GoPIM: GCN-Oriented Pipeline Optimization for PIM Accelerators

Siling Yang[1,2,3,4], Shuibing He[1,2,3,4], Wenjiong Wang[1,2,3,4], Yanlong Yin[1], Tong Wu[1,2,3,4], Weijian Chen[1,2,3,4], Xuechen Zhang[5], Xian-He Sun[6], and Dan Feng[7,8]
[1]The State Key Laboratory of Blockchain and Data Security, Zhejiang University
[2]Zhejiang Lab, [3]Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security
[4]Zhejiang Key Laboratory of Big Data Intelligent Computing, [5]Washington State University Vancouver
[6]Illinois Institute of Technology, [7]Huazhong University of Science and Technology
[8]Wuhan National Laboratory for Optoelectronics
{slingzjunet, heshuibing, wenjiongwang, yinyanlong, wutong, weijianchen}@zju.edu.cn,
xuechen.zhang@wsu.edu, sun@iit.edu, dfeng@hust.edu.cn
* Corresponding Author: Shuibing He (heshuibing@zju.edu.cn)

***Abstract*—Graph convolutional networks (GCNs) are popular for a variety of graph learning tasks. ReRAM-based processing-in-memory (PIM) accelerators are promising to expedite GCN training owing to their in-situ computing capability. However, existing accelerators can be severely underutilized even with pipelines, due to the oversight of the skewed execution times of various GCN stages and the ignorance of skewed degrees of graph vertices. In this work, we propose GoPIM, a GCN-oriented pipeline optimization for PIM accelerators to expedite GCN training. First, GoPIM proposes an ML-based scheme that allocates crossbar resources to the most needed stages to streamline the overall pipeline. Second, GoPIM utilizes a selective vertex updating technique that evenly distributes vertices on crossbars by interleaved mapping. These techniques collectively reduce the overall execution time without losing much accuracy. We also provide a practical architecture design for GoPIM. Our experimental results show that, GoPIM achieves up to 191× speedup and 16.1× energy saving, compared to the state-of-the-art work.**

I. Introduction

Graph convolutional networks (GCNs) are widely used in node classification [1], [20], [46], link prediction [5], [26], [29], and recommendation [18], [34], [48]. GCNs are excellent at extracting information from non-Euclidean graph-structured data by leveraging multiple layers, each of which consists of a *Combination* stage and an *Aggregation* stage. Recent studies indicate that more than 84% of the total execution time on CPU platforms and over 94% on GPU platforms are dedicated to *Aggregation* stages [55]. This is because that, during *Aggregation* stages, there are lots of irregular memory accesses, causing a performance bottleneck in the system.

To address this issue, ReRAM-based processing-in-memory (PIM) architectures have been proposed for GCN training due to their in-situ computation capacity. Specifically, both *Combination* and *Aggregation* kernels can be expressed as Matrix Vector Multiplication (MVM) operations, which are suitable for crossbar-based computations. Compared to conventional accelerators, PIM architectures have shown remarkable performance and energy efficiency for GCN applications [2], [7], [23], [32], [33], [38], [49], [55].

Existing ReRAM-based GCN accelerators expedite the execution process through intra-vertex and inter-vertex parallelism [23], [55], and intra-batch parallelism [2], [38]. By making multiple micro-batchs that belong to the same batch run in parallel (i.e., intra-batch parallelism), these approaches improve resource utilization and reduce execution time to some extent. However, they do not fully resolve the underutilization issues of ReRAM crossbars, due to the oversight of the skewed execution times of various GCN stages. As seen in Section III-A, in three *Aggregation* stages, the crossbars are idle for over 98.47%, 97.50% and 99.03% of the total training time on average across six datasets. The reason is that *Aggregation* and *Combination* stages' execution times differ significantly, and usually long and short stages cannot overlap well in a pipeline. Moreover, data dependencies exist among stages. For example, one layer's *Combination* must wait for the completion of previous layer's *Aggregation*, and the corresponding crossbars remain idle during such waiting.

One obvious method to address this idle resource issue is to *employ unutilized crossbar resources as replicas*. This enhances pipeline parallelism by allowing distinct computations to perform on different crossbar resources of the same data segment, as done in [2], [23], [42]. However, the resource allocation strategies in these works tend to be inflexible. Pipelayer [42] uses the same number of replicas for all stages, and ReGraphX [2] allocates the crossbars for *Aggregation* and *Combination* stages with the ratio of 2:1. They overlook substantial variations in execution times across different stages. Our profiling reveals that, the time ratio between *Aggregation* and *Combination* stages can range from 888× to 1595× on *products* dataset.

Another method to streamline the GCN execution pipeline is *graph sparsification*, which selects a subset of vertices to update, rather than updating all vertices. However, graph sparsification may not always decrease the total execution time. The reason is that graph vertices show significantly skewed degrees. Existing mapping strategies place the vertices in order of the vertex index [2], [38], so that the vertices mapped on

each crossbar exhibit a random degree distribution, and the reduction in workload resulting from graph sparsification is similarly unpredictable. Consequently, after graph sparsification, while some crossbars experience a notable decrease in workload, others either experience no reduction or see a much smaller decrease. However, the overall updating time is determined by the slowest updating subtask. To give an example on the randomness mentioned above: on the *proteins* dataset, the average degree of vertices mapped to each crossbar ranges from 1.6 to 2266.8, as seen in Section III-A.

To enhance the efficiency of executing GCNs on PIM accelerators, we introduce GOPIM, a GCN-oriented Pipeline Optimization for PIM accelerators, focuses on streamlining pipeline execution to improve performance. Its effectiveness hinges on *two core innovations*.

First, we propose a machine-learning-based replica resource allocation scheme, which takes full account of the characteristics of each stage. Based on the predicted execution time of different stages, resources are allocated as needed. Within each stage, these allocated resources are utilized to generate replicas, effectively reducing the execution time of that particular stage. This reduction is tailored to each stage's requirements, ensuring that the overall execution time of the pipeline is minimized.

Second, we devise an adaptive selective vertex updating scheme, which selects the most important vertices and evenly maps them onto the crossbars by interleaved mapping. This reduces the data updating operations for all crossbars and ensures load balance among crossbars, effectively reducing the total update time.

Our major contributions can be summarized as follows:

- Through comprehensive profiling, we discovered the under-utilization of the pipeline and the underlying reasons, such as the disparity in the runtime of different stages.
- We propose a machine-learning-based resource allocation scheme that allocates crossbars to the most needed stages to streamline the overall pipeline.
- We propose an adaptive selective vertex updating method for different graphs, which only updates important vertices and evenly distributes them onto crossbars to reduce the total update time with acceptable accuracy drop.
- We evaluate GOPIM on a range of GCN models and graph datasets. GOPIM outperforms the state-of-the-art SlimGNN-like, ReGraphX, and the ReFlip accelerator by 2.1×, 2.4×, and 45.1× in terms of speedups, and achieves an average energy reduction of 35%, 37%, and 65%, respectively.

II. Background

A. Graph Convolutional Networks (GCNs)

Basic GCN Structure: To process both vertex and edge data within the input graph, GCNs adopt a *Combination-Aggregation* structure, as shown in Figure 1. In this structure, the input graph consists of a series of vertices and edges, represented as vertex feature vectors and an adjacency matrix, respectively. *Combination* stages update vertex features through

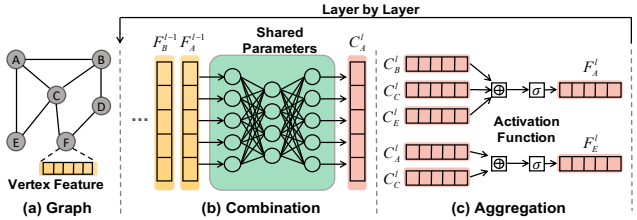


Fig. 1. Illustration of GCN execution process.

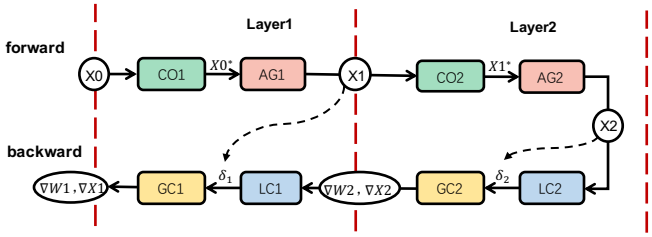


Fig. 2. Illustration of data dependency in GCN training.

a multiplication operation on vertex features and the weight of a Multi-Layer Perceptron (MLP) neural network. *Aggregation* stages aggregate each vertex and its neighbors' feature information by multiplying the adjacency matrix with relevant vertex features [3], [17]. The *Combination-Aggregation* processes are performed layer-wise iteratively. In *Combination* stage, vertex A's features in the $l-1$-th layer (F_A^{l-1}) are updated through linear transformations to C_A^l, as illustrated in Figure 1(b). Then the calculation enters *Aggregation* stage as Figure 1(c) shows. And in this stage, the updated feature information of vertex A and its neighbors (B, C, and E), C_A^l, C_B^l, C_C^l, and C_E^l, are aggregated to a new vertex feature, i.e., A's feature in the l-th layer F_A^l. The *Combination-Aggregation* process in the l-th layer is defined as follows:

$$C_v^l = Combination(F_v^{l-1}, W^l) = F_v^{l-1}\dot{W}^l \quad (1)$$

$$F_v^l = A\dot{C}_v^l \quad (2)$$

where v is the vertices in the graph, C is the output of the *Combination* operation, F is the vertex feature matrix, W is the weight matrix used by the *Combination* operation for linear transformations, and A is the adjacency matrix.

Data Dependencies among Stages: GCN training process comprises forward passes and backward passes [54]. For ease of description, we divide this process into four types of stages: *Combination* (CO), *Aggregation* (AG), loss calculation (LC), and gradient compute (GC). The forward pass contains two stages (i.e., CO and AG) while the backward pass includes another two stages (i.e., LC and GC). Figure 2 illustrates a two-layer GCN training process with eight stages: CO1 → AG1 → CO2 → AG2 → LC2 → GC2 → LC1 → GC1. There is a unidirectional dataflow between stages, such as vertex features ($X0, X1, X2$), loss values ($\delta 1, \delta 2$), and gradient update values (∇W, ∇X). Therefore, the various stages of GCN must be executed in sequence due to data dependencies.

Micro-batch Processing: In deep learning, batch processing and micro-batch processing can affect the parallel granularity of

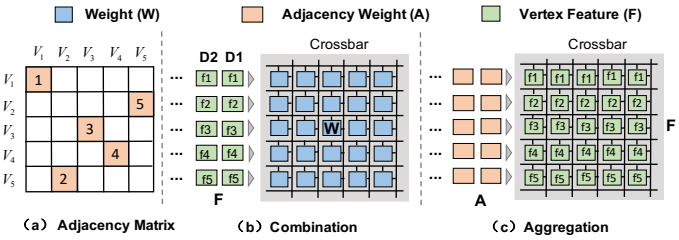


Fig. 3. The basic data mapping strategy for *Combination* stage and *Aggregation* stage of GCN. Note the numbers 1–5 are the logical numbers, and they are transformed into binary numbers on ReRAM cells. F_i denotes the vertex i.

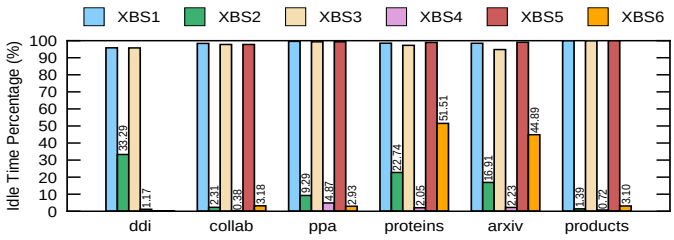


Fig. 4. The comparison of idle time percentage of crossbars for various stages.

model training, which refers to the size of the data processed simultaneously [22]. Batch processing updates the model's weights in each iteration. Micro-batch processing is a variant of batch processing, where each batch is further divided into several micro-batches [22], [36]. The data in each micro-batch are processed individually, and their gradients are accumulated for updating the model's weights. Multiple micro-batches can be processed in parallel, allowing flexible parallel execution strategies and flexible pipelining.

B. ReRAM Crossbars for GCNs

In-situ Computing in ReRAM: The resistive random access memory (ReRAM) is an emerging non-volatile memory with advantages of high density and low power leakage [53]. ReRAM has the ability to facilitate in-situ Matrix Vector Multiplication (MVM) operations [10]. Therefore, ReRAM is naturally suited as an accelerator for GCN training.

For MVM operation, the crossbar cells are programmed with matrix elements in advance. The input of the crossbar is first converted into voltage by digital-to-analog converters (DACs) and then sequentially fed to the wordlines. According to Kirchoff's law, the input voltage is converted into currents and the currents from cells sharing the same bitlines are accumulated. Then, the accumulated values are stored in sample and hold (S&H) circuits and converted to digital representation using analog-to-digital converters (ADC).

Data Mapping Strategy: The data mapping strategy of crossbars is an important factor that affects the performance and energy efficiency of ReRAM-based GCN accelerators. Figure 3 explains a basic mapping strategy. For *Combination* stages, weights (W) are mapped onto the crossbar, with the vertex feature matrix (F) serving as the input. Conversely, for *Aggregation* stages, the adjacency matrix (A) becomes the input, while the crossbar maps the feature matrix (F). This mapping strategy avoids mapping large sparse A onto crossbars, thus avoiding crossbar waste and external data movement overhead of F. When a matrix's size is larger than the crossbar's size, horizontal and vertical tiling extension is used as follows: a long matrix row will be horizontally mapped onto the same row of multiple crossbars; and matrix rows beyond one crossbar's rows will be vertically mapped to new rows provided by other crossbars. REPLIP [23] and GoPIM both use this approach.

C. Graph Sparsification

Graph sparsification can accelerate GCN execution, without significantly altering the structure and functionality of the graph [39]. It reduces computational demands by removing edges that are redundant or less relevant. Existing graph sparsification methods can be categorized into heuristic-based methods and learning-based ones. Heuristic-based methods [41], [43] typically rely on local information of the graph and remove certain edges or nodes to achieve sparsification. Learning-based methods [28], [45], [55] aim to find an optimal sparsification strategy that minimizes information loss during the process.

III. Motivation and Challenges

A. Motivation: The Inefficient Pipeline in ReRAM-based Accelerator

Existing ReRAM-based frameworks (such as ReGraphX [2] and SlimGNN [38]) use simple pipelines to accelerate GNN training. However, their pipelines are inefficient due to under-utilization of crossbars. To thoroughly understand the under-utilization of pipeline design in SlimGNN, we analyze GCN [26] models with six datasets (see detailed experimental setup in Section VII-A) and present the analysis data in Figure 4. It shows the idle time percentage of the crossbars during a forward propagation (XBSi is short for "crossbars for stage i," meaning the crossbars occupied by the GCN model in the i-th stage). *The data not only show that crossbars are heavily under-utilized, but also unveil that the idle time percentage varies largely among stages.* XBS1, XBS3, and XBS5 are the crossbars for mapping *Combination* stage weights, and they are idle for 98.47%, 97.50%, and 99.03% of the total time on average across six datasets, respectively.

This idleness are mainly caused by two reasons. *First, data dependencies exist among stages.* We illustrate it with an example shown in Figure 5. In case (a), for each micro-batch, each stage-2 cannot start until the corresponding stage-1 finishes. *Second, different stages have distinct computation patterns.* Also in Figure 5's case (a), a micro-batch's execution time in stage-1 is much shorter than that in stage-2, causing stage-1's crossbars being idle for long time until stage-2 finishes. In real GCN training, *Combination* stages are much shorter than *Aggregation* stages. Please note, in Figure 5, the communication aspect is not depicted because pipelining overlaps computation and communication [36]. *Inspired by the resource allocation optimization techniques in pipeline [2], we aim to use unoccupied crossbar resources as replicas to*

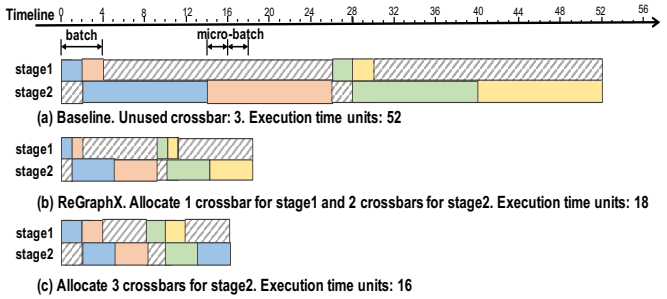


Fig. 5. The comparison of two unused crossbar resource allocation methods. Each interval on the timeline arrow represents one unit of execution time. Each batch includes two micro-batches. Different colors represent the execution time of different micro-batches. Shaded areas represent idle time.

shorten the execution times of longer stages, thus streamlining the overall pipeline.

We also observe that the vertex updating operations—writing mapped vertices onto crossbars (see in Section IV-B)—can be time-consuming. For example, these operations take 52% of the total execution time of AG1 and AG2 on the *ppa* dataset, as shown in Figure 2. *Inspired by graph sparsification approaches that improve performance by training only a subset of edges [55], we also aim to selectively update only a portion of vertices on ReRAM-based PIM architectures to optimize performance.*

B. Challenges

Challenge 1: Existing resource allocation methods lead to suboptimal pipeline performance because they do not consider varying execution times of different stages. For example, ReGraphX [23] allocates resources at a fixed ratio (1:2) between the CO and AG stages. However, this fixed crossbar allocation is suboptimal for different GNN layers, models, and datasets.

Setting different numbers of replicas for different stages can lead to a drastic difference on the overall pipeline execution time. Let's use Figure 5 as an example again. In Figure 5(a), no replicas are created and the pipeline's execution time is 52 time units. Here, we assume that three crossbars are available as unoccupied resources, and the size of data mapped on crossbars of stage 1 and stage 2 is also one. In other words, there are enough resources to create only three replicas of stage 1 or stage 2. In Figure 5(b), ReGraphX allocates crossbars with a fixed ratio of *CO* to *AG*. One crossbar is allocated to stage 1 and two to stage 2. This reduces the execution time for stage 1 by half and for stage 2 by two-thirds, achieving an improvement ratio of approximately 65.4%. The total pipeline execution time is reduced by 34 time units. In Figure 5(c), all three crossbars are allocated to stage 2, further increasing the improvement ratio to ∼69.2%, which surpasses ReGraphX. As a result, the pipeline's total execution time is reduced by 36 time units. This shows that, despite that ReGraphX's resource allocation method is already efficient, there is still potential for improvement. A simple adjustment in the allocation ratio can further enhance performance.

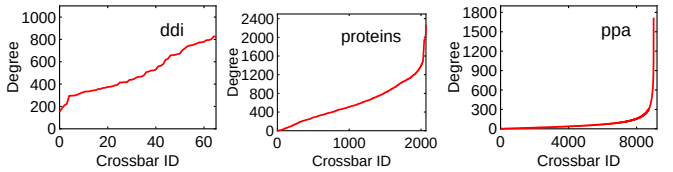


Fig. 6. The average degree of vertices mapped on each crossbar with index-based mapping strategy.

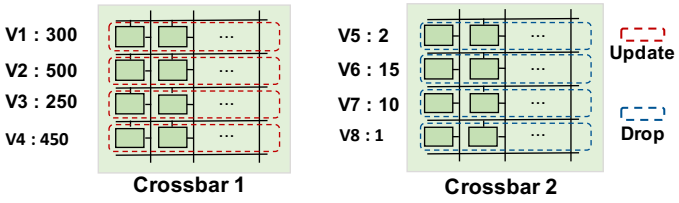


Fig. 7. The selective vertex updating with index-based mapping method (OSU). (V_i : d) indicates that the degree of vertex V_i is d.

The reason behind this effect is that, in GCN training, there is a significant difference in execution time between *Aggregation* and *Combination* stages (up to 888×, with an average of 247×). If the number of replicas is not carefully chosen for each stage, the pipeline execution may not achieve optimality. Please note that Figure 5 is a simplified example with only three replicas to allocate and only two stages to serve. The execution time ratio of stage 1 and stage 2 is 1:6. In real system, there will be more crossbar resources to be allocated [16], [24], [27], and the GCN execution are divided into more stages. In such systems, the execution time ratio of stage 1 and stage 2 could be much smaller than 1:6. Consequently, the improvement ratio could be more pronounced with replicated crossbars.

Challenge 2: Existing mapping strategies for ReRAM-based GCN accelerators, such as ReGraphX and SlimGNN, may negate the latency benefits of selective updating. These frameworks typically use vertex index-based mapping strategies, which place the vertices in order based on their index. However, this approach results in a biased degree distribution of vertices mapped onto each crossbar. Our profiling of the average degree of mapped vertices per crossbar using this strategy, illustrated in Figure 6, reveals significant variations across datasets. For example, in the *ddi*, *proteins*, and *ppa* datasets, the average degrees of mapped vertices per crossbar vary widely, ranging from 151.8 to 827.4, 1.6 to 2266.8, and 1 to 1716.91, respectively.

We name selectively updating part of vertices on ReRAM-based accelerators with index-based mapping strategy as Original Selective Updating (OSU). An example on such challenge is shown in Figure 7. We use vertex degree as the metric to select vertices, because it is an important metric to evaluate the vertex's importance on training [9]. We assume a graph consists of eight vertices (V1–V8) with degrees of 300, 500, 250, 450, 2, 15, 10, and 1, respectively. And we assume each crossbar wordline is large enough for a vertex vector. The default vertex mapping method arranges vertices based on their indices, mapping V1–V4 to the first crossbar and V5–V8 to the second. Crossbar 1 and 2 can process in parallel,

but ReRAM writing operations within the same crossbar are serial [4], [31], [51]. Thus, without sparsification, updating these eight vertices on the ReRAM accelerator takes 4 cycles. After enabling sparsification, following the default stragety of selecting vertices with the highest degree for updating, OSU chooses to update only V1–V4, which still takes 4 cycles. The total updating time is not reduced at all, since crossbar 1 does not receive any workload reduction.

IV. Overview of GoPIM

A. Architecture

GoPIM's architecture is shown in Figure 8. It comprises a 16GB ReRAM array with multiple Tiles, a 128-KB Global Buffer, an SRAM-based *Weight Manager*, an *Activation Module*, a set of *Adder*s, and a *Central Controller*. Specially, we integrate intra-batch and inter-batch pipeline parallelism [36] into the system running on ReRAM-based accelerators to enable faster GCN training. Then, we design an ML-based resource allocation through a *Time Predictor* and a *Resource Allocator*. We also design an interleaved mapping with adaptive selective updating scheme (ISU) to compress the execution time of each training stage.

(1) *Tile:* Each tile consists of 8 PEs, and each PE contains 32 ReRAM crossbars, an input register, an output register, ADCs, DACs, S&Hs, and S+As. The ReRAM tiles are connected through adders and pipeline bus to support the inter-tile data *Aggregation* and transmission.

(2) *Central Controller*: The controller controls the dataflow of GCN training between the off-chip memory and the GoPIM chip. To overlap the latency of off-chip memory accesses, input data are prefetched to the on-chip global buffer. Output results are written back to the off-chip memory in batches to enforce sequential memory accesses.

(3) *Weight Manager*: This unit is made of SRAM and it calculates the weight gradient during the backward propagation. We opt for SRAMs rather than ReRAMs mainly because that there are frequent weight updates in the backward phase and SRAM has faster update speed than ReRAM. Also, SRAM has better endurance, which better suits the frequent surge of writes. SRAM can write 10^{16} times while ReRAM can write 10^8 times during their lifetime.

(4) *Activation Module*: Besides MVM operations, GCN training also includes activation operations. We choose to use the ReLU activation function [11] and integrate an activation module on the GoPIM chip to perform these operations.

(5) *Time Predictor* and *Resource Allocator*: These two components run on the CPU side. First, GoPIM predicts the execution time (without using replicas) of each stage of each layer in the GCN training process based on relevant information derived from the model and dataset. Then, the *Resource Allocator* allocates crossbar replicas for each stage proportionally based on the predicted execution time of the intra-layer *Combination* and *Aggregation* within the constraints of limited crossbar hardware resources.

(6) *ISU Data Mapper:* The interleaved mapping with selective updating scheme runs on the CPU side. First, it determines which vertices to update during each epoch based on their significance. Then it maps the selected vertices on different crossbars in an interleaved way to realize efficient vertex updating. More details are shown in Section VI.

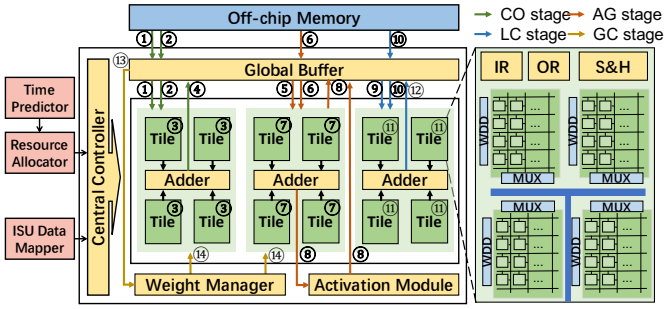


Fig. 8. The architecture and dataflow of GoPIM.

B. Dataflow

We illustrates the dataflow for ReRAM-based accelerator in Figure 8. We will elaborate on them in four types of stages: *Combination* (CO), *Aggregation* (AG), loss calculation (LC), and gradient compute (GC), which are indicated by four colored arrows. **(1) CO**: First, weight matrices are fetched into the global buffer (GB) from off-chip memory (OM), and then transferred to the crossbars (①). Then, GoPIM loads the vertex features to GB from OM (②), and it executes the MVM for vertex feature transformation computation (③). Subsequently, the transformed vertex features are updated to GB (④). **(2) AG**: For *Aggregation* stage next to *Combination*, the latest version vertex features are loaded from the GB to the crossbars (⑤). Next, the adjacency matrix is loaded from OM (⑥) involving MVM operations for *Aggregation* (⑦). The results are handled with *Adder*s and sent to the *Activation Module* to transform the intermediate result, which are then moved to GB (⑧). **(3) LC**: In backward of the training, we adopt the same computation mode as in [40], which reads error from the layer $l+1$ as input to compute the error of layer l. The dataflow of LC stage is the same as the CO stage (⑨ – ⑫). Specifically, the input and output of crossbars are errors of adjacent two layers, while multiple crossbars store weights, vertex features and adjacency matrix to perform MVM operations in order. **(4) GC**: To calculate the weight gradients of layer l, the error from layer $l+1$ will be loaded from OM to *Weight Manager* to do element-wise multiplication and accumulation with the activations from layer l in a channel-to-channel scheme (⑬). Subsequently, the updated gradients are transmitted to the tiles, where they are used to rewrite the weights and vertex features on the crossbars (⑭). This step is considered part of the data loading process during the CO and AG stages.

V. ML-based Resource Allocation for Pipeline

This section presents the details of the design and implementation of ML-based resource allocation. Section V-A explains how GoPIM estimates the resource demands for each stage. Section V-B explains how GoPIM dynamically manages and allocates resources.

TABLE I
FEATURES USED IN THE TIME PREDICTOR.

Feature	Meaning
R_{IFM}^{CO}	the row number of input matrix IFM for *Combination*
C_{IFM}^{CO}	the column number of input matrix IFM for *Combination*
R_{E}^{CO}	the row number of mapped weight matrix for *Combination*
C_{E}^{CO}	the column number of mapped weight matrix for *Combination*
R_{A}^{AG}	the row number of adjacency matrix (A) for *Aggregation*
C_{A}^{AG}	the column number of adjacency matrix (A) for *Aggregation*
R_{E}^{AG}	the row number of mapped feature matrix for *Aggregation*
C_{E}^{AG}	the column number of mapped feature matrix for *Aggregation*
s	the sparsity of the graph
k	the current layer

A. ML-based Execution Time Prediction

The execution time of each stage is the basis for allocating the crossbar resources. Execution Time Predictor needs to estimate the execution time accurately based on the workload characteristics.

Inefficiency of Existing Approaches: Existing approaches have utilized profiling-like methods to estimate execution times during online training processes [35]. However, the profiling technique is time-consuming (e.g., 1688.9 seconds for one profiling on *ppa* dataset). Further, with users submitting diverse models, datasets, vertex dimensions, and hardware configurations, it's infeasible to profile every scenario.

Key Idea: To avoid the high time overhead for profiling and obtain accurate predictions for each stage of every layer, we employ a machine learning approach. Specifically, for the user-submitted workload, we extract the model's architecture and dataset features as input. Then we use a pre-trained and lightweight machine learning model (MLP) to rapidly predict the execution times of each stage after deploying the workload on a crossbar-based PIM accelerator.

The Input Feature Extraction of Workloads: Input features significantly affect the predictor model. We identify the parameters associated with computation. Empirically, the latency of each stage correlates with the dimensions of matrices involved in the MVM operations. For instance, in *Combination* stages, careful consideration should be given to the feature dimensions of weight matrices and input matrices, including both their row and column numbers. Subsequently, we conduct ablation studies, sequentially eliminating one feature at a time, and monitor significant decrease in accuracy. In other words, if the exclusion of some feature causes a large drop in the predictor's accuracy, then we need to keep that feature in the model. Eventually, we choose ten features as the model input, as listed in Table I.

The Predictor Structure: One challenge is to choose a lightweight yet accurate learning-based model. To address this, we evaluate all the regression models available in the scikit-learn library and selecte the top five performing models for our analysis. To benchmark our ML-based predictor, we compare it with these top-performing models. Additionally,

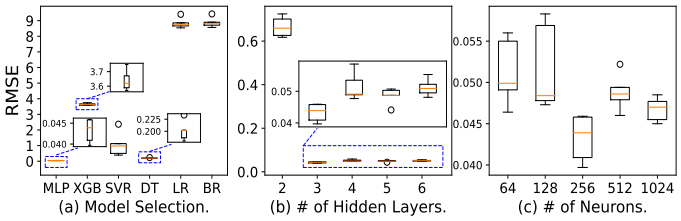


Fig. 9. The comparison of RMSE for different learning-based models. The smaller the RMSE, the better of the model. XGB, SVR, DT, LR, and BR are short for XGBoost, Support Vector Regression, Decision Tree, Logistic Regression, and Bernoulli Regression, respectively.

we include a variety of Multilayer Perceptrons (MLPs) in our comparison. To assess the performance, we utilize the root mean squared error (RMSE) as our primary evaluation metric, consistent with established practices as referenced in [19], [47]. As shown in Figure 9(a), the MLP model outperformes other learning-based approaches. We then systematically vary the number of layers in the MLPs from two to six, as detailed in Figure 9(b). Our findings indicate that a three-layer MLP provides superior performance. Following this, we vary the number of neurons in the hidden layer of the three-layer MLP, with the results depicted in Figure 9(c). It indicates that a configuration with 256 neurons in the hidden layer was the most effective. Therefore, we select a three-layer MLP model as our execution time predictor. This model consists of 10 neurons in the input layer, 256 neurons in the hidden layer, and a single neuron in the output layer.

Training Data Generation and Model Evaluation: The quality of datasets greatly affects the prediction accuracy of the model. We record the execution times of all stages of the workloads on the ReRAM accelerator to create a dataset. To diversify our models, we conduct six workloads for 30 epochs to gather the execution records. Each sample is recorded as $(t_1, t_2, ..., t_n)$ for a model with n stages. We randomly select samples from these six datasets and split them into training and test datasets with an 8:2 ratio. We incrementally increase the number of data samples until we achieve satisfactory prediction accuracy. In our experiments, we find that training with 2,200 samples provides sufficient accuracy. The results show that the RMSE of our predictor is 0.0022 on average.

B. Resource Allocation Strategy

Inefficiency of Existing Approaches: The ultimate goal is to effectively allocate an optimal number of replicated and mapped weight matrices or vertex feature matrices for each *Combination* or *Aggregation* stage, while adhering to hardware resource constraints. Existing methodologies typically employ dynamic programming techniques to make these allocation decisions [27]. However, this approach often incurs significant decision-making time. When dealing with large datasets such as *products*, decisions may take more than 3 days to be finalized.

Key Idea: To avoid long resource allocation time, we propose a max-heap based approach. The key idea is to utilize two max heaps to implement resource allocation strategy based on the greedy algorithm. One max heap stores crossbar resource

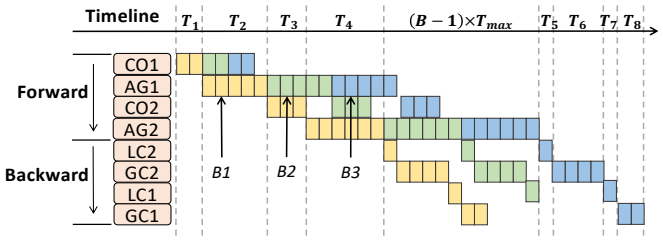


Fig. 10. The pipeline of GCN training on ReRAM-based accelerator.

adjustment values, while the other stores execution durations for all stages. This approach allows GOPIM to quickly identify the stages where adding resource replicas can minimize the total execution time of the pipeline.

Overall Execution Time of the Pipeline: Figure 10 illustrates the pipeline designed for ReRAM-based accelerators in the context of a 2-layer GCN model. For a GCN model with L layers, GOPIM divides it into $4L$ stages. Each layer consists of individual stages: CO, AG, LC and GC. For illustrative purposes, we set the micro-batch size (denoted as B) as 3, with distinct block colors representing three sequential micro-batches of input data. Because of the data dependency of stages within one micro-batch, for the same micro-batch, the execution start time of i-th stage ($T_i^j(start)$) is not earlier than the end time of the $(i-1)$-th stage ($T_{i-1}^j(end)$). Additionally, for the same group of crossbars, the execution of the j-th micro-batch must not occur prior to the execution completion of the $(j-1)$-th micro-batch ($T_i^{j-1}(end)$). B represents the number of total micro-batches. The total execution time (T_A) of the pipeline can be expressed as:

$$T_i^j(start) \geq T_i^{j-1}(end) \tag{3}$$

$$T_i^j(start) \geq T_{i-1}^j(end) \tag{4}$$

$$T_{max}^j = \max(T_i^j), i = 1, 2, 3, \cdots, 4L \tag{5}$$

$$T_A = \sum_{i=1}^{4L}(T_i^j) + (B-1) \times T_{max}^j \tag{6}$$

Resource Allocation Algorithm: To efficiently allocate crossbar resources during training, we employ a max heap-based greedy algorithm. We establish two max heaps, H_v and H_p, where each node is a key-value pair. The keys represent the adjusted values of each stage's replica number and execution time duration, respectively. The values within both heaps denote the index of the respective stage. When unused crossbar resources are available, GOPIM augments the replica count for the stage of the top node in H_p. Then, GOPIM checks whether the stage with the highest adjustment value also has the longest execution time. If so, the adjusted values and execution time for this stege are updated, and subsequently, both max heaps are adjusted from top-down. Otherwise, the adjusted value for the top node of H_v is recomputed. Then, GOPIM finds the corresponding node in H_p and decrements its execution time. Next, adjust two heaps top-down. Subsequently, the unused crossbar resources are reduced. GOPIM iterates through the replica allocation process until the available unused crossbar resources are exhausted. A high-level algorithm for the resource allocation strategy is shown in Algorithm 1.

Algorithm 1: Crossbar allocation algorithm

Input: C_{PIM}: the number of unused crossbars;
N: the number of stage;
$P[1, 2, 3, \ldots, n]$: the execution time for N stages;
$X[1, 2, 3, \ldots, n]$: the number of crossbars for one replica of mapped weight or vertex feature matrix N stages;
Output: $R[1, 2, 3, \ldots, n]$: number of replicas for N stages.

```
V[1,2,...,n] ← computeAdjustValue();
Initiate a max heap H_v according to V, H_v.key = V;
  H_v.value is the index of the corresponding stage;
Initiate a max heap H_p according to P, H_p.key = P,
  H_p.value is the index of the corresponding stage;
while C_PIM > 0 do
    v ← H_v.top();
    p ← H_p.top();
    R[v.value] + +;
    if v.value == p.value then
        v.key ← computeAdjustValue();
        p.key ← adjustExcutionTime();
        shiftHeap(H_v, H_p)
    end
    else
        v.key ← computeAdjustValue();
        p ← findNode(H_p, v.value);
        p.key ← adjustExcutionTime();
        shiftHeap(H_v, H_p)
    end
    C_PIM ← updateTotalUnusedCrossbars();
end
return
```

VI. Interleaved Mapping with Selective Updating

A. Key Idea

Selectively updating the vertices can effectively reduce the number of ReRAM write operations. Low-degree vertices, while having less impact on training accuracy, still consume the same update frequencies (or time) as high-degree vertices during training. By moderately reducing the update frequency of these low-degree vertices, we can speed up the training process without significantly compromising accuracy. Vertices are categorized as important or less important based on their degrees, with a predefined adaptive division ratio (see Section VI-C). Important vertices are updated in every epoch, while less important ones are updated every 20 epochs. Besides accuracy, we also need to consider write balance among crossbars, and we adopt an interleaved approach to map vertices. This ensures that all crossbars receive almost the same amount of write/update reduction, so that their execution time can be reduced all together. We refer to this solution as the interleaved mapping with adaptive selective update method (ISU).

B. Interleaved Mapping

The mapping method of ISU is illustrated in Figure 11. Assuming the graph data have a total of N vertices and the size of each crossbar is M, the interleaved mapping strategy works as follows. First, it calculates the degrees of all vertices and

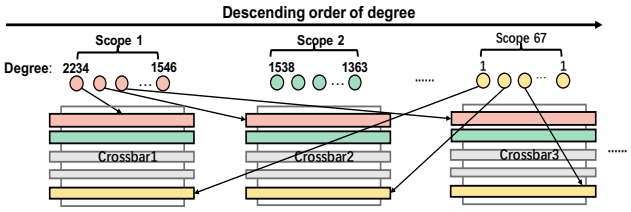


Fig. 11. The mapping strategy of ISU. Take an example for GCN with *ddi* dataset, which includes 4267 vertices. The crossbar size is 64×64.

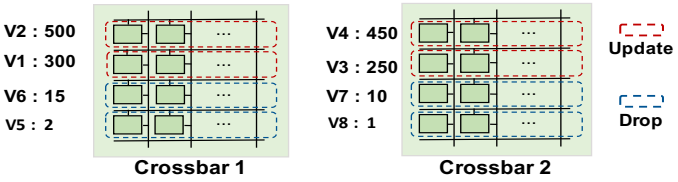


Fig. 12. The illustration of *ISU*.

sorts them in descending order. Then, the reorderd vertices list is divided into K scopes, each containing N/K vertices. Since vertices within the same interval have similar degrees, they can be considered equally or similarly important. The different intervals represent clusters of vertices with varying levels of importance. When vertices are mapped to each crossbar in a round-robin fashion, one vertex from each interval is selected to be assigned to a specific crossbar, and each vertex can only be assigned once. This strategy ensures that each crossbar maps vertices from different intervals, resulting in a more balanced distribution of vertex degrees across all crossbars.

Interleaved mapping, working together with selective updating, not only equalizes the number of update rows for both crossbars, but also ensures that both crossbars drop low-degree non-important vertices, as shown in the example in Figure 12. In this example, four high-degree vertices are categorized into the important group and they are evenly mapped to all crossbars (two vertices on each crossbar). The degrees of the dropped vertices for the two crossbars are 15, 2, 10, and 1, respectively.

C. Adaptive Selective Vertex Updating

Mitigating update frequency within a threshold can effectively decrease execution time. Specifically, GOPIM selectively prioritizes vertex updates, targeting a subset based on their significance—vertices are ranked by their degrees, and only the top θ (a percentage value) are selected for updates. A simple approach would be to use a fixed θ value for all datasets, but it is challenging to determine a proper θ that works across different graphs. For example, using a low θ may yield significant performance gains for dense graphs, as fewer vertices are updated. However, the same low θ may cause a notable drop in accuracy for sparse graphs, as eliminating many useful vertices would be harmful.

To address this, GOPIM employs an adaptive update threshold through the following steps. (1) Accuracy benchmarking: we benchmark the accuray of GOPIM with different θ values. (2) Accuracy analysis: we analyze the impact of different θ values on accuracy, ensuring that the precision loss remains below 1%. Our analysis shows that sparse graphs require a higher threshold to maintain structural integrity, while dense graphs can use a lower threshold due to the higher redundancy of vertex connections. (3) Threshold determination: we determine the optimal θ values for both sparse and dense graphs. For graphs with an average vertex degree of 8 or less (classified as sparse), we set the threshold at 80%. For denser graphs, the update threshold is set at 50% (see Section VII-E).

TABLE II
SPECIFICATIONS OF RERAM-BASED ACCELERATOR SIMULATOR.

Component	Power(mW)	Area(mm2)	Params.	Spec.
PE properties (8 PEs per Tile)				
ADC	64	0.0384	Resolution	8 Bits
			Total	32
DAC	8	0.00034	Resolution	2 Bits
			Total	32×64
S&H	0.02	0.00008	Total	32×64
Crossbar	6.2	0.00051	Bits per Cell	2
			Size	64×64
			Total	32
IR	2.32	0.0038	Size	4KB
OR	0.42	0.0014	Size	512B
S+A	0.8	0.00096	Total	16
Tile properties (65536 Tiles per Chip)				
Tile	654.08	0.36392	Total	65536
Input Buffer	7.95	0.034	Size	32KB
Crossbar Buffer	59.42	0.208	Size	256KB
Output Buffer	1.28	0.0041	Size	4KB
NFU	2.04	0.0024	Total	8
PFU	3.2	0.00192	Total	8
Chip properties				
Weight Computer	99.6	3.21	Width	16 Bits
			Total	1
Activation Module	0.0266	0.0030	Width	16 Bits
			Total	1
Central Controller	580.41	2.65	Total	1

VII. EVALUATION

A. Experimental Setup

Experimental Platform: We implemented GOPIM in a modified PIM-based simulator, NeuroSim [40], to model the proposed pipeline and mapping framework. The simulator models all the microarchitectural characteristics of GOPIM, including the on-chip storage buffers, processing-in-memory (PIM) storage and compute elements, and other peripheral circuits. Table II summarizes the GOPIM configuration, which consists of 65536 tiles, with each tile featuring 8 PEs. Each PE includes 32 ReRAM crossbars organized in a 4×8 matrix layout. Each crossbar is 64×64, and the read and write latencies are 29.31 ns and 50.88 ns, respectively [37]. We utilize the same methods in CACTI [50] and NVSim [13] to estimate latency and energy. We define the crossbar array resource constraint as 16GB, as referred to in [16], [24].

Baselines: We choose the following counterparts, which are the most relevant and recent ReRAM-based GCN accelerators in terms of pipeline resource allocation and graph sparsification. (1) *Serial* represents the sequential execution without pipeline and graph sparsification. (2) *SlimGNN-like* is derived from SlimGNN [38] but without its weight pruning. *SlimGNN-like* optimizes pipeline resource allocation with replicas based

TABLE III
DETAILED INFORMATION OF DATASETS.

Name	Category	Average # Vertices	Average # Edges	Average # Vertex's Deg.	Vertex Feature Dimension
ddi	Link	4267	1334889	500.5	256
collab	Link	235868	1285465	8.2	128
ppa	Link	576289	30326273	73.7	58
proteins	Node	132534	39561252	597.0	8
arxiv	Node	169343	1166243	13.7	128
products	Node	2449029	61859140	50.5	100
Cora	Node	2708	10556	3.9	1433

TABLE IV
GCN MODEL ARCHITECTURE AND TRAINING PARAMETERS.

Name	# Layer	Learning Rate	Dropout	# Input Channels	# Hidden Channels	# Output Channels
ddi	2	0.005	0.5	256	256	256
collab	3	0.001	0	128	256	256
ppa	3	0.01	0	58	256	256
proteins	3	0.01	0	8	256	112
arxiv	3	0.01	0.5	128	256	40
products	3	0.01	0.5	100	256	47
Cora	3	0.005	0.5	256	256	256

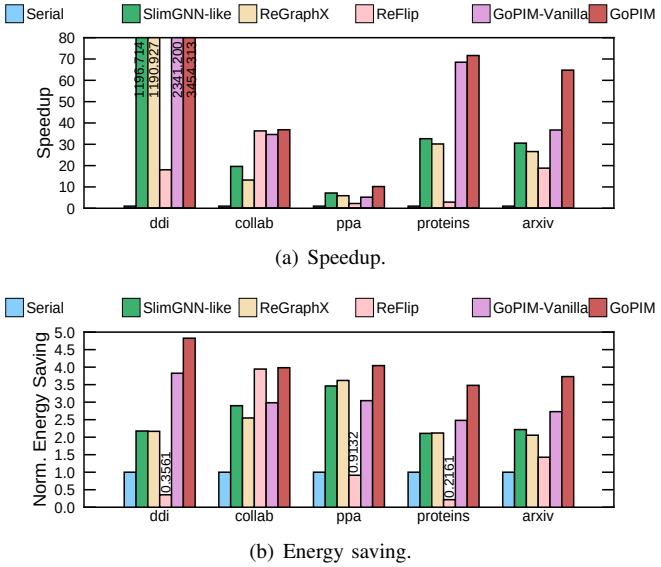
on the space requirements of each stage. It also leverages input subgraph pruning and index-based mapping for graph sparsification. (3) *ReGraphX* sets a fixed resource allocation ratio and it discards graph sparsification; (4) *ReFlip* adopts replicas only in combination phases and without sparsification. (5) To demonstrate the effectiveness of ISU in GOPIM, we also implement GOPIM-Vanilla, which is the version of GOPIM without using ISU. We do not compare CPU/GPU systems because *ReFlip* has outperformed them. For a fair comparison, all accelerators including GOPIM are equipped with the same crossbar resources.

Datasets and Models: Table III presents six datasets from Open Graph Benchmarks (OGB) [21] and one dataset *Cora*, selected to cover two prediction task types. The first three are used for link prediction tasks, and the last four serve node prediction tasks. For each type, we select datasets with three *graph density*: large, medium, and small. The *graph density* is defined as the ratio of the actual number of edges to the maximum possible number of edges in the graph [15]. A denser graph has a higher average number of vertex's degree. We evaluate GOPIM with the most popular GCNs models. The model parameters on different datasets are presented in Table IV. We choose 64 as the micro-batch size by default.

B. Overall Performance and Energy Saving

We first evaluate the performance and energy saving of GOPIM with *Serial*, SlimGNN-like, ReGraphX, and ReFlip in Figure 13. The experimental results do not include the CPU prediction latency. On average, the CPU execution time takes less than 1 second per model, which is negligible compared to the whole GCN training process. The end-to-end execution time and energy consumption are normalized to that of the *Serial* approach.

End-to-End Speedup: Figure 13(a) shows the end-to-end speedup comparison results between five baselines and GOPIM. *First, we can observe that* GOPIM *achieves the best performance among the six schemes for all five datasets.* Specifically, GOPIM achieves 10.2×–3454.3× (727.6× on average), 1.4×–2.9× (2.1× on average), 1.7×–2.9× (2.4× on average), 1.1×–191.4× (45.1× on average), and 1.1×–2.0× (1.5× on average), compared to *Serial*, SlimGNN-like, ReGraphX, ReFlip, and GOPIM-Vanilla, respectively. The observation that GOPIM consistently outperforms GOPIM-Vanilla highlights the effectiveness of ISU.



(a) Speedup.

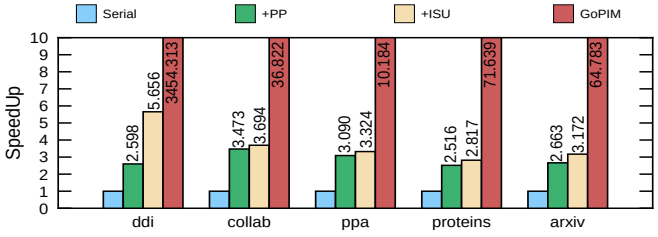
(b) Energy saving.

Fig. 13. The overall performance of GOPIM for GCN with five datasets, compared with baselines.

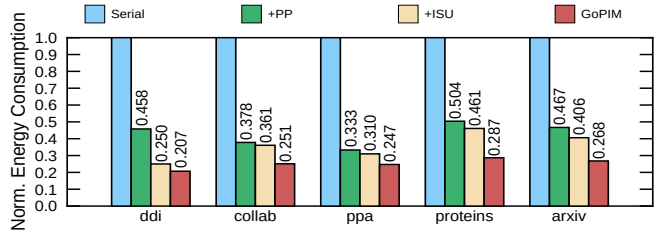
Second, we can observe that the highest speedup is obtained under the case with the dataset that has the smallest number of vertices. Specifically, the *ddi* dataset demonstrates the greatest speedup (i.e., 3454.3×) compared to the other dataset. This is attributed to the fact that *ddi* dataset, with its minimum number of vertices, necessitates fewer crossbars to create a replica, offering more acceleration opportunities for GOPIM.

Third, the dataset with denser graph can provide more opportunities for GOPIM *compared to ReFlip.* The number of vertices in *ppa* surpasses that in *collab*, implying that GOPIM has fewer replicas available to enhance parallelism given the existing amount of unused crossbars. However, despite this limitation, we observe that GOPIM demonstrates a superior speedup on the *ppa* dataset compared to the *collab* dataset. Specifically, GOPIM showcases 4.6× speedups and 1.1× speedups on *ppa* and *collab* datasets, respectively. This can be attributed to the following reasons: (a) ReFlip adopts a hybrid execution model, utilizing the row-major model for high-degree vertices and the column-major model for low-degree ones. The latter execution model leads to the repeated loading of vertices. (b) *ppa* dataset exhibits higher density than *collab* dataset, with the average degree of vertices in being 73.7 and 8.2, respectively. Consequently, ReFlip needs to repeatedly load more source vertices with *ppa*, compared to with *collab*.

Energy Saving: Figure 13(b) presents the energy saving compasion among the five counterparts for the various datasets. *First, we can observe that* GOPIM *achieves the best energy*



(a) Speedup.



(b) Energy Comsumption.

Fig. 14. The performance impact on total execution time and energy consumption with intra-&inter-batch pipeline, ISU, and ML-based resource allocation.

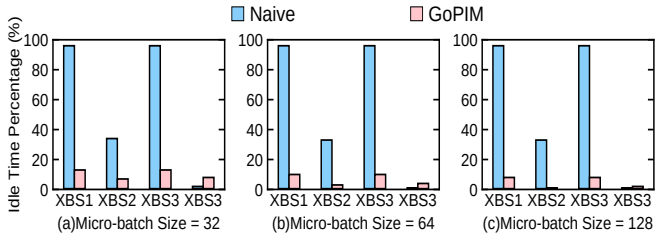


Fig. 15. The idle time percentage of crossbars for various stages with different micro-batches. XBSi is short for "crossbars for stage i," meaning the crossbars occupied by the GCN model in the i-th stage.

TABLE V
THE ACCURACY IMPACT OF GOPIM.

Dataset	***ddi***	***collab***	***ppa***	***proteins***	***arxiv***
GoPIM-Vanilla	39.39%	45.66%	10.87%	71.96%	71.9%
GoPIM	43.40%	45.01%	11.94%	73.58 %	71.7%
Acc Impact	+4.01 %	-0.65 %	+1.07%	+1.62%	-0.2%

saving among the baselines for all datasets. Specifically, compared to *Serial*, GoPIM consumes 4.0× less energy on average across the five datasets, while SlimGNN-like, ReGraphX, ReFlip, and GoPIM-Vanilla achieve average energy savings of 2.6×, 2.5×, 1.4×, and 3.0×, respectively. This is because GoPIM integrates both intra-batch and inter-batch parallelism to shorten the idle time for most crossbars while SlimGNN-like and ReGrahX only support intra-batch pipeline. Additionally, GoPIM uses the ML-based resource allocation strategy to further reducing the idle time for most crossbars. Furthermore, GoPIM uses selective vertex updating strategy to reduce the write operations, which are not supported in other baselines.

Second, we also observe that GoPIM *gains the largest energy saving for ddi dataset.* Specially, compared to *Serial*, GoPIM consumes 4.8×, 4.0×, 4.0×, 3.5×, and 3.7× (4.0× on average) less energy on the five datasets. This is because *ddi* dataset, with its minimal vertex count, requires fewer crossbars for one replica, thereby offering more chances to mitigate crossbar idle time for GoPIM. Additionally, we notice that for *ddi*, *ppa* and *proteins* datasets, ReFlip consumes more energy compared to *Serial*. The reason is that when the product of the number of vertices and the average degree of vertices increases, more vertices are repeatedly loaded onto crossbars. In other words, the number of ReRAM rewrite operations increases.

C. Impact of Individual Techniques

Figure 14 illustrates the impact of each optimization technique within GoPIM on the end-to-end training time and energy consumption across five datasets. The *Serial* version refers to the basic ReRAM-based accelerator for GCN, which operates in a layer-wise and sequential style. The *+PP* variant integrates both intra-batch and inter-batch pipelining to expedite the training process without replicas. The *+ISU* version incorporates an interleaved mapping with selective update (or rewriting) of vertices on the version of *+PP*. GoPIM denotes the version with all optimizations including the ML-based resource allocation strategy and the interleaved mapping with selective update technique.

Figure 14(a) shows *+PP* achieves a speedup of 2.6× over *Serial* on *ddi* dataset. The primary reason is that *+PP* can overlap the stages of each samples in a batch and within the bounded staleness batches. *+ISU* allows *Aggregation* to shorten the loading data duration by decreasing the ReRAM write times. By employing ML-based crossbar replica allocation, GoPIM achieves a speedup of 3472.3×. This is because that using unused crossbars for replica can improve the intra-batch parallelism. Moreover, allocating more replicas to stages with longer durations minimizes idle time in the pipeline.

Figure 14(b) presents the energy savings contributed by each technique. *+PP*, *+ISU*, and GoPIM deliver up to 62%, 75%, and 79% energy reduction, respectively, across all datasets. *+PP* minimizes idle time without additional hardware, maintaining similar power levels. *+ISU* cuts energy use by diminishing ReRAM rewrites during *Aggregation*. Although GoPIM activates more components, its reduction in execution time leads to significant overall energy savings.

D. Effectiveness Analysis

The Idle Time of Crossbars: Figure 15 shows the comparison of idle time percentage of crossbars between $Naive$ and GoPIM. $Naive$ represents the ReRAM-based accelerator with pipelining and vertex index-based mapping strategy. We only present the results on *ddi* for space limitation (we have similar observation on other datasets). We can observe that GoPIM reduces the average idle time percentage of the crossbars for all batch sizes. Specifically, ISU achieves an average reduction of 46.75%, 49.75%, and 51.75% in the crossbars' idle percentage for micro-batch sizes of 32, 64, and 128, respectively. This is because that GOPIM utilizes resource replicas to shorten the duration of longer pipeline stages, thereby minimizing the execution time disparity among different stages. Consequently,

TABLE VI
THE CROSSBAR ALLOCATION DETAILS ON *ddi*. [. . .] DENOTES REPLICA AND CROSSBAR NUMBERS IN STAGES OF A GCN TRAINING PROCESS.

Dataset	Method	Numbers of Replicas	Numbers of Crossbars	Total Crossbars
ddi	Serial	[1, 1, 1, 1, 1, 1, 1, 1]	[32, 534, 32, 534, 32, 534, 32, 534]	2264
	GOPIM	[59, 364, 60, 616, 61, 487, 61, 484]	[1888, 194144, 1920, 328636, 1952, 258146, 1952, 258214]	1046852

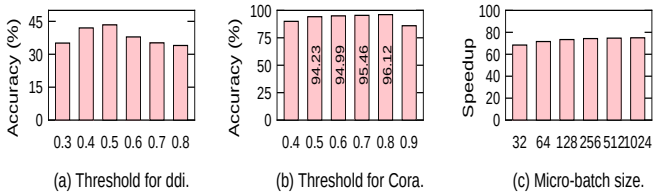


Fig. 16. Evaluation on sensitivity: (a) Accuracy with various selective updating thresholds for dense graph *ddi*; (b) Accuracy with various selective updating thresholds for sparse graph *Cora*; (c) Speedup performance with various batch sizes.

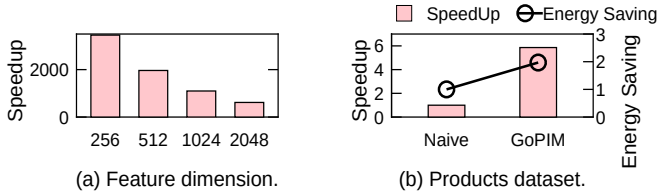


Fig. 17. Evaluation on scalability: (a) The speedups of GOPIM as the dimensions of the vertex features are increased from 256 to 2048; (b) The performance and energy savings on the *products* dataset.

this approach reduces the idle time of the crossbar in each stage.

The Accuracy Impact of ISU: Table V shows the model accuracy impact of GOPIM on various dataset. For *ddi*, *ppa*, and *proteins* datasets, GOPIM improves the accuracy by 4.01%, 1.07%, and 1.62%, respectively. For other datasets, GOPIM reduces the accuracy by 0.2%–0.65%, which is acceptable (commonly an accuracy loss below 1% is acceptable [25]). This is because that ISU makes GOPIM focus on the high-degree vertices' influence to the training and neglects a portion of unimportant low-degree vertices.

The Replica Number Analysis: Table VI presents the details of crossbar resource allocation for each stage. Due to space constraints, we only display the results of *ddi* (similar observations apply to other datasets). A N-layer GCN training process has $4N$ stages (i.e., CO1–AG1, . . . , COn–AGn, LCn–GCn, . . . , LC1–GC1) as mentioned in Section (II). The model on *ddi* uses a 2-layer structure with an 8-stage training process. The *Serial* method maps data to crossbars once per stage, with the number of crossbars determined by parameters like the weight matrix in *Combination* and vertex dimensions in *Aggregation*. GOPIM allocates crossbars for stages within a layer using varying ratios, based on ML-predicted execution times. For example, in the *ddi* model, the crossbar replica ratios for *Combination* and *Aggregation* stages in the first and second layers are 0.162 and 0.097, respectively.

E. Sensitivity Studies

Update Threshold: The update threshold (i.e., θ described in Section VI) can impact the accuracy of GCN model. We study GOPIM's sensitivity to various update thresholds using two datasets: *ddi*, representing dense graphs (average vertex degree > 8), and *Cora*, representing sparse graphs (average vertex degree $<= 8$). Figure 16(a) and Figure 16(b) show that GOPIM achieves the highest accuracy, with an accuracy drop of less than 1%, when θ is set to 50% for *ddi* and 80% for *Cora*. Moreover, GOPIM is not sensitive to the threshold variations within certain ranges. This means that users can select any value within these ranges to maintain both accuracy and performance. Specifically, for dense graphs, setting θ between 40% and 50%, and for sparse graphs, setting θ between 70% and 80%, ensures that the accuracy loss remains below 1% while optimizing performance. These findings highlight the necessity of using an adaptive threshold for different datasets.
Micro-batch Size: We study the impact of the micro-batch sizes on system performance. As Figure 16(b) shows, as the micro-batch size increases, the speedup of GOPIM normlized to *Serial* increases. This is because, while increasing the micro-batch size, the intra-micro-batch pipeline mechanism in GOPIM helps exploit the parallelism of training, which reduces end-to-end time to a greater extent.

F. Scalability Analysis

Dimension Size of Vertex Feature: Figure 17(a) illustrates the speedup outcomes of GOPIM across varying vertex feature dimensions, ranging from 256 to 2048. It is evident that GOPIM consistently demonstrates speedups as the dimension size increases, underscoring its commendable scalability. Nevertheless, the speedups taper off with increasing dimension sizes. This occurs because larger dimensions necessitate more crossbars per replica, leaving less room to exploit the benefits of ML-based resource allocation.
Larger Dataset: We explore the scalability of GOPIM using a larger dataset (i.e., *products* containing 2,449,029 vertices). As shown in Figure 17(b), GOPIM achieves a 5.9× speedup and 1.8× energy saving, compared to *Serial*. Though GOPIM may exhibit diminishing improvements as the dataset size increases due to fewer unoccupied crossbar resources to balance the pipeline, it can be addressed by augmenting the crossbar resources. In addition, GOPIM's selective updating with interleaved mapping method still works well to reduce the overall execution time.
Sparse Dataset: We also evaluate GOPIM on a sparse dataset *Cora*, whose average vertex degree is 3.9. The update threshold θ of ISU is set to 80%. Compared to Serial, SlimGNN-like, ReGraphX, and ReFlip, GOPIM achieves speedups of 3460.5×,

TABLE VII
THE SPEEDUPS (NORMOLIZED TO SERIAL) COMPARISON OF ML AND PROFILING METHOD.

Dataset	*ddi*	*collab*	*ppa*	*proteins*	*arxiv*
ML	3454.31	36.82	10.18	71.64	64.78
Profiling	3469.17	36.82	10.20	71.83	66.20

1.30×, 1.26×, and 1.27×, respectively, along with energy savings of 8%, 3.8%, 3.8%, and 19.5%. Although GOPIM achieves less benefits for sparser graphs due to the fewer dropped vertices for updating, it consistently outperforms all the baselines. Meanwhile, the accuracy loss of GOPIM is 0.28%, which is ignorable. This indicates GOPIM is universal for both dense and sparse graphs. On the other hand, a large number of graphs (e.g., *proteins*, *ddi*, and *ppa*) are denser, where GOPIM can always deliver substantial benefits.

G. Overhead Analysis and Discussion

Time Overhead for Predictor: To train GOPIM's predictor, we need collect the training dataset. The total time for doing this is about 7 days. Then, it takes about 4 minutes to train our predictor. With the trained predictor, it takes only a few milliseconds to predict the execution times of all stages.

Model Generalizability on Unseen Datasets: We train our ML model on five datasets and use the trained model to predict the execution time of each stage of the GCN on the remaining unseen dataset. By comparing the predicted execution time with the actual execution time, we calculate the prediction accuracy. Repeating this process for different unseen datasets, we obtain six sets of experimental results. The results show that GOPIM's ML approach achieves an average prediction accuracy of 93.4% on unseen datasets. This demonstrates that our model has good generalization on unseen datasets.

The Comparison of ML model and profiling: Table VII shows the comparison between the ML approach and the profiling method. We can observe that GOPIM's ML-based approach achieves a speedup comparable to the profiling method, with a maximum difference of only 4.3%, while reducing the time overhead by an average of 94%. This advantage is attributed to ML model's generalizability. In other words, ML model can handle unseen data while profiling cannot. In order to achieve similar performance across multiple datasets, the profiling-based method requires significantly more time (compared to the ML-based method) to gather execution times for each stage in every epoch.

VIII. RELATED WORK

Customized Architectures for GCN: Various crossbar-based processing-in-memory accelerators optimize architecture and dataflow for GCN by vertex data reusing [49], exploring hybrid execution model [23], heterogeneous crossbar size [2], input graph pruning or weight pruning [33], [38], dense data mapping and scheduling strategies [7], edge selection strategy [55]. Instead, in this work, we focus on the learning-based resource allocation and mapping strategies with vertex selective updating.

Exploring Parallelism for GCN Accelerators: Multiple GCN accelerators optimize their performance by leveraging either intra-vertex and inter-vertex parallelism [8], [23], [32], [55], intra-batch parallelism [2], [38], [44], inter-batch parallelism [44], or a *combination* of these techniques [30], [44]. However, either they are not applicable to ReRAM-based accelerators, or they still suffer from issues that fail to fully utilize PIM-based crossbars.In this study, we introduce an ML-based crossbar allocation strategy to mitigate idle time and employ an interleaved mapping strategy with selective updating to achieve a more efficient pipeline.

Removing Redundancy in Graphs: Heuristic-based [41], [43] and model-based [28], [45], [55] graph sparsifications are widely used to accelerate GCN execution by removing less-relevant or redundant edges from graphs. However, the heuristic-based approaches ignore hardware mapping strategies and the model-based methods require extensive times for data collection and model training. In contrast, our approach, GOPIM, incorporates vertex mapping strategies on PIM during graph sparsification. This approach considers both the hardware characteristics of the crossbar and is lightweight.

Pipeline Resource Allocation in DNN vs. GNN: Prior studies accelerate DNN training with optimized resource allocation, such as [6], [12], [14], [52], [56]. Compared to DNN training, GNN training exhibits highly dependent computing mode and longer execution times between stages, making the resource allocation more promising and challenging. GOPIM proposes innovative approaches to maximize resource allocation efficiency in GNN training.

IX. CONCLUSION

GCN is becoming an increasingly important tool for graph learning. In this paper, we propose GOPIM to improve the execution efficiency of GCN on PIM accelerators. GOPIM integrates a machine-learning method to allocate unoccupied resources for streamlining the overall pipeline. GOPIM also employs selective updating of graph vertices with interleaved mapping, to reduce training time and ensure workload balance across all crossbars. GOPIM integrates the two core techniques with intra-batch and inter-batch parallelism to form its architecture design. Experimental results show that the proposed architecture can meet our design goal, achieving up to 191× speedup and 16.1× energy saving.

ACKNOWLEDGMENTS

We sincerely thank the anonymous reviewers for their constructive suggestions. This work was supported in part by the National Key Research and Development Program of China (2023YFB4502100), the National Science Foundation of China (62172361), the Major Projects of Zhejiang Province (LD24F020012), the Open Project Program of Wuhan National Laboratory for Optoelectronics (2023WNLOKF005), and the Pioneer and Leading Goose R&D Program of Zhejiang Province (2024SSYS0002).

REFERENCES

[1] S. Abu-El-Haija, A. Kapoor, B. Perozzi, and J. Lee, "N-gcn: Multi-scale graph convolution for semi-supervised node classification," in *uncertainty in artificial intelligence*. PMLR, 2020, pp. 841–851.

[2] A. I. Arka, J. R. Doppa, P. P. Pande, B. K. Joardar, and K. Chakrabarty, "Regraphx: Noc-enabled 3d heterogeneous reram architecture for training graph neural networks," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2021, pp. 1667–1672.

[3] A. Auten, M. Tomei, and R. Kumar, "Hardware acceleration of graph neural networks," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.

[4] G. W. Burr, R. M. Shelby, S. Sidler, C. Di Nolfo, J. Jang, I. Boybat, R. S. Shenoy, P. Narayanan, K. Virwani, E. U. Giacometti *et al.*, "Experimental demonstration and tolerancing of a large-scale neural network (165 000 synapses) using phase-change memory as the synaptic weight element," *IEEE Transactions on Electron Devices*, vol. 62, no. 11, pp. 3498–3507, 2015.

[5] L. Cai, B. Yan, G. Mai, K. Janowicz, and R. Zhu, "Transgcn: Coupling transformation assumptions with graph convolutional networks for link prediction," in *Proceedings of the 10th international conference on knowledge capture*, 2019, pp. 131–138.

[6] X. Cai, Y. Wang, X. Ma, Y. Han, and L. Zhang, "Deepburning-seg: Generating dnn accelerators of segment-grained pipeline architecture," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2022, pp. 1396–1413.

[7] N. Challapalle, K. Swaminathan, N. Chandramoorthy, and V. Narayanan, "Crossbar based processing in memory accelerator architecture for graph convolutional networks," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 2021, pp. 1–9.

[8] C. Chen, K. Li, X. Zou, and Y. Li, "Dygnn: Algorithm and architecture support of dynamic pruning for graph neural networks," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021, pp. 1201–1206.

[9] J. Chen, Z. Gong, W. Wang, C. Wang, Z. Xu, J. Lv, X. Li, K. Wu, and W. Liu, "Adversarial caching training: Unsupervised inductive network representation learning on large-scale graphs," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 33, no. 12, pp. 7079–7090, 2021.

[10] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "Prime: A Novel Processing-in-memory Architecture for Neural Network Computation in Reram-Based Main Memory," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 27–39, 2016.

[11] W.-L. Chiang, X. Liu, S. Si, Y. Li, S. Bengio, and C.-J. Hsieh, "Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks," in *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, 2019, pp. 257–266.

[12] C. Dong, S. Hu, X. Chen, and W. Wen, "Joint optimization with dnn partitioning and resource allocation in mobile edge computing," *IEEE Transactions on Network and Service Management*, vol. 18, no. 4, pp. 3973–3986, 2021.

[13] X. Dong, C. Xu, Y. Xie, and N. P. Jouppi, "NVSim: A circuit-level performance, energy, and area model for emerging nonvolatile memory," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 7, pp. 994–1007, 2012.

[14] Y. Duan and J. Wu, "Optimizing resource allocation in pipeline parallelism for distributed dnn training," in *2022 IEEE 28th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 2023, pp. 161–168.

[15] M. Ester, H.-P. Kriegel, J. Sander, X. Xu *et al.*, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *kdd*, vol. 96, no. 34, 1996, pp. 226–231.

[16] R. Fackenthal, M. Kitagawa, W. Otsuka, K. Prall, D. Mills, K. Tsutsui, J. Javanifard, K. Tedrow, T. Tsushima, Y. Shibahara *et al.*, "19.7 a 16gb reram with 200mb/s write and 1gb/s read in 27nm technology," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. IEEE, 2014, pp. 338–339.

[17] T. Geng, A. Li, R. Shi, C. Wu, T. Wang, Y. Li, P. Haghi, A. Tumeo, S. Che, S. Reinhardt *et al.*, "Awb-gcn: A graph convolutional network accelerator with runtime workload rebalancing," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 922–936.

[18] X. He, K. Deng, X. Wang, Y. Li, Y. Zhang, and M. Wang, "Lightgcn: Simplifying and powering graph convolution network for recommendation," in *Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval*, 2020, pp. 639–648.

[19] T. O. Hodson, "Root mean square error (rmse) or mean absolute error (mae): When to use them or not," *Geoscientific Model Development Discussions*, vol. 2022, pp. 1–10, 2022.

[20] F. Hu, Y. Zhu, S. Wu, L. Wang, and T. Tan, "Hierarchical graph convolutional networks for semi-supervised node classification," *arXiv preprint arXiv:1902.06667*, 2019.

[21] 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," *Advances in neural information processing systems*, vol. 33, pp. 22 118–22 133, 2020.

[22] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu *et al.*, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," *Advances in neural information processing systems*, vol. 32, 2019.

[23] Y. Huang, L. Zheng, P. Yao, Q. Wang, X. Liao, H. Jin, and J. Xue, "Accelerating graph convolutional networks using crossbar-based processing-in-memory architectures," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2022, pp. 1029–1042.

[24] M. Jagasivamani, C. Walden, D. Singh, L. Kang, S. Li, M. Asnaashari, S. Dubois, D. Yeung, and B. Jacob, "Design for reram-based main-memory architectures," in *Proceedings of the International Symposium on Memory Systems*, 2019, pp. 342–350.

[25] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos, "Stripes: Bit-serial deep neural network computing," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–12.

[26] K. Lei, M. Qin, B. Bai, G. Zhang, and M. Yang, "Gcn-gan: A non-linear temporal link prediction model for weighted dynamic networks," in *IEEE INFOCOM 2019-IEEE conference on computer communications*. IEEE, 2019, pp. 388–396.

[27] B. Li, Y. Wang, and Y. Chen, "Hitm: High-throughput reram-based pim for multi-modal neural networks," in *Proceedings of the 39th International Conference on Computer-Aided Design*, 2020, pp. 1–7.

[28] J. Li, T. Zhang, H. Tian, S. Jin, M. Fardad, and R. Zafarani, "Sgcn: A graph sparsifier based on graph convolutional networks," in *Advances in Knowledge Discovery and Data Mining: 24th Pacific-Asia Conference, PAKDD 2020, Singapore, May 11–14, 2020, Proceedings, Part I 24*. Springer, 2020, pp. 275–287.

[29] Z. Li, Z. Liu, J. Huang, G. Tang, Y. Duan, Z. Zhang, and Y. Yang, "Mv-gcn: multi-view graph convolutional networks for link prediction," *Ieee Access*, vol. 7, pp. 176 317–176 328, 2019.

[30] Z. Li, X. Jian, Y. Wang, Y. Shao, and L. Chen, "Daha: Accelerating gnn training with data and hardware aware execution planning."

[31] J. Liang and H.-S. P. Wong, "Cross-point memory array without cell selectors—device characteristics and data storage pattern dependencies," *IEEE Transactions on Electron Devices*, vol. 57, no. 10, pp. 2531–2538, 2010.

[32] C. Liu, H. Liu, H. Jin, X. Liao, Y. Zhang, Z. Duan, J. Xu, and H. Li, "Regnn: a reram-based heterogeneous architecture for general graph neural networks," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, 2022, pp. 469–474.

[33] Y. Luo, P. Behnam, K. Thorat, Z. Liu, H. Peng, S. Huang, S. Zhou, O. Khan, A. Tumanov, C. Ding *et al.*, "Codg-reram: An algorithm-hardware co-design to accelerate semi-structured gnns on reram," in *2022 IEEE 40th International Conference on Computer Design (ICCD)*. IEEE, 2022, pp. 280–289.

[34] K. Mao, J. Zhu, X. Xiao, B. Lu, Z. Wang, and X. He, "Ultragcn: ultra simplification of graph convolutional networks for recommendation," in *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*, 2021, pp. 1253–1262.

[35] J. Mohan, A. Phanishayee, J. Kulkarni, and V. Chidambaram, "Looking beyond {GPUs} for {DNN} scheduling on {Multi-Tenant} clusters," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022, pp. 579–596.

[36] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "Pipedream: Generalized pipeline parallelism for dnn training," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 1–15.

[37] D. Niu, C. Xu, N. Muralimanohar, N. P. Jouppi, and Y. Xie, "Design of cross-point metal-oxide reram emphasizing reliability and cost," in *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2013, pp. 17–23.

[38] C. Ogbogu, A. I. Arka, L. Pfromm, B. K. Joardar, J. R. Doppa, K. Chakrabarty, and P. P. Pande, "Accelerating graph neural network training on reram-based pim architectures via graph and model pruning," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2022.

[39] H. Peng, D. Gurevin, S. Huang, T. Geng, W. Jiang, O. Khan, and C. Ding, "Towards sparsification of graph neural networks," in *2022 IEEE 40th International Conference on Computer Design (ICCD)*. IEEE, 2022, pp. 272–279.

[40] X. Peng, S. Huang, H. Jiang, A. Lu, and S. Yu, "Dnn+ neurosim v2. 0: An end-to-end benchmarking framework for compute-in-memory accelerators for on-chip training," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 40, no. 11, pp. 2306–2319, 2020.

[41] Y. Rong, W. Huang, T. Xu, and J. Huang, "Dropedge: Towards deep graph convolutional networks on node classification," *arXiv preprint arXiv:1907.10903*, 2019.

[42] L. Song, X. Qian, H. Li, and Y. Chen, "Pipelayer: A pipelined reram-based accelerator for deep learning," in *2017 IEEE international symposium on high performance computer architecture (HPCA)*. IEEE, 2017, pp. 541–552.

[43] R. S. Srinivasa, C. Xiao, L. Glass, J. Romberg, and J. Sun, "Fast graph attention networks using effective resistance based graph sparsification," *arXiv preprint arXiv:2006.08796*, 2020.

[44] J. Sun, L. Su, Z. Shi, W. Shen, Z. Wang, L. Wang, J. Zhang, Y. Li, W. Yu, J. Zhou *et al.*, "Legion: Automatically pushing the envelope of {Multi-GPU} system for {Billion-Scale}{GNN} training," in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, 2023, pp. 165–179.

[45] G. Wan and H. Kokel, "Graph sparsification via meta-learning," *DLG@ AAAI*, 2021.

[46] H. Wang and J. Leskovec, "Unifying Graph Convolutional Neural Networks and Label Propagation," *arXiv preprint arXiv:2002.06755*, 2020.

[47] L. Wang, Q. Weng, W. Wang, C. Chen, and B. Li, "Metis: Learning to schedule long-running applications in shared container clusters at scale," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–17.

[48] Y. Wei, X. Wang, L. Nie, X. He, R. Hong, and T.-S. Chua, "Mmgcn: Multi-modal graph convolution network for personalized recommendation of micro-video," in *Proceedings of the 27th ACM international conference on multimedia*, 2019, pp. 1437–1445.

[49] Y. Wei, X. Wang, S. Zhang, J. Yang, X. Jia, Z. Wang, G. Qu, and W. Zhao, "Imga: Efficient in-memory graph convolution network aggregation with data flow optimizations," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2023.

[50] S. J. Wilton and N. P. Jouppi, "CACTI: An enhanced cache access and cycle time model," *IEEE Journal of solid-state circuits*, vol. 31, no. 5, pp. 677–688, 1996.

[51] H.-S. P. Wong and S. Salahuddin, "Memory leads the way to better computing," *Nature nanotechnology*, vol. 10, no. 3, pp. 191–194, 2015.

[52] Y. Xiang and H. Kim, "Pipelined data-parallel cpu/gpu scheduling for multi-dnn real-time inference," in *2019 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2019, pp. 392–405.

[53] C. Xu, D. Niu, N. Muralimanohar, R. Balasubramonian, T. Zhang, S. Yu, and Y. Xie, "Overcoming the Challenges of Crossbar Resistive Memory Architectures," in *2015 IEEE 21st international symposium on high performance computer architecture (HPCA)*. IEEE, 2015, pp. 476–488.

[54] S. Xu, Z. Shao, C. Yang, X. Liao, and H. Jin, "Accelerating backward aggregation in gcn training with execution path preparing on gpus," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 12, pp. 4891–4902, 2022.

[55] T. Yang, D. Li, F. Ma, Z. Song, Y. Zhao, J. Zhang, F. Liu, and L. Jiang, "Pasgcn: An reram-based pim design for gcn with adaptively sparsified graphs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 1, pp. 150–163, 2022.

[56] S. Ye, L. Zeng, X. Chu, G. Xing, and X. Chen, "Asteroid: Resource-efficient hybrid pipeline parallelism for collaborative dnn training on heterogeneous edge devices," in *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*, 2024, pp. 312–326.