NDPGNN: A Near-data Processing Architecture for GNN Training and Inference Acceleration

Haoyang Wang*, Shengbing Zhang*, Xiaoya Fan*, Zhao Yang†, Meng Zhang*
* Northwestern Polytechnical University, Xi'an, China. † Chang'an University, Xi'an, China.

***Abstract*—Graph Neural Networks (GNN) require a large number of fine-grained memory accesses, which results in inefficient use of bandwidth resources. In this paper, we introduce a near-data processing architecture tailored for GNN acceleration, named NDPGNN. NDPGNN provides different operating modes to meet the acceleration needs of various GNN frameworks while ensuring the configurability and scalability of the system. NDPGNN takes advantage of data locality characteristics to repeatedly distribute and utilize data, thereby reducing memory access requirements, and further improving memory access efficiency by combining a subgraph sparse node scheduling strategy with intermediate result reuse. We use data packaging to provide a higher effective data ratio for long-distance data transmission, thereby improving the utilization of the system's limited bandwidth resources. Compared with the previous method, NDPGNN brings 5.68 times improvement in system performance while reducing energy consumption overhead by 8.49 times.**

***Index Terms*—Graph Neural Networks, Hardware Accelerator, Near-Data Processing.**

I. Introduction

THE application of Graph Neural Networks (GNNs) has gained popularity in recent years as a powerful extension of Deep Neural Networks (DNNs) [12], [31]–[33]. GNNs operate on graphs, which consist of interconnected vertices and are utilized in various graph abstraction levels including Link Prediction, Vertex Classification, and Graph Clustering [1]–[3], [8], [16], [17]. In contrast to typical DNNs, the properties of GNNs bring new challenges for computing systems. GNNs have been widely applied in practical scenarios, such as social network analysis [25], [26], autonomous driving [27], and recommendation systems [28], [29]. Several tasks within these applications necessitate low-latency and energy-efficient inference, particularly in edge computing contexts like autonomous driving [27], [30].

GNNs require a sparse adjacency matrix [15] to represent the graph connectivity relationship. Graph Convolutional Networks (GCNs) are classic computing models consisting of two operations in each layer, aggregation and combination. GCN model uses the message-passing method to collect characteristics of neighbor nodes and updates vertex data with matrix multiplication (MM) [51]–[53].

The calculation processes of different graph neural network algorithm models are different. There are several GNN accelerators designed to mitigate the irregular acceleration

Manuscript received March 31, 2024; accepted July 15, 2024. This work was partly supported by National Key R&D Program of China (2022YFB2901103) and Research fund of CIE-SmartChip (20220370-0060). Corresponding author: Meng Zhang(e-mail: zhangm@nwpu.edu.cn).

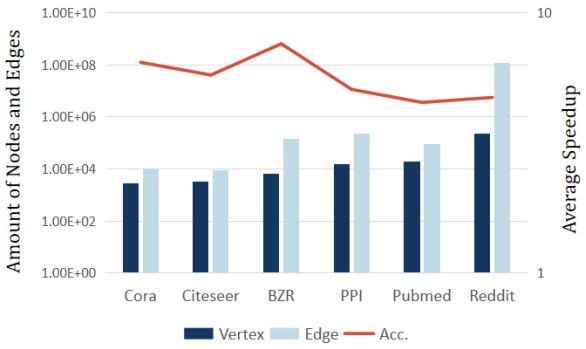

Fig. 1. Average Speedup of NDPGNN on Different Datasets.

requirement of GNN and improve performance and energy efficiency [9]–[11]. However, the existing GNN accelerator design cannot well support the computing requirements of different GNN models and does not have good scalability. At the same time, graph neural network training, especially, is not considered. Our research rethinks what parts of existing work can be optimized and designs a hardware acceleration architecture with better GNN adaptability.

For GNN training, the characteristics of graph data present several difficulties [38]–[42], particularly in terms of the under-utilization of compute and bandwidth resources brought by irregular operations and fine-grained memory access. These challenges are compounded when training on large graphs due to the increased demands on system scalability [47]–[50]. A sample-based training method is proposed as a solution [3]. Instead of processing the entire graph, this method divides the workload into several node-wise mini-batches, which are more scalable and effective than whole-graph training.

In recent years, the research of near-data processing technology has provided new ideas for solving the problems existing in data processing tasks [18]–[22]. The near-data processing mode uses a method that arranges simple computing components near data storage to accelerate data-intensive simple computing operations. Data-intensive tasks can significantly diminish data transmission delays with the support of near-data processing architecture, resulting in enhanced computational performance.

Our design adopts a heterogeneous architecture composed of near-data modules and convolutional computing units to adapt to the different computing characteristics of Combination and Aggregation operations. The advantage of this architecture is that it can meet both data needs and computing

power needs and optimize the execution efficiency of various operations by rationally allocating hardware resources. In addition, we further improve system performance through a series of design methods. For example, repeated distribution technology that takes advantage of data locality characteristics can reduce memory access requirements, thereby reducing data transmission delays and improving overall computing speed. In addition, the combination of subgraph sparse node scheduling strategy and intermediate result reuse can further improve memory access efficiency and reduce resource waste.

In order to further optimize the bandwidth resource utilization of the system, we adopt the method of data packing, thereby increasing the effective ratio of data transmission over long distances. This packaging technology effectively reduces the idle time during data transmission, allowing limited bandwidth resources to be more fully utilized. Through the combination of these design methods and optimization strategies, our system achieves a higher performance level while meeting various acceleration requirements, providing reliable technical support for large-scale graph data processing.

Our research has made important contributions in the following aspects:

a) First: We analyzed the adaptability of GNN models to different hardware acceleration methods and discussed specific strategies for optimizing aggregation and combination operations to enhance overall acceleration performance. We considered the varying acceleration requirements of these operations and evaluated the issues and challenges in GNN acceleration from multiple dimensions. Additionally, we reassessed the shortcomings of previous GNN accelerator designs and established clear design objectives.

b) Second: We design NDPGNN, a near-data hybrid acceleration architecture that aims to address the challenges of GNN acceleration and achieve significant performance improvements. NDPGNN fully leverages the advantages of near-data processing architecture and traditional computing-intensive acceleration architecture to achieve efficient system operation. The design of this architecture takes into account the characteristics of the GNN model and combines efficient data processing and computing acceleration mechanisms to provide powerful acceleration capabilities for GNN applications. At the same time, it reduces the modification of high-cost devices, reduces design costs, and improves the versatility of the system.

c) Third: NDPGNN supports multiple working modes including inference and training, and can provide flexible acceleration solutions for GNN frameworks with different acceleration requirements. Whether it is for data sets with a huge number of nodes or an exponential increase in the number of connecting edges, NDPGNN has demonstrated excellent performance and stable acceleration. Figure 1 shows the average speedup of NDPGNN on various GNN datasets, highlighting its strong scalability and applicability.

II. GNN Algorithm: Inference and Training

In this section, we assess the intricacies of GNN inference and training.

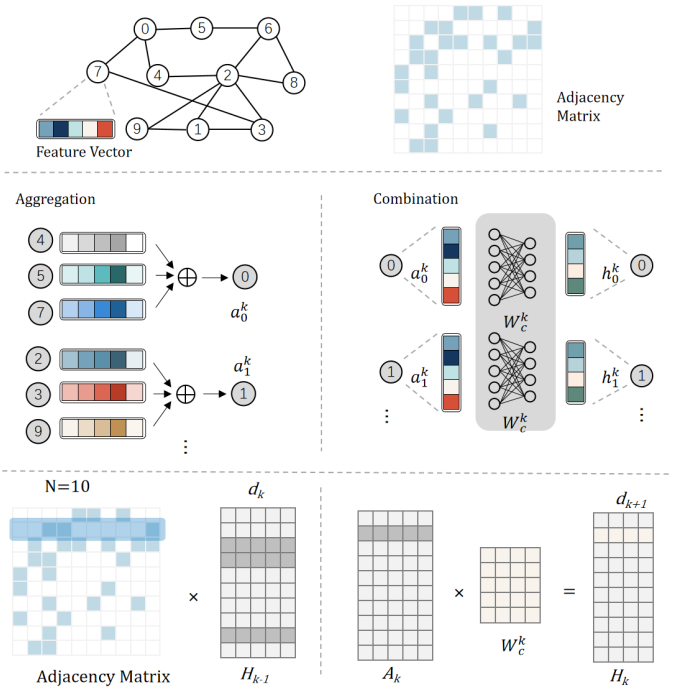

Fig. 2. GNN Algorithm: Aggregation and Combination, Represented by Connection Relationship and Matrix Multiplication.

During inference, GNN typically employs the following two fundamental operations: Aggregation and Combination. For each node, aggregate the features of neighboring nodes into the current node using a defined message-passing function. Typical message-passing functions include summation, averaging, weighted sums, etc. Upon receiving information transmitted from neighboring nodes, combination operation updates the current node's feature representation. A common feature update function is multilayer perception.

Next, we introduce the training method for GNN. The training process for GNN involves forward propagation, loss computation, backward propagation, and parameter optimization.

A. Graph Neural Networks

Figure 2 shows a standard GNN algorithm [34]. The graph structure is represented by the Adjacency Matrix composed of connection relationships between nodes. Within a GNN model, there exist numerous graph convolutional layers. In each layer, vectors undergo two distinct computations: aggregation and combination. Within a layer operation cycle, a node accumulates information from its neighboring nodes through aggregation and subsequently updates its vector through combination. The aggregation and combination operations for the node v in layer k can be succinctly characterized as follows:

$$\begin{aligned} a_v^k &= \textbf{Aggregate}(h_u^{k-1} | u \in \widetilde{\mathcal{N}}(v)), \\ h_v^k &= \textbf{Combine}(a_v^k, W_c^k). \end{aligned} \tag{1}$$

Here, h_v^k signifies the hidden feature vector of node v at the k-th layer, while a_v^k represents the aggregation feature

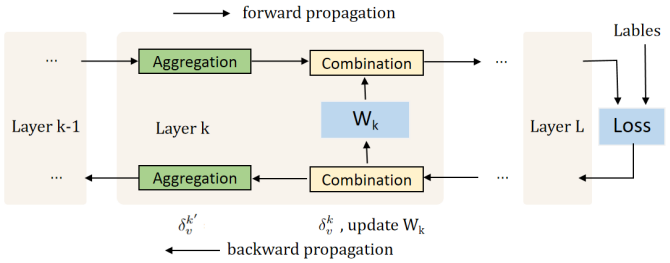

Fig. 3. GNN Training Process, Forward Propagation, Loss Computation, Backward Propagation, and Parameter Optimization.

vector. W_c^k stands for the shared weight parameters used for combination operation in layer k, and $\widetilde{\mathcal{N}}(v)$ encompasses all the neighboring nodes of the central node. Moreover, there are some symbols in Fig. 2. N is the total amount of nodes in the graph, d_k is the dimension of the node vector in layer k. H_K and A_k represent the stitching matrix of the hidden feature vector and the aggregation feature vector in layer k, respectively.

$$a_v^k = \sum_{u \in \widetilde{\mathcal{N}}(v)} \frac{1}{\sqrt{D_v * D_u}} h_u^{k-1}, h_v^k = \mathrm{ReLU}\left(a_v^k \cdot W^k\right) \quad (2)$$

The equations 2 delineate the particular computation within the typical GCN model [5]. Within the equation, D_v and D_u represent the degrees of the vertex v and each respective neighbor u. The aggregation of neighbor features is accomplished through summation alongside degree-based normalization. The combination operation is obtained through node-weight convolutional computations, followed by a ReLU operation. The weights W are acquired through training. In the GCN model, all nodes calculated in each layer utilize the same weight values for the combination operation.

Edge updating is another prevalent function in GNNs, and formula 3 elucidates the procedure involved in edge updating.

$$e_{u,v} = \sigma(W_e^k \cdot h_u^{k-1}). \quad (3)$$

where $e_{u,v}$ denotes the resultant edge update vector for the node pair u and v, h_u^{k-1} signifies the hidden feature vector of node u at the $(k-1)$-th layer, and W_e^k represents the shared weight parameters employed in the edge update process of layer k.

Numerous GNN models necessitate edge update processing, but there are also GNN models that don't involve edge update operations. Allocating dedicated hardware computing resources for edge updates may lead to a waste of computational resources.

B. GNN Training

GNN training typically involves four main steps: forward propagation, loss computation, backward propagation, and parameter optimization, shown in Fig. 3.

a) Forward Propagation:: Input node features from the graph into the GNN model for forward propagation computation. Continuously update node representations during the aggregation and combination processes. The process of forward propagation is similar to the inference process, using the weight values of the current round for calculation.

b) Loss Computation:: Compare the GNN's output node representations with the ground truth labels to compute the loss function. During the training process, the Loss value can be reduced through continuous iteration, thereby improving the system prediction accuracy.

c) Backward Propagation:: Utilize the backward propagation algorithm to calculate gradients of the loss function. Different from traditional CNN calculations, the calculation of each layer of GNN calculation consists of two sub-calculations. Therefore, the calculation process of aggregation and combination operations needs to be analyzed separately during the backward propagation process. The calculation in the backward propagation process can be expressed as:

$$\delta_v^{k'} = \mathrm{mask}\left(\sum_{u \in \widetilde{\mathcal{N}}(v)} \frac{1}{\sqrt{D_u * D_v}} \delta_u^{k+1}\right) \quad (4)$$

$$\delta_v^k = \delta_v^{k'} \cdot {W^{k+1}}^T, \frac{\partial \mathcal{L}}{\partial W^k} = \frac{\partial \mathcal{L}}{\partial W^k} + {a_v^k}^T \cdot \delta_v^k \quad (5)$$

Where the symbol δ_u^{k+1} represents the gradients of features for vertex u at layer $l+1$, which is equivalent to $\frac{\partial \mathcal{L}}{\partial h_u^{l+1}}$. The function mask() refers to gradients associated with activation functions such as ReLU(). At the start of each training epoch, the weight gradients $\frac{\partial \mathcal{L}}{\partial W^k}$ are initialized to zero.

d) Parameter Optimization:: Utilize optimization algorithms to update model parameters and reduce the loss function.

$$W^k = W^k - \eta \frac{\partial \mathcal{L}}{\partial W^k} \quad (6)$$

where η represents the learning rate.

The training process is iterative for multiple epochs until convergence. In the training process of graph neural network (GNN), due to a large amount of calculation, the Sample-based mini-batch training method is usually used [43]–[46]. This method is to reduce the computational complexity and memory usage during the training process, thereby improving training efficiency and speed.

III. Challenges

In this part, we first introduce the challenges that exist in the design process of GNN accelerators, and then we introduce some related work in the same field and their shortcomings.

A. Challenges

The GNN task is different from the traditional DNNs. Because of its irregular data organization and operation characteristics, von Neumann's architecture is not suitable for GNN acceleration. There is a mismatch between acceleration structure and computing requirements, which is primarily reflected in the following:

a) ***Challenge 1: Acceleration Requirements Differences of Two Basic Computations*** *:* Combination and aggregation are the two fundamental processes of the GNN algorithm, and they both have distinct acceleration requirements [35]–[37]. Matrix multiplication is used in combination tasks, which is a computationally demanding operation. Aggregation is a data-intensive activity that requires a lot of data and uses basic calculations, and the data has low reusability characteristics. To meet the processing acceleration needs of GNNs, it is not sufficient to choose an acceleration approach that adjusts to a single computation feature.

b) ***Challenge 2: Fine-grained Memory Access****:* According to the natural connection relationship of graphs, the GNN operations require the support of fine-grained memory access. Operations between pairs of nodes call for hopping access in memory, which brings memory access requirements to all surrounding nodes that are connected to one central node. Simply increasing the memory access bandwidth won't result in a significant performance gain because the node's neighbors are dispersed over the whole graph and the spatial locality of the data they link to in the storage system is very poor. By increasing the memory access bandwidth, more data can be accessed, but since a considerable portion of the extra data is worthless, the total efficiency of memory access has not risen.

c) ***Challenge 3: Significant Overhead of Full-graph Training****:* A natural method of GNN training is full-graph training, which takes into account all neighbors of every central node. This strategy has some drawbacks. First of all, full-graph training brings substantial difficulties to storage bandwidth and computational power because it requires a large number of calculations when training big graphs. Additionally, the full-graph training method is not scalable enough to handle the needs of large-scale distributed GNN as graph size grows with the development of GNN. Sampling is a good way to deal with the GNN's irregular computation and sparse properties, but it requires hardware that can support the mini-batch computing technique.

B. Related Works

Based on the analysis of GNN accelerator design problems, we rethink the aspects of previous work that can be improved.

a) GNN Inference Accelerating: Numerous custom accelerators have been introduced to accelerate GNNs. Notable accelerators in this field include HyGCN [9], AWB-GCN [10], and HaGNN [11].

HyGCN [9] employs a hybrid architecture comprising an aggregation phase and a combination phase. The aggregation phase exhibits dynamic and irregular execution characteristics, whereas the combination phase demonstrates static and regular execution patterns. To address these phases, HyGCN introduces an edge-centric and matrix-vector multiplication (MVM) programming model, leveraging various forms of parallelism while ensuring hardware transparency. It integrates two efficient engines to optimize these phases and coordinates the pipeline between engines to minimize latency and energy consumption. However, due to the lack of in-depth analysis of the acceleration requirements for Aggregation and Combination, HyGCN does not fully realize the performance potential of its hybrid architecture.

AWB-GCN [10] addresses the workload imbalance when processing large-scale real-world graphs by introducing three hardware-based auto-tuning techniques: dynamic distribution smoothing, remote switching, and row remapping. These techniques continuously monitor sparse graph patterns, dynamically adjust workload distribution among a large number of processing units, and reuse the optimal configuration after convergence. Nevertheless, HyGCN and AWB-GCN can not support Edge-update operations in GNN models, limiting its applicability for accelerating models like GraphSage and G-GCN.

HaGNN [11] presents an accelerator architecture specifically designed for GNNs, addressing the unique memory access and data movement requirements of GNNs. This architecture includes dedicated hardware units for efficiently handling the irregular data movements essential in graph computations while providing the high computational throughput required by GNN models. However, HaGNN cannot identify and store intermediate results of edge update and aggregation, resulting in the movement of invalid data and the wasting of bandwidth resources.

b) GNN Training Accelerating: GNNear [24] is considered one of the most advanced works currently and provides strong support in GNN training acceleration. However, despite its excellent performance in accelerating GNN training, there are some areas worthy of improvement. Especially in terms of utilizing algorithmic features in aggregation operations, GNNear has not fully exploited its potential and failed to maximize the use of limited bandwidth resources. This makes it less than ideal for achieving optimal training acceleration. Although GNNear has made important progress in the field of GNN training acceleration, there is still room for improvement, especially in optimizing algorithm feature utilization and bandwidth resource utilization.

c) ReRAM Accelerators for GNNs: In-data processing is a method of performing internal computations within memory structures using novel circuit components that possess both storage and computational capabilities, such as memristors. There have been some studies leveraging new ReRAM storage to accelerate graph processing applications [18] [19]. In-data processing is an important branch of graph neural network acceleration, but the expensive design costs associated with novel memory structures pose a challenge to its widespread adoption in the short term, which does not meet our design goals.

C. Design Goal

During the system design process, we need to consider the following design goals to achieve better performance and scalability:

a) High Acceleration Energy Efficiency: Our system should be able to provide efficient acceleration capabilities to support fast and efficient computing processes while maintaining little energy consumption.

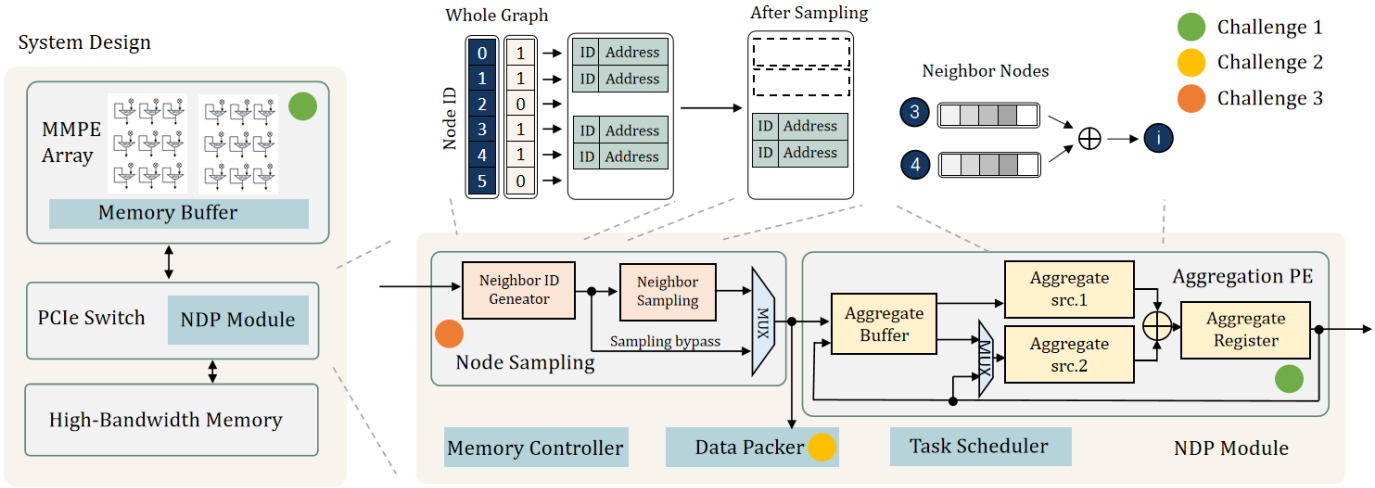

Fig. 4. NDPGNN Architecture System Design and Module Description. Including the specific design of Sample Engine and Aggregation PE. The corresponding relationship between each design module and challenges is marked in different colors.

b) Solve the Bottleneck Problem of Memory Bandwidth: In order to avoid fine-grained memory bandwidth utilization becoming the bottleneck of the system, we need to reduce unnecessary long-distance data transfer and increase the proportion of effective data participating in the operation.

c) Flexible System Design Concept: Our system should support GNNs in different sizes and operation modes to accelerate calculations. To do this, we need to design flexible and scalable systems to adapt to complex and changing application scenarios.

d) Reduce the Changes to High-cost Components: In the system design process, we need to minimize the changes to high-cost components such as memory chips to ensure that the acceleration system has better versatility and scalability.

IV. Proposed Architecture: NDPGNN

For the needs of GNN acceleration, we present NDPGNN, a near-data hybrid acceleration architecture.

A. System Design

The system design of NDPGNN is shown in Fig. 4. It comprises a high-bandwidth memory, a PCIe switch with a near-data processing module, and a matrix multiplication array connected to the PCIe switch. The NDP module is utilized to handle computation-intensive tasks related to data-dense features in GNNs. Matrix-Multiplication Processing Engine (MMPE) is responsible for processing compute-intensive tasks, such as combination and edge updates. By allocating tasks associated with different features to their respective units, NDPGNN not only addresses the challenge of dissimilar feature operations during the graph neural network computation but also ensures the system's capability to support various GNN models without wasting hardware resources.

Following design goals, NDPGNN organizes function modules, each designed to address challenges in GNN inference and training acceleration processes. For different memory access and computational requirements during GNN aggregation and combination processes, we have designed two different modules tailored to their features. To address bandwidth bottlenecks in the GNN computation process, we have designed a near-data fine-grained access architecture and also devised a data packaging module to enhance the utilization of limited bandwidth further. Also, we designed a node ID generate module that supports sample-based training processes to tackle the issue of low efficiency in full-graph training.

NDPGNN supports various computing needs in different GNN inference and training processes through configurable data paths. For example, in the Node Sampling Module, you can choose whether to perform sampling through the data gating device, so the system can support high-precision reasoning and sample-based training at the same time. As another example, NDPGNN can configure the data paths of two Matrix Multiplication Processing Engines (MMPE) to make them both process combination workloads at the same time or process combination and edge updates separately. Configurable datapath strategy makes NDPGNN not only support multiple types of models but also not cause hardware resources waste by load imbalance.

Unlike traditional DNNs, the computation process and workload of GNNs are dependent on the input features, making static data flow dependencies and task allocation unsuitable for GNN acceleration needs. Scheduling GNN computational tasks requires dynamic scheduling techniques. We achieve dynamic scheduling of graph tasks by incorporating a Sample Engine and Task Scheduler into the system.

The NDP module consists of five main components: the Sample Engine, Aggregation Processing Engine (Aggregation PE), Memory Controller, Task Scheduler, and Data Packer. Among them, the Sample Engine and Aggregation PE are computational modules used for training and inference in graph neural networks. The Task Scheduler is used for subtask partitioning and overall system task scheduling. The Memory Controller is responsible for controlling data access to the high-bandwidth memory. The Data Packer is responsible for

aggregating fine-grained memory access data, ensuring that data transmitted over long distances consists only of valid data, thereby improving bandwidth utilization.

B. Sample Engine

Figure 4 describes the workflow of Sample Engine. Sample Engine is used to convert the sparse neighbor node information of the entire graph into the correlation between aggregation operations between nodes. It takes the full graph adjacency matrix as input and extracts the connection information between the central node and its neighboring nodes. Retrieve the adjacency matrix of the corresponding row according to the ID of the central node. The node with 1 in the corresponding position is the neighbor node connected to the central node, so the ID of the corresponding neighbor node can be generated. This transformation converts the sparse connection information of the entire graph into a dense mini-batch data structure, simplifying irregular operations within the graph structure. In addition, it will also filter out the data of neighbor nodes that do not participate in aggregation during the training process, thereby achieving sample-based training. For example, in the case where random sampling is applied, the neighbor sampling module proportionally removes some edges between the central node and its neighbors through group sampling. When the GNN model does not require sampling to perform a high-precision inference process, this sampling step can be skipped through a bypass circuit.

In different computing models, the specific computing process of GNN is different. The order of operations based on node data can be divided into two types: Aggregation first and Combination first. Sample Engine can support these two different acceleration requirements. Through the generated configurable data stream of neighbor node data, the system can send it to the Aggregation Engine or the MMPE array for Combination operation.

C. Aggregation Processing Engine

The Aggregation Processing Engine (Aggregation PE) is used to implement the aggregation operation in GNN. It receives the neighbor node information in the mini-batch structure, and the intermediate results are used as input for further processes. By saving reusable intermediate aggregation results, we use a redundancy elimination method [14] to improve system performance. The same central node's aggregation tasks are given to a single aggregation engine unit, and the aggregation workloads of several central nodes are processed concurrently by various units.

The same node may be a neighbor of multiple central nodes at the same time. If this property cannot be well utilized, multiple unnecessary data accesses may occur, thereby increasing bandwidth pressure. NDPGNN uses the node data reuse method to distribute the same data to the parallel Aggregation PE of all current neighbor nodes, as shown in Fig 5. GNN data does not have the reuse characteristics of traditional computing, so it is not suitable for the multi-level Cache structure in the von Neumann structure. Finding the data reuse characteristics in the GNN acceleration process

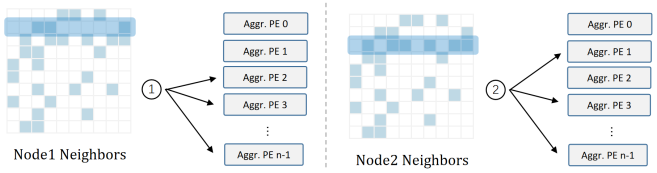

Fig. 5. Distribute Nodes to Other Neighbor Nodes in Parallel Aggregation PE to Take Advantage of Data Time Locality.

can avoid repeated access to the same data multiple times, and use the time locality of data to improve system operating efficiency.

D. Data Packer

The memory access of GNN data during the calculation process has fine-grained characteristics, and the data nodes accessed each time are generated by the connection relationships of the graph. For a central node, the IDs of its neighbor nodes may be discontinuous, thus bringing about fine-grained memory access requirements.

$$\text{Efficiency}_{\text{pipe}}\left[t_k\right] = \left(\sum_{i=0}^{N[t_k]} \frac{D_{\text{eff}}\ \left[t_k\right]}{L_B}\right) / N\left[t_k\right] \tag{7}$$

Equation 7 shows the memory access efficiency of t_k round. Where L_B represents the memory access bit width, D_{eff} represents the length of valid data, and N is the number of memory accesses calculated this time.

Among the data accessed in a single time, the data that needs to be used in this round of calculation only accounts for a small part, as shown in Fig. 6, and more data is the data that is not used in this round of calculation. If all this data is transmitted over long distances, limited bandwidth resources will be wasted.

NDPGNN uses data packing to solve the bandwidth problem caused by fine-grained memory access, prevents the long-distance transmission of invalid data and the occupation of bandwidth resources, and packs sparse information into dense data, thereby improving system bandwidth utilization and reducing This eliminates the impact of bandwidth bottlenecks on the overall acceleration of the system.

Nodes that have completed the Aggregation phase proceed to the packing stage to be sent to the MMPE for the Combination operation. The Combination process for each node is relatively independent, with no data dependency or requirement on the order of node operations. Once Aggregation is completed, nodes can be packed, and when a fixed-length pack is ready, it is sent to the MMPE. Although there is a waiting period during the packing process, this wait is not a stall but a continuous flow of data, without the bubbles caused by data dependencies.

E. Task Scheduler

The main responsibility of the Task Scheduler is to manage the overall computing process of the system, including selecting data paths and controlling data scheduling. In graph neural

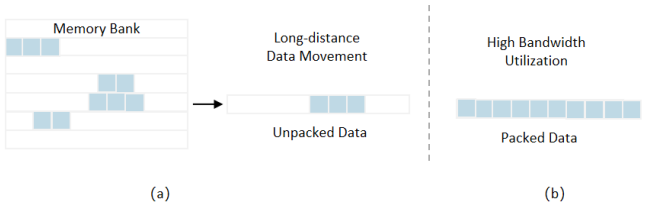

Fig. 6. (a) The Proportion of Currently Required Valid Data in Unpackaged Data is Small. (b) Packed Dense Data Helps Improve Bandwidth Utilization.

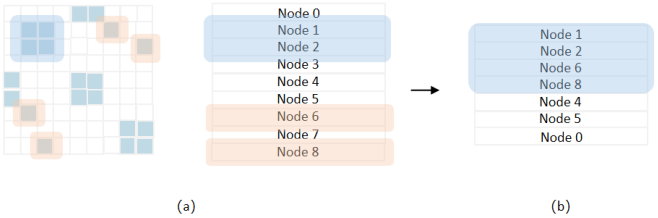

Fig. 7. (a) Sparse Connections between Dense Sub-graphs May Increase the Amount of Memory Accesses. (b) Rearrange the Sparse Neighbors of the Sub-graph to Improve Data Spatial Locality.

networks, the task scheduler is even more crucial, because it not only coordinates the order of various calculation modules and ensures that calculations are performed in the correct order to effectively process and transmit information, but also controls the way data is arranged, thereby better-improving system bandwidth utilization. In terms of computing processes, the task scheduler can control the order of aggregation and combination to support the computing needs of different GNN models.

In order to reduce the sparsity in graph calculation and avoid repeated calculation of intermediate results, NDPGNN adopts the strategy of sub-graph calculation. We used the METIS [54] method to identify and create subgraphs. This hierarchical partitioning algorithm is based on the adjacency matrix, with the core idea being to continuously coarsen nodes and edges of the original graph to reduce its size. After reaching a certain degree of reduction, the coarsened graph is partitioned, and the partitioned subgraphs are then projected back to the original graph structure. This ensures the balance of each subgraph while maintaining model accuracy. Sub-graph technology saves the aggregation intermediate results of dense sub-graphs and directly uses these results when needed, which can avoid repeated aggregation calculations for multiple neighbor nodes of the sub-graph, thereby improving computing efficiency.

However, using sub-graph intermediate results for calculation may bring more sparse connections to the graph structure. At the same time, the process of generating sub-graph intermediate results requires multiple accesses to sparse neighbor node data to calculate the value of the sub-graph intermediate results. These accesses to sparse neighbor nodes are fine-grained, that is, two sparse neighbor nodes may require two visits to obtain because their locations are not continuous, so the bandwidth of each memory access cannot be efficiently utilized.

During the data arrangement process, NDPGNN also places the necessary sparse nodes of dense sub-graphs near the sub-graph to reduce the number of memory accesses and increase the locality of the data space, as shown in Fig. 7. This optimization method further improves the bandwidth utilization of the system and makes the graph computing process more efficient and feasible, especially when processing large-scale graph data. It has significant advantages. In summary, task scheduler and sub-graph technology play an indispensable role in graph neural networks. Through reasonable scheduling and optimization methods, the overall performance and computing efficiency of the system can be effectively improved, and the processing of complex graph data can be provided. strong support and solutions.

F. Matrix Multiplication Processing Engine

Typical near-data processing architecture is suitable for computational tasks with high data demands and simple computational components, but it is not suitable for larger-scale multiplication-accumulate matrices. The characteristics of the NDP determine the NDPGNN architecture, which consists of a near-data processing module and an MMPE, forming a heterogeneous system mode.

The MMPE is utilized for handling Combination and Edge update operations, comprising multiplication-accumulate matrices. The calculation of Combination and Edge update, centered around convolution, is crucial within GNN computations. It is noteworthy that certain GNN models do not involve Edge update operations. Thus, designing a dedicated computational module for Edge update allows for supporting a variety of GNN computational model requirements. However, this approach may lead to idle circuits during the computation of models without Edge update operations, resulting in hardware resource wastage.

NDPGNN employs two homogeneous MMPEs to handle Combination and Edge update operations. When dealing with GNN models that include Edge update, the two MMPEs execute Combination and Edge update operations separately, thereby supporting multiple model types. Conversely, when processing models without Edge update, the two MMPEs can collaborate on Combination operations through data path configuration, preventing circuit idle states.

The data received by the MMPE is provided by the Data Packing module within the NDP module. Data packing mitigates the low bandwidth utilization issue caused by GNN fine-grained access features, enabling a higher proportion of effective data in limited bandwidth resources for long-distance transmission. Consequently, this resolves bandwidth bottlenecks encountered during GNN acceleration processes.

G. Execution Mode

NDPGNN can support GNN models that require and do not include edge updates. The nodes removed during the sample operation don't have to participate in the edge update operation, ensuring that no unused intermediate results are

TABLE I
DIFFERENT GNN MODELS

Algorithms	Edge Update	Aggregation	Combination
GCN	Null	$a_v^k = \sum_{u \in \mathcal{N}(v)} h_u^{k-1}$	$h_v^k = \sigma\left(W^k \cdot a_v^k\right)$
GraphSage-Pool	$e_{u,v} = \sigma\left(W_1^k \cdot h_u^{k-1}\right)$	$a_v^k = \max_{u \in \mathcal{N}(v)} \left\{e_{(u,v)}\right\}$	$h_v^k = \sigma\left(W_2^k \cdot a_v^k\right)$
G-GCN	$e_{u,v} = \sigma\left(W_1^k \cdot h_u^{k-1} + W_2^k \cdot h_v^{k-1}\right)$	$a_v^k = \sum_{u \in \mathcal{N}(v)} e_{u,v} \odot h_u^{k-1}$	$h_v^k = \sigma\left(W_3^k \cdot a_v^k\right)$

generated. During the GNN model execution process without edge updates, after-sampling neighbor data based on the central node is directly sent to Aggregation PE to operate, after which the combination task is packed and then distributed to two MMPE units for operations. In this configuration, the two MMPEs computing components are all used, and no computing resources are idled. For models with edge update operations, Sample Engine separates the entire graph into mini-batch-based tasks before transmitting packed data to MMPE for edge update. After the edge update, node vectors are sent to Aggregation PE and then MMPE again for the combination operation. Two MMPE arrays can perform edge updates and combinations in this mode, respectively.

NDPGNN not only supports GNN inference but also supports GNN training. In the training process of large-scale graphs, training acceleration is very challenging, so the mini-batch sample-based training mode is adopted. Different from the inference process, the Sample Engine chooses to enter sampling mode rather than bypass mode during the training process.

In NDPGNN, the Sample Engine will choose to enter the sampling mode during the training phase, perform data aggregation and update operations based on the sampled data, calculate gradients, and update weights until the training purpose is achieved. During the training process of GNN, the mini-batch Sample-based training mode can effectively improve the training speed and efficiency. In this mode, the system will randomly select a part of nodes and edges from the entire graph to form a mini-batch for training based on the preset sampling strategy. This can reduce the amount of data that needs to be processed during the training process, reduce the computing load, speed up model convergence, and save computing resources.

V. EXPERIMENTS AND EVALUATION

A. Experimental Setup

We've instantiated NDPGNN in Verilog and conducted synthesis using the Synopsys Design Compiler (DC) with the TSMC 12 nm standard VT library. The overall on-chip memory is 22MB. We employed Ramulator [13] to estimate HBM timings and energy consumption. In evaluating NDPGNN, we selected three prominent GNN models, namely GCN [5], GraphSage-Pool [6], and G-GCN [7]. The operations of each model are shown in Table I. It's worth noting that both GraphSage-Pool and G-GCN include edge-update operations, and GraphSage-Pool and G-GCN have different edge-update operations. Table II shows the GNN datasets we use. The six datasets we employed exhibit significant differences in size, manifested in terms of node count and node dimensionality. This approach enables a better assessment of the system's capability to handle different feature datasets while also highlighting whether the system maintains scalability with increasing dataset size, thereby showcasing its ability to deliver stable performance improvements.

TABLE II
GRAPH DATASETS

Dataset	Vertex	Edge	Dimension
Cora(CA)	2,708	10,556	1,433
Citeseer(CS)	3,327	9,104	3,703
BZR(BR)	6,519	137,734	3
PPI(PI)	14,755	225,270	50
Pubmed(PB)	19,717	88,648	500
Reddit(RD)	232,965	114,615,892	602

We conducted a comparative analysis, pitting NDPGNN against PyTorch Geometric [4] implementations on both CPU and GPU. Additionally, we compared NDPGNN with three GNN accelerators, including HyGCN [9], AWB-GCN [10], and HaGNN [11]. The CPU platform was equipped with two Intel Xeon(R) CPU E5-2680 v3 processors and 500GB of DRAM, while the GPU platform featured an NVIDIA Tesla V100 GPU. To ensure the fairness of our comparisons, we standardized the configurations, operating frequencies, and fabrication processes of the baseline accelerators. This alignment helps to avoid performance discrepancies caused by varying hardware conditions. Table III provides the system configurations of baseline accelerators. In order to evaluate the training acceleration effect of NDPGNN, we use DGL [23] model training running on CPU and GPU, and SOTA training accelerator GNNear [24] as baselines, and compare the performance of NDPGNN and baseline training acceleration methods. GNNear utilizes the TSMC 28nm process. To maintain fairness in our comparisons, we employed identical experimental configurations across all evaluated accelerators when assessing NDPGNN's training acceleration effects. Also, we adjusted the operating frequency of GNNear to the same 1GHz as in our work instead of 700M in the original article, to get a fairer comparison of the acceleration effect. In addition, we use our acceleration environment and GNNear's acceleration strategy to evaluate its training acceleration effect on the G-GCN model.

B. Experimental Results and Evaluation

a) Inference Speedup: We conducted an evaluation of NDPGNN by comparing it to three GNN frameworks: GCN,

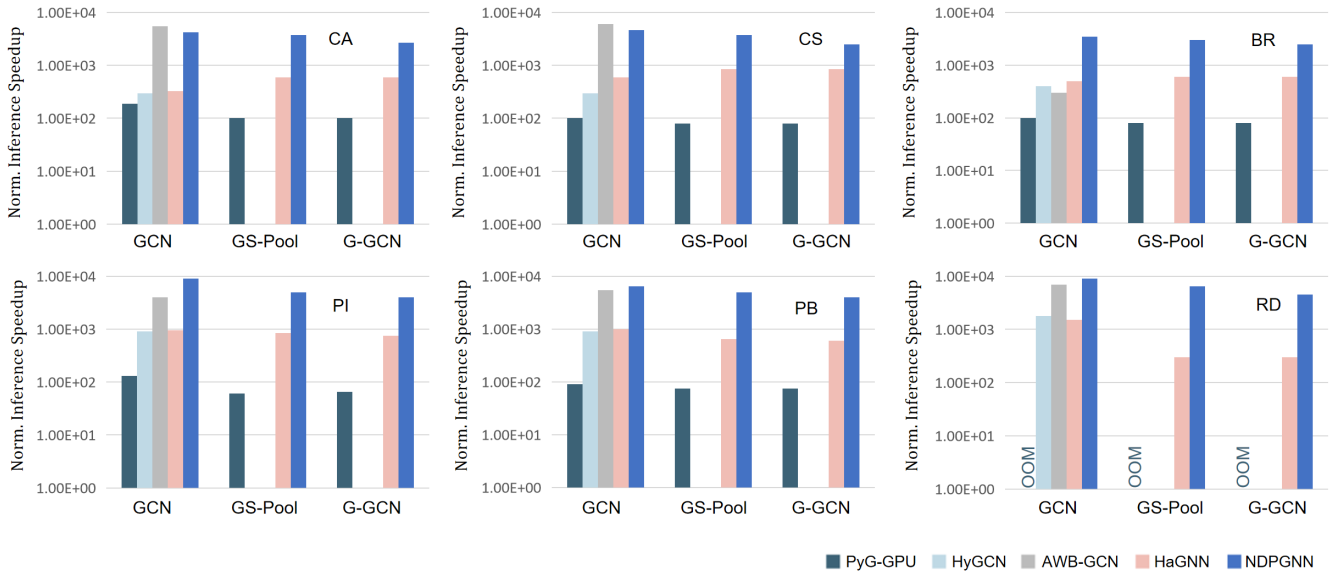

Fig. 8. Inference Speedup Compared with Other Baselines on Different Datasets, Normalized to PyG-CPU.

TABLE III
SYSTEM CONFIGURATIONS OF ACCELERATORS.

	System Configurations
HyGCN	1GHz @ 32 SIMD 16 cores and 32×128 arrays 256 GB/s HBM
AWG-GCN	1GHz @ 4k PEs 256 GB/s HBM
HaGNN	1GHz @ 512 32-bit ALUs and 32×128 PEs 256 GB/s HBM
NDPGNN	1GHz @ 512 Aggregation-PEs and 32×128 arrays 256 GB/S HBM

GraphSage-Pool, and G-GCN. GraphSage-Pool and G-GCN models incorporate edge update operations, a feature that the HyGCN and AWB-GCN accelerators lack the capability to support.

The performance metrics obtained from our evaluations, as delineated in Fig. 8, have been normalized relative to the PyG-CPU. NDPGNN showcases remarkable speed enhancements, outperforming HyGCN, AWB-GCN, and HaGNN by factors of 7.90×, 2.44×, and 6.73×, respectively, across a diverse array of accelerator configurations. These results underscore NDPGNN's prowess in terms of computational efficiency and performance scalability.

Unlike HyGCN and AWB-GCN, NDPGNN not only supports GCN-based models but also facilitates edge update operations through data path scheduling, thus catering to the diverse acceleration requirements of different GNN models. This flexibility not only expands the system's applicability but also ensures high efficiency in handling various graph neural network tasks while avoiding reduced overall system efficiency due to idle circuits. Therefore, NDPGNN demonstrates unique advantages and potential in accelerating graph neural networks.

Moreover, we evaluate NDPGNN's scalability, seeking to unveil its adaptability and efficiency across varying dataset sizes. Fig. 1 portrays the relationship between different dataset dimensions and the corresponding speedup achieved by NDPGNN. The absence of a discernible inverse correlation between speedup and graph size indicates that the magnitude of GNN datasets does not exert a discernible influence on the acceleration effect offered by NDPGNN. This observation is significant as it underscores NDPGNN's capacity to maintain consistent acceleration capabilities across datasets of varying complexities and magnitudes.

In small graph tasks, AWB-GCN achieves slightly better acceleration than NDPGNN, benefiting from its use of graph sparsity strategies to enhance computational efficiency. However, as the graph size increases, NDPGNN demonstrates superior acceleration, showcasing better scalability for large graph acceleration tasks. Additionally, NDPGNN supports GraphSage-Pool and G-GCN models that include Edge-Update operations, which AWB-GCN cannot support, highlighting NDPGNN's scalability across different GNN models.

b) Energy Consumption: Regarding energy consumption, as illustrated in Fig. 9, NDPGNN employs a node-wise processing method, resulting in energy savings of 8.00×, 8.90×, and 8.57× compared to HyGCN, AWB-GCN, and HaGNN, respectively.

The reduction in energy consumption can be attributed primarily to two factors. Firstly, there is a direct impact on energy consumption from improvements in performance. For instance, the acceleration of the computational process in Graph Neural Networks (GNNs) leads to an overall reduction in energy consumption, as the time required for computation

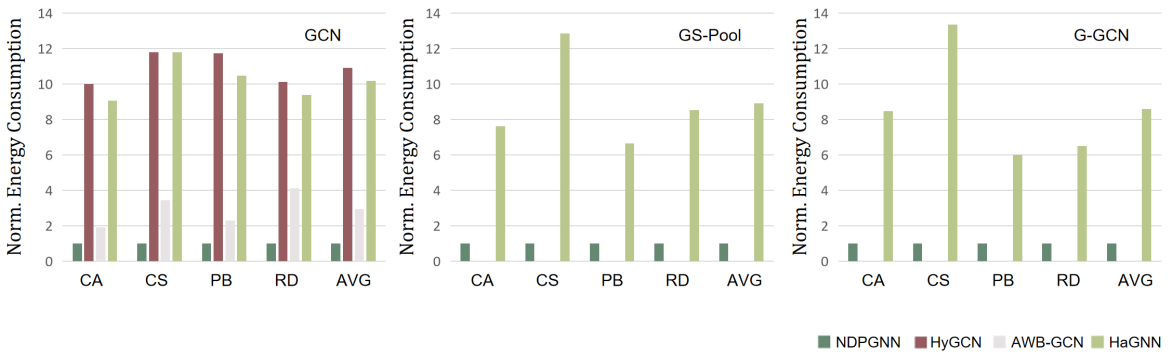

Fig. 9. Energy Consumption Compared with Other Baseline Accelerators, Normalized to NDPGNN.

significantly decreases. Secondly, optimizing the memory access process can effectively reduce the number of memory accesses, thereby lowering the energy consumption of the memory subsystem. This can be achieved by increasing the proportion of effective data in memory, allowing the system to utilize memory resources more efficiently and consequently reducing energy consumption. Thus, through optimization in these two aspects, the system can simultaneously improve performance while lowering energy consumption, demonstrating effective energy optimization outcomes.

c) Training Speedup: Figure 10 depicts the training acceleration capabilities of NDPGNN in the context of GNN training. We conducted a comprehensive comparative analysis by evaluating the training acceleration capabilities of NDPGNN against DGL-CPU, DGL-GPU, and the current state-of-the-art GNN training accelerator GNNear. To assess NDPGNN's performance, we employed three popular GNN models: GCN, GraphSage-Pool, and G-GCN. All performance metrics were normalized to the DGL-CPU benchmark to ensure a consistent evaluation framework.

The comparative results presented in Fig. 10 unequivocally demonstrate that NDPGNN exhibits superior training acceleration capabilities compared to the state-of-the-art GNN accelerator. These findings underscore the effectiveness and efficiency of NDPGNN in accelerating GNN training processes, highlighting its potential as a high-performance solution in the field of graph-based machine learning tasks.

d) Ablative Analysis: In order to deeply analyze the impact of various optimization strategies on the acceleration performance of NDPGNN, we conducted a series of ablative experiments. We compared the GCN model on the Cora dataset with HyGCN, presenting the performance improvements brought by the near-memory heterogeneous design method proposed in this paper, as well as the three optimization strategies during GNN inference acceleration. Figure 11 presents the results of these ablative analyses. In these experiments, we use HyGCN as the baseline accelerator. NDP Vanilla means using the original version of the near-data processing method for GNN acceleration but does not adopt the optimization strategy proposed in this article. This method still consists of two parts: the NDP module and MMPE, forming a heterogeneous system. Compared to NDP Vanilla, NDP Packing, NDP Aggr., and NDP-SN (NDPGNN) sequentially add one optimization method over the previous model. These methods are data-packing, data scheduling in aggregation, and sparse neighbors' rearrangement, respectively. This allows us to evaluate the contribution of each optimization strategy to system performance. The final NDP-SN incorporates all optimization strategies and therefore represents the proposed NDPGNN architecture. NDP Packing introduces data packaging technology on the basis of NDP Vanilla, thereby providing more efficient data for long-distance data transmission, thus improving the overall bandwidth utilization of the system. Based on NDP Packing, NDP Aggr. further enhances the data scheduling and optimization in the Aggregation operation. By increasing the number of data reuses on the chip, it effectively utilizes the time locality of the data and reduces the system's memory access requirements, thus significantly improving the efficiency of the system. NDPGNN is the complete system design described in this article, including all optimization strategies introduced in the article. Specifically, compared with NDP Aggr., NDPGNN adds strategies such as sub-graph result reuse and sub-graph sparse connection node arrangement in terms of system scheduling optimization, which further improves the overall performance of the system. The above data are normalized to HyGCN. These ablative experimental results reveal the respective contributions of different optimization strategies in the acceleration performance of NDPGNN, providing a useful reference for understanding system optimization, and also better reflecting the optimization proposed in this article. effectiveness of the strategy.

VI. Discussion

A. Opportunity for Large-scale Graph Processing

As research in GNNs advances, the size of graph datasets continues to grow, leading to a substantial increase in the demand for memory bandwidth. To effectively address this challenge, distributed storage structures have gradually emerged, offering robust support for the processing of expansive graph datasets. CXL technology, with its high-speed memory access and data transfer capabilities, in conjunction with high-bandwidth memory technology, has the capacity to significantly enhance the system's data throughput [22]. Leveraging the support of CXL technology, the NDPGNN architecture

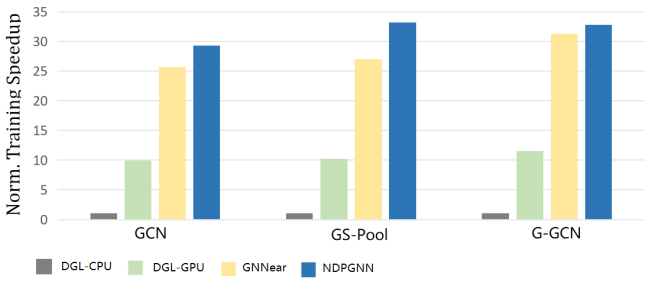

Fig. 10. Training Speedup. Normalized to DGL-CPU.

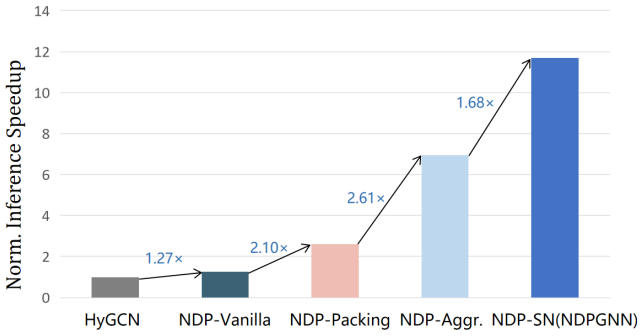

Fig. 11. Ablative Experiment. Showing the Contribution of Each Method to the Improvement of Performance.

approach holds the potential for achieving more impressive results.

B. Cost-effectiveness for the Architecture Based on Common Components

In-memory computing plays a crucial role in accelerating graph neural networks. However, the high costs associated with the design of new memory systems limit its widespread adoption in the short term. In contrast to traditional in-memory computing, near-data processing doesn't require costly modifications and production of storage chips.

VII. Conclusion

In this work, we introduce NDPGNN. It employs near-data processing modules to handle data-intensive computations and utilizes matrix-multiplication arrays for compute-intensive tasks. To overcome the difficulty of fine-grained memory access, it improves bandwidth resource usage through data packaging. NDPGNN adopts a node-centered mini-batch training method to enhance overall computational efficiency in the GNN training aspect. Overall, NDPGNN improves 5.68× system performance and reduces 8.49× energy consumption overhead compared to previous methods.

References

[1] Z. Lin, et al., 2020. PaGraph: Scaling GNN training on large graphs via computation-aware caching. In SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020. ACM, 401–415. DOI: 10.1145/3419111.3421281

[2] I. Gog, et al., 2016. Firmament: Fast, centralized cluster scheduling at scale. In Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation. 99–115.

[3] M. Serafini and H. Guan, "Scalable Graph Neural Network Training: The Case for Sampling," Oper. Syst. Rev., vol. 55, no. 1, pp. 68–76, 2021. DOI: 10.1145/3469379.3469387

[4] M. Fey, et al., Fast graph representation learning with pytorch geometric. arXiv:, 2019.

[5] T. N. Kipf, et al., "Semi-supervised classification with graph convolutional networks," International Conference on Learning Representations, 2017.

[6] W. Hamilton, et al., "Inductive representation learning on large graphs," Adv. Neural Inf. Process. Syst., 2017.

[7] X. Bresson, et al., "Residual gated graph convnets," arXiv preprint arXiv:, 2017.

[8] M. Zhang, et al., "Link prediction based on graph neural networks," Adv. Neural Inf. Process. Syst., vol. 31, pp. 5165–5175, 2018.

[9] M. Yan, et al., "Hygcn: A gcn accelerator with hybrid architecture," in 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA). IEEE, 2020, pp. 15–29. DOI: 10.1109/HPCA47549.2020.00012

[10] T. Geng, 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. DOI: 10.1109/MICRO50266.2020.00079

[11] A. Auten, et al., "Hardware acceleration of graph neural networks," in 2020 57th ACM/IEEE Design Automation Conference (DAC). IEEE, 2020, pp. 1–6. DOI: 10.1109/DAC18072.2020.9218751

[12] Zonghan Wu, et al. 2020. A comprehensive survey on graph neural networks. IEEE Transactions on Neural Networks and Learning Systems (2020).

[13] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible dram simulator," IEEE Comput. Archit. Lett., vol. 15, no. 1, pp. 45–49, 2015. DOI: 10.1109/LCA.2015.2414456

[14] Z. Jia, et al., (2019). Redundancy-Free Computation Graphs for Graph Neural Networks. arXiv.

[15] J. E. Gonzalez, et al., "PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs," in Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI), 2012, pp. 17–30.

[16] P. Veličković, et al., 2018. Graph Attention Networks. In Proceedings of the 6th International Conference on Learning Representations (ICLR'18).

[17] V. G. Satorras, et al., 2018. Few-Shot Learning with Graph Neural Networks. In Proceedings of the 6th International Conference on Learning Representations (ICLR'18).

[18] N. Challapalle, et al., "GaaS-X: Graph Analytics Accelerator Supporting Sparse Data Representation using Crossbar Architectures," in Proceedings of the International Symposium on Computer Architecture (ISCA), 2020, pp. 433–445. DOI: 10.1109/ISCA45697.2020.00044

[19] Y. Huang, et al., "RAGra: Leveraging Monolithic 3D ReRAM for MassivelyParallel Graph Processing," in Proceedings of the Design, Automation & Test in Europe (DATE), 2019, pp. 1273–1276. DOI: 10.23919/DATE.2019.8715192

[20] Z. Wu, et al., "A comprehensive survey on graph neural networks," IEEE Trans. Neural Netw. Learn. Syst., 2020.

[21] K. Xu, et al., "How powerful are graph neural networks?" arXiv preprint arXiv:, 2018.

[22] W. Huangfu, et al., BEACON: Scalable Near-Data-Processing Accelerators for Genome Analysis near Memory Pool with the CXL Support, 2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO), Chicago, IL, USA, 2022, pp. 727-743.

[23] DGL, [n.d.][n.d.]. DGL Framework. https://github.com/dmlc/dgl.

[24] Z. Zhou, et al., 2023. GNNear: Accelerating Full-Batch Training of Graph Neural Networks with near-Memory Processing. In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT '22). Association for Computing Machinery, New York, NY, USA, 54–68. https://doi.org/DOI: 10.1145/3559009.3569670

[25] W. Fan, et al., "Graph neural networks for social recommendation," in The World Wide Web Conference, 2019, pp. 417–426. DOI: 10.1145/3308558.3313488

[26] A. Lerer, et al., "Pytorch-biggraph: A large scale graph embedding system," Proceedings of Machine Learning and Systems, vol. 1, pp. 120–131, 2019.

[27] X. Weng, et al., "Joint 3d tracking and forecasting with graph neural network and diversity sampling," arXiv preprint arXiv:, vol. 2, no. 6.2, p. 1, 2020.

[28] B. Jin, et al., "Multi-behavior recommendation with graph convolutional networks," in Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval, 2020, pp. 659–668.
[29] Z. Yang and S. Dong, "Hagerec: Hierarchical attention graph convolutional network incorporating knowledge graph for explainable recommendation," Knowl. Base. Syst., vol. 204, p. 106194, 2020. DOI: 10.1016/j.knosys.2020.106194
[30] M. Klimke, et al., "Cooperative behavior planning for automated driving using graph neural networks," in 2022 IEEE Intelligent Vehicles Symposium (IV). IEEE, 2022, pp. 167–174. DOI: 10.1109/IV51971.2022.9827230
[31] K. He, et al., "Deep residual learning for image recognition," in Proceedings of the IEEE Conference on computer vision and pattern recognition, 2016, pp. 770–778.
[32] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," Commun. ACM, vol. 60, no. 6, pp. 84–90, 2017. DOI: 10.1145/3065386
[33] C. Szegedy, et al., "Going deeper with convolutions," in Proceedings of the IEEE conference on computer vision and pattern recognition, 2015, pp. 1–9.
[34] J. Gilmer, et al., "Neural message passing for quantum chemistry," in International conference on machine learning. PMLR, 2017, pp. 1263–1272.
[35] A. Abou-Rjeili and G. Karypis, "Multilevel algorithms for partitioning power-law graphs," in Proceedings 20th IEEE International Parallel and Distributed Processing Symposium. IEEE, 2006, pp. 10–pp.
[36] J. E. Gonzalez, et al., "Powergraph: Distributed graph-parallel computation on natural graphs," in Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12), 2012, pp. 17–30.
[37] C. Xie, et al., "Distributed power-law graph computing: Theoretical and empirical analysis," Advances in neural information processing systems, vol. 27, 2014.
[38] J. Ahn, et al., 2015. A scalable processing-in-memory accelerator for parallel graph processing. In Proceedings of the 42nd Annual International Symposium on Computer Architecture. 105–117. DOI: 10.1145/2749469.2750386
[39] G. Dai, T. Huang, Y. Chi, J. Zhao, G. Sun, Y. Liu, et al., "Graphh: A processing-in-memory architecture for large-scale graph processing," IEEE Trans. Comput. Aided Des. Integrated Circ. Syst., vol. 38, no. 4, pp. 640–653, 2018. DOI: 10.1109/TCAD.2018.2821565
[40] L. Nai, et al., 2017. GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks. In 2017 IEEE International Symposium on High Performance Computer Architecture, HPCA 2017, Austin, TX, USA, February 4-8, 2017. IEEE Computer Society, 457–468. https://doi.org/DOI: 10.1109/HPCA.2017.54
[41] M. Zhang, et al., 2018. GraphP: Reducing communication for PIM-based graph processing with efficient data partition. In 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA). IEEE, 544–557. DOI: 10.1109/HPCA.2018.00053
[42] Y. Zhuo, et al., 2019. Graphq: Scalable pim-based graph processing. In Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture. 712–725. DOI: 10.1145/3352460.3358256
[43] J. Chen, et al., 2018. Fastgcn: fast learning with graph convolutional networks via importance sampling. arXiv preprint arXiv: (2018).
[44] W.-L. Chiang, et al., 2019. 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 and Data Mining. 257–266. DOI: 10.1145/3292500.3330925
[45] W. Hamilton, et al., Inductive representation learning on large graphs. In Advances in neural information processing systems. 1024–1034.
[46] H. Zeng, et al., 2019. Graphsaint: Graph sampling based inductive learning method. arXiv preprint arXiv: (2019).
[47] Z. Cai, et al., DGCL: an efficient communication library for distributed GNN training. In EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021, Antonio Barbalace, Pramod Bhatotia, Lorenzo Alvisi, and Cristian Cadar (Eds.). ACM, 130–144. https://doi.org/DOI: 10.1145/3447786. 3456233 DOI: 10.1145/3447786.3456233
[48] Z. Jia, et al., "Improving the accuracy, scalability, and performance of graph neural networks with roc," Proceedings of Machine Learning and Systems, vol. 2, pp. 187–198, 2020.
[49] M. Vasimuddin, et al., 2021. DistGNN: Scalable Distributed Training for Large-Scale Graph Neural Networks. arXiv preprint arXiv: (2021).
[50] A. Tripathy, et al. 2020. Reducing communication in graph neural network training. In SC20: International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE, 1–14. DOI: 10.1109/SC41405.2020.00074
[51] R. Hwang, et al., "Grow: A row-stationary sparse-dense gemm accelerator for memory-efficient graph convolutional neural networks," in 2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA). IEEE, 2023, pp. 42–55. DOI: 10.1109/HPCA56546.2023.10070983
[52] R. Sarkar, et al., "Flowgnn: A dataflow architecture for real-time workload-agnostic graph neural network inference," in 2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA). IEEE, 2023, pp. 1099–1112. DOI: 10.1109/HPCA56546.2023.10071015
[53] M. Yoo, et al., "Sgcn: Exploiting compressed-sparse features in deep graph convolutional network accelerators," in 2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA). IEEE, 2023, pp. 1–14. DOI: 10.1109/HPCA56546.2023.10071102
[54] G. Karypis and V. Kumar. Metis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices, 1997.

Haoyang Wang is a Ph.D candidate in School of Computer Science, Northwestern Polytechnical University. She received M.E. in Northwestern Polytechnical University. Her main research interests include Computer Architecture, Machine Learning and Neuromorphic Architecture.

Shengbing Zhang received the M.E. and Ph.D. degree in computer science and technology from Northwestern Polytechnical University, Xi'an, China, in 1991 and 1998, respectively. He is currently a Full Professor with Northwestern Polytechnical University. His main research interests include Architecture, Computer engineering and technology and Neuromorphic Architecture, VLSI.

Xiaoya Fan received the M.E. and Ph.D. degree in computer science and technology from Northwestern Polytechnical University, Xi'an, China, in 1985 and 1989. He is currently a Full Professor with Northwestern Polytechnical University. His research interests include Digital Circuit Design, SoC Design, and Mixed-signal Circuit Design.

Zhao Yang is an assistant professor in School of Future Transportation, Chang'an University. His research interests include computer architecture, machine learning, and perva- sive computing. He is currently exploring the challenges of federated learning on heterogeneous mobile edge computing platforms.

Meng zhang , Ph.D, Associate professor, Member of China Computer Federation. His main research interests include Computer Architecture, Machine Learning and Neuromorphic Architecture, VLSI.