven data transmission, shifting from sending instructions and waiting for data to directly transmitting data. This change has significantly reduced access latency and saved costs.

B. Combined Execution Model

In order to support the remote merging and enable data reuse across stages while ensuring the overall efficiency of GCNs, we shall contemplate two aspects: 1) the execution order between phases and 2) the matrix multiplication method in the combination phase.

Execution Order: GCNim employs a combination-first execution order. This is because parallel tasks involved in the multiply stage do not produce complete matrices, leading to the combination having to wait until the entire multiply and merge stages are completed before they can start. If partial results are directly sent to the combination, it could result in redundant computations. Furthermore, the combination first results in lower computational overhead for most datasets.

Combination Phase: We propose employing the row-wise product during the combination phase instead of utilizing either the outer or inner products. The fundamental idea behind our choice is that row-wise products can load the corresponding rows of matrix W based on the nonzero element

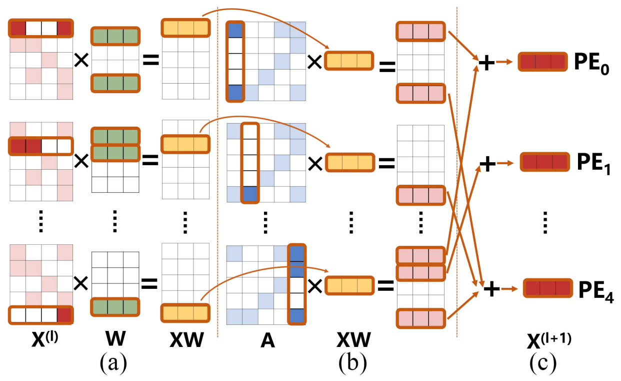


Fig. 6. Computation model of GCNim follows the combination-first order. The GCNs inference process is redefined as a three-stage process, where the combination stage utilizes the row-wise product method. The aggregation phase is split into a multiply and merge stage using the outer product method. (a) Combination stage. (b) Multiply stage. (c) Merge stage.

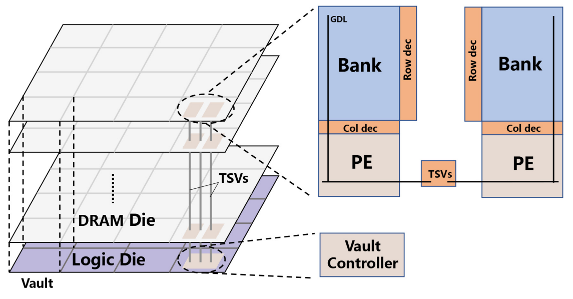


Fig. 7. Architecture overview of the GCNim.

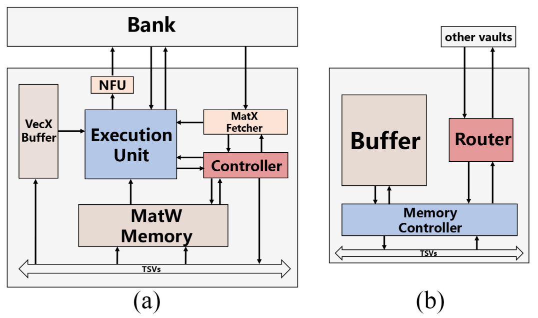


Fig. 8. Detailed hardware architecture of GCNim: (a) PE of the DRAM layer and (b) vault controller in the base logic layer.

indices within each row of matrix X. Thus, there is no need to access the entirety of W. Moreover, the complete row of XW obtained through row-wise product computations can immediately engage in computations during the aggregation phase, ensuring the comprehensive spatial reuse of the XW matrix.

On the one hand, the outer product method fails to yield a complete matrix, necessitating the aggregation phase to await the completion of the entire combination phase before commencing. Directing partial results to the aggregation would lead to redundant computations. On the other hand, employing the inner product method would require computing each row of X with every column of W, causing X to access the entirety of W during each row computation. However, due to the sparsity in the feature matrix X, a portion of the data within each loaded column of W is unnecessary for computation. Furthermore, the inner product method involves element-wise calculations, which, to be implemented, demand additional reduction operations. Introducing dedicated hardware for these reduction operations would result in significant area overhead. Neither the outer product nor the row-wise product methods impose such requirements.

The overview computational model of GCNim is depicted in Fig. 6, which embodies the critical processes of GCNs inference through a sequence of matrix operations. The model adheres to the combination-first execution order and comprises three consecutive stages: 1) combination; 2) multiply; and 3) merge. Specifically, during the combination, the rows of X are multiplied with matrix W in parallel, employing the row-wise product method, to obtain rows of XW. This ensures that each nonzero element of X, typically sparse, is accessed only once. In the multiply stage, the multiplication of rows of XW and columns of A yields several partial matrices, further divided into partial vectors. These vectors are dispatched to different PEs, where all partial vectors with the same row indices are accumulated during the merge stage, and the final complete output features are obtained.

IV. GCNim Architecture

This section presents the GCNim architecture to support the proposed computation model. It aims to facilitate GCNs inference by executing three phases in a data-parallel manner. The section first presents a comprehensive exposition of the overarching architecture of GCNim, succeeding by exploring the design of its DRAM layers and base logic layers. Finally, this section provides a detailed description of the complete workflow for GCNs inference on GCNim.

A. Accelerator Overview

We observe that by redividing the inference of each GCNs layer into three stages: 1) the combination stage involves dense vector multiply–accumulate (MAC) operations; 2) the multiply stage involves scalar multiplication of dense vectors; and 3) the merge stage is dense vector accumulation operations. All three operations can be seen as dense vector operations, allowing us to use a set of MAC units to support these three operations, thus, enabling the design of a unified PE architecture.

Fig. 7 provides an overview of the GCNim architecture, comprising a base logic layer and several DRAM layers. The entire memory cube is partitioned into multiple vaults. Within each vault, the memory cells in the DRAM layers are partitioned into bank groups that share a common TSV, allowing for communication between the vault layers. GCNim incorporates a PE alongside each bank and a vault controller in the logic layer. The PEs are primarily responsible for the execution of GCNs inference computations, while the vault controller manages communication and data forwarding between various DRAM layers across different vaults.

B. DRAM Layer PE Design

As illustrated in Fig. 8(a), the architecture of the PE in the DRAM layer consists of five main components, namely, the EUs, the controller, the MatW memory (MWM), the MatX

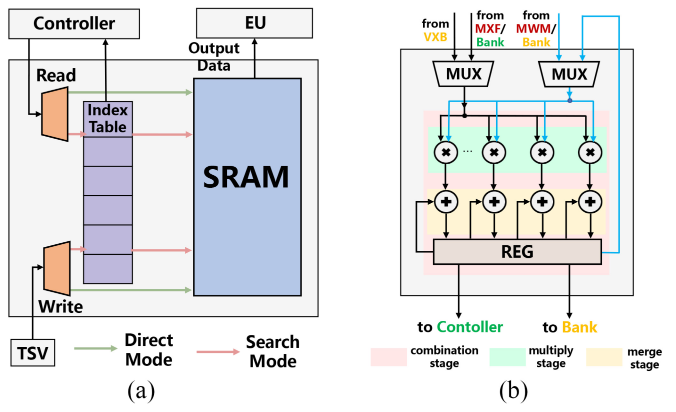


Fig. 9. Microarchitecture of (a) MWM and (b) EU.

fetcher (MXF), and the VecX buffer. In the subsequent, we will present a comprehensive description of each component of a PE.

1) Execution Unit: Each EU comprises a dual set of 16 FP32 multipliers and adders, as depicted in Fig. 9(b). These enable the execution of three fundamental operations: 1) MAC; 2) MUL; and 3) ADD. Data is directed from different memory hierarchies to the EU, which the controller organizes. Moreover, the EU is equipped with several registers that are employed for instruction decoding and data staging.

2) Controller: The controller has two primary responsibilities: first, it manages the transfer of data between the bank, PE, and logic layer, and the internal data transfer within the PE. Second, it orchestrates three computational stages. As the merge stage involves handling data from remote PEs, arrival time and sequence are uncertain. Hence, the controller needs to coordinate when to start the merge stage.

3) MatW Memory: The MWM is mainly composed of a content addressable memory (CAM), and it also supports direct mapping, as illustrated in Fig. 9(a). Due to the sparsity in feature vectors, the row-wise product approach in the combination stage results in repetitive and random access to weight parameters. These parameters may be shared among all vertices. To improve data reusability and reduce access to the logic layer, we implement a specialized CAM-based scratchpad within the PE to store and buffer the weight parameters. This serves to alleviate the bandwidth pressure on the TSV.

4) MatX Fetcher: The MXF is a buffer for prefetching and storing the feature vectors. Its primary objective is to allow the PE to predict which weight parameters are needed for matrix multiplication and preload them into the MWM before computation begins based on the column indexes of the nonzero elements of feature vectors. However, due to the constrained area allocation resulting from the integration of PE next to the bank, it may not be possible for the MXF and MWM to store an entire feature vector and the corresponding weight parameters. Consequently, the controller iterates through the elements in the MXF to verify if the weight parameters are present in the MWM.

5) VectorX Buffer: The VectorX buffer (VXB) serves to receive partial feature vectors transmitted from other PEs and arranges them in order according to their corresponding row indices. When a certain number of these partial vectors are received, and the other two stages have been completed, the partial vectors are then sent to the EU for accumulation.

In addition, to maintain generality, a nonlinear function unit (NFU) is also configured within each PE. Each layer's final complete output vectors will be activated through NFU and written back to the banks.

C. Logic Layer Design

The composition of the logic layer in each vault is consistent and includes a router, a memory controller, and a buffer, as depicted in Fig. 8(b). Its primary function is to manage requests and forward and deliver data to the relevant PE. The matrix W is stored in the buffer. When W is small enough, it can be entirely stored in each logic layer. In contrast, when W is relatively large, GCNim distributes the weight parameters evenly among different vaults.

The logic layer handles two primary types of data requests and forwarding. The first is retrieving and forwarding the weight parameters required during the combination stage. The memory controller retrieves the necessary parameters from the buffer and transmits them to the corresponding PE based on the request. If each vault stores the entire W, there is no need for intervault communication. However, if W needs to be stored separately, the controller of each vault maintains the distribution information of W. The controller is first based on the index of the required weight parameters to ascertain whether they are located in the local buffer or other vaults. Then, it will either directly transmit the data or request them from other vaults. The second type of data forwarding is partial vectors generated in the multiply stage. This includes sending vectors to and receiving vectors from other vaults and forwarding them to different PEs of DRAM layers. Each vault controller maintains the correspondence between the feature vectors in the local vault and the banks and between nonlocal feature vectors and the vault. This information is assigned statically before the computation begins and remains unchanged during execution.

D. Workflow

In this section, we explain how GCNim performs the inference task of GCNs comprehensively and systematically. The three matrices are stored in different formats. The adjacency matrix A is stored in a compressed sparse column (CSC) format, and the feature matrix X is stored in a compressed sparse row (CSR) format. Column and row vectors of A and $X^{(l)}$, which share the same indices for the outer product multiplication, are stored in the same memory bank. For the row vectors of $X^{(l+1)}$, the merging and access occur within the same bank with matching $X^{(l)}$ indices. The weight matrix W is stored in a dense row-major format in the Buffer of the logic layer. Smaller weight matrices are entirely stored in one vault, while larger ones are evenly distributed across multiple vaults.

In the first combination stage, matrix multiplication is performed between X and W. Each PE involves taking all nonzero elements in a row of X and performing scalar multiplication with multiple rows of the required W matrix, followed by vector addition to obtain an intermediate result, a row of XW. The nonzero elements of X are cached in MXF, and the rows of W are stored in MWM. The lack of caching of the corresponding rows of W in the MWM, when nonzero elements of X are computed sequentially, can lead to calculation pauses. Therefore, GCNim employs a nonblocking method to handle this issue.

The controller of GCNim cyclically checks each element in MXF and verifies whether the corresponding W row exists in MWM based on its column index. If a match is found, the element and the corresponding row of W are immediately sent to the EU for MAC operations. Specifically, as shown in Fig. 9(b), the nonzero element of X is broadcasted to all MAC units, while the elements in the row vector of W are sequentially forwarded to each MAC unit. Each EU provisioned eight multipliers and adders. The dimensionality of the weight parameters determines the length of the output feature vector. The time required for each MAC operation also depends on this length. If the length exceeds the number of multipliers and adders, the entire operation will be iterated for multiple rounds. If a match is not found in MWM, the controller sends a command to the logic layer to retrieve the corresponding W row while skipping the current element and moving on to the next one. This design is supported by the fact that the execution order of the elements in a feature does not impact the correctness of the result using the row-wise method. However, for sparse matrix X, the distribution of nonzero elements is irregular. Some feature vectors may have an unusually high number of *nonzero elements* (*nnz*) that cannot fit entirely in MXF. Moreover, the poor locality in X may result in frequent data requests, putting excessive pressure on TSV. To address this issue, for feature vectors with a large *nnz*, GCNim no longer traverses and searches MXF but instead divides MXF and MWM into double buffers, with half the data involved in the processing and the other half in data preparation. In this case, multiple rows of W are loaded into the buffer simultaneously, and the controller operates in order directly without verifying the rows of W.

After processing all nonzero elements in a row of X, the intermediate result XW row is temporarily stored in the register of the EU. In the multiply stage, columns of A are fetched from the banks to the EU, then multiplied by the XW to generate partial matrices through the outer product. In other words, the data generated in the combination stage remains stored in the registers instead of entering the global buffer or DRAM and is directly involved in the multiply stage calculation. This can reduce the area overhead caused by caching intermediate matrices or avoid expensive DRAM access, providing dual benefits of energy and performance.

The partial matrices are fragmented into multiple partial vectors, and the controller subsequently forwards these partial vectors to the logical layer, which then assigns them to the corresponding PE via row indexing. Upon receiving the partial vectors, each PE accumulates them in the merge stage.

We employ a vector buffer VXB and a data reordering technique to prevent potential conflicts. Partial output vectors

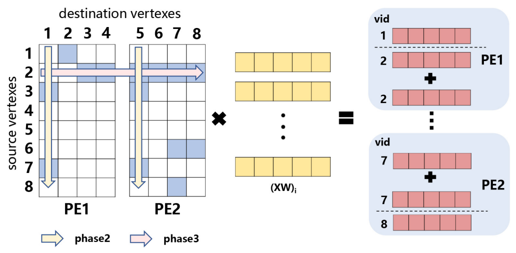


Fig. 10. Example of the allocation of workload for the second and third stage PEs.

are temporarily stored in VXB instead of being immediately merged. The merging process occurs after the combination and multiply stages when the data in VXB reaches a specific threshold. If there is limited data in VXB, the combination and multiply stages proceed consecutively, bypassing the merge stage. During merging, the EU retrieves the data from VXB, matches them based on their indices, and accumulates them with the old partial sums stored in the bank. Subsequently, the new partial sums are written back to their original positions in the banks. To minimize data movement within the bank, we reorder the partial output vectors according to their indices rather than relying on the order of vector arrivals. Once all partial vectors have been merged, the resulting output vector is activated by the NFU and stored back in the bank.

GCNim system does not stagnate in computation due to data waiting, as the data either resides in the EU or is prepared in the memory of the PE. The controller orchestrates the computational pipeline and data movement across the three stages. The combination and multiply stages are closely coupled. Interstage pipelining is achieved by having the output of the combination stage reside in the registers of the EU, participating in the subsequent multiply stage. Merging does not occur after every multiply stage; its execution depends on the amount of data in VXB. Therefore, the combination or merging stage can execute after the multiply stage, and the EU is reused across these three stages. The combination and multiply stages use computations performed in other stages to hide the latency of data transfer in these stages. These approaches improve the utilization of both PE and DRAM bandwidths and minimize the number of incoming memory accesses.

V. Preprocessing

In this section, we introduce a lightweight workload mapping method to distribute the nonzero elements of matrices A and X into GCNim's memory banks for EUs to process. GCNim integrates a PE next to each memory bank, and all PEs simultaneously handle nonzero elements. Therefore, performance is bounded by the slowest PE, which needs workload balance among the PEs.

In prior research, offline preprocessing techniques [32], [33], [34] were employed to reconstruct the graph and partition the vertices. This approach aimed to enhance data locality and achieve a balanced workload distribution. However, this approach introduced significant latency overheads based on complex software reordering algorithms and was only feasible during offline processing. Instead of relying on software algorithms, we harness data locality by designing our hardware architecture. First, in the combination stage, we utilize prefetchers, buffer for W matrices, and hidden data transfers after calculations to provide locality advantages. Second, in the multiply stage, all elements of the adjacency matrix are parallel without data dependencies, while in the merge stage, we achieve complete data locality by sending partial vectors to the corresponding storage PE for merging.

After static data allocation, A and $X^{(0)}$ do not move during system runtime, and all PEs execute in parallel. Performance depends on the slowest PE, so balancing the workload among all PEs through preprocessing is necessary. By analyzing the workload of each PE in the three stages, we propose an efficient and lightweight processing method.

The workload of each PE in each stage depends on nnz in X allocated in its bank, nnz in A, and the number of X partial vectors received. Specifically, for the first stage, the multiplication and addition operations time depends on nnz in X and the length of the W vector. As the length of the dense matrix W row is fixed, only X's nnz needs to be considered. For the second stage, the time for multiplication operations is similar to the first stage and depends on nnz in A and the length of the (XW) vector. For directed graphs, nnz in A's column represents the in-degree of a vertex, which determines the workload of the second stage. Although partial vectors are received from remote PEs for the third stage, the total computation for the third stage can be preknown. This is because the partial vectors are sent based on A and X's row indices. The number of generated partial vectors in the second stage depends on the nnz in A's row vectors, representing the out-degree of a vertex in the directed graph. Therefore, the total computation of the PE depends on the degree of the assigned vertices and nnz in their feature vectors. Fig. 10 provides an example of workload allocation for PEs in the second and third stages. Based on this, we propose a simple heuristic data placement algorithm. Typically, the algorithm employs a greedy strategy, and the pseudocode is demonstrated in Algorithm 1.

First, we allocate an empty container for each bank. We compute each vertex's workload as the sum of its degree and nnz in its feature vector. Then, we place the vertices into different containers according to the capacity of each container and use a minimum priority rule. Once all vertices are assigned, we will renumber the vertex IDs of the entire graph to ensure that vertices in the same container have continuous indexes, making it easier for the logic layer's vault controller to store corresponding information.

Algorithm 1 Matrices A and X Assignment to Banks

```
Input: The adjacency matrix $A$, feature matrix$X$, the number
    of PEs $\#PE_s$
Output: The partitioned $A$ and $X$
 1: $n \leftarrow$ the number of vertices of $A$
 2: $P \leftarrow MinFirstQueue(\#PE_s)$
                                  ▷ Initialize each container
 3: for $i \leftarrow 0$ to $n$ do
 4:     $D_i \leftarrow$ the degree of vertex $i$
 5:     $N_i \leftarrow nnz$ in $i$-th row of $X$
 6:     $p \leftarrow P.extract\_\min()$        ▷ Filling the container
 7:     $p.vertex.idxset(i)$
 8:     $p.workload \leftarrow p.workload + D_i + N_i$
 9:     $P.insert(p)$
10: end for
11: $count = 0$
12: for each $p \in P$ do                ▷ Renumber the vertices
13:     for each $v \in p.vertex$ do
14:         $v.index \leftarrow count$
15:         $count \leftarrow count + 1$
16:     end for
17: end for
```

VI. Evaluation

In this section, we begin by introducing the experimental setup. Subsequently, we elaborate on the comprehensive performance, power, and area results of GCNim against state-of-the-art GCNs software and hardware solutions. Following that, we discuss the scalability of GCNim and the sensitivity studies of hardware configurations.

A. Experimental Methodology

Hardware Configuration: We follow the specification [35] for 3-D stacking memory to implement our architecture design. We use the configuration specified in the previous 3-D stacking memory characterization study [36]. A memory cube consists of a logic die, and eight DRAM dies. The cube is segregated into 32 vaults, each graced with its controller positioned at the logic die. These controllers establish connections with the DRAM dies via 32 TSVs. Each DRAM layer contains two banks per partition, each with a capacity of 16 MB. Hence, the number of banks within each cube entity amounts to 512, boasting a collective capacity of 8 GB. We set a PE next to each bank; the FPU of each PE contains 16 pairs of multipliers and adders, a 2-kB MXF, and the size of the VecX Buffer is also 2 kB. The size of the DRAM layer MWM and logic layer Buffer is 4 and 128 kB, respectively.

Simulation Configuration: We implement an in-house cycle-accurate simulator to perform performance and power simulations of our architecture. It also supports the prior work [16], [23] for bank-level computing. We use CACTI-3DD [37] to model 3D-stacked DRAM, interconnect components (including TSVs and routers), and on-chip memory elements (including MXF, VecX Buffer, PMatW Memory, and logic layer MatW Buffer) and estimate its area, power, and latency. To estimate the overhead of GCNim's logic parts, we implemented them using Verilog RTL and synthesized them using the Synopsys toolchain with the TMSC 28-nm standard library. Furthermore, we estimate the power consumption using Synopsys PrimeTime PX. GCNim runs at a frequency of 312.5 MHz.

Datasets: The five datasets most widely used in GCNs studies are shown in Table I. Among the datasets utilized, we incorporate Cora (CR), Citeseer (CS), and Pubmed (PB).

TABLE I
GRAPH DATASET

Datasets	Cora	Citeseer	Pubmed	Nell	Reddit
#Vertex	2,708	3,327	19,717	65,755	232,965
#Edge	10,556	9,104	88,648	266,144	114,615,892
Feature length	1,433	3,703	500	5,415	602
Density of A	0.18%	0.11%	0.028%	0.0073%	0.21%
Density of $X^{(0)}$	1.27%	0.85%	10.0%	0.011%	51.6%
Density of $X^{(1)}$	78.0%	89.1%	77.6%	86.4%	60.0%
Density of W	100%	100%	100%	100%	100%

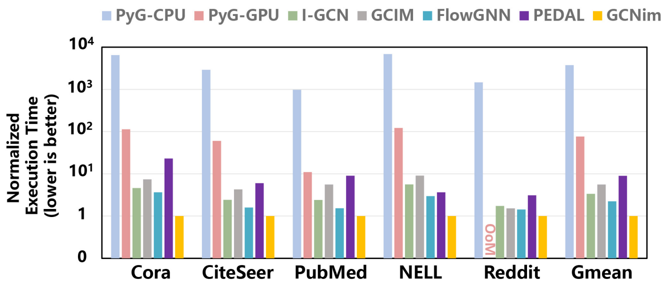


Fig. 11. Execution times of GCNim against the state-of-the-art GCNs solutions. All results are normalized to GCNim.

They represent three widely recognized collections of paper citation networks. Nell (NE) is a knowledge graph. Reddit (RD) is a social network graph extracted from various Reddit forums [38].

Baselines: We compared GCNim and the state-of-the-art graph neural network framework PyG [39] on CPU and GPU platforms. PyG is a library built on top of PyTorch for writing and training graph neural networks easily and powerfully. We evaluated PyG-CPU and PyG-GPU using a Linux workstation with two Intel Xeon CPUs E5-2680 v4 and one Nvidia Tesla V100 GPU.

Furthermore, we compared GCNim with four other advanced solutions: 1) a novel hardware accelerator I-GCN [18]; 2) a generic dataflow architecture FlowGNN [40]; 3) a power-efficient accelerator PEDAL [41]; and 4) a PIM-based accelerator GCIM [23]. To ensure fairness, we integrated the I-GCN and PEDAL solution into 3D-stacked PIM architecture, establishing them as the baseline scheme.

B. Overall Performance

We begin by assessing the performance of GCNim. Fig. 11 illustrates the comprehensive performance outcomes of GCNim compared to PyG-CPU, PyG-GPU, I-GCN, FlowGNN, PEDAL, and GCIM.

1) GCNim Versus PyG-CPU: In contrast to PyG-CPU, GCNim demonstrates a significantly faster performance, ranging from 976.36× to 6862.84× (3736.06× on average) due to the hybrid execution model and the near-data processing method adopted by GCNim architecture. The speedup obtained by GCNim is closely associated with the shape of the graph. Notably, NE shows the highest speedup ratio (6862.84×) due to the sparsity of its graph and input features. This means that most of the aggregation and first layer combination phases will show irregular memory access, resulting in poor performance for PyG-CPU. GCNim mitigates irregular memory access by splitting the aggregation into two stages, where data from the previous stage is directly transmitted to other PEs for the subsequent computation in the next stage. In the combination phase, we hide memory access latency by designing two-layer buffers and pipelines in three stages.

2) GCNim Versus PyG-GPU: GCNim outperforms PyG-GPU by a factor of 11.02×–121.82× (76.56× on average). Despite the GPU's vast number of cores, the sparsity of the graph during GCNs inference causes the GPU to generate strided memory access, resulting in multiple memory transactions during a single computation step. This makes it challenging for PyG-GPU to leverage the available parallelism fully. For GCNim, through the static mapping before execution and the outer product method adopted in the aggregation phase, FPU only needs to multiply or accumulate data stored in the local bank and buffers to avoid irregular memory access and achieve a reasonably high execution efficiency. Similarly, GCNim performs best on the NE graph, where the large graph size prevents complete on-chip memory storage in the GPU, thus incurring off-chip communication overhead. GCNim's near-data processing architecture can effectively solve this problem.

3) GCNim Versus I-GCN: I-GCN suggests a dynamic reordering scheme called Islandization based on a breadth-first search. In addition, it reuses the overlapping computations within the aggregation phase to reduce the computational complexity. In comparison, GCNim outperforms I-GCN by 1.73×–5.61× (3.35× on average). GCNim performs best on the NE graph but less well on the RD graph. This is due to the sparsity of the feature vectors in the first layer of NE. In contrast, the overall sparsity of it in RD is between 50% and 60%, resulting in the calculation in the combination phase of the RD not being memory-bound. This provides no advantage for the 3D-stacked memory with fewer processing units.

4) GCNim Versus GCIM: GCIM integrates MAC arrays in the logic and DRAM layers to support the two phases. In comparison, GCNim achieves a better speedup of 1.52×–9.08× (5.58× on average) compared to GCIM. Although GCIM employs a bank-level approach for aggregation, accessing vertices and their feature vectors can result in severe cross-vault communication between different vaults. Despite the use of replicas to reduce remote communication, this approach is ineffective for GCNs as long vertex feature vectors can lead to unaffordable storage overhead. GCNim performs better on the NE graph with sparse feature vectors, with the most extended feature vector length. GCIM follows an aggregation-combination execution order, whereas GCNim employs a combination-aggregation sequence. In most datasets, initiating with combination reduces computations due to the smaller output feature dimension than the input feature dimension. However, there is an exception: in the NE dataset's second layer, output features are longer than input features. Despite this, GCNim performs better on the NE dataset. This indicates that our performance enhancement is not solely due to reducing operations by changing the execution order.

5) GCNim Versus FlowGNN: FlowGNN is a novel and scalable dataflow architecture with a configurable dataflow optimized for GNN models with a message-passing

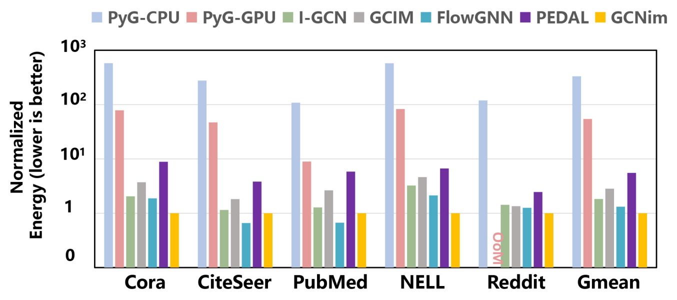


Fig. 12. Energy consumption of GCNim against the state-of-the-art GCN solutions. All results are normalized to GCNim.

mechanism. In comparison, GCNim outperforms FlowGNN by 1.43×–3.66× (2.24× on average). FlowGNN targets real-time applications with zero preprocessing and partitioning. GCNim is tailored for GCNs and achieves load balancing through a lightweight preprocessing strategy. Our design significantly reduces latency during the aggregation phase, particularly for large graphs like NE.

6) GCNim Versus PEDAL: PEDAL is a power-efficient accelerator supporting multiple dataflows. PEDAL chooses the best-fit dataflow and phase ordering based on input graph characteristics and GCNs algorithm, achieving both efficiency and flexibility. We compare the best-performing dataflows and phase order of PEDAL in GCN. In comparison, GCNim outperforms PEDAL by 3.10×–23.06× (8.97× on average). The architecture of PEDAL with separate engines and a design that supports multiple dataflows makes the execution of GCN less efficient. But in terms of generality, PEDAL can also support the GCN that employs nonlinear aggregation functions.

C. Energy Consumption and Area

We estimate energy consumption mainly from four main factors: 1) EU; 2) DRAM bank access; 3) on-chip SRAM access; and 4) I/O link in the network. Fig. 12 depicts the energy results. Thanks to GCNim's near-data processing capability significantly reduces the cost of data movement, GCNim consumes 2704.27×–14445.72× (8292.46× on average) less than PyG-CPU. GCNim saves 13.43×–124.17× (81.45× on average) more energy than PyG-GPU. GCNim exhibits a notable energy advantage over PEDAL, consuming 2.46× to 8.86× (with an average of 5.53×) less energy. Similarly, compared to I-GCN, FlowGNN, and GCIM, GCNim showcases superior energy efficiency, averaging 1.83×, 1.32×, and 2.83× less energy consumption. Compared with GCIM, GCNim adopts different multiplication methods to eliminate data movement overhead between the two phases while avoiding high energy consumption caused by irregular and redundant cross-partition communication through remote merging.

Area: Table II shows the area of the hardware components in each bank. The PE area cost of GCNim in each bank of the DRAM layer is only 0.1843 mm^2, accounting for just 7.58% of the bank area. The total area of all components in the logic layer is only 11.4048 mm^2, representing 11.88% of the logic layer area. The base logic die in the 3-D memory has a 10%–30% area budget [42]. Therefore, our design is within the acceptable range of the 3D-stacked memory area budget.

TABLE II
AREA AND POWER OF COMPONENTS IN A BANK OR A VAULT

Module	Component	Area(mm^2)	Power(mW)
DRAM layer	MACs	0.1328	32.9041
	MatX Fetcher	0.0059	0.4107
	VecX Buffer	0.0078	0.3441
	MatW Memory	0.0356	1.5575
	NFU	0.0022	0.4826
logic layer	Buffer	0.3564	49.2936

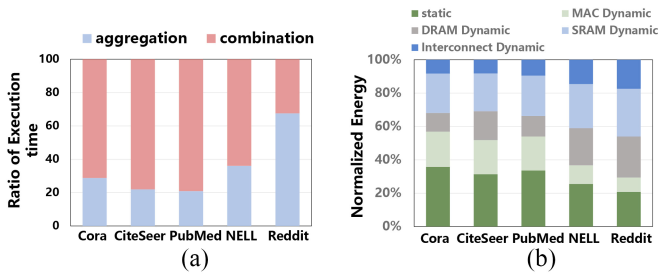


Fig. 13. Breakdown of (a) execution time for the aggregation and combination and (b) energy breakdown for GCNim.

D. Execution Time and Energy Breakdown

To gain a deeper understanding of the effectiveness of our design, we conducted an evaluation that included the decomposition of execution time and energy consumption at different phases.

Latency Breakdown: Fig. 13(a) illustrates the decomposition of the execution time ratio for the two phases of GCN. The findings indicate that GCNim's main performance advantage comes from significantly reducing latency in aggregation. Except for the RD graph, the combination phase exhibits a higher proportion in the remaining datasets.

Energy Breakdown: Fig. 13(b) depicts the detailed decomposition of energy consumption for GCNim. We estimate energy consumption mainly from five main factors: 1) static; 2) MAC dynamic; 3) DRAM dynamic; 4) SRAM dynamic; and 5) interconnect dynamic. For small graphs, the energy consumption of static and SRAM dynamics is relatively high. With the graph size expanding, the overhead from interconnections grows, leading to a decrease in the proportion of energy consumption of the computing units.

E. Scalability

Previous works [43], [44], [45] have also studied the architecture of multiple 3D-stacked memory interconnects. The cubes can be connected to other devices or each other, with external bandwidth between cubes reaching 320 GB/s. We measured the scalability of GCNim as the number of cubes increased, as shown in Fig. 14. When the number of cubes is small, all graphs show good scalability with near-linear expansion. However, when the number of cubes increases to more than 8, the performance growth of small graphs slows down while large graphs continue to perform well. When the

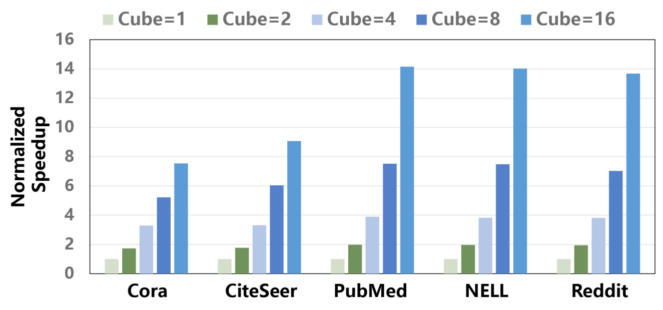


Fig. 14. Scalability of GCNim with the increase of the number of cubes.

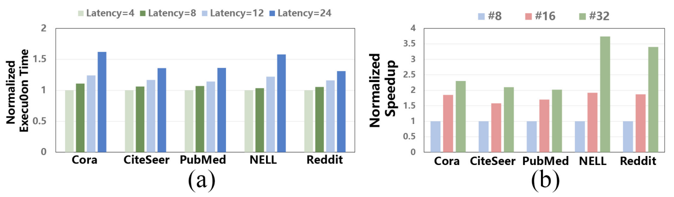


Fig. 15. Sensitivity of performance to (a) TSV transfer latency and (b)number of MAC.

number of cubes increases to 16, our architecture still shows a nearly 14× speedup compared to a single cube.

Our design exhibits good scalability on large graphs because we avoid remote random access during execution. In the combination phase, the weight matrix required for PE is obtained mainly from the local cube and read from the local vault. During the aggregation phase, the latency of sending remote feature vectors can be hidden through computation, avoiding computational stagnation caused by vertex aggregation randomly accessing remote data.

F. Sensitivity to Hardware Parameters

We conducted a sensitivity study to examine the impact of a subset of architectural parameters on inference latency.

The vast majority of data requests and accesses occur in local vaults, making the transfer latency of TSVs crucial. We study the sensitivity of TSV delay in 3D-stacked memory by setting the data transmission delay in the simulator. Fig. 15(a) shows the trend of increasing execution time for different datasets as TSV latency increases. When the transmission delay of TSV increases from 4 to 8, the actual execution time increases by only about 6%. When the delay increases to 12, the execution time increases by about 18%. When the delay increases to 24, the execution time increases by an average of 40%, with a maximum rise of 60% in some graphs. This indicates that when the TSV transmission capacity is strong, the data transmission can overlap with calculation, and when the delay increases, the time spent waiting for data will increase. However, due to our three-stage pipelining, our architecture design remains resistant to high latency.

On the other hand, different datasets have different lengths of feature vectors, and changes in the number of EUs can have different impacts. Fig. 15(b) showcases the performance outcomes obtained by augmenting the number of multipliers and adders. The results showed that when the number of logical components increased from 8 to 16, the average acceleration ratio was 1.784×. When the number continued to expand to 32, it only increased by an average of 1.523× compared to 16. This is because a multiplier and an adder process one feature element at a time. In the case of small graphs, the length of the output feature vector is usually less than or equal to 16. Therefore, the acceleration is insignificant when the number of computing components increases to 32. However, for large graphs, the output feature vector expands to a point where augmenting the number of logical components becomes advantageous.

VII. Related Work

GCNs Accelerators: In recent years, numerous efficient architectures have been proposed to accelerate GCNs. Among them, HyGCN [15] proposes a hybrid engine accelerator. It elucidates the necessity of GCNs accelerators and discusses the distinctive critical features of the two phases of GCNs inference. AWB-GCN [26] relies on the column-wise product execution method and explores the impact of execution order. It transforms GCNs inference into SpMM and employs various dynamic load-balancing strategies. I-GCN [18] adopts a novel online graph reordering algorithm, Islandization, to improve data locality and minimize repetitive calculations. GCNAX [16] proposes a flexible GCNs dataflow to maximize the utilization of computing engines and minimize data movement. Additionally, GCoD [46] introduces a co-design framework that requires retraining GCNs to obtain dense and sparse regions amenable to acceleration. FlowGNN [40] is the first generic and flexible accelerator framework for a wide range of GNNs. PEADL [41] is a power-efficient accelerator for GCNs inference supporting multiple dataflows.

PIM Accelerators Related to Graph Processing: Many PIM-based 3D-stacked memory graph processing accelerators have typically integrated digital logic units into the logic layer. Tesseract [42] represents the first graph accelerator based on 3D-stacked memory, a scalable PIM accelerator used for extensive graph computation. GraphPIM [47] demonstrates that PIM's key performance advantage in graph processing is reducing atomic overheads by offloading expensive atomic operations into 3-D memory with an extended minor architecture. GraphP [45] is a software and hardware co-design accelerator that proposes a graph partition method and further optimizes communication between cubes. GraphQ [44] is an enhanced PIM-based graph processing architecture that achieves static and structured communication through batched communication orders and simplified processing models, fundamentally eliminating irregular data movements. Another exploration is integrating processing units near the bank of the DRAM layer. GCIM [23] is the first accelerator to utilize bank-level processing of GCNs. It leverages the unique characteristics of the aggregation and combination phase to perform at both the logic and DRAM layers. SpaceA [22] is a customized accelerator for SpMV that integrates PEs near the banks to utilize bank-level bandwidth. In a vertex-centric paradigm, the graph algorithm can be represented in numerous rounds of SpMV.

VIII. Conclusion

In this article, we propose GCNim, an accelerator designed for GCNs on the 3D-stacked memory PIM architecture. GCNim adopts a novel GCNs computational model based on the PIM architecture, which employs different multiplication methods in the aggregation and combination phases of GCNs, enabling remote merging and pipelined interphase fusion. This approach significantly reduces data movement within and between stages. Additionally, GCNim integrates unified PEs at the bank level, concurrently supporting alternating computations of aggregation and combination kernels, thereby obliterating load imbalances caused by dynamic workload variations between phases. Our experiments demonstrate that GCNim exhibits superior performance and energy efficiency compared to state-of-the-art CPU, GPU, and accelerator solutions.

References

[1] A. Voulodimos, N. Doulamis, A. D. Doulamis, and E. Protopapadakis, "Deep learning for computer vision: A brief review," *Comput. Intell. Neurosci.*, vol. 2018, pp. 1–13, 2018, Art. no. 7068349.

[2] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 779–788.

[3] S. Vashishth, "Neural graph embedding methods for natural language processing," 2019, *arXiv:1911.03042*.

[4] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 770–778.

[5] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. Conf. Neural Inf. Process. Syst.*, 2012, pp. 1106–1114.

[6] A. Lerer et al., "PyTorch-BigGraph: A large-scale graph embedding system," 2019, *arXiv:1903.12287*.

[7] T. Hamaguchi, H. Oiwa, M. Shimbo, and Y. Matsumoto, "Knowledge transfer for out-of-knowledge-base entities: A graph neural network approach," 2017, *arXiv:1706.05674*.

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

[9] A. García-Durán and M. Niepert, "Learning graph representations with embedding propagation," 2017, *arXiv:1710.03059*.

[10] H. Wang, L. Dong, and M. Sun, "Local feature aggregation algorithm based on graph convolutional network," *Front. Comput. Sci.*, vol. 16, no. 3, 2022, Art. no. 163309.

[11] D. Duvenaud et al., "Convolutional networks on graphs for learning molecular fingerprints," in *Proc. Conf. Neural Inf. Process. Syst.*, 2015, pp. 2224–2232.

[12] A. Fout, J. Byrd, B. Shariat, and A. Ben-Hur, "Protein interface prediction using graph convolutional networks," in *Proc. Conf. Neural Inf. Process. Syst.*, 2017, pp. 6530–6539.

[13] H. Dai, Z. Kozareva, B. Dai, A. J. Smola, and L. Song, "Learning steady-states of iterative algorithms over graphs," in *Proc. Int. Conf. Mach. Learn.*, 2018, pp. 1114–1122.

[14] M. Yan et al., "Alleviating irregularity in graph analytics acceleration: A hardware/software co-design approach," in *Proc. Annu. IEEE/ACM Int. Symp. Microarchit.*, 2019, pp. 615–628.

[15] M. Yan et al., "HyGCN: A GCN accelerator with hybrid architecture," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2020, pp. 15–29.

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

[17] Y. Huang et al., "Accelerating graph convolutional networks using crossbar-based processing-in-memory architectures," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2022, pp. 1029–1042.

[18] T. Geng et al., "i-GCN: A graph convolutional network accelerator with runtime locality enhancement through Islandization," in *Proc. Annu. IEEE/ACM Int. Symp. Microarchit.*, 2021, pp. 1051–1063.

[19] S. H. Lee et al., "Hardware architecture and software stack for PIM based on commercial DRAM technology: Industrial product," in *Proc. ACM/IEEE Annu. Int. Symp. Comput. Archit.*, 2021, pp. 43–56.

[20] M. Lenjani et al., "Fulcrum: A simplified control and access mechanism toward flexible and practical in-situ accelerators," in *Proc. Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2020, pp. 556–569.

[21] S. Li, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie, "DRISA: A DRAM-based reconfigurable in-situ accelerator," in *Proc. Annu. IEEE/ACM Int. Symp. Microarchit.*, 2017, pp. 288–301.

[22] X. Xie et al., "SpaceA: Sparse matrix vector multiplication on processing-in-memory accelerator," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2021, pp. 570–583.

[23] J. Chen et al., "GCIM: Toward efficient processing of graph convolutional networks in 3D-stacked memory," *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 41, no. 11, pp. 3579–3590, Nov. 2022.

[24] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Proc. Conf. Neural Inf. Process. Syst.*, 2017, pp. 1024–1034.

[25] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" in *Proc. Int. Conf. Learn. Rep.*, 2019, pp. 1–17.

[26] T. Geng et al., "AWB-GCN: A graph convolutional network accelerator with runtime workload Rebalancing," in *Proc. Annu. IEEE/ACM Int. Symp. Microarchit.*, 2020, pp. 922–936.

[27] J. Chen, G. Lin, J. Chen, and Y. Wang, "Towards efficient allocation of graph convolutional networks on hybrid computation-in-memory architecture," *Sci. China Inf. Sci.*, vol. 64, no. 6, 2021, Art. no. 160409.

[28] M. Wang et al., "Deep graph library: Towards efficient and scalable deep learning on graphs," 2019, *arXiv:1909.01315*.

[29] N. Srivastava, H. Jin, J. Liu, D. Albonesi, and Z. Zhang, "MatRaptor: A sparse-sparse matrix multiplication accelerator based on row-wise product," in *Proc. Annu. IEEE/ACM Int. Symp. Microarchit.*, 2020, pp. 766–780.

[30] C. Weis, N. Wehn, I. Loi, and L. Benini, "Design space exploration for 3D-stacked DRAMs," in *Proc. Design, Autom. Test Eur.*, 2011, pp. 1–6.

[31] X. Qian, "Graph processing and machine learning architectures with emerging memory technologies: A survey," *Sci. China Inf. Sci.*, vol. 64, no. 6, 2021, Art. no. 160401.

[32] K. Li, W. Yang, and K. Li, "Performance analysis and optimization for SpMV on GPU using probabilistic modeling," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 1, pp. 196–205, Jan. 2015.

[33] N. Sedaghati, T. Mu, L.-N. Pouchet, S. Parthasarathy, and P. Sadayappan, "Automatic selection of sparse matrix representation on GPUs," in *Proc. ACM Int. Conf. Supercomput.*, 2015, pp. 99–108.

[34] P. Fang et al., "An efficient memory data organization strategy for application-characteristic graph processing," *Front. Comput. Sci.*, vol. 16, no. 1, 2022, Art. no. 161607.

[35] (Micron Tech., Inc., Boise, ID, USA). *Hybrid Memory Cube Specification 2.1.* (2015). [Online]. Available: https://www.nuvation.com/sites/default/files/Nuvation-Engineering-Images/Articles/FPGAs-and-HMC/HMC-30G-VSR_HMCC_Specification.pdf

[36] R. Hadidi, B. Asgari, B. A. Mudassar, S. Mukhopadhyay, S. Yalamanchili, and H. Kim, "Demystifying the characteristics of 3D-stacked memories: A case study for hybrid memory cube," in *Proc. IEEE Int. Symp. Workload Charact.*, 2017, pp. 66–75.

[37] K. Chen, S. Li, N. Muralimanohar, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, "CACTI-3DD: Architecture-level modeling for 3D die-stacked DRAM main memory," in *Proc. Design, Autom. Test Eur. Conf. Exhibit.*, 2012, pp. 33–38.

[38] W. Hu et al., "Open graph benchmark: Datasets for machine learning on graphs," in *Proc. Conf. Neural Inf. Process. Syst.*, 2020, pp. 22118Ű-22133.

[39] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch geometric," 2019, *arXiv:1903.02428*.

[40] R. Sarkar, S. Abi-Karam, Y. He, L. Sathidevi, and C. Hao, "FlowGNN: A dataflow architecture for real-time workload-agnostic graph neural network inference," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2023, pp. 1099–1112.

[41] Y. Chen, A. Khadem, X. He, N. Talati, T. A. Khan, and T. Mudge, "PEDAL: A power efficient GCN accelerator with multiple DAtafLows," in *Proc. Design, Autom. Test Eur. Conf. Exhibit.*, 2023, pp. 1–6.

[42] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proc. Annu. Int. Symp. Comput. Archit.*, 2015, pp. 105–117.

[43] X. Zhang, S. L. Song, C. Xie, J. Wang, W. Zhang, and X. Fu, “Enabling highly efficient capsule networks processing through a PIM-based architecture design,” in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2020, pp. 542–555.
[44] Y. Zhuo et al., “GraphQ: Scalable PIM-based graph processing,” in *Proc. Annu. IEEE/ACM Int. Symp. Microarchit.*, 2019, pp. 712–725.
[45] M. Zhang et al., “GraphP: Reducing communication for PIM-based graph processing with efficient data partition,” in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2018, pp. 544–557.
[46] H. You, T. Geng, Y. Zhang, A. Li, and Y. Lin, “GCoD: Graph convolutional network acceleration via dedicated algorithm and accelerator co-design,” in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2022, pp. 460–474.
[47] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, “GraphPIM: Enabling instruction-level PIM offloading in graph computing frameworks,” in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2017, pp. 457–468.

Runze Wang is currently pursuing the Ph.D. degree with the School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China.

His current research interests include graph processing and processing-in-memory.

Ao Hu received the B.S. degree from the Huazhong University of Science and Technology, Wuhan, China, in 2021, where he is currently pursuing the M.S. degree with the School of Computer Science and Technology.

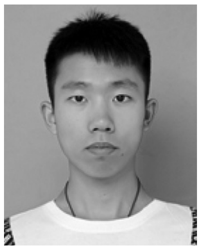
His research interests focus on processing-in-memory architecture and hypergraph processing.

Long Zheng (Member, IEEE) received the Ph.D. degree from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 2016.

He is an Associate Professor with the School of Computer Science and Technology, HUST. His current research interests include program analysis, runtime systems, and heterogeneous computing with a particular focus on graph processing.

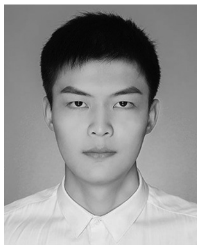
Qinggang Wang received the Ph.D. degree from the Huazhong University of Science and Technology, Wuhan, China, in 2023.

He is currently working as a Postdoctoral Fellow with Zhejiang Laboratory, Hangzhou China. His current research interests include graph processing and reconfigurable computing.



Jingrui Yuan is currently pursuing the Ph.D. degree with the School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China.

His current research interests include techniques and applications of graph processing.



Haifeng Liu received the B.E. degree in computer science from Wuhan University, Wuhan, China, in 2020. He is currently pursuing the Ph.D. degree with the School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan.

His current research interests include the techniques and applications of in/near-memory processing.

Linchen Yu received the Ph.D. degree in computer science from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 2012.

She is currently an Associate Professor with the School of Cyber Science and Engineering, HUST. Her research interests include graph processing and system security.

Xiaofei Liao (Member, IEEE) received the Ph.D. degree in computer science and engineering from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 2005.

He is currently a Professor with the School of Computer Science and Technology, HUST. His research interests are in the areas of system virtualization, system software, and cloud computing.

Hai Jin (Fellow, IEEE) received the Ph.D. degree in computer engineering from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 1994.

He is a Chair Professor of Computer Science and Engineering with HUST. He worked with The University of Hong Kong, Hong Kong, from 1998 to 2000, and as a Visiting Scholar with the University of Southern California, Los Angeles, CA, USA, from 1999 to 2000. He has coauthored more than 20 books and published over 900 research papers. His research interests include computer architecture, parallel and distributed computing, big data processing, data storage, and system security.

Dr. Jin was awarded a German Academic Exchange Service Fellowship to visit the Technical University of Chemnitz in Germany in 1996. He was awarded the Excellent Youth Award from the National Science Foundation of China in 2001. He is a Fellow of CCF and a Life Member of ACM.