GraNDe: Efficient Near-Data Processing Architecture for Graph Neural Networks

Sungmin Yun†, Hwayong Nam†, Jaehyun Park†, Byeongho Kim§,
Jung Ho Ahn†, *Senior Member, IEEE* and Eojin Lee‡
†Seoul National University, ‡Inha University, §Samsung Electronics

Abstract—Graph Neural Network (GNN) models have attracted attention, given their high accuracy in interpreting graph data. One of the primary building blocks of a GNN model is aggregation, which gathers and averages the feature vectors corresponding to the nodes adjacent to each node. Aggregation works by multiplying the adjacency and feature matrices. The size of both matrices exceeds the on-chip cache capacity for many realistic datasets, and the adjacency matrix is highly sparse. These characteristics lead to little data reuse, causing intensive main-memory accesses during the aggregation process. Thus, aggregation exhibits memory-intensive characteristics and dominates most of the total execution time.

In this paper, we propose GraNDe, an NDP architecture that accelerates memory-intensive aggregation operations by locating NDP modules near DRAM datapath to exploit rank-level parallelism. GraNDe maximizes bandwidth utilization by separating the memory channel path with the buffer chip in between so that pre-/post-processing in the host processor and reduction in NDP modules operate simultaneously. By exploring the preferred data mappings of the operand matrices to DRAM ranks, we architect GraNDe to support adaptive matrix mapping that applies the optimal mapping for each layer depending on the dimension of the layer and the configuration of a memory system. We also propose adj-bundle broadcasting and re-tiling optimizations to reduce the transfer time for adjacency matrix data and to improve feature vector data reusability by exploiting tiling with consideration of adjacency between nodes. GraNDe achieves 3.01× and 1.69× on average, and up to 4.00× and 1.98× speedups of GCN aggregation over the baseline system and the state-of-the-art NDP architecture for GCN, respectively.

Index Terms—Near-data processing, DRAM, graph neural networks

1 Introduction

Existing neural network models (e.g., CNN, RNN, and Transformer [4], [6], [10], [20], [35]) demonstrate high performance for processing data expressed as vectors in a Euclidean space, such as image and speech data. However, these models are not suitable for processing non-Euclidean graph data, which is more complex in structure compared to image or speech data. Graph Neural Networks (GNNs) have recently emerged to process graph data.

A GNN model receives an input graph and infers the meaning of the entire graph, the nodes, or the links between the nodes in a graph. A graph data consists of the graph structure and the features of nodes, which are represented as an adjacency matrix and a feature matrix, respectively. The GNN model consists of several layers, each mainly composed of a combination phase and an aggregation phase. The combination phase is typically implemented by a multi-layer perceptron (MLP). The aggregation phase gathers the features of adjacent nodes for each node and reduces the gathered features into a single intermediate vector through the element-wise average or summation.

These two phases of the layer have different characteristics. The combination phase is compute-intensive, so it is well accelerated by traditional accelerators [15], [21]. In contrast, the aggregation phase is memory-intensive in general because the size of the adjacency matrix and the feature matrix, which can exceed hundreds of GBs depending on the dataset, mostly do not fit in on-chip memory [2], [3]. This characteristic continues as the graph dataset becomes larger [12]. Moreover, the adjacency matrices of typical GNN datasets are highly sparse, with a density of less than 0.01% for the graphs in Table 1. Therefore, the gathered feature vectors in the aggregation phases are rarely reused. These characteristics make the aggregation phase account for a large portion of the total execution time. So it is crucial to accelerate the aggregation phase. Although a large body of SpMM accelerators [5], [11], [26], [30], [31], [32], [39], [43] has been proposed, they do not focus on GNN models.

Prior studies [7], [24], [27], [41] accelerate GNN models by adopting high bandwidth memory (HBM) to meet the memory bandwidth demand for processing aggregation. However, the size of the graph dataset (up to hundreds of GB) exceeds the capacity of HBM [12] devices. Therefore, a memory system with a capacity larger than that of HBM (e.g., a dual in-line memory module (DIMM)-based memory system) is required to perform GNN. Using tens or hundreds of GPUs (HBMs) could also cover the required memory capacity. However, the aggregation operation is

• *A preliminary version of this paper was published at IEEE Computer Architecture Letters (CAL) [42]. This work was partly supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (NRF-2018R1A5A1059921 and NRF-2022R1F1A1062826), Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) [NO.2021-0-01343, Artificial Intelligence Graduate School Program (Seoul National University)], and the Samsung Electronics. Eojin Lee is the corresponding author.*

TABLE 1
The GNN datasets [12], [23]. The adjacency matrices are in the CSR format. Each feature vector consists of 256 FP32 elements.

Dataset	arxiv	amazon	mag	products	papers
Number of nodes (N)	169 K	410 K	736 K	2.45 M	111 M
Number of edges	2.48 M	5.29M	11.53 M	126.17 M	3.34 B
Dimension of an input feature vector (d_1)	128	96	128	100	128
Number of output classes	40	22	349	47	172
Average degree	14.7	12.9	15.7	51.5	30.1
Adjacency matrix density	8.7E-5	3.1E-5	2.1E-5	2.1E-5	2.7E-7
Adjacency matrix size	20.0 MB	42.7 MB	93.0 MB	1.0 GB	26.8 GB
Feature matrix size	173.4 MB	420.1 MB	754.1 MB	2.5 GB	113.7 GB

simple (e.g., element-wise summation) and does not require powerful computational units. For this reason, using numerous expensive GPUs is inefficient. Also, it is difficult to expect performance gain on large datasets with previous ASIC-based accelerator studies because of their small on-chip memory size.

To solve these problems, Near-Data Processing (NDP) architectures for GNN [34], [45] have emerged recently, which simultaneously perform data access and aggregation operation in multiple DRAM ranks. They place processing units for each rank in the buffer chip to accelerate aggregation by utilizing rank-level parallelism. The NDP architectures perform an aggregation phase through three stages in common. First, the host should generate instructions and send them to processing units (pre-processing). Second, the processing units decode the instructions, read feature vectors from the DRAM, and compute them (reduction). Finally, the host reads the partial sum vectors computed by each processing unit, performs final reduction, and writes them to DRAM again (post-processing).

Prior NDP architectures focused only on accelerating the reduction using amplified bandwidth between a buffer chip and DRAM devices. Unlike reduction, pre-/post-processing in these architectures must use a path between the host and the buffer chips. A single channel shares the path between the host and the buffer chip, enforcing pre- and post-processing to be performed sequentially. When the number of ranks in a channel increases, the time required for pre-/post-processing also increases, which leads to less performance gains. Also, they lack consideration for DRAM mapping in the adjacency matrix and feature matrix, which decides the time spent on pre-/post-processing.

In this paper, we propose GraNDe, a DIMM-based NDP architecture to accelerate the aggregation phase of GNN. GraNDe implements NDP modules for each DRAM rank in the buffer chip of DIMM. Each NDP module performs an aggregation operation in parallel using the feature matrix portion stored in the corresponding rank, reducing GNN execution time effectively. GraNDe locates buffers that can be accessed by both the host processor and NDP modules. These buffers separate the memory channel path to the host-side and DRAM-side. Using these buffers and host-/DRAM-side path, GraNDe performs pre-processing, post-processing, and reduction simultaneously, which is unable in prior NDP architectures. Moreover, we explore granular mappings of operand matrices to DRAM ranks in the aggregation phase and propose adaptive operand matrix mapping. Therefore, each NDP module operates the aggregation phase using the preferred mapping for each layer based on the feature vector dimension. Also, GraNDe manages the movement of distributed adjacency matrix data differently depending on each feature matrix mapping.

We also propose two optimizations for GraNDe to improve memory bandwidth utilization and data reusability. To reduce the time to transmit adjacency information (adj-bundle) that is common to multiple ranks sequentially, GraNDe supports *adj-bundle broadcasting* that allows the host processor to send adj-bundle to all ranks in a channel simultaneously. Also, we propose a *re-tiling* mechanism that constructs tiles considering the adjacency between nodes and reuses the feature vector data repeated in a tile. It effectively decreases execution time by reducing the number of DRAM reads for aggregation without a costly graph reconstruction process.

Our evaluation demonstrates that GraNDe accelerates the aggregation phase of GCN, the most representative model of GNN, across various graph datasets and layer dimensions. Using layer-by-layer optimal data mapping, which efficiently exploits higher memory bandwidth with rank-level parallelism, and applying two optimization techniques, GraNDe shows aggregation performance improvement of up to 4.00× and 3.01× on average compared to the baseline system using ogbn [12] and amazon [23] datasets. Also, including the time for combination and activation operation processed in the host processor, end-to-end GCN performance is improved up to 3.02× and 1.91× on average in GraNDe.

We make the following key contributions:

- We propose GraNDe, an NDP architecture for GNN, which accelerates the aggregation phase by choosing the most effective mapping scheme and by decoupling the memory channel to operate data movement and reduction simultaneously.
- We propose adj-bundle broadcasting and re-tiling optimizations to improve memory bandwidth utilization and data reusability.
- Our evaluation shows that GraNDe improves GCN aggregation performance up to 4.00× and 1.98× compared to the baseline and the state-of-the-art NDP architecture, respectively.

2 Graph Neural Network

Graph Neural Network (GNN) is the deep-learning-based model used for analyzing graphs. It takes graph data as input and infers the meaning of the entire graph, the nodes, or the links between the nodes in the graph. Graph data consists of a feature matrix and an adjacency matrix. The

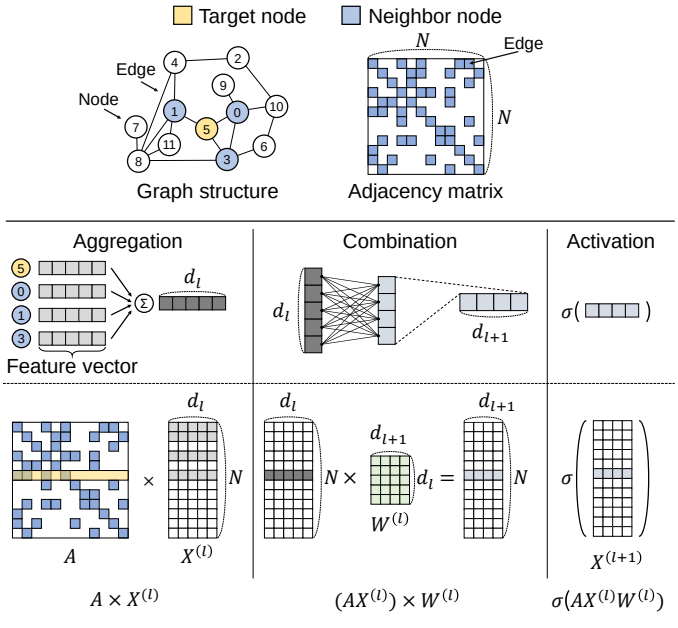

Fig. 1. Graph data organization and overview of a layer in GCNs. Adjacency matrix (A) expresses edge information between N number of nodes in a graph. $X^{(l)}$, d_l, and $W^{(l)}$ represent the input feature matrix, feature vector dimension, and weight matrix of l^{th} layer, respectively. σ indicates activation function, such as ReLU.

feature matrix comprises the feature vectors of the nodes, each representing each node's characteristics. Therefore, the column size of the feature matrix is determined by the feature vector dimension, where a larger vector dimension can encode a node feature in a more detailed manner. The adjacency matrix is a square matrix that expresses the edge information, where both the row index and column index indicate the node index. Because the adjacency matrix is sparse, it is expressed in a sparse format, such as CSR (compressed sparse row).

The GNN models consist of several layers, and each layer consists of two phases: aggregation and combination. During aggregation, each node (target node) gathers (e.g., calculates the average) the feature vectors of its neighbor node and generates a single feature vector through an aggregated function. Then, the aggregated individual feature vectors are passed through an MLP layer called a combination phase. The result from the combination phase of each node goes through an activation function to produce a single output vector. The aggregated function and combination operation are slightly different for each GNN model (e.g., GCN [19], GIN [40], and SAGEConv [9]). Hereafter, we will explain based on GCN, which is the most representative of GNN models [45].

The operation of a GNN layer is expressed as follows:

$$X^{(l+1)} = \sigma(\widehat{A} \times X^{(l)} \times W^{(l)})$$

Here, $\widehat{A}$ is a normalized adjacency matrix, and $X^{(l)}$ and $W^{(l)}$ are the feature matrix and the weight matrix of the l^{th} layer, respectively. Hereafter, A denotes the normalized adjacency matrix for simplicity. The former matrix multiplication $(A \times X^{(l)})$ corresponds to the aggregation phase. The latter multiplies the output value of aggregation by the weight matrix $(A \times X^{(l)}) \times W^{(l)}$, corresponding to the combination phase. $\sigma(X)$ is an activation function (e.g., ReLU).

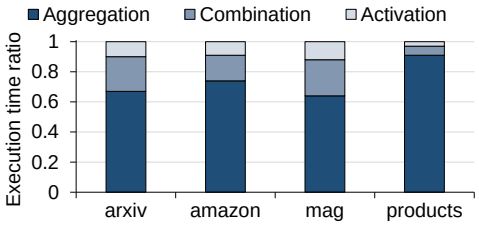

Fig. 2. Execution time breakdown of GCN with the evaluated datasets [12], [23] on Intel Xeon Gold 6138 with four memory channels (DDR4-2400).

$X^{(l+1)}$ is used as the input feature matrix of the $(l+1)^{th}$ layer. As one node is only connected to less than 0.1% of the nodes in most graphs we target, the adjacency matrix is expressed in a sparse format, such as CSR (compressed sparse row), whereas the feature and weight matrices are dense. Therefore, the aggregation operation takes the form of sparse-dense matrix multiplication (SpMM), and the combination takes the form of general dense-dense matrix multiplication (GEMM).

In this paper, we use realistic datasets [12], [23] described in Table 1 to evaluate our proposal. The datasets have hundreds of thousands or more nodes, millions or more symmetrically located (i.e., undirected) edges, and the density of the adjacency matrix is less than 0.1%. As the adjacency matrix and the feature matrix each have a size of 100s of MBs to 100s of GBs, they cannot fit into the on-chip memory of the host system. Instead, they must be stored in the main memory.

The aggregation phase is memory-intensive and accounts for a large portion of the total execution time. Besides the large sizes of the adjacency and feature matrices, the data of these matrices are rarely reused, meaning that the SpMM requires access to the main memory almost every time. These make the memory bandwidth determine the performance of the aggregation phase. As shown in Figure 2, our experiment on a real system (Intel Xeon Gold 6138 with four memory channels, each equipped with two DDR4-2400 DIMMs with two ranks per DIMM) demonstrates that aggregation accounts for more than two-thirds of the total execution time on average (*arxiv*: 67%, *amazon*: 74%, *mag*: 64%, and *products*: 91%). Moreover, processing a larger graph can increase the portion of aggregation in execution time due to the memory bandwidth bottleneck. *Therefore, it is important to accelerate the aggregation phase to improve the GNN performance.*

3 DIMM-based Memory Systems and NDP Architectures

A modern main-memory system consists of one or more channels, each connecting a memory controller (MC) to one or more DIMMs. A DIMM has multiple ranks, each consisting of DRAM devices that operate in tandem by receiving the same command/address (C/A) signal. The DIMM for servers employs a buffer device to repeat the C/A signal to mitigate signal integrity issues when populating multiple DRAM devices. The buffer device also repeats the data signal if necessary.

Because multiple ranks are connected to an MC through a shared datapath (forming a memory channel), only one

rank can occupy the datapath and transfer data at any given time. However, if we exploit the near-data processing (NDP) architecture that locates the processing unit (PU) at a buffer device located in the middle of the DRAM datapath [1], [29], multiple ranks can concurrently transfer data to the PU. Then, the PUs in all ranks can operate simultaneously, accelerating the target operation by utilizing the amplified internal bandwidth that is identical to the channel bandwidth multiplied by the number of ranks.

GNNear [45] and G-NMP [34] improve GNN performance given their use of the DIMM-based NDP architecture. Similar to the existing NDP studies [16], [17], [22], [29], these architectures use custom instruction set architecture for NDP operations. The host processor generates instructions for NDP and sends them to a PU. The PU decodes the instructions, generates DRAM commands, reads data from DRAM, and processes the GNN operation. The size of an NDP instruction is large as it includes the address and size information of the operand data for GNN. Thus, the command/address (C/A) path has insufficient bandwidth to fully utilize PUs, so these previous works proposed transferring NDP instructions through the data (DQ) path.

NDP architectures mostly require pre-/post-processing, which is done by the host. In the case of GNN, because each PU produces an incomplete output feature matrix (e.g., partial sum or a fraction of output), post-processing is required to send the incomplete result to the host and obtain the complete output feature matrix by reduction or concatenation. Moreover, after post-processing, the final output feature matrix is used as the input feature matrix for the next layer, so it must be written from the host back to DRAM. However, as listed in Table 1, the size of the feature matrix is up to several hundred GBs, so it takes a long time to write the output matrix to DRAM. Because the data path of a memory channel is occupied during this time, sending another instruction to a PU through the DQ path is impossible, preventing PUs from utilizing the amplified bandwidth. The same problem occurs when reading adjacency matrix data, whose size reaches tens of GB, from DRAM to host for the pre-processing operation (i.e., generating instructions for NDP).

Another important consideration in the NDP architecture for GNNs is operand matrix mapping on DRAM. Previous works [34], [45] considered the data mapping of the feature matrix suitable for DIMMs. The operand matrices don't fit in one rank due to their large size; instead, they must be stored over multiple ranks. GNNear partitioned a feature matrix horizontally and mapped each partition to each DIMM. G-NMP proposed an adaptive mapping that partitions the feature matrix vertically or horizontally depending on the feature vector dimension. However, feature vector mappings with finer grains must be considered for more efficient NDP operations. Moreover, it is necessary to consider the adjacency matrix mapping because it is too large to be stored within one rank.

4 GraNDe: An NDP Architecture for GNNs

We propose GraNDe, a DIMM-based NDP architecture that accelerates Graph Convolutional Networks. GraNDe locates processing units on DIMM's buffer devices and utilizes memory-channel's internal bandwidth to accelerate the aggregation phase of GNNs. This section describes the overall hardware architecture of GraNDe that fully utilizes the memory bandwidth. Then, we explore the data mapping strategies of operand matrices for NDP, including the adjacency matrix, and describe how GraNDe applies adaptive mapping for each layer in the aggregation operation. We also propose a re-tiling technique that can effectively improve data reuse when performing aggregation phases, maximizing the performance of GraNDe. GraNDe focuses on inference, but it can also be applied to the backpropagation phases of GNN training.

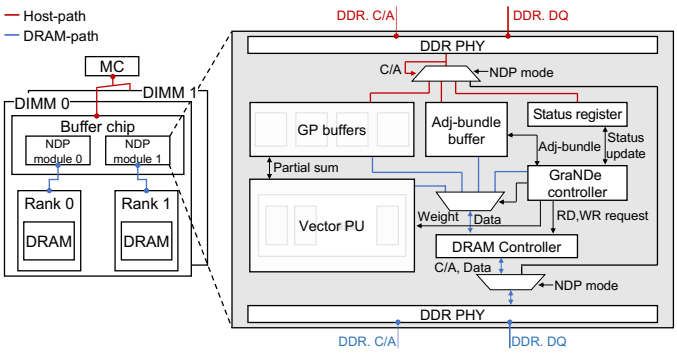

Fig. 3. An overview of the GraNDe architecture. A buffer chip on DIMM locates NDP modules for each rank, and each NDP module is mainly composed of a vector PU and buffers. The memory channel path is separated into host and DRAM paths with the buffer chip in between.

4.1 Overview of the GraNDe Architecture

GraNDe has NDP modules for each rank on the DIMM's buffer chip, which can handle adaptive feature vector mapping (see Figure 3). An NDP module consists of the adj-bundle buffer, four general purpose (GP) buffers, the status register, the vector processing unit (vector PU), the DRAM controller, the re-tiling unit, and the control unit. An adj-bundle buffer temporarily stores an adj-bundle, an adjacency matrix portion that is read from the DRAM rank where it resides. Four GP buffers of the same size store the output feature vectors or adj-bundles from different ranks, behaving as double-buffers. The status register represents the current status of the NDP module. These buffers and register separate the memory-channel path from a host processor to NDP modules (*Host-path*) and that from the NDP module to DRAM devices (*DRAM-path*). The vector PU is composed of multiply-accumulators and performs weighted vector summation for aggregation. The DRAM controller generates DRAM commands (e.g., read or write) for an NDP operation, and the control unit manages each component to operate according to the aggregation execution flow.

GraNDe maximizes the utilization of amplified bandwidth during the aggregation phase by exploiting the *Host-path* and the *DRAM-path* simultaneously. GraNDe places the buffers between *Host-path* and *DRAM-path*, which can be accessed by both host and DRAM in the NDP module; details of how the host accesses the buffers in the NDP module are described in Section 6. Thereby, GraNDe enables NDP modules to perform the reduction in *DRAM-path*, simultaneously with the pre-/post-processing operations in

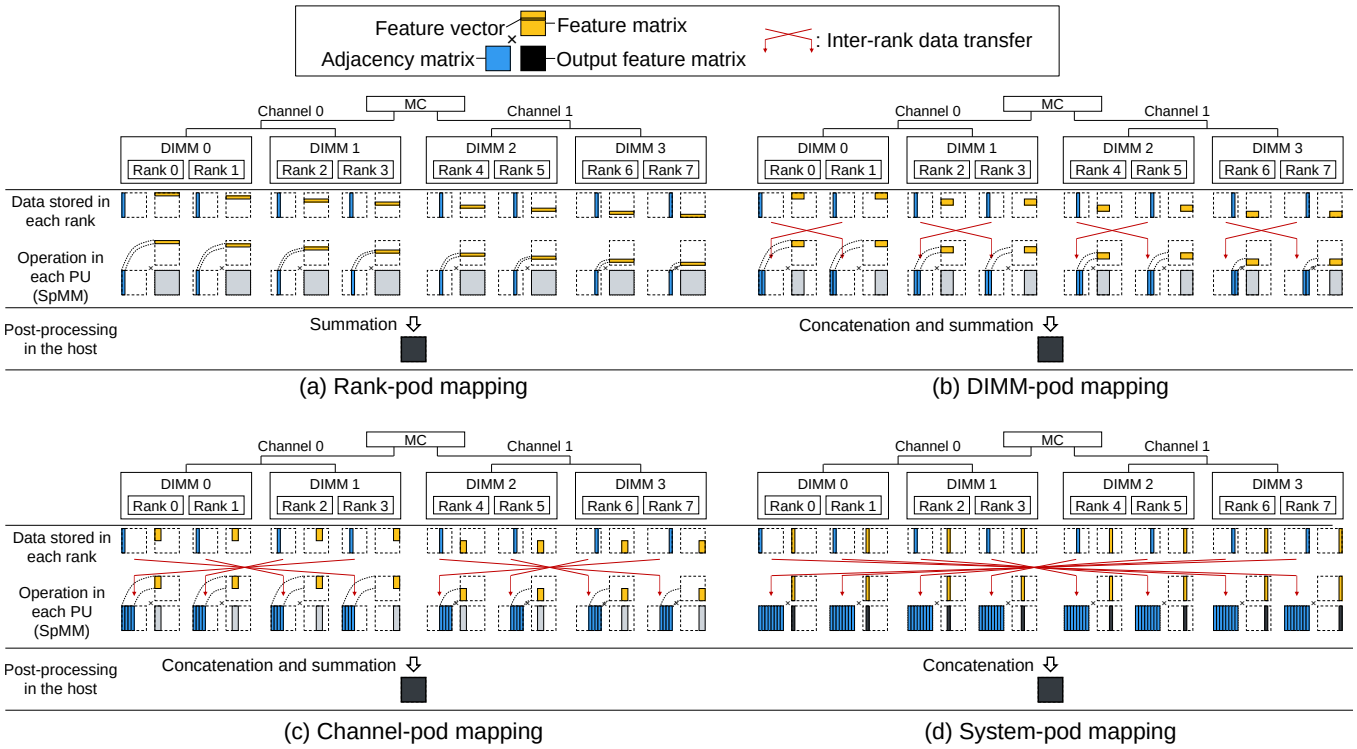

Fig. 4. Feature matrix mapping methods and SpMM (multiplying adjacency matrix and feature matrix) operation flow of (a) rank-pod, (b) DIMM-pod, (c) channel-pod, and (d) system-pod mapping. The adjacency matrix is partitioned vertically to minimize inter-rank data transfer. The type of post-processing in the host differs in each mapping.

the host processor using *Host-path*. Also, GraNDe supports aggregation operations for various data mappings, and exploits adaptive data mapping that switches between the data mappings for each layer. We describe the operation for each data mapping in the following section.

4.2 Exploring Data Mapping for GraNDe

The operand matrices can be partitioned and mapped to multiple ranks in the form advantageous to NDP instead of the conventional memory-interleaving. When the adjacency and feature matrices are distributed to multiple DRAM ranks, data movement among ranks is required. It is because each NDP module cannot calculate the output feature matrix entirely by using only the data in its rank. In GraNDe, adjacency matrix data would be transferred between ranks instead of the feature matrices because the size of the feature matrix is larger than the adjacency matrix in most GNN datasets (see Table 1).

In the case of the feature matrix, we assort matrix partitioning and mapping as four types of feature matrix mapping (rank, DIMM, channel, and system levels) depending on which memory-component units (hereafter, pod) store one feature vector entirely. A feature vector is distributed across the ranks in a pod, and each NDP module on each rank performs an aggregation phase. Therefore, the operation of NDP modules and data movement between ranks vary according to the distribution of the feature matrix. We explore differences in the operation for the four feature matrix mapping types.

Rank-pod mapping: In rank-pod mapping, the feature matrix is partitioned horizontally, where one feature vector is mapped to a single rank, and each matrix partition is mapped to each rank. Therefore, each NDP module performs incomplete aggregation using only the feature vectors allocated to the corresponding rank (Figure 4(a)) and produces partially computed output feature vectors. Therefore, the host post-processes the final output feature vector by adding the partial sums of the output feature vectors from all ranks.

System-pod mapping: When the feature matrix is partitioned vertically, each rank only has a fraction of each feature vector (Figure 4(d)). Thus, a pod consists of all ranks in the memory system. Each NDP module aggregates the fraction and generates a portion of the complete output feature vector. The host performs post-processing to compute the final output feature vector by concatenating the result of each rank.

DIMM-pod mapping, channel-pod mapping: The feature matrix can be partitioned both vertically and horizontally. An adjacency matrix is horizontally partitioned and mapped to each pod. Then each partition is vertically partitioned again and mapped to the rank inside the pod (Figure 4(b) and (c)). Partially computed output feature vectors are produced in each pod, and each NDP module in the pod calculates a portion of those vectors. The host reads the partial sum from all the pods and produces final output feature vectors through reduction. GraNDe can also use various pod mappings such as two-channel-pod and four channel-pod between channel-pod and system-pod mapping.

We also explore the mapping method of the adjacency matrix. In DIMM-pod, channel-pod, and system-pod mappings, each NDP module requires the adjacency matrix portion, which overlaps with different NDP modules. If each

rank stores adjacency information as much as necessary and hence in a duplicative manner, the required memory space would significantly exceed the capacity of a typical memory system. Especially for system-pod mapping, each rank must store the whole adjacency matrix. Therefore, the adjacency matrix is better partitioned and distributed to each rank to mitigate this capacity problem. In this case, an inter-rank data transfer is needed, where a rank receives the part of the adjacency matrix required for aggregation (henceforth adj-bundle) from other ranks. In rank-pod mapping, each rank stores the entire feature vectors for the part of graph nodes that differ from each other. Each NDP module requires a portion of the adjacency matrix, vertically partitioned, containing the corresponding node indices. Therefore, rank-pod mapping does not need inter-rank data transfer if the adjacency information of the feature vectors mapped to a rank is stored in the same rank. Considering these points, *GraNDe always partitions adjacency matrices vertically* and saves them with CSR format to minimize the costly inter-rank data transfer and avoid capacity issues.

The appropriate feature vector mapping for NDP differs depending on the feature vector dimension. With the rank-pod mapping (also DIMM- and channel-pod mappings), a load imbalance occurs in the number of feature vectors processed by NDP modules of each rank, exacerbating execution time. When performing the aggregation phase with system-pod mapping, the number of feature vectors processed in each rank is identical, so load balancing is not an issue. Instead, if the dimension of the feature vector is small, the size of the feature vector fraction stored in each rank could become smaller than the DRAM read granularity (e.g., 64 bytes in DDR5), so the remaining portion of the DRAM read data is not involved in aggregation. Also, system-pod mapping involves a lot of adj-bundle transfer to perform aggregation. Load imbalance is exacerbated in the order of system-, channel-, DIMM-, and rank-pod mapping, whereas DRAM bandwidth waste intensifies in the opposite order. Considering these trade-offs, *GraNDe supports multiple pod mapping and selects the better-performing mapping strategies depending on the dimension of a layer adaptively*.

4.3 Execution Flow

GNN initialization: During initialization, input operand matrices are loaded into DRAM using the proper mapping for GraNDe. The adjacency matrix is vertically partitioned and distributed to each rank, and the feature matrix is partitioned and mapped according to the feature vector dimension of the first layer. The weight matrix is only used in the combination phase, which is not the target of GraNDe, so it is distributed across multiple ranks with ordinary memory interleaving.

Prior systems for processing GNN: Figure 5 presents each timing diagram when we perform aggregation on a baseline system without NDP architecture, previous NDP architectures [34], [45], and GraNDe. We assume that the system equips only two ranks through one memory channel for the following explanation. In the baseline system, the host processor reads both adjacency and input feature matrix data required from DRAM and performs aggregation operations. In this case, the host can read data from only one rank per channel at a time (see Figure 5(a)).

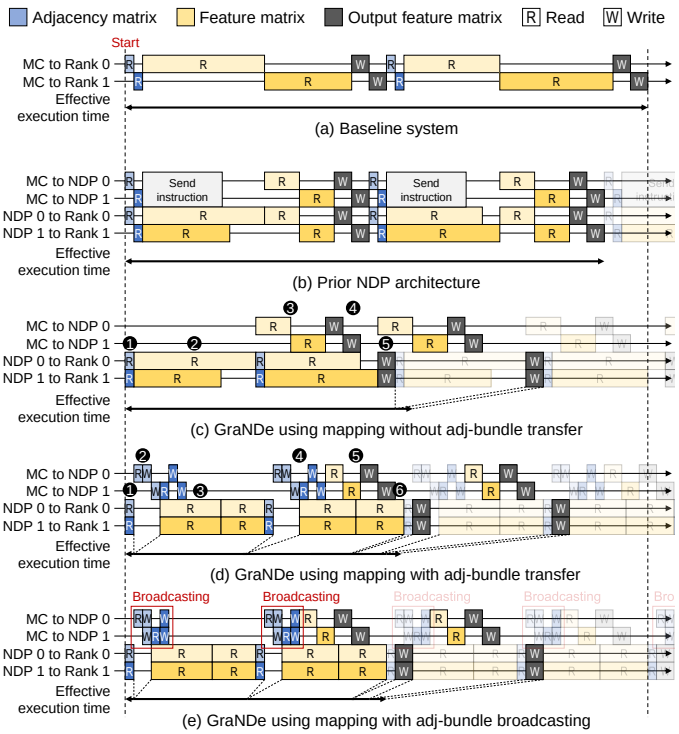

Fig. 5. An exemplar aggregation operation in the baseline system, previous NDP architectures [34], [45], and GraNDe with each pod mapping in a memory system with two ranks. The aggregation operation is divided into multiple execution windows, and the first two windows are colored clearly while the subsequent windows are transparent.

In the NDP architecture, the NDP module for each rank can read or write data from each DRAM rank simultaneously, reducing the execution time of the aggregation phase. Figure 5(b) depicts the execution flow of the previous NDP architecture. First, the host processor reads the adjacency matrix data, generates an instruction for NDP modules, and then sends the instruction to the NDP modules. The NDP modules receiving the instruction read the feature vectors simultaneously from each rank and perform aggregation. After NDP modules complete the operation corresponding to the instruction received, the host reads the result from each rank's NDP module, performs post-processing, such as reduction or concatenation, and writes the final output result back to the DRAM. During this process, the DQ path between the MC and NDP modules is occupied by transactions for post-processing, so the host can neither read the subsequent adjacency matrix data nor send the instructions, delaying the following aggregation operation.

GraNDe execution window: When GraNDe performs aggregation, it divides the total aggregation phase into multiple windows considering the size of the output buffer of an NDP module so that it can store all of the produced output feature vectors of a window. The host performs post-processing when each window completes. Within one window, the detailed execution sequence differs among the feature vector mappings that require inter-rank adj-bundle transfer (i.e., DIMM-, channel-, and system-pod mapping) and do not require (i.e., rank-pod mapping).

Execution flow without adj-bundle transfer: Figure 5(c)

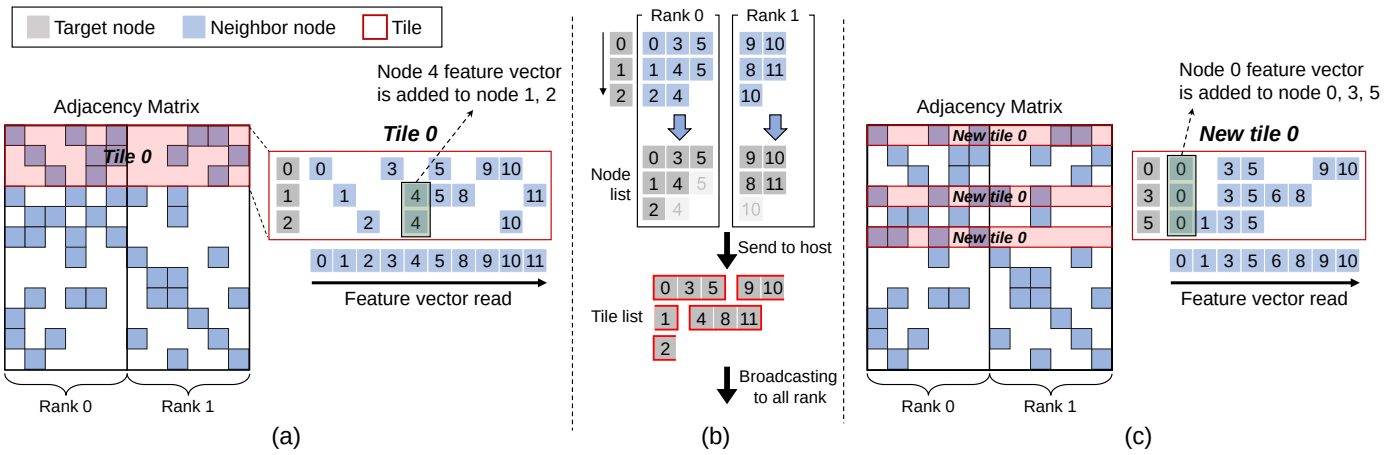

Fig. 6. The comparison between typical tiling and re-tiling. (a) Typical tiling groups nodes in the order of node index. (b) Re-tiling reads the adjacency matrix, lists nodes adjacent to each node sequentially, and makes a new tile list. (c) The new tile with the re-tiling technique has more overlapped nodes than typical tiling.

shows the timing diagram of executing GraNDe with rank-pod mapping where the adj-bundle transfer is not needed. ① When aggregation begins, the NDP module of each rank reads the portion of the adjacency matrix from the DRAM cells in each rank and stores the adj-bundle in the adj-bundle buffer. ② Then, the NDP modules fetch the adj-bundles from the adj-bundle buffer and perform aggregation. Because the adj-bundle used by each NDP module is different, the time for aggregation per NDP module can be different from each other due to a load imbalance problem. When the aggregation process for a window finishes, adjacency-matrix read and aggregation operations for the next window are performed sequentially. Because *DRAM-path* and *Host-path* are separated, ③ the host reads the output feature vectors of the previous window from the output buffer, aggregates them, and ④ writes to the output buffers, simultaneously. ⑤ Finally, the NDP module writes the final output feature vectors of the previous window in the output buffer to the DRAM cells.

Execution flow with adj-bundle transfer: Figure 5(d) presents the execution flow of system-pod mapping, in which all ranks perform an aggregation phase with the same adj-bundle, so NDP modules must receive an adj-bundle from other ranks. ① First, identically to rank-pod mapping, NDP modules read the adjacency matrix from DRAM cells and store an adj-bundle in the adj-bundle buffer. ② The host reads the adj-bundle from the adj-bundle buffer of one rank and then sends it to the broadcasting buffers in all ranks sequentially. ③ The control unit fetches the adj-bundle from the broadcasting buffer, performs aggregation, and stores the output vectors in the output buffer. Concurrently, the host reads the adj-bundle of the next rank and sends it to all ranks sequentially. GraNDe stores this adj-bundle for the next rank in another broadcasting buffer (i.e., double-buffering). While the NDP module performs aggregation for the current window by repeating the above steps (①, ②, and ③), ④ the host reads the output vectors of the previous window from the output buffer, concatenates them, and writes to the output buffers. Then, ⑤ each NDP module writes the final output feature vectors of the previous window in the output buffer to the DRAM cells. For the DIMM- and channel-pod mappings, adj-bundle transfer is also necessary. The execution flow is equal to the explanation above, besides the adj-bundle is transferred only within the pod.

If the size of the adj-bundle exceeds the buffer size because target nodes with many edges exist in a window, the NDP module processes the loaded portion of the adj-bundle first. Then, the module loads another portion of the adj-bundle and processes it. As intermediate values are all stored in the output buffer, the module repeats the aforementioned steps until the entire window is aggregated.

5 Optimizing GraNDe

5.1 Adj-bundle Broadcasting

The execution flow of system-pod mapping requires adj-bundle transfer to all ranks, increasing the execution time of the aggregation phase. In conventional memory systems, multiple ranks equipped with one memory channel share the C/A and DQ path. The path between a memory channel and the ranks forms a multi-drop bus structure. When an MC transfers a signal (i.e., command/address or data) to a specific memory channel, the signal travels to all of the connected ranks. The rank that is not the target of the signal drops a transferred signal. Therefore, an MC should send the data to each rank sequentially, even if it is the same data. The effect of adj-bundle transfer on performance grows as the number of ranks increases; the time required to send adj-bundles to more ranks takes longer, while the amount of work for each NDP module is reduced.

We propose *adj-bundle broadcasting* to mitigate the effect of adj-bundle transfer on execution time. In system-pod mapping, every adj-bundle data transmitted to the broadcasting buffer of each NDP module is identical. If each rank does not drop this adj-bundle data intentionally, it can be written to all NDP modules in a channel at the same time. Therefore, we implement adj-bundle broadcasting in a straightforward manner as follows. If an NDP module receives a write command to the broadcasting buffer, it recognizes this command as broadcasting and writes the data to the buffer, regardless of the target address corresponding to its rank. Figure 5(e) shows how adj-bundle broadcasting reduces the execution time of the aggregation phase. The

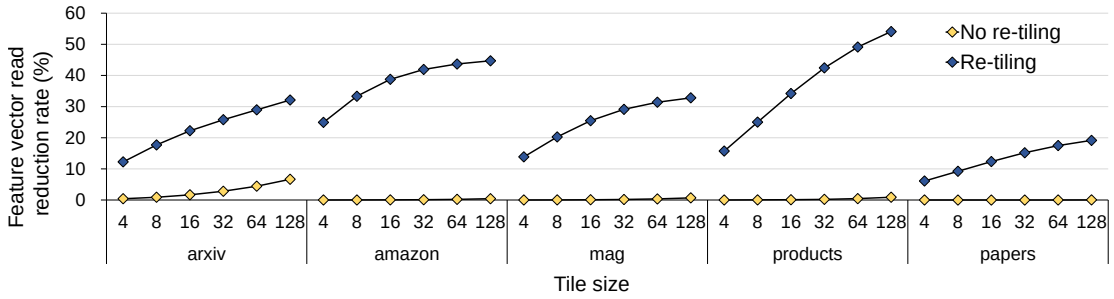

Fig. 7. The reduction in the feature vector read when varying the tile size from 4 to 128 gradually for *arxiv*, *amazon*, *mag*, *products*, and *papers* dataset. Re-tiling achieves a much higher reduction rate than No re-tiling (typical tiling).

adj-bundle broadcasting further reduces the execution time as more ranks are populated.

This adj-bundle broadcasting method can also be used in channel-pod mapping. In DIMM-pod mapping, NDP modules broadcast the adj-bundle without using the memory channel because the NDP modules in the same buffer device can transfer data to each other without communicating with the host processor.

5.2 Re-tiling Technique

GraNDe supports tiling for aggregation to reduce DRAM reads by increasing data reusability. It groups multiple target nodes into one tile and performs aggregation of the nodes in a tile together. Therefore, if different target nodes in a tile have the same neighbor node, the feature vector of that node can be reused (see Figure 6(a)).

In GraNDe, the typical tiling method that groups the nodes in the order of the index is ineffective. Node indices are determined regardless of adjacency between nodes, so the nodes of consecutive indices in the sparse adjacency matrix have few common neighbor nodes. This characteristic reduces the data reusability obtained by tiling, which is more severe in the larger graph with a high sparsity. Figure 7 shows the change in DRAM read reduction for feature vectors in the aggregation phase with typical tiling when varying the tiling size. Increasing the number of nodes belonging to one tile increases the number of feature vectors reused within a tile, so the number of DRAM reads further reduces up to 6.66% with 128-node tiling on the *arxiv*. However, in the larger graphs, such as *amazon*, *mag*, *products*, and *papers*, the probability that target nodes in one tile have the common adjacent node is decreased, exhibiting less than 1% of DRAM read reduction with tiling.

We propose re-tiling that groups the nodes considering the adjacency between nodes without the reconstruction process of the adjacency matrix in GraNDe. The re-tiling operation consists of 1) creating a node list in the order of nodes with adjacency and 2) creating a tile list based on the node lists of all ranks (see Figure 6(b)). First, re-tiling units in each NDP module read the adjacency matrix fraction in each rank sequentially and fill the node list. In this process, nodes that were already included are not added to the list again. Then, the host processor reads the node lists from each NDP module, aggregates them, and creates a tile list by cutting the aggregated node list into a tiling size unit. Each tile consists of nodes in a different order from the index order of the adjacency matrix (see Figure 6(c)). The host broadcasts the tile list to all ranks, and each NDP module reads the adjacency matrix in this tile order, creates an adj-bundle, and performs an aggregation operation. Because re-tiling constructs a tile with nodes adjacent to a particular node, the nodes in a tile have at least one common adjacent node. Also, it makes the nodes in a tile more likely to be adjacent than tiling in index order. GraNDe performs the re-tiling operation in a unit of the number of nodes processed in one window.

Re-tiling requires extra data structure in addition to node lists and tile lists. When creating a node list, the re-tiling unit must determine whether each node in the adjacency matrix is already in the node list. Therefore, GraNDe manages this information using a 1-bit flag per node. Because the adjacency matrix is vertically partitioned and stored in each rank, each re-tiling unit requires flag information for different nodes from each other, and the size of the flag structure for each re-tiling unit is (the number of nodes)/(the number of ranks)-bits. It is too large to be stored in an NDP module (e.g., 13.8 MB for the *papers* dataset), so GraNDe stores this flag information inside DRAM and accesses the required flags during the re-tiling process.

GraNDe performs re-tiling while aggregating the first layer, stores the generated tile list in DRAM, and reuses it in the other aggregation phases within one inference as the adjacency matrix does not change during the inference. GraNDe with re-tiling can effectively reduce the number of DRAM reads for the aggregation phase by reading once the feature vector data for the repeated node inside a tile and reusing the data. As shown in Figure 7, the re-tiling technique is effective even if the graph size increases. This effect increases as the tiling size increases, reducing the number of DRAM reads up to 54.1% in the *products* dataset with a 128-tiling size. There is much more data reuse than typical tiling, which shows a reduction rate lower than 1% on the same configuration.

6 GraNDe Control

The host and the NDP modules communicate through status register. When the host sets the start bit of the status register, the NDP modules begin aggregation. While checking the value in the status register, the host and the NDP modules performs pre-/post-processing and reduction (See Figure 8(a))

The host accesses the status register and buffers through memory-mapped I/O [17]. BIOS reserves the memory address space used for GraNDe in the course of booting. The

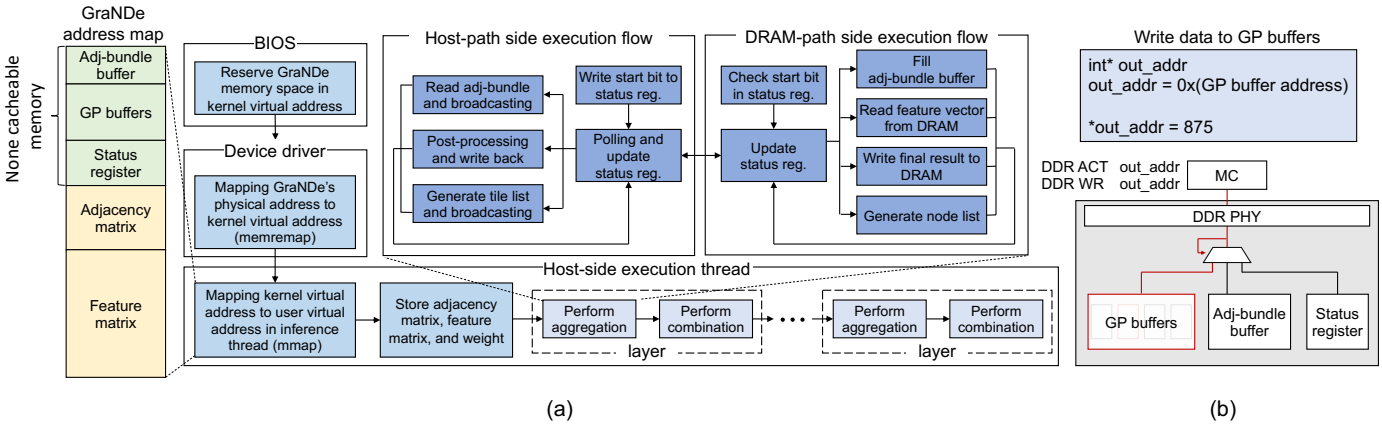

Fig. 8. Overview of the software stack for GraNDe. With the support of the BIOS and the device driver, the host can access the buffers in the NDP modules in the same way as normal memory access.

device driver maps GraNDe's physical address to the kernel virtual address, and the user thread maps the kernel virtual address to the user virtual address. With support of BIOS and a device driver, the physical memory is mapped with the virtual address in the user thread, and the buffers and the status register in the NDP module are accessed like ordinary memory access, as shown in the Figure 8(b). The NDP module checks the address of the DDR command and determines the buffers to access.

For adj-bundle broadcasting, the host reads an adj-bundle from the adj-bundle buffer of the source NDP module and then writes it to the broadcasting buffers of all NDP modules in a pod. During re-tiling, the host reads the node lists created by the re-tiling unit of each NDP module, makes a new tile list, and broadcasts it to all NDP modules. When the aggregation process is finished in the NDP module, the host reads the output buffer, performs post-processing, and writes the final output feature vectors to the output buffer of each NDP module. The NDP modules then fetch the final output feature vectors in the output buffer and write them into the DRAM cells. Because the *Host-path* is separated from the *DRAM-path*, adj-bundle broadcasting, making a new tile list, and post-processing by the host using the *Host-path* can be performed concurrently while each NDP module reads data from or writes data to DRAM cells with the *DRAM-path*.

During aggregation, the MC does not have direct access to DRAM and can only access the buffers in the NDP module. As shown in Figure 3, the MC's signals are multiplexed by the NDP mode signals. When operating in NDP mode, the signals (i.e., command, address, and data) from the MC cannot reach the DRAM. Therefore, the data in the DRAM is only modified by the NDP module during aggregation. Also, before starting aggregation, the host flushes feature matrix stored in the cache. It ensures that the matrix data is only stored in DRAM, maintaining data consistency between DRAM and cache.

7 Experimental Setup

Simulation framework: We modeled GraNDe with a trace-driven, cycle-level simulator by modifying Ramulator [18] to evaluate the performance of GraNDe compared to the baseline system and the state-of-the-art NDP architectures. The host system consists of 20 cores (32KB L1$, 256KB L2$), 32MB LLC, and four memory channels, each with two DDR5-4800 DIMMs and two ranks per DIMM. We used the timing parameters of DDR5 JEDEC specification [14]. We implemented GCN using the Intel MKL library [13] and extracted the memory trace of the aggregation phase using Intel Pin [28]. We put these traces into the modified Ramulator and evaluated the baseline system. We modeled GNNear by applying DIMM-pod mapping [45] and G-NMP [34] by adopting the best pod mapping. We used the tile size of 16 for GNNear, G-NMP, and GraNDe.

Benchmarks: We used Open Graph Benchmark (OGB [12]), which provides large-scale graph datasets, and amazon0505 [23] for evaluation. We used *arxiv*, *amazon*, *mag*, *products*, and *papers* as datasets. Concerning *papers*, the entire simulation could not be performed due to its huge size. Accordingly, aggregation operation of only a tenth of the total node was measured. We also excluded re-tiling evaluation on *papers* because re-tiling can be processed only for entire graph nodes. We used a GCN model with three layers [12] for evaluating the end-to-end aggregation phase speedup. The feature vector dimension of the first layer in the aggregation is the same as the feature dimension of each dataset in Table 1. The feature vector dimensions of the second and the third layer are set to 128 and 256, respectively.

8 Evaluation

8.1 GraNDe Performance for GCN Aggregation

First, we evaluated the GCN layer-by-layer speedups of GraNDe adopting rank-, DIMM-, channel-, two-channel-, and system-pod mapping compared to the baseline system (**Base**) (see Figure 9(a)). We measured the aggregation time of the entire GCN, varying the feature vector dimension of each layer from 16 to 256 to identify the performance improvement depending on the layer size. Also, to verify the effect of each mapping on performance, we modeled GraNDe without two optimizations (**GraNDe**), adj-bundle broadcasting and re-tiling, in this experiment.

The preferred mapping that performs best differs according to the feature vector dimension. When the dimension is small, the pod of a smaller size performs better (i.e., rank-pod with dimension 16). In contrast, the pod of a larger size

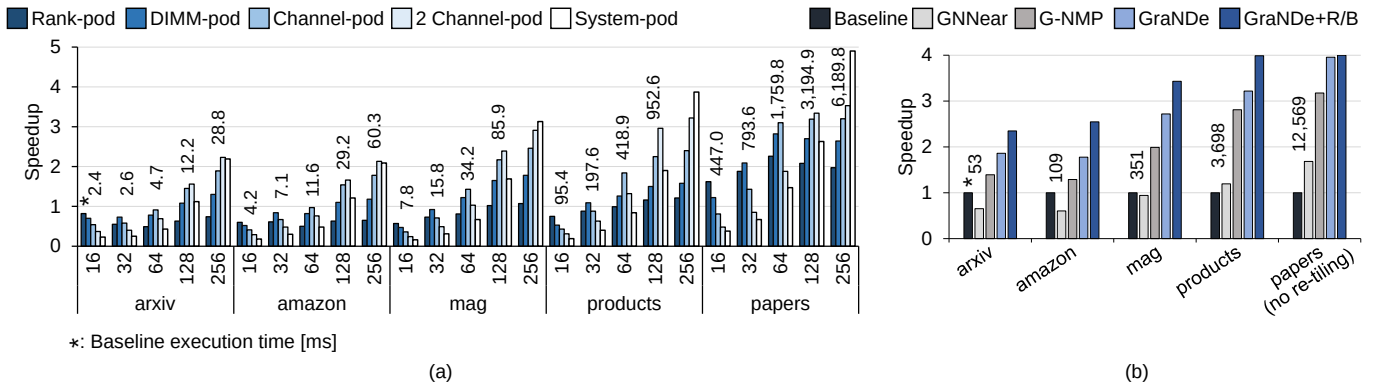

Fig. 9. (a) Aggregation speedup when using rank-, DIMM-, channel-, two-channel-, and system-pod mapping of **GraNDe** compared to **Base** for one layer with feature vector dimensions of 16, 32, 64, 128, and 256 for each dataset. (b) The performance improvements of **GNNear**, **G-NMP**, **GraNDe**, and **GraNDe+R/B** compared to the **Base** in aggregation time in the entire GCN. **GraNDe+R/B** of *papers* is the performance with only adj-bundle broadcasting (**GraNDe+B**) optimization.

achieves better performance with a large dimension (i.e., two-channel- or system-pod with dimension 256). Using the largest pod, which does not occur the DRAM bandwidth waste problem described in Section 4.2, is most advantageous for performance. For example, the channel-pod is a preferred mapping for a feature vector dimension of 64, where each feature vector consists of 64 FP32 elements. Because there are four ranks in a channel, each rank stores 16 FP32 elements (64 bytes) of each feature vector allocated to the channel. This size equals the DRAM access granularity, so it is the minimum size to access data without DRAM bandwidth waste. Mapping a larger portion of a feature vector to a pod also does not incur a bandwidth waste problem, but performance degrades due to the load imbalance problem. Other dimensions also show the best performance when each rank stores 16 elements for each feature vector.

Figure 9(b) shows the speedup of **GNNear**, **G-NMP**, **GraNDe**, and GraNDe with re-tiling and adj-bundle broadcasting (**GraNDe+R/B**) compared to the **Base** for the entire GCN aggregation. We set **GraNDe** to use the two-channel-pod mapping for the first and second layers and the system-pod mapping for the last layer. We used 16 as the tile size, considering output buffer size and computing power. **GraNDe** achieves a speedup of 2.58× on average over the **Base** and up to 3.96× when using *papers*. **GraNDe+R/B** further accelerates GCN aggregation, improving the performance by 3.19× on average across datasets and a maximum of 3.99× using *products*. As we measured the operation of only a tenth of the total nodes in *papers* due to its huge size, we could not apply re-tiling to *papers*, and **GraNDe** with adj-bundle broadcasting shows a speedup of 4.00× over the **Base** using *papers*.

GraNDe outperforms the state-of-the-art NDP architectures, **GNNear** and **G-NMP**. **GraNDe** achieves up to 2.92× and 1.37× higher performance of GCN aggregation over **GNNear** and **G-NMP**, respectively, thanks to the amplified memory bandwidth by separating the memory channel path. With re-tiling and adj-bundle broadcasting optimizations, **GraNDe+R/B** improves performance up to 1.98× than **G-NMP**. **GNNear** shows inferior performance due to fixed DIMM-pod mapping for all layers. **G-NMP** applies adaptive mapping for each layer, similar to **GraNDe**, so it achieves higher speedup than **GNNear**.

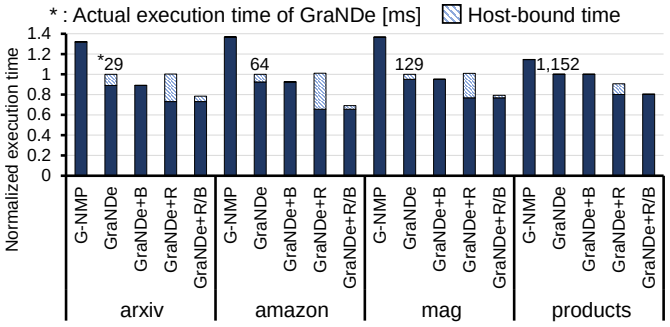

Fig. 10. Normalized execution time of aggregation in **G-NMP**, **GraNDe+B**, **GraNDe+R** and **GraNDe+R/B** based on **GraNDe** for each dataset. *Host-bound time*, the extra time bound to the *Host-path*, is marked with slashes.

8.2 Effect of Adj-bundle Broadcasting and Re-tiling

Figure 10 shows the normalized aggregation execution time in the end-to-end GCN of **GraNDe**, GraNDe with re-tiling (**GraNDe+R**), and GraNDe with both re-tiling and adj-bundle broadcasting (**GraNDe+R/B**). GraNDe can independently utilize the path between the host and NDP module (*Host-path*) and the path between NDP modules and DRAM devices (*DRAM-path*). In the *Host-path*, adj-bundle transfer and post-processing are performed, and feature computation is taking place in the *DRAM-path*. The time of adj-bundle transfer and post-processing in *Host-path* can take longer than the feature computation in *DRAM-path*. Hereafter, the extra time bound to the *Host-path* in total execution time is defined as *Host-bound time*.

In *arxiv*, *amazon*, and *mag*, it takes longer for adj-bundle transfer and post-processing than the time for feature computation, resulting in *Host-bound time*. Using adj-bundle broadcasting reduces adj-bundle transfer time (also *Host-bound time*), so the performance of **GraNDe+B** is better than **GraNDe**. As for **GraNDe+R**, the feature computation time is reduced, but the performance does not increase due to *Host-bound time*. When we apply adj-bundle broadcasting, the time to transfer adj-bundle is reduced, so *Host-bound time* decreases.

In the case of *products*, **GraNDe** is not bounded to *Host-path*. The difference from the other datasets is that *products* has a high average degree of a graph, which is the average

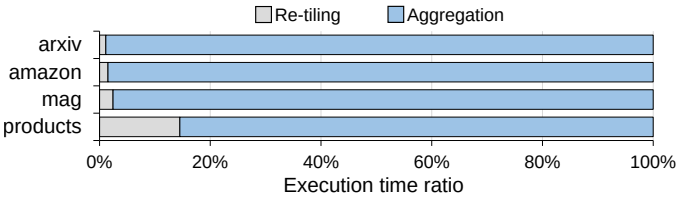

Fig. 11. The ratio of time spent on re-tiling over the aggregation execution time of **GraNDe+R/B**.

number of adjacent nodes per node. With a high degree of a graph, more feature vectors need to be read when performing reduction. Therefore, the feature computation takes longer, and *Host-bound time* does not occur in **GraNDe**. In **GraNDe+R**, re-tiling accelerates the feature computation, reducing the reduction time in NDP modules. However, the pre-/post-processing time in the host processor does not shrink, so the degree of host bound is increased compared to **GraNDe**, making no significant difference in total execution time except for *products*. Because re-tiling reduces the reduction time using *DRAM-path*, and adj-bundle broadcasting reduces the transfer time using *Host-path*, **GraNDe+R/B** exhibits meaningful performance improvement.

Figure 11 shows the percentage of time spent for re-tiling among **GraNDe+R/B**. In *arxiv*, *amazon*, *mag*, and *products*, the re-tiling time is 1.1%, 1.5%, 2.4%, and 14.5%, respectively. As described in Section 5.2, re-tiling is performed only in the first layer, and the generated re-tiling information is reused in the later layers. Therefore, the performance gain from re-tiling is greater than its performance overhead, effectively reducing the aggregation execution time in the NDP modules, as shown in Figure 10.

8.3 Design Overhead

GraNDe's NDP module, which is in charge of one rank, consists of 32 single-precision floating-point multiply-accumulators, a 16KB adj-bundle buffer, four 16KB GP buffers, and the control logic. We designed arithmetic units using Verilog and synthesized them using the Synopsys Design Compiler with 40nm CMOS technology at a frequency of 300MHz. We used CACTI [25] to model SRAM-based buffers. In an NDP module, the area of the arithmetic unit is 0.22mm^2, and that of the buffers is 0.28mm^2. These values correspond to 1.02% of the buffer chip considering two ranks per DIMM [1]. Also, the power of the arithmetic unit in the NDP module is 64mW, and that of the buffer is 47.8mW, which is negligible.

GraNDe places NDP modules in buffer chips without modifying DRAM, not affecting the peak power of DRAM. Instead, these NDP modules and the simultaneous accesses to multiple ranks increase the power consumption at the DIMM level. However, GraNDe dissipates less DRAM static energy as the execution time decreases. Also, when GraNDe does not undertake aggregation, the signal bypasses the NDP modules so that the buffer chip operates as a conventional buffer chip, with no effect on the ordinary DRAM access latency.

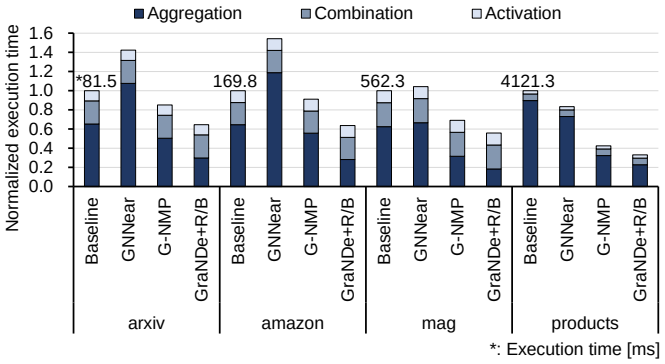

Fig. 12. Normalized GCN end-to-end execution time of **GraNDe+R/B** compared with **Base** for each dataset.

9 End-to-end GCN Operation with GraNDe

GraNDe is for accelerating an aggregation phase, so the combination and activation operations of GCN must be performed by host processors such as CPU and GPU or existing accelerators (e.g., the ones implementing systolic arrays) for GEMM operation. When GraNDe does not perform the aggregation phase, it acts like a conventional DIMM, so the host processor can read the data stored in DRAM using a typical memory access protocol. Therefore, GraNDe stores the output feature matrix of the aggregation phase in DRAM, and the host performs the combination phase by reading it and the weight matrix. If the size of the weight matrix is small enough to fit in the on-chip memory, the process becomes simpler. When the host reads the output feature vector of aggregation for the post-processing (i.e., concatenation or reduction), the result of post-processing can be multiplied directly with the weight matrix located on-chip. Therefore, while the host performs the combination operation immediately followed by post-processing, GraNDe can perform the aggregation operation of the next window, making the aggregation and combination phase be performed simultaneously.

The result of the combination and activation process is used as an input feature matrix for aggregation of the following layer. The dimension of this feature matrix can differ from that of the previous layer, so the host writes it to the memory using the best mapping for the new feature vector dimension. Then, GraNDe performs aggregation of the next layer by reading the feature matrix from DRAM. The entire GCN is performed by repeating the above process.

We evaluate the impact of aggregation acceleration on the end-to-end GCN inference time, including the combination and activation processes. Based on the execution time breakdown measured in the hardware experiment environment (Intel Xeon Gold 6138 with four memory channels, two DDR4-2400 DIMM with two ranks per channel) used for Figure 10, we calculated reduction in execution time using the aggregation simulation result. The simulation was conducted on the same configuration as the hardware experiment. Figure 12 shows that GraNDe improves the performance of the end-to-end GCN inference by 1.55×, 1.57×, 1.79×, and 3.02× with *arxiv*, *amazon*, *mag*, and *products* compared to the baseline system without GraNDe, respectively. For small datasets of 1 GB or less (*arxiv*, *amazon*, and

products), the performance improvement is relatively small, due to the increased data reuse by the cache. However, for large graph datasts such as *products*, which are GraNDe's primary target, the performance improvement is significant, more than 3×.

We also compared the performance of GraNDe with a GPU. We used a V100 (HBM2, 900GB/s, 16GB) and measured execution times using a deep graph library [37]. GPU shows 3.90×, 4.28×, and 4.01× speedups on *arxiv*, *amazon*, and *mag* than GraNDe because they have higher memory bandwidth than amplified bandwidth of GraNDe (307.2GB/s). However, as the size of the dataset grows (i.e., products and papers), the GPU cannot perform the inference due to memory size limitations, while GraNDe shows significant performance improvement by utilizing amplified bandwidth for these large graph datasets. Moreover, GraNDe can increase amplified bandwidth at a lower cost than GPU by equipping more DIMMs or implementing more ranks per DIMM.

10 Related work

Graph-aware methods: Previous studies reduced the execution time of aggregation by reconstructing the graph considering the adjacency between graph nodes to increase data reusability. GNNAdvisor [38] used node renumbering to accelerate the aggregation phase of GNN in a single GPU by improving data reuse. BNS-GCN [36] used graph partitioning to reduce traffic between GPUs in training. Graph partitioning is efficient if the continuous input graph structures are identical because the reconstructed graph can be reused. However, real-world graphs are frequently updated and dynamically generated [8]. As a graph becomes larger, the high restructuring overhead of graph partitioning has a more significant impact on the interference time of GNN. GraNDe differs from these studies in that it extracts adjacency information online during the tiling operation without graph reconstruction.

Broadcasting interface in NDP architectures: Previous studies have attempted to use broadcasting in the NDP structure. TensorDIMM [22], an NDP architecture that accelerates recommendation systems, sends its NDP instruction in a broadcast manner as the NDP cores in all DIMMs use the operands in the same memory address space in each rank. However, TensorDIMM does not target graph applications, so it does not consider broadcasting the data being written (e.g., post-processed data in the graph). ABC-DIMM [33] proposed an inter-DIMM broadcast to alleviate the bottleneck of communication between DIMMs in an NDP architecture. It writes broadcasted data to the DRAM cells, so its NDP modules cannot read data from the DRAM cells while performing the broadcasting unlike GraNDe, which stores the broadcasted data (e.g., adj-bundle) to the specific buffer in the NDP modules. Graphp [44] is another NDP architecture exploiting broadcasting for graph application, but it targets Hybrid Memory Cube (HMC), and its broadcasting scheme targets network-on-chip.

11 Conclusion

We have proposed GraNDe, a DIMM-based near-data-processing architecture that accelerates the memory-intensive aggregation phase of Graph Neural Networks. It effectively exploits the amplified DRAM bandwidth offered by concurrently accessing the ranks in a memory channel and separating the memory channel path to both sides with the buffer chip in the middle. GraNDe supports an adaptive operand matrix mapping that distributes the adjacency and feature matrices across all ranks using a preferred mapping layer-by-layer according to the feature vector dimension of each layer. We further improved GraNDe by adopting the broadcasting mechanism for transferring adjacency matrix data to all ranks simultaneously and the re-tiling technique to increase the data reusability of aggregation operation through tiling. GraNDe improves GCN aggregation performance up to 4.00× and 1.98× compared to the baseline system and the state-of-the-art NDP architecture, respectively.

References

[1] M. Alian, S. W. Min, H. Asgharimoghaddam, A. Dhar, D. K. Wang, T. Roewer, A. McPadden, O. O'Halloran, D. Chen, J. Xiong, D. Kim, W.-m. Hwu, and N. S. Kim, "Application-Transparent Near-Memory Processing Architecture with Memory Channel Network," in *MICRO*, 2018, pp. 802–814.

[2] M. Andersch, G. Palmer, R. Krashinsky, N. Stam, V. Mehta, G. Brito, and S. Ramaswamy, "NVIDIA Hopper Architecture In-Depth," 2022. [Online]. Available: https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/

[3] M. Arafa, B. Fahim, S. Kottapalli, A. Kumar, L. P. Looi, S. Mandava, A. Rudoff, I. M. Steiner, B. Valentine, G. Vedaraman, and S. Vora, "Cascade Lake: Next Generation Intel Xeon Scalable Processor," *IEEE Micro*, vol. 39, no. 2, pp. 29–36, 2019.

[4] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language Models Are Few-Shot Learners," in *Proceedings of the 34th International Conference on Neural Information Processing Systems*, 2020.

[5] N. Challapalle, S. Rampalli, L. Song, N. Chandramoorthy, K. Swaminathan, J. Sampson, Y. Chen, and V. Narayanan, "GaaS-X: Graph Analytics Accelerator Supporting Sparse Data Representation using Crossbar Architectures," in *ISCA*, 2020, pp. 433–445.

[6] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," *arXiv:1810.04805*, 2018.

[7] T. Geng, A. Li, R. Shi, C. Wu, T. Wang, Y. Li, P. Haghi, A. Tumeo, S. Che, S. Reinhardt, and M. C. Herbordt, "AWB-GCN: A Graph Convolutional Network Accelerator with Runtime Workload Rebalancing," in *MICRO*, 2020, pp. 922–936.

[8] T. Geng, C. Wu, Y. Zhang, C. Tan, C. Xie, H. You, M. Herbordt, Y. Lin, and A. Li, "I-GCN: A Graph Convolutional Network Accelerator with Runtime Locality Enhancement through Islandization," in *MICRO*, 2021, pp. 1051–1063.

[9] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30. Curran Associates, Inc., 2017. [Online]. Available: https://proceedings.neurips.cc/paper/2017/file/5dd9db5e033da9c6fb5ba83c7a7ebea9-Paper.pdf

[10] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[11] R. Hojabr, A. Sedaghati, A. Sharifian, A. Khonsari, and A. Shriraman, "SPAGHETTI: Streaming Accelerators for Highly Sparse GEMM on FPGAs," in *HPCA*, 2021, pp. 84–96.

[12] 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," *arXiv:2005.00687*, 2020.

[13] Intel, "Intel(R) oneAPI Math Kernel Library," 2021. [Online]. Available: https://github.com/oneapi-src/oneMKL

[14] JEDEC, "DDR5 SDRAM Standard," 2020.

[15] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-Datacenter Performance Analysis of a Tensor Processing Unit," in *ISCA*, 2017, pp. 1–12.

[16] L. Ke, U. Gupta, B. Y. Cho, D. Brooks, V. Chandra, U. Diril, A. Firoozshahian, K. Hazelwood, B. Jia, H.-H. S. Lee, M. Li, B. Maher, D. Mudigere, M. Naumov, M. Schatz, M. Smelyanskiy, X. Wang, B. Reagen, C.-J. Wu, M. Hempstead, and X. Zhang, "RecNMP: Accelerating Personalized Recommendation with Near-Memory Processing," in *ISCA*, 2020, p. 790–803.

[17] L. Ke, X. Zhang, J. So, J.-G. Lee, S.-H. Kang, S. Lee, S. Han, Y. Cho, J. H. Kim, Y. Kwon, K. Kim, J. Jung, I. Yun, S. J. Park, H. Park, J. Song, J. Cho, K. Sohn, N. S. Kim, and H.-H. S. Lee, "Near-Memory Processing in Action: Accelerating Personalized Recommendation with AxDIMM," *IEEE Micro*, no. 01, pp. 116–127, 2021.

[18] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A Fast and Extensible DRAM Simulator," *IEEE Computer Architecture Letters*, vol. 15, no. 1, pp. 45–49, 2016.

[19] T. N. Kipf and M. Welling, "Semi-supervised Classification with Graph Convolutional Networks," *arXiv:1609.02907*, 2016.

[20] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Advances in Neural Information Processing Systems*, F. Pereira, C. Burges, L. Bottou, and K. Weinberger, Eds., vol. 25, 2012.

[21] H. Kung and C. Leiserson, "Algorithms for VLSI processor arrays," *Introduction to VLSI systems*, pp. 271–292, 1980.

[22] Y. Kwon, Y. Lee, and M. Rhu, "TensorDIMM: A Practical Near-Memory Processing Architecture for Embeddings and Tensor Operations in Deep Learning," in *MICRO*, 2019, p. 740–753.

[23] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford Large Network Dataset Collection," http://snap.stanford.edu/data, 2014.

[24] J. Li, A. Louri, A. Karanth, and R. Bunescu, "GCNAX: A Flexible and Energy-efficient Accelerator for Graph Convolutional Neural Networks," in *HPCA*, 2021, pp. 775–788.

[25] S. Li, J. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures," in *MICRO*, 2009, p. 469–480.

[26] S. Li, E. Hanson, X. Qian, H. H. Li, and Y. Chen, "ESCALATE: Boosting the Efficiency of Sparse CNN Accelerator with Kernel Decomposition," in *MICRO*, 2021, p. 992–1004.

[27] S. Liang, Y. Wang, C. Liu, L. He, H. LI, D. Xu, and X. Li, "EnGN: A High-Throughput and Energy-Efficient Accelerator for Large Graph Neural Networks," *IEEE TC*, vol. 70, no. 9, pp. 1511–1525, 2020.

[28] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *PLDI*, vol. 40, no. 6, 2005.

[29] J. Park, B. Kim, S. Yun, E. Lee, M. Rhu, and J. Ahn, "TRiM: Enhancing Processor-Memory Interfaces with Scalable Tensor Reduction in Memory," in *MICRO*, 2021, p. 268–281.

[30] J. Pavon, I. V. Valdivieso, A. Barredo, J. Marimon, M. Moreto, F. Moll, O. Unsal, M. Valero, and A. Cristal, "VIA: A Smart Scratchpad for Vector Units with Application to Sparse Matrix Computations," in *HPCA*, 2021, pp. 921–934.

[31] G. Rao, J. Chen, J. Yik, and X. Qian, "SparseCore: Stream ISA and Processor Specialization for Sparse Computation," in *ASPLOS*, 2022, p. 186–199.

[32] A. Rucker, M. Vilim, T. Zhao, Y. Zhang, R. Prabhakar, and K. Olukotun, "Capstan: A Vector RDA for Sparsity," in *MICRO*, 2021, p. 1022–1035.

[33] W. Sun, Z. Li, S. Yin, S. Wei, and L. Liu, "ABC-DIMM: Alleviating the Bottleneck of Communication in DIMM-based Near-Memory Processing with Inter-DIMM Broadcast," in *ISCA*. IEEE, 2021, pp. 237–250.

[34] T. Tian, X. Wang, L. Zhao, W. Wu, X. Zhang, F. Lu, T. Wang, and X. Jin, "G-NMP: Accelerating Graph Neural Networks with DIMM-based Near-Memory Processing," *Journal of Systems Architecture*, vol. 129, p. 102602, 2022.

[35] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is All you Need," in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30, 2017.

[36] C. Wan, Y. Li, A. Li, N. S. Kim, and Y. Lin, "BNS-GCN: Efficient Full-Graph Training of Graph Convolutional Networks with Partition-Parallelism and Random Boundary Node Sampling," *Proceedings of Machine Learning and Systems*, vol. 4, pp. 673–693, 2022.

[37] M. Wang, D. Zheng, Z. Ye, Q. Gan, M. Li, X. Song, J. Zhou, C. Ma, L. Yu, Y. Gai, T. Xiao, T. He, G. Karypis, J. Li, and Z. Zhang, "Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks," *arXiv:1909.01315*, 2019.

[38] Y. Wang, B. Feng, G. Li, S. Li, L. Deng, Y. Xie, and Y. Ding, "GNNAdvisor: An Adaptive and Efficient Runtime System for GNN Acceleration on GPUs," in *15th USENIX Symposium on Operating Systems Design and Implementation*, 2021, pp. 515–531.

[39] X. Xie, Z. Liang, P. Gu, A. Basak, L. Deng, L. Liang, X. Hu, and Y. Xie, "SpaceA: Sparse Matrix Vector Multiplication on Processing-in-Memory Accelerator," in *HPCA*, 2021, pp. 570–583.

[40] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How Powerful are Graph Neural Networks?" in *International Conference on Learning Representations*, 2019. [Online]. Available: https://openreview.net/forum?id=ryGs6iA5Km

[41] M. Yan, L. Deng, X. Hu, L. Liang, Y. Feng, X. Ye, Z. Zhang, D. Fan, and Y. Xie, "HyGCN: A GCN Accelerator with Hybrid Architecture," in *HPCA*, 2020, pp. 15–29.

[42] S. Yun, B. Kim, J. Park, H. Nam, J. H. Ahn, and E. Lee, "GraNDe: Near-Data Processing Architecture With Adaptive Matrix Mapping for Graph Convolutional Networks," *IEEE Computer Architecture Letters*, vol. 21, no. 2, pp. 45–48, 2022.

[43] G. Zhang, N. Attaluri, J. S. Emer, and D. Sanchez, "Gamma: Leveraging Gustavson's Algorithm to Accelerate Sparse Matrix Multiplication," in *ASPLOS*, 2021, p. 687–701.

[44] M. Zhang, Y. Zhuo, C. Wang, M. Gao, Y. Wu, K. Chen, C. Kozyrakis, and X. Qian, "GraphP: Reducing Communication for PIM-based Graph Processing with Efficient Data Partition," in *HPCA*, 2018, pp. 544–557.

[45] Z. Zhou, C. Li, X. Wei, X. Wang, and G. Sun, "GNNear: Accelerating Full-Batch Training of Graph Neural Networks with Near-Memory Processing," in *PACT*, 2022.

Sungmin Yun received the BS degree with Department of Integrated Information Technology from Yonsei University. He is currently working toward the PhD degree in Interdisciplinary Program in Artificial Intelligence from Seoul National University. His research interests include memory system, near-data processing architecture, processing in memory, and AI accelerator.

Hwayong Nam received the BS degree in electrical and electronics engineering from Chung-Ang University. He is currently working toward the PhD degree with the Department of Intelligence and Information, Seoul National University. His research interests include memory system, near-data processing, and memory reliability.

Jaehyun Park received the BS degree in electrical and computer engineering from Seoul National University. He is currently working toward the PhD degree with the Department of Intelligence and Information, Seoul National University. His research interests include memory system, in-memory accelerator architecture and logic design.

Byeongho Kim received the BS degree in electrical and computer engineering and the PhD degree in intelligent convergence systems from Seoul National University. He is currently a staff engineer in Samsung Electronics DRAM design team, where he is working on designing memory systems for processing-in-memory (PIM) architecture. His research is focused on improving performance of memory system in computer systems.

Jung Ho Ahn (Senior Member, IEEE) received the BS degree in electrical engineering from Seoul National University and the MS and PhD degrees in electrical engineering from Stanford University, Stanford, CA, USA. He is currently a Professor with the Graduate School of Convergence Science and Technology, Seoul National University, Seoul, South Korea, where he leads the Scalable Computer Architecture Laboratory. He is interested in bridging the gap between the performance demand of emerging applications and the performance potential of modern and future massively parallel systems.

Eojin Lee received the BS degree in electrical and computer engineering and the PhD degree in intelligent convergence systems from Seoul National University. He is currently an assistant professor with the Department of Computer Engineering at Inha University, South Korea. His research interests include architecting better memory systems and designing computer architectures for accelerating emerging applications.