PyGim: An Efficient Graph Neural Network Library for Real Processing-In-Memory Architectures

CHRISTINA GIANNOULA, University of Toronto, Canada, ETH Zürich, Switzerland, Vector Institute, Canada, and CentML, Canada
PEIMING YANG, University of Toronto, Canada
IVAN FERNANDEZ, Barcelona Supercomputing Center, Spain, Universitat Politècnica de Catalunya, Spain, and ETH Zürich, Switzerland
JIACHENG YANG, University of Toronto, Canada and Vector Institute, Canada
SANKEERTH DURVASULA, University of Toronto, Canada and Vector Institute, Canada
YU XIN LI, University of Toronto, Canada
MOHAMMAD SADROSADATI, ETH Zürich, Switzerland
JUAN GOMEZ LUNA, NVIDIA, Switzerland
ONUR MUTLU, ETH Zürich, Switzerland
GENNADY PEKHIMENKO, University of Toronto, Canada, Vector Institute, Canada, and CentML, Canada

Graph Neural Networks (GNNs) are emerging models to analyze graph-structure data. The GNN execution involves both compute-intensive and memory-intensive kernels. The memory-intensive kernels dominate execution time, because they are significantly bottlenecked by data movement between memory and processors. Processing-In-Memory (PIM) systems can alleviate this data movement bottleneck by placing simple processors near or inside memory arrays. To this end, we investigate the potential of PIM systems to alleviate the data movement bottleneck in GNNs, and introduce PyGim, an efficient and easy-to-use GNN library for real PIM systems. We propose intelligent parallelization techniques for memory-intensive kernels of GNNs tailored for real PIM systems, and develop an easy-to-use Python API for them. PyGim employs a cooperative GNN execution, in which the compute- and memory-intensive kernels are executed in processor-centric and memory-centric computing systems, respectively, to fully exploit the hardware capabilities. PyGim integrates a lightweight tuner that configures the parallelization strategy of the memory-intensive kernel of GNNs to provide high system performance, while also enabling high programming ease. We extensively evaluate PyGim on a real-world PIM system that has 16 PIM DIMMs with 1992 PIM cores connected to a Host CPU. In GNN inference, we demonstrate that it outperforms prior state-of-the-art PIM works by on average 4.38× (up to 7.20×), and the state-of-the-art PyTorch implementation running on Host (on Intel Xeon CPU) by on average 3.04× (up to 3.44×). PyGim improves energy efficiency by 2.86× (up to 3.68×) and 1.55× (up to 1.75×) over prior PIM and PyTorch Host schemes, respectively. In memory-intensive kernel of GNNs, PyGim provides 11.6× higher resource utilization in PIM system than that of PyTorch library (optimized CUDA implementation) in GPU systems. Our work provides useful recommendations for software, system and hardware designers. PyGim is publicly and freely available at https://github.com/CMU-SAFARI/PyGim to facilitate the widespread use of PIM systems in GNNs.

Key Words: machine learning, graph neural networks, sparse matrix-matrix multiplication, library, multicore, processing-in-memory, near-data processing, memory systems, data movement bottleneck, DRAM, benchmarking, real-system characterization, workload characterization

1 Introduction

Graph Neural Networks (GNNs) [49, 67, 135, 147] have emerged as state-of-the-art Machine Learning (ML) models to depict dependent relations in graph-structure data, providing high accuracy in vertex classification and link (edge) prediction tasks [93, 115, 126, 140]. Thus, they have been adopted to many real-world applications, including point-cloud analysis [110], recommendation systems [134], social network analysis [29], and drug discovery [121]. GNNs comprise a few layers,

and each layer consists of two steps: *aggregation* and *combination*. The former aggregates the input feature vectors of the neighboring vertices for each vertex in the graph via a permutation-invariant operator (e.g., average). The latter processes the aggregated vectors of all vertices through a small neural network (e.g., a multilayer perceptron [9]) to produce the output feature vectors, which will be fed as input feature vectors to the next layer.

The key operators of combination are dense matrix matrix multiplications (GEMMs), while aggregation degenerates to a Sparse Matrix Matrix Multiplication (SpMM) kernel, processing the graph data that is represented as a sparse matrix [43, 141, 148]. In §2.1, we profile the GNN execution in a high-end GPU system, and find that aggregation dominates execution time and exhibits high memory intensity, while combination is compute-intensive. The compute-intensive combination fits to be executed in processor-centric systems (CPUs or GPUs). However, aggregation is significantly bottlenecked by data movement between memory and processors in such systems, since SpMM is typically memory-bandwidth-bound in CPUs and GPUs [43, 45, 141] (See also §2.1).

A promising way to alleviate the data movement cost is Processing-In-Memory (PIM) [2, 4, 12, 14, 18, 19, 23, 25–27, 30, 31, 34, 37, 39–42, 47, 48, 53, 57, 60, 64, 70, 72, 75, 84, 86, 89, 95–98, 112, 116, 145, 149] computing paradigm. PIM enables computation to be performed close to the application data by equipping memory chips with processing capabilities (in-memory processors). To provide significantly higher memory bandwidth for the in-memory processors than standard DRAM modules, manufacturers have commercialized *near-bank* PIM designs [25]. Near-bank PIM memory modules tightly couple a PIM core with one (or a few) DRAM bank, exploiting bank-level parallelism to expose the high aggregated on-chip memory bandwidth of standard DRAM to processors. A real PIM system supports multiple near-bank PIM memory modules, which are connected to a CPU or GPU, henceforth referred to as *Host*. The UPMEM PIM architecture [25] is the first PIM system to become commercially available. HBM-PIM [74] and AiM [75] are near-bank PIM systems that have been prototyped and evaluated in real systems.

A few works [128, 141, 148] propose hybrid Host-PIM accelerators for GNNs. However, none of them considers real-world PIM systems. These works design new microarchitectures for *near-rank* PIM systems, i.e., accelerator cores are placed at each rank of memory modules. Near-rank PIM designs have not been commercialized yet, and are not always able to provide significantly higher memory bandwidth for processors than standard DRAM [4, 74]. In the software level, these works have simple *fixed* parallelization strategies for GNN aggregation in PIM cores, which would cause out-of-memory errors for medium-/large-size graphs (See §2.3) or achieve very low performance in *real* near-bank PIM systems, as we show in Figs. 13 and 14 of §4. Moreover, these works use software emulators for their evaluations (not a real PIM system), and do not describe the engineering efforts needed to deploy GNNs in their accelerators.

Our **goal** in this work is to efficiently map GNNs on near-bank PIM systems and quantify the potential of *real* PIM architectures in GNN executions. Efficiently executing GNNs in real PIM systems encounters three key challenges. 1) GNN execution has repeated compute-intensive (combination) and memory-intensive (aggregation) kernels. On the one hand, executing both types of kernels in PIM cores would incur high performance overheads in combination, since PIM cores are low-area and low-power cores with relatively low computation capabilities [4, 47, 74]. On the other hand, executing combination on Host cores and aggregation on PIM cores, respectively, necessitates minimizing the overheads of passing the output result of one kernel as input to the next kernel. 2) Real-world graphs exhibit diverse characteristics, e.g., the average, min or max vertex neighboring degrees vary across different graphs. Therefore, as discussed in prior works [39, 62, 77, 122, 138], the execution behavior of sparse kernels, such as the SpMV/SpMM, depends on the particular characteristics of the input given, and there is no typically one-size-fits-all parallelization solution that performs best across various inputs [39]. 3) Programming a real near-bank PIM system for a

high-level application is a hard task [14, 55, 61], since software stacks for PIM systems are still in an early stage. Thus, ML programmers may need to distribute data of GNNs across thousands of DRAM banks in a fine-grained and careful way, have expertise of the PIM hardware [14, 61] and/or program the PIM cores using low-level APIs [14, 47].

To address the aforementioned challenges, we design PyGim [44][1], a high-level ML library to efficiently execute GNNs in real PIM systems. PyGim provides high system performance in *real* Host-PIM executions of GNNs, and bridges the gap between ML engineers, who prefer high-level programming interfaces (e.g., Python), and real PIM systems, that typically provide complex and low-level APIs and may need deep knowledge of PIM hardware.

PyGim co-designs a **Cooperative Acceleration (CoA)** model with a novel **Parallelism Fusion (PaF)** method. CoA runs heterogeneous kernels to the best-fit underlying hardware: the processor-centric Host (CPU/GPU) system executes the compute-intensive GNN combination, and the memory-centric PIM system executes the memory-intensive aggregation. PaF serves a dual purpose: it (i) strives a balance between computation and data transfer costs in GNN aggregation executed in PIM cores, minimizing the overheads of passing the output result of combination as input to aggregation, and vice versa, and (ii) provides various parallelization techniques to cover many real-world graphs with diverse characteristics. Specifically, in GNN aggregation, we enable three parallelism levels on the hardware PIM side and, at each level, we provide different parallelization techniques on the software side. 1) We group the available PIM cores of the system in clusters, and design *edge-* and *feature-level* parallelism *across PIM clusters*. 2) We enable *vertex-* or *edge-level* parallelism across cores *within PIM cluster*. 3) We employ either *vertex-* or *edge-level* parallelism across threads *within a PIM core*. The technique of the first parallelism level reduces data transfer overheads to/from PIM memory modules, thus reducing costs when passing the output of one GNN kernel as input to the next one. The techniques of the second and third parallelism levels enable load balancing schemes that provide high compute balance across low-power PIM cores and across threads of a PIM core. PaF enables various GNN aggregation configurations and load balancing strategies, by configuring the number of PIM cores per cluster, vertex- or edge-level parallelism within a PIM cluster or within a PIM core, such that to efficiently support diverse real-world graphs.

We design PyGim to adapt to the graph's characteristics with minimal programmer intervention. We integrate in PyGim a **lightweight tuner** that predicts the best-performing PaF aggregation configuration based on the particular characteristics of the input graph. PyGim's tuner employs effective performance models to estimate performance of different GNN aggregation configurations in PIM systems at low cost. This way, we automate the selection of the PyGim PaF configuration and eliminate the need for manual programmer intervention, while also providing high system performance. We develop a PIM backend for our optimized implementations and expose them with a **handy Python API** (See Alg. 2), so that programmers can easily use them. We integrate our API with PyTorch [105] (it can be integrated to other frameworks [1, 15, 46, 58]) to support either CPU or GPU as the Host (GPU-PIM systems are expected to be commercialized) in GNN PIM-based executions. PyGim supports two widely-used compression formats for real-world graphs. To our knowledge, PyGim is *the first easy-to-use and high-level ML library to deploy GNN models in real PIM systems*, and is available as open-source to enable the wide use of PIM systems in GNNs.

We comprehensively characterize GNN execution on the UPMEM PIM system, the first real-world PIM architecture, which has 16 PIM DIMMs with 1992 PIM cores connected to Host CPU. We evaluate our techniques in terms of scalability, data transfer costs, aggregation kernel and inference performance and energy efficincy using various real-world graphs and various GNN models. We

[1]PyGim is publicly available at https://github.com/CMU-SAFARI/PyGim.

compare PyGim over prior state-of-the-art PIM-based works for GNNs and show that it achieves significantly higher performance by on average 4.38× (up to 7.20×) and higher energy efficiency by on average 2.86× (up to 3.68×) in GNN inference. PyGim improves GNN inference performance and energy efficiency over the state-of-the-art PyTorch scheme running on Host by on average 3.04× (up to 3.44×) and 1.55× (up to 1.75×), respectively. Moreover, PyGim achieves on average 11.6× higher resource utilization on PIM system than that of PyTorch's backend on GPU systems, which is an optimized CUDA implementation from pytorch_sparse library [32]. This means that PyGim uses the PIM system more effectively than PyTorch's backend library uses the GPU system. Our extensive study provides recommendations to improve multiple aspects of future PIM hardware, systems and software. We hope that our ML library encourages further research and deployment of GNNs and sparse ML models in real PIM systems.

Overall, we make the following contributions:

- We investigate the challenges of efficiently implementing GNNs on real-world PIM architectures, and propose an easy-to-use high-level GNN library, named PyGim, for such systems. PyGim is open-source to enable further research.
- We combine the execution of heterogeneous kernels running on Host and PIM cores with a multi-level parallelization model. We enable three levels of parallelism and reduce the data transfer overheads from/to PIM memory modules in a heterogeneous GNN execution. We provide various parallelization strategies and load balancing schemes to cover diverse real-world graphs.
- We design a fast tuning mechanism to adapt the parallelization configuration of GNNs in PIM systems to the particular characteristics of the input graph, eliminating the need for manual programmer intervention. We expose our optimized PIM backend implementations as a handy Python API that can be integrated with state-of-the-art ML frameworks such as PyTorch.
- We extensively study the potential of real-world PIM architectures in GNNs using various real-world graphs and models. We show that PyGim significantly outperforms prior approaches both in performance and energy efficiency, and provides high resource utilization on real PIM systems.

2 Background & Motivation

2.1 GNNs in Commodity Systems

GNNs are emerging ML models that analyze graph-structured data (e.g., knowledge graphs, social and road networks). A GNN has a few layers. Each layer takes as input (i) the graph $G = (V, E)$, where V and E represent the graph's vertices and edges (connections between vertices), respectively, which is stored as a matrix, referred to as **adjacency matrix** A, and (ii) the *feature* matrix F, that has one feature vector per vertex in the graph, each vector encodes the vertex's characteristics. Most real-world graphs are typically sparse (less than 1% density) [39, 141], i.e., they have relatively few connections between vertices compared to the total number of possible connections. Thus, the adjacency matrix is stored in memory in a compressed format, e.g., Compress Sparse Row (CSR) [10]. The feature matrix is dense with size $N \times K$, where N is the number of vertices and K is the number of features per vertex (henceforth referred to as **hidden size**).

Fig. 1 shows the GNN layer execution that has two steps: the *aggregation* and *combination*. In aggregation, each vertex gathers the feature vectors of its neighbors, and produces an aggregated vector though an operator (e.g., average). In combination, the aggregated vectors of all vertices are processed through a small neural network, that typically has dense operators (e.g., GEMMs) and finishes with an activation. The output feature vectors of all vertices serve as an input feature matrix of the next layer in GNN model.

The aggregation and combination operators vary slightly across GNN models. For example, GCN [67] uses a *weighted sum* function for aggregation, while GIN [135] uses a *sum* function. GIN

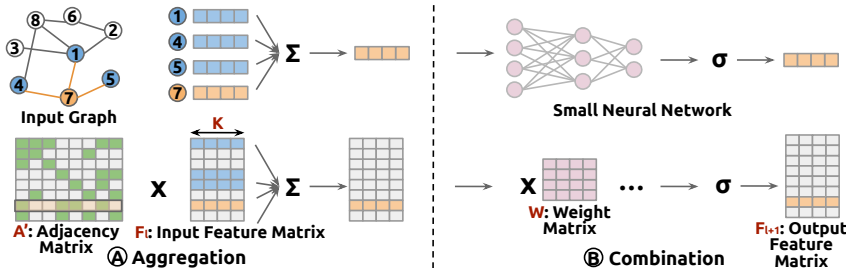

Fig. 1. Overview of the GNN layer execution workflow.

uses an MLP for combination, while SAGE [49] uses a fully-connected operator. Assuming that A' is a normalized adjacency matrix based on the aggregation function of each particular model, F^l is the input feature matrix of a layer l with hidden size K, and W_i^l matrices are weight matrices used in the small neural network of combination, the GNN computation can be expressed as:

$$F^{l+1} = \sigma(\sigma((A' * F^l) * W_1^l)... * W_w^l)$$

F^{l+1} will be the input feature matrix of the next layer. The aggregation step corresponds to the computation $A' * F^l$, which is a Sparse Matrix Matrix Multiplication (SpMM).

Recent works [128, 141, 148] show that GNN aggregation takes the largest portion of the execution time, because it is bottlenecked by memory bandwidth in processor-centric systems (CPUs/GPUs). We evaluate a 3-layer GNN in a RTX 3090 GPU and observe that aggregation takes ~91% of the inference time. Fig. 2 shows the roofline model, when executing a GNN layer in the GPU. Even in a high-end GPU with more than 900GB/s bandwidth, aggregation is highly limited by memory bandwidth. Moreover, as we show in Table 2, the resource utilization in aggregation is very low, i.e., on average 0.44% and 1.19% in CPU and GPU systems, respectively, due to the bottleneck of moving data from memory to processors. Therefore, we conclude that GNN aggregation is significantly limited by data movement in processor-centric systems like CPUs and GPUs.

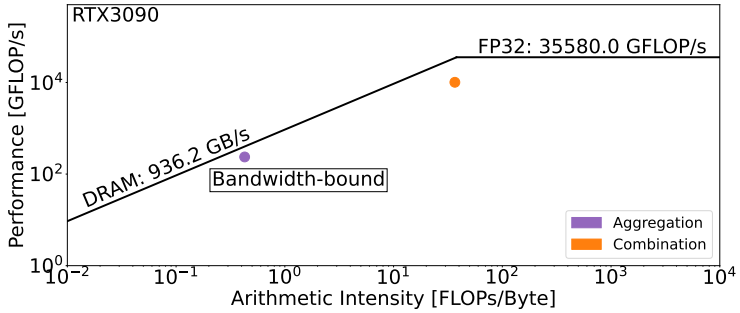

Fig. 2. Roofline model in the NVIDIA RTX 3090 GPU for aggregation and combination kernels.

2.2 Processing-In-Memory (PIM) Systems

PIM computing paradigm [4, 74, 96, 130] enables memory-centric computing systems: processing units (general-purpose cores or specialized accelerators) are placed near or inside memory arrays. PIM is a practical solution to alleviate the data movement bottleneck of processor-centric systems, and in this work, we study the potential of real PIM systems in GNNs. According to the location of PIM cores, PIM architectures could be classified into two categories: (i) *near-rank* in which PIM cores are placed at the buffer chip of the DIMM and have access to all DRAM banks of the DIMM,

and (ii) *near-bank* in which each PIM core is tightly coupled with one (or a few) DRAM banks, and can access data placed in its local bank(s). Placing PIM cores at a lower level provides larger aggregated memory bandwidth, enabling higher levels of parallelism. For example, UPMEM near-bank PIM can provide up to ~80GB/s internal bandwidth per DIMM [47], while TensorDIMM [71] near-rank PIM only 22GB/s per DIMM. Therefore, several manufacturers [25, 74, 75] target the commercialization of near-bank PIM designs to enable high levels of (bank-level) parallelism and support *thousands* of PIM cores. UPMEM PIM [25] has already commercialized a PIM product that has a general-purpose core near each memory bank of a DDR4 DRAM chip. HBM-PIM [74] and AiM [50, 75] have been prototyped and evaluated in real systems. HBM-PIM proposes a SIMD unit with 16-bit floating-point support between every two banks in memory layers of HBM stack. AiM is a GDDR6-based PIM system with near-bank cores that support multiply-and-accumulate and activation operations.

These real-world PIM systems have some important common characteristics, shown in Fig. 3. First, there is a Host processor (CPU or GPU) typically having a deep cache hierarchy, which is connected to standard main memory and PIM-enabled memory. Second, the PIM-enabled memory module has one (or a few) memory devices (rank of 2D DRAM or stacked layer of 3D-stacked DRAM). Each PIM device contains multiple processing elements (PIM cores), that have access to memory banks with higher bandwidth and lower latency than the Host cores. Third, the PIM cores (general-purpose cores, SIMD units, or specialized processors) run at only a few hundred megahertz, and have relatively small (or no) scratchpad or cache memory. Fourth, PIM cores may not be able to directly communicate with each other (UPMEM, HBM-PIM or AiM in different chips), and communication between them typically happens via the Host.

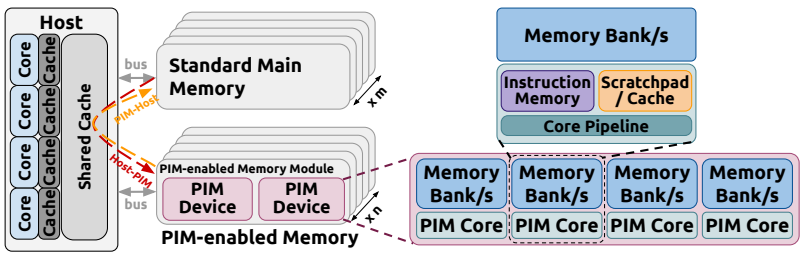

Fig. 3. Overview of a real near-bank PIM system. Host has access to *m* standard and *n* PIM-enabled modules.

In our evaluation, we use UPMEM PIM [25], the first PIM system that has been commercialized on real hardware. UPMEM PIM uses 2D DRAM arrays and combines them with general-purpose cores, called *DPUs*, on the same chip. Each PIM-enabled module has two ranks (devices), each rank has 8 chips, and each chip has 8 DPUs. Each DPU is tightly coupled to a DRAM bank, has a 14-stage pipeline, and supports multiple threads (up to 24), called *tasklets*.

DPUs have a 32-bit RISC-style general-purpose instruction set, and natively support in hardware 32-bit integer addition/subtraction and 8-bit/16-bit multiplication. More complex operations, e.g., 32-bit integer multiplication/division, and floating-point operations are software emulated [48]. Each DPU has access to its own (1) 64MB DRAM bank, called MRAM, (2) 24KB instruction memory, and (3) 64KB scratchpad memory, called WRAM. The Host CPU can access the MRAM banks to copy input data (from main memory to MRAM, i.e., CPU-DPU) and retrieve results (from MRAM to main memory, i.e., DPU-CPU). The CPU-DPU/DPU-CPU transfers can be performed in parallel across multiple MRAM banks, if the size of data transferred from/to all MRAM banks is the same. There is no direct communication channel between DPUs, and inter-DPU communication takes place via the Host [61].

In our paper, we use generic terminology since our optimization strategies can generally apply to near-bank PIM systems (e.g., HBM-PIM or AiM), like the generic one shown in Fig. 3, and not exclusively to the UPMEM PIM. Thus, we use the terms PIM device, PIM core, PIM thread, DRAM bank, scratchpad, and Host-PIM/PIM-Host data transfer, corresponding to PIM rank, DPU, tasklet, MRAM bank, WRAM, and CPU-DPU/DPU-CPU data transfer in UPMEM's terminology.

2.3 Prior PIM-Based GNN Accelerators

A few prior works [128, 141, 148] propose hardware accelerators for GNN aggregation; however, they do not consider real-world PIM systems. They target near-rank PIM systems that have not been developed as proof-of-concept silicon yet, and typically provide lower memory bandwidth than near-bank PIM systems [74], which have already been commercialized. Moreover, these prior works focus on the ASIC design of the processing units, and implement simple parallelization strategies at the software level, which have been evaluated in software simulators (instead of a real system). Specifically, these works do not comprehensively evaluate the performance overheads of transferring data to/from PIM memory modules (i.e., Host-PIM and PIM-Host data transfer costs).

We find that applying the simple parallelization strategies of prior PIM-based GNN accelerators [128, 141, 148] is not suitable or efficient for real near-bank PIM systems. In detail, GNNear [148] equally distributes the graph's vertices across PIM cores, and the large dense feature matrix is copied at each PIM device of the system. Applying this approach at near-bank PIM systems would necessitate to replicate the large dense matrix of size N (vertices) $\times$ K (hidden size) at *each* (or a few) PIM memory bank, and would cause out-of-memory errors for medium-/large-size graphs: e.g., assuming an UPMEM PIM bank of 64MB and 256 hidden size, GNNear's approach would support only small graphs with maximum ~64K vertices. G-NMP [128] equally distributes the feature matrix across PIM units. On the one hand, distributing the hidden size dimension K of the feature matrix across PIM cores in near-bank PIM systems would leave *many* PIM cores *idle* causing low PIM utilization: GNN layers typically have a much smaller hidden size (e.g., 128 or 256 [43, 49, 54, 67, 135, 138, 147]) than the available PIM cores (thousands of cores) of near-bank PIM systems (e.g., UPMEM PIM system has 2560 PIM cores). On the other hand, distributing the vertex dimension N of the feature matrix across the PIM cores would create a partial result of size $N \times K$ for the output feature matrix of aggregation at *each* (or a few) PIM bank. Similarly to GNNear's scheme, this approach would cause out-of-memory errors for medium-/large-size graphs (e.g., graphs with only up to ~64K vertices could be supported in UPMEM PIM), while also a large number of partial results would need to be merged by Host cores. Finally, considering all prior PIM-based works in GNNs, GraNDe is the state-of-the-art work, and the most optimized in terms of its parallelization strategy. GraNDe [141] demonstrates that a 2D distribution scheme provides the highest on average performance in near-rank PIM systems. In this scheme, the vertex dimension N of the feature matrix is distributed across PIM modules (e.g., PIM devices (ranks) in a near-bank PIM system) and the hidden size dimension K of the feature matrix is distributed across PIM units of the same device (e.g., PIM cores of the same PIM device (rank)). We evaluate this scheme in a real near-bank PIM system, and show that it achieves very low performance, being on average 6.3× (Fig. 14) worse in GNN inference compared to our proposed PyGim approach. This is because GraNDe is tailored for near-rank PIM systems, rather than near-bank PIM systems.

3 PyGim: Detailed Design

PyGim is a easy-to-use ML library to efficiently execute GNNs on real near-bank PIM systems. See common characteristics in §2.2. PyGim improves system efficiency by running compute-bound and memory-bound kernels on processor-centric and memory-centric hardware, respectively (§3.1), and providing highly efficient parallelization strategies in GNN aggregation tailored for real PIM

systems (§3.2). Moreover, PyGim adapts to the characteristics of the real-world graph without any programmer intervention via a lightweight tuning mechanism (§3.3) that automates the aggregation configuration. PyGim enables high programming ease via a high-level ML-friendly interface (§3.4) that is integrated with state-of-the-art ML frameworks.

3.1 Cooperative Acceleration Model (CoA)

GNN execution alternates between sparse and dense operators: the aggregation step degenerates to an SpMM kernel, which is bottlenecked by memory bandwidth in processor-centric systems (Fig. 2), while the combination step mainly comprises compute-heavy kernels, e.g., GEMMs. We employ a *Cooperative Acceleration* (*CoA*) model that efficiently maps and executes each step to the best-fit underlying hardware. For *each* GNN layer, PyGim executes the SpMM kernel of aggregation on PIM cores to leverage immense memory bandwidth available on PIM system, and the compute-heavy kernels of combination on Host (CPU or GPU) to exploit large processing capabilities available on processor-centric systems. Since aggregation and combination are repeated one after the other in multiple consecutive GNN layers, the **key challenge** is how to minimize the overheads of passing the output of the one step as input to the next step. We discuss how we address it in the next subsection. Note that while PIM cores are running aggregation, Host cores are idle, until the dependent computation is finished, and vice versa. We leave for future work the extension of PyGim to offload part of aggregation and combination computations on Host and PIM cores, respectively, to minimize idleness.

3.2 Parallelism Fusion (PaF)

Fig. 4 shows the GNN aggregation execution on a real PIM system that can be broken down in four execution steps: (1) the time to transfer the input feature matrix of combination from Host into DRAM banks of PIM-enabled memory (**Host-PIM**), (2) the time to execute computational kernel on PIM cores (**Kernel**), (3) the time to retrieve from DRAM banks of PIM-enabled memory to the Host the results for the output (**PIM-Host**), and (4) the time to merge partial results and assemble the final output feature matrix on the Host (**Merge**). The graph (adjacency matrix) is pre-loaded into PIM-enabled memory *once*, i.e., when reading the graph file from the disk and loading it to DRAM (pre-processing step). The same graph (adjacency matrix) is *reused* across all layers.

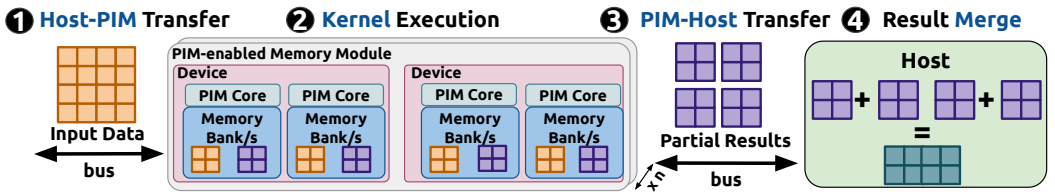

Fig. 4. Execution of aggregation step on a real PIM system.

We design *Parallelism Fusion* (PaF) to mitigate data transfer and kernel time performance costs. We enable three levels of parallelism, each level implements a different strategy, shown in Fig. 5. The first-level strategy reduces data transfer overheads (Host-PIM and PIM-Host), thus addressing the aforementioned key challenge, while the second- and third-level strategies reduce computation overheads (kernel time) to achieve high PIM performance. This way PaF provides the sweet-spot and strives a balance between computation and data transfer costs. Moreover, in sparse computational kernels, including SpMM, the execution behavior depends on the particular characteristics of the input given, i.e., the input graph in GNNs. Therefore, PyGim enables various data partitioning approaches and load balance schemes, such that to tune the parallelization strategy based on the

particular characteristics of the input graph, thus achieving performance that is as close as possible to the optimal performance for that given graph.

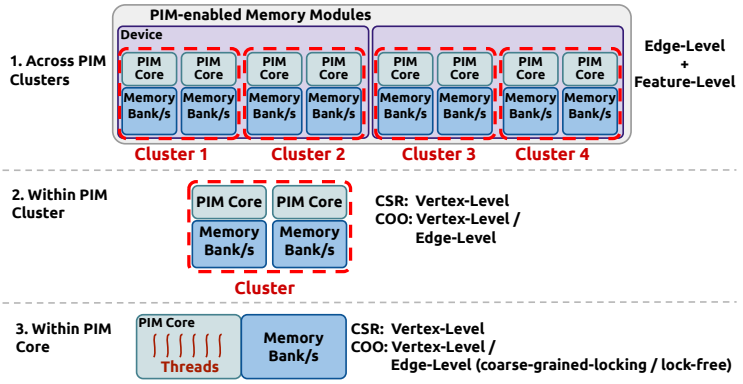

Fig. 5. PaF overview.

Across PIM Clusters. We group the available PIM cores into clusters, named **PIM clusters**, and execute a part of the SpMM aggregation at each cluster. A PIM device, i.e., a rank in 2D DRAM (e.g., UPMEM PIM system) or a stacked layer in 3D DRAM (e.g., HBM-PIM system) can contain multiple PIM clusters, while all cores of the same cluster belong to the same PIM device (grouping cores located to different PIM devices is inefficient, since it would need multiple *separate* Host-PIM and PIM-Host transfers for the *same* cluster, causing high transfer and launch overheads).

We parallelize SpMM across PIM clusters via a hybrid **edge-level** and **feature-level** approach, as shown in Fig. 6. Each cluster processes a subset of the graph's edges and a subset of the vertices' features to minimize Host-PIM and PIM-Host transfer costs. We create vertical partitions (edge-level parallelism) on the adjacency sparse matrix, the number of which is henceforth referred to as **sparse partitions**. The adjacency matrix is distributed vertically across multiple PIM clusters, where each vertical partition is assigned to multiple PIM clusters. To minimize the amount of partial results produced for the final output matrix, and thereby minimizing the PIM-Host transfers, we combine edge-level parallelism with feature-level parallelism: we also create vertical partitions on the feature dense matrix, the number of which is henceforth referred to as **dense partitions**. Edge- and feature-level parallelism split the feature matrix both vertically and horizontally. This way the feature matrix is distributed into 2D tiles, the number of which is equal to the number of PIM clusters used, i.e., each 2D tile is assigned to a PIM cluster. Multiple PIM clusters process in parallel the 2D tiles of the feature matrix with their corresponding vertical partitions of the adjacency matrix, executing a part of the SpMM. Multiple PIM clusters produce dense output matrices that correspond to partial results for the final output matrix. These partial results are merged in the Host cores (merge step) via matrix matrix addition.

Let's assume the number of sparse partitions is s and the number of dense partitions is d. The sparse matrix is split into s vertical partitions, and the feature matrix is split into s horizontal partitions and d vertical partitions, creating $s \times d$ 2D tiles. Each PIM cluster will be assigned to process a 2D tile of the feature matrix with the corresponding vertical partition of the adjacency matrix. The clusters that are assigned to 2D tiles of the same vertical partition in the feature matrix create partial results for the output matrix. The values of sparse partitions s and dense partitions d, and the number of PIM clusters per PIM device can be configured either manually by the user or automatically by the PyGim tuner, as described in §3.3. The PIM clusters per device multiplied with the number of PIM devices used in SpMM execution needs to be equal to the number $s \times d$. By

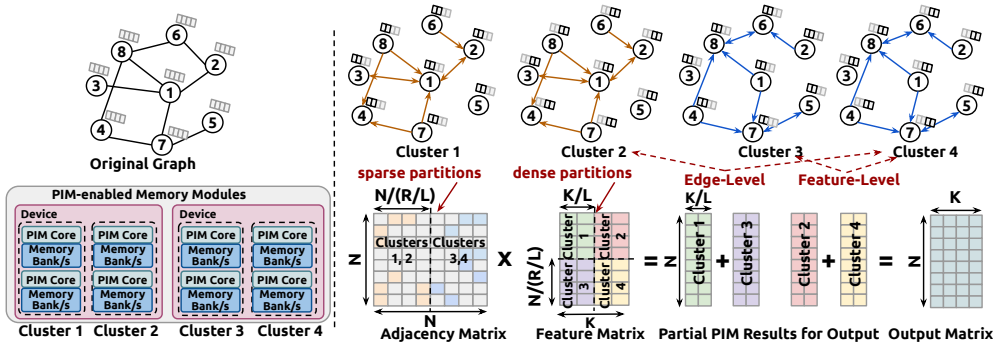

Fig. 6. Example of edge-level and feature-level parallelism across PIM clusters in PyGim.

carefully selecting the values of sparse partitions s and dense partitions d, as well as the number of PIM clusters per PIM device, PaF technique can significantly minimize the Host-PIM and PIM-Host data transfer costs.

Fig. 6 shows an example of two sparse partitions and two dense partitions and a PIM system of four PIM clusters. In this example, there are four 2D tiles in the feature matrix each assigned to a different PIM cluster. The clusters 1 and 2 are assigned to 2D tiles that belong in the first 2D tile row of the feature matrix, while the clusters 3 and 4 are assigned to 2D tiles that belong in the second 2D tile row of the feature matrix. Therefore, in the adjacency matrix, clusters 1 and 2 are assigned to process the first vertical partition, i.e., the vertical partition that is associated with the orange edges of the input graph, while clusters 3 and 4 are assigned to process the second vertical partition, i.e., the vertical partition that is associated with the blue edges of the graph. The clusters 1 and 3 create partial results for the final output matrix corresponding to the first two columns of the final output matrix, while the cluster 2 and 4 create partial results for the final output matrix corresponding to the last two columns of the final output matrix. Assuming a graph with N vertices, hidden size K and R PIM clusters, when creating L equal partitions on the feature matrix ($L < R$), each PIM cluster processes a feature matrix tile of size $(N/(R/L)) \times (K/L)$, which corresponds to the Host-PIM transfer cost for this cluster, and produces a partial output matrix of size $N \times (K/L)$, which corresponds to the PIM-Host transfer cost for this cluster. Although in the example of Fig. 6 the sparse and dense partitions have the same column width, PyGim can support variable-sized vertical partitions on the adjacency and feature matrices. However, with variable-sized partitions, PIM clusters process variable-sized 2D feature tiles and produce variable-sized partial output results. Our exploratory evaluations showed that this approach incurs high load imbalance in Host-PIM/PIM-Host transfers, causing high overheads. Thus, in our evaluations we present equal-sized partitions.

Within PIM Cluster. PyGim encodes the adjacency matrix in CSR [10, 108] and COO [108, 114] formats, the most widely-used compressed matrix storage formats for sparse matrices [56, 73, 94, 107]. We parallelize smaller SpMMs across PIM cores of the same cluster, by enabling *vertex-level* parallelism, if the adjacency matrix is stored in CSR, and either *vertex-level* parallelism or *edge-level* parallelism, if it is stored in COO. This way we provide compute balance across cores of the same cluster, minimizing the kernel time. The corresponding 2D feature matrix tile is replicated at each core of the same PIM cluster. Fig. 7 presents an example of parallelization across multiple cores of the *same* cluster with CSR and COO formats.

The CSR format (Fig. 7 left) sequentially stores the edges (non-zero elements) in a vertex-wise (row) order. A column index array (*colind[]*) and a value array (*values[]*) store the column index and the value of each non-zero element, respectively. An array *rowptr[]*, stores the location of the

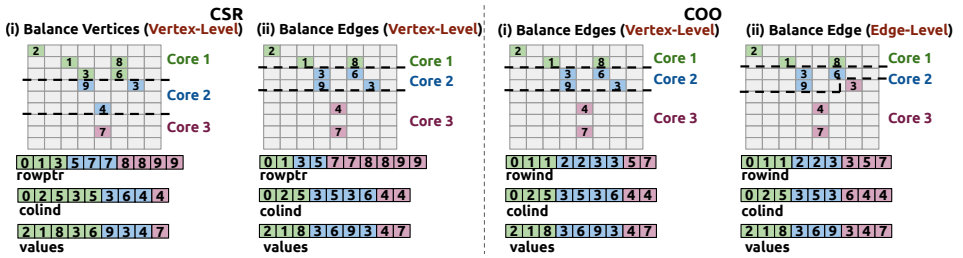

Fig. 7. Vertex- and edge-level parallelism across PIM cores within cluster. The gray cells represent zero values, while the green, blue and pink cells represent non-zero values (edges).

first non-zero element of each row within the *values[]* array. An adjacent pair *rowptr[i, i+1]* stores the number of the non-zero elements of the i-th row. Since in CSR the adjacency matrix is stored in vertex-wise order, we perform *vertex-level* parallelism across PIM cores of the same cluster: each core processes a subset of the vertices, i.e., consecutive rows in the adjacency matrix. We enable load balance across cores via two schemes: (a) equally balancing the number of vertices (rows) across PIM cores (Fig. 7 left i), or (b) equally balancing the number of edges (non-zero elements) across PIM cores at vertex (row) granularity (Fig. 7 left ii).

The COO format (Fig. 7 right) uses three arrays to store edges (non-zero elements): the row index (*rowind[]*), column index (*colind[]*) and value (*values[]*) arrays store the row index, column index and value of each non-zero element, respectively. Since in COO the adjacency matrix is stored in *non-zero-element-wise order* (edge-wise order), we enable either *vertex-level* parallelism, i.e., each PIM core of the cluster processes a subset of the vertices, or *edge-level* parallelism, i.e., each PIM core processes a subset of the edges. We enable load balance across cores via two schemes: (a) equally balancing the number of edges (non-zero elements) across cores at a vertex (row) granularity (Fig. 7 right i), or (b) equally balancing the number of edges (non-zero elements) across cores by enabling splitting a vertex (row) across two (or more) neighboring cores to provide near-perfect edge-level balance (Fig. 7 right ii). In (b), when a vertex (row) is split between neighboring PIM cores, the cores produce partial results for the same row of the output matrix, which are merged by Host cores.

Within PIM Core. We enable a similar scheme across threads of a PIM core with that enabled across PIM cores of the same cluster (Fig. 7). We enable high compute balance across threads of the same core to further minimize the kernel time. In CSR, we perform *vertex-level* parallelism by either (i) equally balancing the number of vertices (rows) across threads of a core, or (ii) equally balancing the number of edges (non-zero elements) at a vertex (row) granularity across threads. In COO, we either (i) equally balance the number of edges (non-zero elements) at a vertex (row) granularity across threads of a core (vertex-level parallelism), or (ii) equally balance the number of edges (non-zero elements) across threads, enabling splitting a vertex (row) across two (or more) PIM threads (edge-level parallelism). In the latter case, when a vertex (row) is split across two (or more) PIM threads, PIM threads perform write accesses to the *same* elements of the output matrix, thus synchronization among threads is necessary. We provide two synchronization schemes:

- **Coarse-grained locking**: one global mutex (lock) protects all the elements of the output matrix.
- **Lock-free**: given that we assign consecutive rows to each thread (consider Fig. 7 COO (ii) with threads instead of cores), race conditions might arise only when a vertex (row) is split across two (or more) threads. These vertices (rows) are proportional to the number of threads, which are only a few per core. Thus, the number of partial results for the same final output element are a few, and threads can temporarily store partial results in the scratchpad memory (e.g., WRAM in the UPMEM PIM system). Then, only one single thread merges the partial results by reading them from scratchpad memory, and writes the final result to the DRAM bank/s with no synchronization.

Kernel Implementation. We describe how threads access the data involved in SpMM. There are three types of data arrays: (i) the arrays that store the non-zero elements of the adjacency matrix, i.e., their values (*values[]*) and their positions (*rowptr[]*, *colind[]* for CSR, and *rowind[]*, *colind[]* for COO), (ii) the array that stores the elements of the feature matrix, and (iii) the array that stores the partial results for the output matrix. First, SpMM performs streaming memory accesses to the arrays that store the non-zero elements. To exploit PIM's immense internal bandwidth, each thread reads the non-zero elements (their values and positions) by fetching large chunks of bytes in a coarse-grained manner from DRAM bank to scratchpad (e.g., from MRAM to WRAM in the UPMEM PIM system). Then, it accesses data element by element via scratchpad. In the UPMEM PIM system, we fetch large chunks of 128-bytes/256-bytes, as suggested by prior work [47]. Second, SpMM processes the feature matrix elements at a row granularity, as a chunk of hidden size elements in the tile assigned to the PIM core, e.g., a chunk of K/L elements in Fig. 6. To exploit spatial locality, each thread reads the feature matrix elements by fetching chunks of tile hidden size × data type bytes from DRAM bank to scratchpad, and performs multiply-and-add. Third, threads temporarily store partial results for the elements of the same output matrix row in scratchpad, until all non-zero elements of the same row of the adjacency matrix are processed. This way we exploit temporal locality for multiple updates on the *same* output matrix elements. Then, the produced results are written from scratchpad to DRAM bank/s as a chunk of tile hidden size × data type bytes.

Merge Step. PyGim merges partial results created across PIM clusters and across cores within PIM cluster (merge step) on the Host cores. In our CPU-PIM system, we use the OpenMP API [21] to parallelize Merge, and perform (i) 2D block copy on the final output matrix for the partial results of the PIM clusters assigned to the first 2D block row of the feature matrix (Clusters 1,2 in Fig. 6) and (ii) 2D block reduction (add) operation on the final output matrix (matrix-matrix addition) for the partial results of the PIM clusters assigned to the remaining 2D block rows of the feature matrix (Clusters 3,4 in Fig. 6).

3.3 PyGim Tuner

PyGim's PaF is designed to support various parallelization and load balancing strategies to efficiently cover various real-world graphs: as shown in prior works [39, 62, 77, 138], the execution behavior of sparse kernels that process input data with diverse characteristics, such as the SpMM of GNN aggregation that processes real-world graphs with varying neighboring degrees, diameters etc, depends on the particular characteristics of the input. Therefore, to enable high performance by adapting the PaF strategy to the particular graph's characteristics and avoid programmer's intervention, we integrate in PyGim a lightweight tuner that selects the aggregation configuration to be used: the user selects the compression format (CSR/COO), and then the tuner predicts the best-performing aggregation configuration, i.e., the sparse and dense partitions, the groups per device, the selection of vertex- or edge-level parallelism across cores of a PIM cluster and across threads of a PIM core.

To enable the tuner, we first run a few microbenchmarks on the PIM system to collect information on hardware characteristics. These microbenchmarks run within less than ~30 secs, and are executed only *once* per PIM server to gather runtime characteristics of the underlying hardware. Our approach is similar to prior prediction and profiling tools for ML executions [7, 16, 35, 63, 144]. We devise four microbenchmarks. 1) The *Host-PIM-BW-byte* and *PIM-Host-BW-byte* measure the Host-PIM and PIM-Host bandwidth, respectively, when using multiple PIM devices and transferring M bytes to/from each PIM core (*PCore*). We vary M from 64KB to 8MB, collecting 16 different byte sizes. 2) The *Host-BW-byte* measures the Host bandwidth of standard DRAM modules, when copying M bytes from one memory area (i.e., allocated matrix) to another area. We vary M from 8B to 2KB and collect 9 different byte sizes. 3) *FMA-PCore-chunk* is the fused multiply-add (FMA) throughput

achieved by a PIM core (*PCore*), i.e., number of FMA operations executed per second, when the PIM core performs FMA operations on data values that are transferred from the DRAM bank to scratchpad memory as chunks of M elements. We vary M from 2 to 512 elements collecting 9 different chunk sizes. 4) *ADD-Host-block* is the addition (ADD) throughput achieved by Host cores, when accumulating (ADD) the M elements of a block to a larger allocated matrix. We vary M from 2 to 512 elements collecting 9 different block sizes. After collecting data on hardware characteristics, the tuner estimates the execution time of an aggregation configuration using the following analytical models:

$$T_{\text{total}} = T_{\text{Host-PIM}} + T_{\text{Kernel}} + T_{\text{PIM-Host}} + T_{\text{Merge}}$$

$$T_{\text{Host-PIM}} = \frac{\text{PCores x max-bytes-to-PCore}}{\text{Host-PIM-BW-byte (closest)}}$$

$$T_{\text{Kernel}} = \text{max-NNZs-PCore} \times \text{FMA-PCcore-chunk (closest)}$$

$$T_{\text{PIM-Host}} = \frac{\text{PCores} \times \text{max-bytes-from-PCore}}{\text{PIM-Host-BW-byte (closest)}}$$

$$T_{\text{Merge}} = \frac{\text{dp x cluster-2Dtile-byte}}{\text{Host-BW-byte (closest)}} + \frac{\text{(sp-1) x dp x cluster-2Dtile}}{\text{ADD-Host-block (closest)}}$$

The *max-bytes-to/from-PCore* and *max-NNZs-PCore* are the maximum bytes sent/received to/from a PIM core in Host-PIM/PIM-Host transfers and the maximum non-zero elements (*NNZs*) processed by a PIM core, respectively. The *sp* and *dp* are the number of sparse and dense partitions created. The *cluster-2Dtile* and *cluster-2Dtile-byte* represent the number of elements and bytes of the 2D tile for partial results created by a PIM cluster (e.g., $N \times (K/L)$ in Fig. 6), respectively. For the Host-PIM-BW, PIM-Host-BW and Host-BW, we use the collected bandwidth measurement associated with the data size that is closest to the data size of the aggregation configuration we are estimating. For FMA and ADD throughput, we use the collected throughput measurement associated with the chunk of elements/block size that is closest to the chunk of elements/block size of the aggregation configuration we are estimating.

Alg. 1 presents a brief description of the PyGim tuner which iterates over all possible aggregation configurations and predicts the best-performing configuration. Specifically, the tuner iterates over all divisors of the available PIM devices in the system (line 5), creates 1, 2, or 4 clusters per PIM device, and it computes the number of dense partitions (line 7). If the number of dense partitions is larger than the hidden size, this configuration is not valid and is omitted (lines 9). Otherwise, the tuner iterates over the possible load balance strategies within the PIM cluster (line 10) and within the PIM core (line 11). For each current configuration, the tuner estimates the performance using the described analytical models (line 12), and keeps the lowest estimated execution time and its corresponding configuration in local variables (lines 13-15). When the tuner has examined all possible configurations, it returns the estimated best-performing configuration (line 16).

The PyGim tuner is optional, providing flexibility to the programmer who may choose to utilize it or not. If the tuner is not used, the programmer is responsible for manually selecting and providing the desired aggregation configuration. In Alg. 2 (an example of GCN inference), line 18 needs to be replaced with a manually tuned configuration. If the tuner is enabled, it takes around ~33 secs and is executed *once*, when reading the graph file from the disk and loading it into PIM devices in the pre-processing step. Then, users can submit multiple GNN inference requests to query properties of vertices/edges or the existence of edges between graph's vertices [49, 67, 132, 137, 143]. Finally, the tuner's goal is to estimate the best-performing aggregation configuration by leveraging simple analytical models. In our evaluations (Fig. 12), we evaluate the efficiency of our tuner by comparing the performance achieved by the predicted configuration of tuner versus the best-performing

```
def tune(graph, hidden_size, device_info):
    clst_cfg = ['ver', 'edg'] # load balance within PIM cluster
    core_cfg = ['ver', 'edg'] # load balance within PIM core
    best = inf, best-cfg = []
    for sp in divisors of num_pim_devices: # sparse partitions
        for grp in [1, 2, 4]: # clusters per PIM device
            dp = num_pim_devices / sp * grp # dense partitions
            if dp > hidden_size:
                continue
            for cl in clst_cfg:
                for cr in core_cfg:
                    Ttotal = predict(graph, hidden_size, grp, sp, dp, cl, cr, device_info)
                    if (Ttotal < best):
                        best = Ttotal
                        best-cfg = [sp, dp, grp, cl, cr]
    return best-cfg
```

Algorithm 1. PyGim tuner for the aggregation operator.

manually tuned configuration (oracle prediction), and show that the simple analytical models used by the tuner are highly effective.

3.4 PyGim API and Integration

Combination comprises a small neural network, thus PyGim leverages existing optimized ML kernels from PyTorch to execute the corresponding ML operators of GNN combination on Host cores. We integrate PyGim with PyTorch, as we explain in the next paragraph. Note that although we only have access to a CPU-PIM system for our evaluations, PyGim can support GPU-PIM GNN executions (GPU-PIM systems that are expected to be available in the market) by leveraging PyTorch's supported backends (CPU and GPU).

To interact with PIM devices (e.g., Host-PIM/ PIM-Host transfers) in aggregation, Host code needs to be implemented. The Host code implements (developed in C language) the parallelization approaches and the corresponding data partitioning schemes proposed in §3.2, when loading the graph into PIM-enabled memory modules (pre-processing step). The kernel code that PIM cores are running is implemented using the UPMEM PIM interface, since this is the only commercially available real PIM system. This interface is also written using the C language, and it can be easily ported to other PIM systems with similar interfaces to UPMEM. We create a PIM backend for PIM aggregation and expose this software runtime as a handy Python API so that programmers can easily use it via a high-level programming interface. We combine our Python-like API with PyTorch [105] to enable efficient CoA execution. PyGim's PIM aggregation can be also easily integrated to other ML frameworks, such as TensorFlow [1], Keras [46], MXNet [15] and Caffe [58].

Alg. 2 presents an example of GCN inference with PyGim. Programmers need to (i) allocate PIM devices (line 15), (ii) load graph data into PIM-enabled memory (lines 17-19), (iii) create a GNN model (lines 21-25), and (iv) run GNN inference by configuring aggregation and combination to be executed on the PIM cores (line 9) and Host cores (line 11), respectively. The allocated PIM resources are released, when the program exits.

4 Evaluation

4.1 Methodology

System. We use the UPMEM PIM architecture, a real-world PIM system. The system consists of a Host CPU (2-socket Intel Xeon with 8-cores each and a total of 32 threads at 2.10 GHz), standard DDR4 memory (128 GB), and 16 PIM DIMMs of 2 ranks (124.5 GB and 1992 PIM cores at 350 MHz),

```
import torch, pygim as gyn
class GCNConv(torch.nn.Module):
  def __init__(self, hidden_size):
    self.linear = torch.nn.Linear(hidden_size, hidden_size)

  def forward(self, graph_pim, in_dense):
    # Execute Aggregation in PIM
    dense_parts = col_split(in_dense)
    out_dense = gyn.pim_run_aggr(graph_pim, dense_parts)
    # Execute Combination in Host
    out = self.linear(out_dense)
    return out

# Allocate PIM Devices
gyn.pim_init_devices(num_pim_devices, groups_per_device)
# Load graph in PIM devices
data = load_dataset()
graph_parts, config = gyn.tune(data.graph, hidden_size, device_info)
graph_pim = gyn.load_graph_pim(graph_parts, config)
# Create GNN model
model = torch.nn.Sequential([Linear(in_channels, hidden_size),
  GCNConv(hidden_size),
  GCNConv(hidden_size),
  GCNConv(hidden_size),
  Linear(hidden_size, out_channels) ])
model.forward(graph_pim, data.features)
```

Algorithm 2. Example of GCN execution with PyGim API.

each rank has 64 cores. There are 56 faulty cores in the evaluated system that cannot be used, but they do not affect the correctness of our results (they are not used in our experiments).

Models and Datasets. We evaluate the GCN [67], GIN [135] and SAGE [49] models. The multiplication of floating point data types is software emulated in the UPMEM PIM system. Thus, we present detailed evaluations with 32-bit integer (**int32**) data type, since it has the same byte width with 32-bit float (**fp32**), its arithmetic operations are more effectively supported in UPMEM PIM hardware, and provides high accuracy (having int32 for both computation and memory representation results to less than 1% accuracy drop in all models and datasets over fp32 using the quantization scheme of Ctranslate2 [20]). Quantization [20, 90, 146] is orthogonal to our optimizations, and we expect that future PIM systems (e.g., HBM-PIM) will provide native floating-point arithmetic support or optimized quantization schemes will provide high accuracy with fixed-precision data types. We evaluate real-world sparse matrices from the Sparse Matrix Suite Collection [24], when using one PIM core and one PIM cluster, and present large-scale experiments with three real-world graph datasets: ogbn-proteins [124], Reddit [49] and AmazonProducts [142]. See also Appendix A.7 for detailed matrix and graph dataset characteristics.

Comparison Points. We use the PyG library [109] for GNN implementation. In GNN inference, combination runs on the Host CPU with PyTorch's default backend implementation. In GNN aggregation, we compare PyGim with prior software schemes for PIM systems proposed in the literature. In PyGim and other PIM-based software schemes, when the kernel step of GNN aggregation is running on PIM cores, the host CPU cores are idle. We also compare PyGim with a CPU-only scheme, that runs on *same* system with PIM schemes, i.e., the Host CPU side of the UPMEM PIM server, similarly to the methodology of prior state-of-the-art PIM works [14, 39, 57, 61]. Overall, we compare PyGim with four schemes:

- **PyTorch**: the PyTorch's backend which is the state-of-the-art matmul operator from pytorch_sparse library [32]. We evaluate the latest default implementation of matmul that uses the optimized Intel MKL library. We run this scheme using all 32 threads of the 32-thread Intel Xeon CPU.

- **GraNDe** [141]: the best-performing PIM scheme of prior PIM-based work for GNNs [141] that equally distributes the vertex (row) dimension of the feature matrix across PIM devices, and then equally distributes the hidden size of the feature matrix across cores of the same PIM device.
- **SP1** and **SP2** [39]: two SpMV-based schemes of prior work [39] for real PIM systems. SparseP [39] proposes SpMV kernels for PIM systems, and shows that their optimized COO.nnz-lf kernel performs best, when using ∼2 PIM devices. We run aggregation as an SpMV execution: for each column of the feature matrix, we execute one SpMV kernel using either one PIM device (**SP1**) or two PIM devices (**SP2**), and parallelize multiple SpMVs for the multiple columns of the feature matrix using multiple PIM devices.

4.2 Within PIM Core Analysis

We evaluate SpMM for int32 and fp32 data types with multiple threads of a PIM core. Fig. 8 shows scalability of CSR when equally balancing the vertices (**RV**) or edges at vertex granularity (**RE**) across threads, and COO when equally balancing the edges at vertex granularity (**CE**) or via near-perfect edge balance using the coarse-grained locking (**CP-cg**) or lock-free (**CP-lf**) schemes.

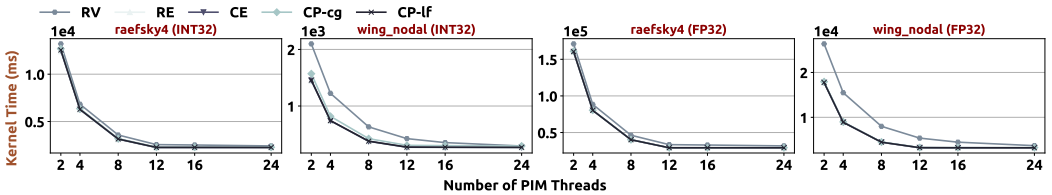

Fig. 8. Scalability of all schemes with of a PIM core in int32 (left) and fp32 (right) data types, as the number of threads of a PIM core increases.

We draw three findings. First, all schemes scale up to 16 threads, because the PIM core pipeline is fully utilized after 16 threads. In wing_nodal with 16 threads, only one thread processes many more edges than the rest, thus RV slightly scales to 24 threads (by 0.09%), because it exhibits better compute balance across threads. Second, int32 data type provides at least one order of magnitude better performance than fp32 data type. The UPMEM PIM core does not support in hardware floating-point operations, while they are software emulated using integer arithmetic units. Thus, fp32 SpMM achieves much lower performance than int32 SpMM due to the excessive amount of computations. Third, RV provides worse performance than other schemes, since balancing the vertices across threads incurs high edge (non-zero element) imbalance, thus causing high disparity in the amount of computations performed across threads (high compute imbalance).

Recommendation 1:
PIM cores typically have low compute capabilities, thus we recommend programmers to design algorithms that minimize the amount of computations performed and support parallelization schemes that enable high compute balance across threads.

Recommendation 2:
Programmers can leverage quantization in ML models, if PIM cores have limited precision and arithmetic operation support in hardware, and can design quantized data types that enable low compute requirements (e.g., replacing multiplications with logical/shift/add operations).

4.3 Within PIM Cluster Analysis

Fig. 9 evaluates SpMM in one PIM cluster of 64 cores with int32 data type for CSR, when equally balancing the vertices (**RV**) or edges at vertex granularity (**RE**) across PIM cores, and for COO when equally balancing the edges at vertex granularity (**CE**) or via near-perfect edge balance (**CP**). Within each PIM core, we use 16 threads with edge-balance across threads in CSR and near-perfect edge balance across threads (lock-free synchronization) in COO. We present the execution time of the breakdown steps of Fig. 4, and sort matrices with increasing irregularity, i.e., standard deviation of non-zero elements among rows.

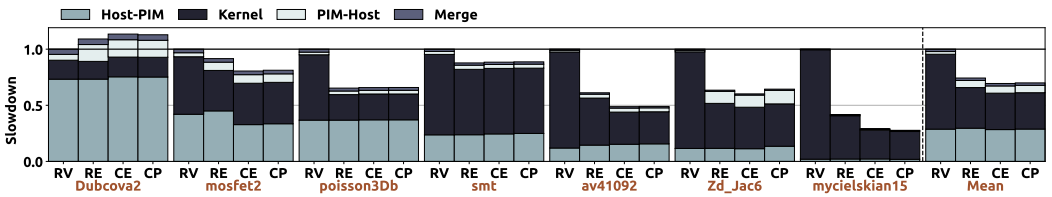

Fig. 9. Comparison of various schemes using one PIM cluster of 64 PIM cores and various sparse matrices.

We draw three findings. First, the vertex-balance scheme (RV) incurs higher kernel time than edge-balance schemes (RE, CE, CP) by 1.96×, because the latter provide high compute balance, i.e., similar number of edges (non-zeros) are processed across cores. Second, edge-balance schemes incur higher PIM-Host data transfer costs over vertex-balance by 2.63×. In UPMEM PIM, PIM-Host data transfers can be performed in parallel across multiple cores, if the transfer sizes from all DRAM banks are the *same*. To leverage parallel data transfers, we perform padding with empty bytes (zeros) at the granularity of a PIM device, when transferring data from/to Host. Edge-balance schemes have higher disparity in the number of vertices assigned to PIM cores, i.e., PIM cores produce different amount of partial results for the output matrix, thus they suffer from higher zero padding costs in PIM-Host data transfers. Based on first and second findings, we observe that if a single parallelization scheme, e.g., only vertex- or edge-parallelism, is used across all available PIM cores in the system (thousands of PIM cores), performance would be sub-optimal, since it would cause either high kernel (e.g., RV) or high data transfer time (e.g., RE, CE, CP). This is key experimental observation that inspired our PaF approach: PaF enables multiple parallelization strategies to trade off computation and data transfer costs in PIM executions. Third, most matrices have a power-law distribution [125], i.e., only a few vertices have a very large number of neighbors (edges), thus edge-balance schemes provide best end-to-end performance by significantly improving kernel time. Dubcova2 is a relatively regular matrix, thus the vertex-balance scheme provides enough compute balance across PIM cores, achieving 1.11× better total performance than edge-balance schemes.

Recommendation 3:

Commodity DRAM has multiple hierarchy levels (e.g., DIMM-, rank/layer-, bank group-, bank-level), each level has different characteristics in circuitry design. PIM architects can enable different hardware optimizations or accelerator cores at each level of hierarchy (e.g., adding processing capabilities both before and after the sense amplifiers of memory arrays). Then, system and software engineers can design different optimization techniques (e.g., different parallelization strategies) for *each* different level of hardware hierarchy (similar to our proposed PaF approach) to enable high performance in PIM executions via hardware software co-design.

Recommendation 4:
Data transfers to/from PIM memory are typically expensive (since they are performed via the common memory bus), and are on the critical path in hybrid Host-PIM executions. Thus, hardware architects and system engineers can explore mechanisms for PIM systems that (i) overlap data transfers to/from PIM memory with computation on PIM cores, and (ii) minimize the zero padding amount needed in parallel Host-PIM/PIM-Host data transfers.

4.4 Across PIM Cluster Analysis

We evaluate SpMM using multiple PIM clusters and within cluster we select edge-balance schemes to minimize kernel time. Fig. 10 presents the performance using real-world graphs, 128 hidden size, 32 PIM devices, and a *fixed* number of 1992 PIM cores, while varying the parallelization scheme used: each triple of values shows the number of sparse partitions, the number of dense partitions and the number of PIM clusters per PIM device, respectively. We show breakdown steps of Fig. 4, and the stacked bar "Other" corresponds to the time needed to partition the dense matrix.

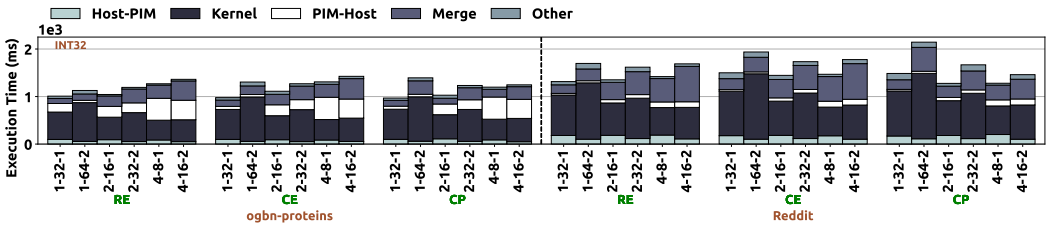

Fig. 10. Performance of edge-balance schemes varying the number of sparse, the number of dense partitions, and the number ofPIM clusters per device.

We note four key points. First, having 2 PIM clusters per device with ~28 cores per cluster increases the kernel time by 1.30× on average over having 1 cluster per device of ~56 cores. Using a smaller number of cores per cluster results in higher compute costs, since *each* PIM core processes a larger number of edges (non-zeros), executing many more computations. Second, creating a larger number of sparse partitions (e.g., 4 or larger) typically increases the PIM-Host data transfer and merge overheads, since PIM clusters create more partial results for the output matrix. Third, our proposed PaF strategy effectively provides low data transfer costs to/from PIM memory modules. We observe that in the best-performing configurations (e.g., 1-32-1 for ogbn-proteins dataset), the Host-PIM and PIM-Host data transfers account for ~14% of the total time, while most of the time (~67%) is the actual sparse matrix matrix multiplication. Fourth, we identify two patterns: in ogbn-proteins, best performance is achieved using CP with 1 sparse partition, while in Reddit, best performance is achieved using CP with 2 sparse partitions (1.11× better over having 1 sparse partition). In ogbn-proteins, there is a high disparity in the number of vertices assigned to PIM cores, causing a large amount of zero padding in PIM-Host transfers. When increasing the sparse partitions from 1 to 2, the vertex disparity and amount of zero padding increase, thus incurring worse performance. Thus, we find the graph's characteristics affect the best-performing parallelization strategy. Our analysis shows that tuning mechanisms and ML compilers need to be developed to optimize performance of sparse workloads in PIM systems based on the characteristics of each particular input.

Recommendation 5:
System performance of sparse workloads in PIM systems highly depends on the particular patterns of each input given. Therefore, software and system engineers can deploy intelligent heuristics, prediction and automation tools (similar to our proposed tuner) that tune the optimization strategies of sparse workloads at each particular given input at low cost, such that to provide high system performance on real PIM systems.

Scalability of PyGim Implementations. Fig. 11 presents the scalability of PyGim's edge-balance schemes, i.e., RE, CE and CP, using int32 data type in SpMM. In these experiments, we have 1 sparse partition, 128 hidden size and 2 PIM clusters per PIM device, and increase the number of PIM devices: we evaluate 8, 16, and 32 PIM devices, i.e., the number of PIM cores increases from 456 up to 1992 (each PIM device has ~56 PIM cores). We find that all PyGim's edge-balance schemes scale well: when we double the number of PIM devices used (double the PIM cores used), the kernel time and total performance improve by on average 1.47× and 1.38×, respectively. Thus, we conclude that PyGim is a scalable GNN library for real PIM systems with a very large number of PIM cores and PIM devices.

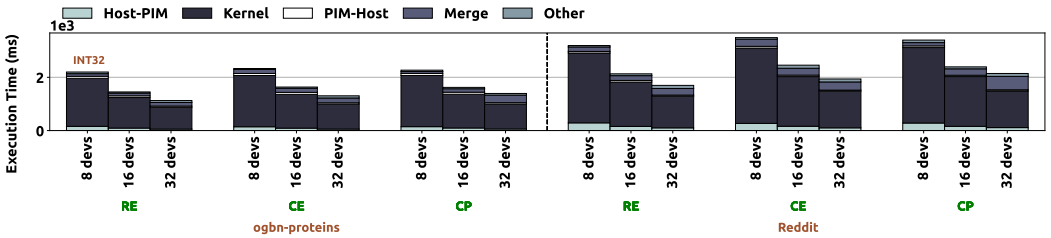

Fig. 11. Scalability of edge-balance schemes, as the number of PIM devices (PIM cores) increases.

4.5 PyGim Tuner Efficiency

Fig. 12 evaluates the PyGim tuner efficiency for CSR (See Fig. 16 in Appendix for COO) by comparing the performance slowdown achieved by its predicted aggregation configuration (predicted) versus an oracle prediction using various datasets and hidden sizes. For the oracle prediction performance, we exhaustively collect the execution times of all possible configurations, and we present in Fig. 12 the best-performing execution time among them (oracle). The predicted aggregation configuration by the tuner achieves similar performance with the oracle configuration, being only 0.72% and 1% worse on average across all datasets and hidden sizes for CSR and COO, respectively. Thus, PyGim tuner effectively tunes the aggregation configuration in GNN executions, eliminating the programmer's intervention and providing high performance.

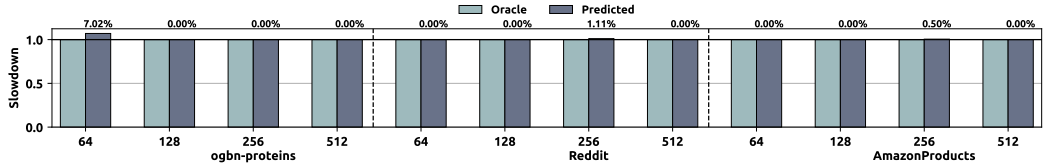

Fig. 12. Slowdown of the predicted CSR aggregation configuration by tuner over oracle prediction.

4.6 GNN Aggregation Performance

Fig. 13 shows the performance of all comparison points described in §4.1, in one aggregation operator using real-world graph datasets and common hidden sizes in feature matrix (x-axis). In PIM executions, we use 32 PIM devices (~56 cores per cluster). In PyGim, we evaluate both CSR and COO schemes and we enable the tuner to set the aggregation configuration. Please see Appendix A.2 (Fig. 17) for energy consumption evaluation in GNN aggregation.

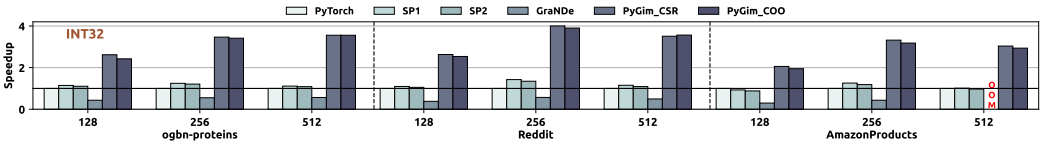

Fig. 13. Performance of all comparison points in one aggregation, using various graphs and hidden sizes.

We draw fourth findings. First, GraNDe's [141] optimized scheme for simulated near-rank PIM systems achieves very low performance in real near-bank PIM systems, being up to 0.57× of PyTorch implementation. PyGim provides significant performance benefits over GraNDe, because the GraNDe's parallelization strategy is tailored for near-rank PIM systems, rather than real near-bank PIM systems. Second, SP1 and SP2 achieve small speedups over PyTorch (on average 1.13×), since they are optimized for SpMV kernel. Instead, aggregation by its nature performs SpMM, thus PyGim schemes provide significant performance speedups, on average 3.09× and up to 4.00× over PyTorch implementation. PyGim outperforms prior PIM-based schemes (SP1, SP2, GraNDe) by on average 4.10× and up to 7.70×. Third, we find that the best-performing PyGim scheme selected by the tuner varies across datasets due to different connection characteristics between vertices of the graph. Fourth, in Fig. 17 (Appendix A.2), we show that PyGim provides higher energy efficiency by on average 4.08× and 1.39× over prior PIM-based schemes (SP1, SP2, GraNDe) and PyTorch, respectively. Overall, PyGim provides high performance and energy efficiency benefits in GNN aggregation, significantly outperforming prior existing schemes across various graph datasets and hidden sizes.

4.7 End-to-End GNN Inference

Performance. Fig. 14 evaluates all comparison points in GNN inference using int32 data type and various graph datasets. We evaluate 3 GNN models, each model has 3 layers of 256 hidden size. In PIM executions, we use 32 PIM devices, having in total 1992 cores. In PyGim, we evaluate both CSR and COO schemes and enable the tuner to set the aggregation configuration. All comparison points of Fig. 14 produce correct output data values and provide the same accuracy, as presented in Appendix A.4. Note that PyGim can be also used to execute GNN training. Please also see Appendix A.5 for GNN training results.

From Fig. 14, we find that PyGim schemes provide significant performance speedups over PyTorch running on Host by 3.04× (up to 3.44×). PyGim outperforms prior state-of-the-art PIM schemes, being 2.46× (up to 2.68×) better compared to SP1 and SP2, and 6.3× (up to 7.2×) better over GraNDe. Note that the UPMEM PIM core does not include a complete 32 × 32-bit multiplier to efficiently support int32 data type in hardware: multiplications of 32-bit operands are implemented using bit shifting and addition and take ~32 cycles. Thus, in Appendix A.3, we also evaluate end-to-end GNN inference using int8 and int16 data types, in which the arithmetic operations are natively supported by PIM hardware, as well as using the fp32 data type, in which the arithmetic operations are software emulated. PIM GNN execution achieves low performance with fp32 values, since their arithmetic operations are software emulated in the UPMEM PIM hardware. However, ML-oriented

PIM systems [4, 74] are expected to be in the market, and natively support higher precision data types. Instead, Figs. 19, 20 in Appendix evaluate int8 and int16 values (their multiplications are natively supported by PIM hardware), and show that PyGim provides superior speedups: PyGim outperforms prior state-of-the-art PIM approaches by 3.59× (up to 9.89×) and 3.60× (up to 8.90×) for int8 and int16, respectively, and outperforms PyTorch running on Host by on average 4.49× (up to 5.54×) and 4.03× (up to 4.63×) for int8 and int16, respectively. We conclude that PyGim provides significant performance benefits in GNN inference over prior approaches.

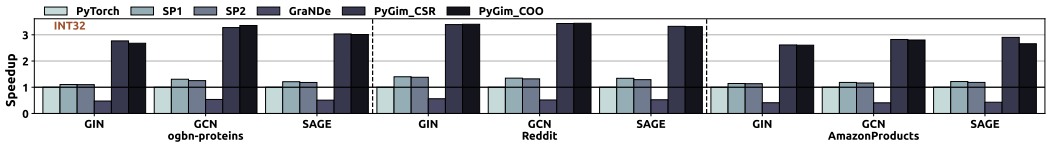

Fig. 14. Performance of all comparison points in GNN inference, using various graph datasets and models.

Energy Consumption. Fig. 15 presents the energy consumption (in Joules) of all comparison points in end-to-end GNN inference using int32 data type, and various GNN models and datasets. In PIM executions, we use 32 PIM devices, having in total 1992 cores, and PyGim's tuner is enabled. We use Intel RAPL [65] to measure energy in CPU execution parts, which are (i) the whole PyTorch scheme, and (ii) in PIM schemes, the combination operator as well as the load, retrieve, and merge steps of the aggregation operator. For the kernel step of aggregation, we measure the energy consumed in PIM-enabled chips using the methodology described in a recent paper [28] written by the UPMEM PIM manufacturer: the power of each UPMEM PIM DIMM is 23.22W, thus the total energy of kernel time is conservatively calculated as the $kernel\_time \times \#PIM\_DIMMs \times power$.

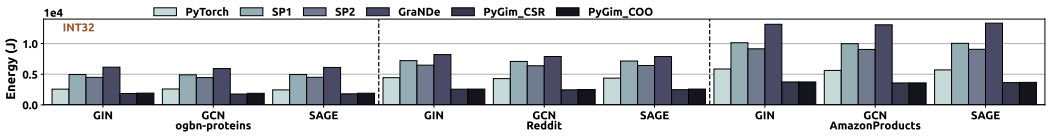

Fig. 15. Energy consumption of all comparison points in the end-to-end inference, using various models.

We draw three findings. First, PyGim provides significant energy benefits on average 2.86× (up to 3.68×) over prior PIM schemes. Second, PyGim improves energy efficiency over PyTorch scheme by on average 1.55× (up to 1.75×). Although PyGim provides 3.04× better performance over PyTorch, it provides 1.55× better energy efficiency, because the manufacturing process of PIM chips is still in an early stage. For example, UPMEM PIM chips have been manufactured with a larger technology node, i.e., at least 20nm, than the 14nm technology node used for CPU hardware (See Table 1). Future real PIM products could advance the technology node to become more energy-efficient. Third, in aggregation, SP1 and SP2 schemes have a time-consuming kernel step executed on PIM cores. GraNDe has a more time-consuming merge step executed on Host. Thus, although SP1 and SP2 improve performance over GraNDe by on average 2.56×, they are on average 1.25× better than than GraNDe in energy efficiency.

4.8 Evaluation of GNN Aggregation in GPU Systems

We propose a GNN library for real near-bank PIM systems, and evaluate it over prior software parallelization schemes/libraries running on the same computing system, the UPMEM PIM server. To judge and compare different libraries tailored for different types of computing systems, we present the resource utilization as a representative metric: resource utilization measures how well the software maps and uses the available capabilities of the underlying hardware. With this metric, we compare how well PyGim library maps to the evaluated PIM system versus how well the PyTorch's

backend libraries map to GPU systems. Resource utilization is defined as the number of operations performed divided by the execution time, and normalized as a percentage of the theoretical peak performance of the system. We focus our evaluations in GNN aggregation operator, because GNN combination uses implementations from existing ML frameworks (e.g., PyTorch) executed on Host side, and thus characterizing the efficiency of existing implementations in various Host systems is out of the scope of our work. In GNN aggregation, for fairness among all implementations, we calculate the number of operations performed as *edges* × *hidden_size*, which is the theoretical arithmetic operations of SpMM. Table 1 shows the characteristics of various computing systems. For peak performance and memory bandwidth, we use peakperf [106] and stream [123] in CPUs/GPUs, and the microbenchmarks from open-source works [39, 48] for UPMEM PIM system.

System	Total Cores	Freq.	INT32 Peak Performance	FP32 Peak Performance	Memory Capacity	Total Bandwidth	Technology Node
CPU Intel Xeon 4215	2x8 x86 cores	2.5 GHz	0.64 TOPS	1.28 TFLOPS	128 GB	23.1 GB/s	14nm
UPMEM PIM	1992 PIM cores	350 MHz	115.93 GOPS	24.85 GFLOPS	124.5 GB	1.39 TB/s	at least 20nm
GPU GTX 1080 Ti	3584 CUDA cores	1.48 GHz	13.25 TOPS	13.25 TFLOPS	11 GB	359.9 GB/s	16nm
GPU RTX 2080 Ti	4352 CUDA cores	1.35 GHz	16.94 TOPS	16.94 TFLOPS	11 GB	558.1 GB/s	12nm
GPU RTX 3090	10496 CUDA cores	1.40 GHz	17.79 TOPS	35.58 TFLOPS	24 GB	936.2 GB/s	8nm

Table 1. Characteristics of CPU, PIM and GPU systems.

Table 2 shows the hardware utilization achieved by different libraries executed on the corresponding computing system in one aggregation operator using int32 and fp32 data types, various datasets, and with 256 hidden size. For CPU and GPU systems, we evaluate PyTorch's pytorch_sparse library [32], which employs SpMM implementations from Intel MKL library for CPUs, and optimized CUDA implementations for GPUs. For the UPMEM PIM system, we evaluate PyGim and account for all breakdown steps of Fig. 4.

Dataset and data type / Software library	OGBN INT32	RDT INT32	AMZ INT32	OGBN FP32	RDT FP32	AMZ FP32
pytorch_sparse - Intel MKL (CPU Intel Xeon 4215)	0.74%	0.63%	0.67%	0.26%	0.22%	0.20%
pytorch_sparse - CUDA (GPU GTX 1080 Ti)	2.15%	0.62%	0.71%	2.02%	0.62%	0.71%
pytorch_sparse - CUDA (GPU RTX 2080 Ti)	1.45%	0.68%	0.71%	1.45%	0.67%	0.71%
pytorch_sparse - CUDA (GPU RTX 3090)	3.03%	1.56%	1.32%	1.58%	0.78%	0.67%
PyGim (UPMEM PIM)	**14.09%**	**13.86%**	**12.32%**	**8.21%**	**9.13%**	**8.84%**

Table 2. Resource utilization in various systems for GNN aggregation with 256 hidden size, the ogbn-proteins (**OGBN**), Reddit (**RDT**), and AmazonProducts (**AMZ**) datasets, and INT32 and FP32 data types.

We make two observations. First, PyGim achieves 12.9×, 13.2×, and 8.8× larger utilization on the UPMEM PIM system than that of the pytorch_sparse CUDA library on GTX 1080 Ti, RTX 2080 Ti and RTX 3090 GPUs, respectively, and provides 29.4× larger utilization on the UPMEM PIM than that of pytorch_sparse Intel MKL library on Intel Xeon CPU. Thus, PyGim uses the PIM system more effectively than PyTorch's optimized backend libraries use the CPU and GPU systems. Second, across three GPU generations (Table 1), GPU architects advance the technology node, and increase the number of cores and the available memory bandwidth. However, the resource utilization in the memory-intensive GNN aggregation still remains low in all GPUs. Comparing RTX 2080 Ti over GTX 1080 Ti, both the compute throughput and the memory bandwidth increased by 1.30× and 1.55×, respectively, however resource utilization in aggregation is similar: on average 0.95% for RTX 2080 Ti and 1.13% for GTX 1080 Ti. Comparing RTX 3090 over RTX 2080 Ti, the fp32 compute throughput and memory bandwidth increased by 2.1× and 1.68×, respectively, however resource utilization in fp32 aggregation remains similarly low: on average 1.01% for RTX 3090 and 0.94% for

RTX 2080 Ti. Resource utilization in int32 aggregation is ~2× compared that in fp32 aggregation in RTX 3090, since the int32 compute throughput from RTX 2080 Ti to RTX 3090 is similar (1.05×), while memory bandwidth increases by 1.68×. These observations show that GPU architectures are not necessarily evolving to better support memory-intensive workloads, such as GNN aggregation. Overall, we conclude that PyGim running GNN aggregation on real PIM systems provides a more cost-effective solution than PyTorch running GNN on Host systems.

Finally, we also report the performance (seconds) and energy consumption (Joules) metrics to show the readers how much absolute performance and energy efficiency the evaluated UPMEM PIM system achieves on GNN aggregation over commodity GPU systems. UPMEM PIM with PyGim library is worse than GPUs with pytorch_sparse library by 9.9×, 10.5× and 25.8× in performance and by 5.0×, 5.3× and 11.6× energy consumption over 1080Ti, 2080Ti, and 3090 GPUs, respectively, for int32 data type. In int8 data type (natively supported in UPMEM hardware), UPMEM PIM with PyGim provides better performance and energy efficiency compared to that provided in int32 data type: e.g., in Reddit data set with 256 hidden size, it is 2.5×, 4.0×, 9.5× worse in performance and 2.4×, 2.9×, 5.6× worse in energy consumption than 1080Ti, 2080Ti, and 3090 GPU with pytorch_sparse library, respectively[2].

In Appendix A.6, we present the detailed evaluation results over GPUs in GNN aggregation and end-to-end GNN inference. Please note that these results are provided for completeness and *not* for competition purposes. Directly comparing this UPMEM PIM system over GPU systems is not a fair comparison. The evaluated UPMEM PIM system is available on the market only ~1.5 years, it is in its first generation manufactured with a large technology node of at least 20nm, and is not yet well optimized for multiplication operations [47, 55]. Instead, GPU systems have been optimized for 15+ years, especially in multiplication operations, and there has been invested large financial budgets from major technology industry leaders to improve GPU architecture across its generations. Moreover, although our *goal* in this work is to quantify the potential of a *real* PIM system in GNN executions, our proposed PaF optimizations cover near-bank PIM systems (See the described characteristics in §2.2), and thus could be evaluated on other near-bank PIM systems with potentially better computation capabilities and energy efficiency than the evaluated UPMEM PIM system. UPMEM [130] has already announced a second generation product (expected to be released in the market), where PIM-enabled DIMMs are integrated with a more powerful CPU server (Ice Lake platform instead of Intel Xeon), the system will support 28 PIM DIMMs with ~1.8× more cores, i.e., 3584 cores (instead of 16 PIM DIMMs with 2K cores) and each PIM core will have ~1.7× higher frequency, i.e., 600MHz core frequency (instead of 350MHz). The current evaluated UPMEM PIM also uses a relatively large technology node of **at least 20nm** for manufacturing (GPU RTX2080 Ti uses **12nm**), and advancing the technology node could improve energy efficiency. Additionally, according to [80], the near-bank HBM-PIM and AiM (both have been prototyped) systems can achieve 1.2 TFLOPS and 1 TFLOPS compute throughput, which is 48.3× and 40.2× higher than that of the evaluated UPMEM PIM system. Hence, running GNN aggregation with our proposed PaF optimizations in upcoming real PIM systems could potentially lead to better performance and energy efficiency compared to the results reported in the above GPU comparisons. Finally, HBM-based PIM memory modules [50, 74, 75] can be integrated into modern GPUs and provide much larger memory bandwidth for the PIM cores than the memory bandwidth available for GPU cores. In such HBM-based PIM systems, when executing GNN aggregation in PIM cores, we could potentially expect important performance and energy benefits compared to executing GNN aggregation on the GPU cores.

[2]In same aggregation configuration, PyGim on UPMEM PIM improves performance and energy efficiency by 2.3× and 1.3× respectively, over an Intel Xeon 4314 CPU (10nm technology) with 2x16 cores (2.4GHz) and 227 GB/s memory bandwidth.

5 Related Work

To our knowledge, our work is the first to design an ML library and tuner for GNN executions on near-bank PIM systems, propose efficient GNN aggregation schemes tailored for such systems, and extensively characterize GNNs on the first real-world PIM system. We briefly discuss prior work.

PIM-Based Accelerators. A few works [13, 128, 141, 148] design PIM-based GNN accelerators. Their custom microarchitecture designs for host and PIM cores are orthogonal to PyGim software library. These works target *near-rank* PIM architectures and use simulators for their evaluations. PyGim targets *near-bank* PIM systems, which typically provide much larger memory bandwidth than near-rank PIMs [75], and evaluates GNNs on a real system. Finally, implementing the software data mappings of these works to near-bank PIM systems would cause out-of-memory errors (See § 2.3) or would be inefficient: as shown in our evaluations, PyGim significantly outperforms the best-performing strategy of GraNDe [141]. Li et al. [79] design a tool to explore PIM design configurations (e.g., near-DIMM, near-rank, near-bank) for different application scenarios, which is orthogonal to our work. A few works [76, 78, 104] propose in-storage PIM designs that sample the large graphs inside the disk (SSDs) to reduce the amount of graph data sent from disk to host CPU/GPU cores. These works can work synergistically with ours: PyGim can be used to efficiently process the produced smaller graph in DRAM. Finally, a few works [51, 80, 81, 103] design custom PIM-based architectures for large language models, however they do not support GNNs and the SpMM kernel.

System Support and Software for PIM Systems. Prior works [14, 23, 26, 38, 39, 47, 48, 57, 60, 84, 112, 116] design optimized linear algebra, graph processing, database, array iterators, ML training, bioinformatics, and image processing kernels for PIM systems. The closest work to ours is SparseP [39], that is an efficient SpMV library for PIM systems. We use the best-performing SparseP kernel in GNNs, show that it has worse performance and energy efficiency than PyGim. A few works [101, 127] propose efficient communication collectives for future PIM systems. Moreover, a few works [55, 61] discuss scalability issues and hardware limitations of real near-bank PIM systems, and propose architectural features for future PIM systems. These prior works [55, 61, 101, 127] are orthogonal to PyGim: PyGim can be used in their proposed future PIM designs and/or leverage optimized communication collectives to enable high efficiency in GNN executions.

GNNs and SpMM in Commodity Systems. Prior works optimize GNNs and SpMM on CPUs [3, 43, 52, 68, 131, 133, 139], GPUs [22, 33, 54, 68, 88, 99, 100, 136, 138] by leveraging the shared memory model of CPUs/GPUs and deep cache hierarchies (on-chip caches). Their optimizations cannot be applied in PIM systems, that have a distributed memory model and shallow cache hierarchy. Prior works [8, 11, 59, 69, 85, 87, 91, 92, 111, 113, 129, 147] optimize GNNs and SpMM on CPU-GPU, multi-CPU/multi-GPU systems by minimizing communication costs among cores, and/or overlapping computation with communication. However, real PIM systems may not support direct communication among PIM cores [61], and there are no real PIM systems that can overlap computation with communication across on PIM cores. Thus, well-tuned GNN and SpMM kernels for distributed processor-centric systems either cannot be directly applied in PIM systems, or their fine-grained inter-PIM-core communication (e.g., implemented over Host) would cause high performance overheads.

Custom Accelerators for GNNs and SpMM. Prior works [5, 6, 36, 62, 66, 82, 83, 102, 117–120] propose custom hardware accelerators for GNNs and SpMM, but they target processor-centric systems with low available memory bandwidth. Instead, PyGim provides software-level optimizations for GNNs, and targets memory-centric PIM systems.

6 Conclusion

We propose PyGim, an efficient ML library for GNNs executions in PIM systems, and conduct a comprehensive characterization study of GNNs on a real-world PIM system. We design a hybrid GNN execution on processor- and memory-centric computing systems, intelligent parallelization techniques for GNN aggregation in near-bank PIM systems and a lightweight tuner to enable programming ease in GNN deployment for PIM systems. In GNN inference, PyGim achieves 3.04× (up to 3.44×) and 4.38× (up to 7.20×) speedup over the state-of-the-art PyTorch and PIM-based schemes, respectively, and 1.55× (up to 1.75×) and 2.86× (up to 3.68×) higher energy efficiency than PyTorch and PIM-based schemes, respectively. In GNN aggregation, PyGim provides on average 11.6× higher resource utilization in PIM system than that of the PyTorch CUDA library in GPUs. We hope that our parallelization strategies for GNNs, in-depth PIM analysis, and open-source library will enable further research on optimizing GNNs and other sparse ML models in memory-centric computing systems.

Acknowledgments

We thank UPMEM for generously providing hardware resources to perform this research. We thank the anonymous reviewers of SIGMETRICS 2025, and our shepherd, Bo Jiang, for their comments and suggestions. We thank Andreas Moshovos for valuable feedback and Bojian Zheng for technical support. We thank the SAFARI Research Group and EcoSystem members for providing a stimulating intellectual environment. Ivan Fernandez is partially supported by the Spanish Ministry of Science and Innovation (projects PID2019-107255GB-C21 and PID2019-107255GB-C22). We acknowledge the generous gifts from our industrial partners, including Google, Huawei, Intel, Microsoft and AWS. This work is supported in part by the Semiconductor Research Corporation (SRC), the ETH Future Computing Laboratory (EFCL), the European Union's Horizon program for research and innovation [101047160 - BioPIM], and the AI Chip Center for Emerging Smart Systems, sponsored by InnoHK funding, Hong Kong SAR (ACCESS). This paper is also supported in part by Vector Institute Research grants, the Canada Foundation for Innovation JELF grant, NSERC Discovery grant, AWS Machine Learning Research Award, Facebook Faculty Research Award, Google Scholar Research Award, and VMware Early Career Faculty Grant. The PyGim library is publicly available at https://github.com/CMU-SAFARI/PyGim.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. 2016. Tensorflow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *arXiv*, 2016.

[2] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi 2015. A Scalable Processing-In-Memory Accelerator for Parallel Graph Processing, In *ISCA*.

[3] Kadir Akbudak and Cevdet Aykanat 2017. Exploiting Locality in Sparse Matrix-Matrix Multiplication on Many-Core Architectures. *TPDS*, 2017.

[4] Hadi Asghari-Moghaddam, Young Hoon Son, Jung Ho Ahn, and Nam Sung Kim 2016. Chameleon: Versatile and Practical Near-DRAM Acceleration Architecture for Large Memory Systems, In *MICRO*.

[5] Adam Auten, Matthew Tomei, and Rakesh Kumar 2020. Hardware Acceleration of Graph Neural Networks, In *DAC*.

[6] Daehyeon Baek, Soojin Hwang, Taekyung Heo, Daehoon Kim, and Jaehyuk Huh 2021. InnerSP: A Memory Efficient Sparse Matrix Multiplication Accelerator With Locality-Aware Inner Product Processing, In *PACT*.

[7] Riyadh Baghdadi, Massinissa Merouani, Mohamed-Hicham Leghettas, Kamel Abdous, Taha Arbaoui, Karima Benatchba, et al. 2021. A Deep Learning Based Cost Model for Automatic Code Optimization. *MLSys*, 2021.

[8] V. Bharadwaj, A. Buluc, and J. Demmel 2022. Distributed-Memory Sparse Kernels for Machine Learning, In *IPDPS*.

[9] Christopher M Bishop 1995. *Neural Networks for Pattern Recognition*. Oxford University Press.

[10] Åke Björck 1996. Numerical Methods for Least Squares Problems, In *SIAM*.

[11] Charles Block, Gerasimos Gerogiannis, Charith Mendis, Ariful Azad, and Josep Torrellas 2024. Two-Face: Combining Collective and One-Sided Communication for Efficient Distributed SpMM, In *ASPLOS*.

[12] Amirali Boroumand, Saugata Ghose, Youngsok Kim, Rachata Ausavarungnirun, Eric Shiu, Rahul Thakur, Daehyun Kim, Aki Kuusela, Allan Knies, Parthasarathy Ranganathan, and Onur Mutlu 2018. Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks, In *ASPLOS*.

[13] Dan Chen, Haiheng He, Hai Jin, Long Zheng, Yu Huang, Xinyang Shen, and Xiaofei Liao 2023. MetaNMP: Leveraging Cartesian-Like Product to Accelerate HGNNs with Near-Memory Processing, In *ISCA*.

[14] Jinfan Chen, Juan Gómez-Luna, Izzat El Hajj, Yuxin Guo, and Onur Mutlu 2023. SimplePIM: A Software Framework for Productive and Efficient Processing-in-Memory, In *PACT*.

[15] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang 2015. Mxnet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *arXiv*, 2015.

[16] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy 2018. Learning to Optimize Tensor Programs. *NIPS*, 2018.

[17] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh 2019. Cluster-GCN: An Efficient Algorithm for Training Deep and Large Graph Convolutional Networks, In *SIGKDD*.

[18] Benjamin Y. Cho, Yongkee Kwon, Sangkug Lym, and Mattan Erez 2020. Near Data Acceleration with Concurrent Host Access, In *ISCA*.

[19] Jiwon Choe, Amy Huang, Tali Moreshet, Maurice Herlihy, and R. Iris Bahar 2019. Concurrent Data Structures with Near-Data-Processing: An Architecture-Aware Implementation, In *SPAA*.

[20] Ctranslate2 2023. Ctranslate2. https://github.com/OpenNMT/CTranslate2

[21] Leonardo Dagum and Ramesh Menon 1998. OpenMP: An Industry-Standard API for Shared-Memory Programming, In *IEEE Comput. Sci. Eng.*

[22] Steven Dalton, Luke Olson, and Nathan Bell 2015. Optimizing Sparse Matrix—Matrix Multiplication for the GPU. *ACM Trans. Math. Softw.*, 2015.

[23] Prangon Das, Purab Ranjan Sutradhar, Mark Indovina, Sai Manoj Pudukotai Dinakarrao, and Amlan Ganguly 2022. Implementation and Evaluation of Deep Neural Networks in Commercially Available Processing in Memory Hardware, In *SOCC*.

[24] Timothy A. Davis and Yifan Hu 2011. The University of Florida Sparse Matrix Collection, In *TOMS*.

[25] F. Devaux 2019. The True Processing In Memory Accelerator, In *Hot Chips*.

[26] Safaa Diab, Amir Nassereldine, Mohammed Alser, Juan Gómez Luna, Onur Mutlu, and Izzat El Hajj 2023. A Framework for High-Throughput Sequence Alignment Using Real Processing-in-Memory Systems. *Bioinformatics*, 2023.

[27] Mario Drumond, Alexandros Daglis, Nooshin Mirzadeh, Dmitrii Ustiugov, Javier Picorel, Babak Falsafi, Boris Grot, and Dionisios Pnevmatikatos 2017. The Mondrian Data Engine, In *ISCA*.

[28] Yann Falevoz and Julien Legriel 2023. Energy Efficiency Impact of Processing in Memory: A Comprehensive Review of Workloads on the UPMEM Architecture, In *Euro-PAR*. Springer.

[29] Wenqi Fan, Yao Ma, Qing Li, Yuan He, Eric Zhao, Jiliang Tang, and Dawei Yin 2019. Graph Neural Networks for Social Recommendation, In *The World Wide Web Conference*.

[30] Ivan Fernandez, Christina Giannoula, Aditya Manglik, Ricardo Quislant, Nika Mansouri Ghiasi, Juan Gómez-Luna, Eladio Gutierrez, Oscar Plata, and Onur Mutlu 2024. MATSA: An MRAM-Based Energy-Efficient Accelerator for Time Series Analysis. *IEEE Access*, 2024.

[31] Ivan Fernandez, Ricardo Quislant, Christina Giannoula, Mohammed Alser, Juan Gómez-Luna, Eladio Gutiérrez, Oscar Plata, and Onur Mutlu 2020. NATSA: A Near-Data Processing Accelerator for Time Series Analysis, In *ICCD*.

[32] Matthias Fey and Jan E. Lenssen 2019. Fast Graph Representation Learning with PyTorch Geometric, In *ICLR*.

[33] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen [n. d.]. Sparse GPU Kernels for Deep Learning, In *SC*.

[34] Mingyu Gao, Grant Ayers, and Christos Kozyrakis 2015. Practical Near-Data Processing for In-Memory Analytics Frameworks, In *PACT*.

[35] X Yu Geoffrey, Yubo Gao, Pavel Golikov, and Gennady Pekhimenko 2021. Habitat: A Runtime-Based Computational Performance Predictor for Deep Neural Network Training, In *ATC*.

[36] Gerasimos Gerogiannis, Serif Yesil, Damitha Lenadora, Dingyuan Cao, Charith Mendis, and Josep Torrellas 2023. SPADE: A Flexible and Scalable Accelerator for SpMM and SDDMM, In *ISCA*.

[37] Saugata Ghose, Amirali Boroumand, Jeremie Kim, Juan Gómez-Luna, and Onur Mutlu 2019. Processing-in-Memory: A Workload-Driven Perspective, In *IBM JRD*.

[38] Christina Giannoula, Ivan Fernandez, Juan Gómez-Luna, Nectarios Koziris, Georgios Goumas, and Onur Mutlu 2022. Towards Efficient Sparse Matrix Vector Multiplication on Real Processing-In-Memory Architectures, In *SIGMETRICS*.

[39] Christina Giannoula, Ivan Fernandez, Juan Gómez Luna, Nectarios Koziris, Georgios Goumas, and Onur Mutlu 2022. SparseP: Towards Efficient Sparse Matrix Vector Multiplication on Real Processing-in-Memory Architectures. *POMACS*, 2022.

[40] Christina Giannoula, Nandita Vijaykumar, Nikela Papadopoulou, Vasileios Karakostas, Ivan Fernandez, Juan Gómez-Luna, Lois Orosa, Nectarios Koziris, Georgios Goumas, and Onur Mutlu 2021. SynCron: Efficient Synchronization Support for Near-Data-Processing Architectures, In *HPCA*.

[41] Maya Gokhale, Scott Lloyd, and Chris Hajas 2015. Near Memory Data Structure Rearrangement, In *MEMSYS*.

[42] Juan Gómez-Luna, Izzat El Hajj, Ivan Fernandez, Christina Giannoula, Geraldo F. Oliveira, and Onur Mutlu 2021. Benchmarking a New Paradigm: An Experimental Analysis of a Real Processing-in-Memory Architecture, In *CoRR*. https://arxiv.org/abs/2105.03814

[43] Zhangxiaowen Gong, Houxiang Ji, Yao Yao, Christopher W. Fletcher, Christopher J. Hughes, and Josep Torrellas 2022. Graphite: Optimizing Graph Neural Networks on CPUs through Cooperative Software-Hardware Techniques, In *ISCA*.

[44] SAFARI Research Group 2022. PyGim Software Package. https://github.com/CMU-SAFARI/PyGim

[45] Zhixiang Gu, Jose Moreira, David Edelsohn, and Ariful Azad 2020. Bandwidth Optimized Parallel Algorithms for Sparse Matrix-Matrix Multiplication Using Propagation Blocking, In *SPAA*.

[46] Antonio Gulli and Sujit Pal 2017. *Deep Learning with Keras*. Packt Publishing Ltd.

[47] Juan Gómez-Luna, Yuxin Guo, Sylvan Brocard, Julien Legriel, Remy Cimadomo, Geraldo F. Oliveira, Gagandeep Singh, and Onur Mutlu 2023. Evaluating Machine Learning Workloads on Memory-Centric Computing Systems, In *ISPASS*.

[48] Juan Gómez-Luna, Izzat El Hajj, Ivan Fernandez, Christina Giannoula, Geraldo F. Oliveira, and Onur Mutlu 2022. Benchmarking a New Paradigm: Experimental Analysis and Characterization of a Real Processing-in-Memory System. *IEEE Access*, 2022.

[49] Will Hamilton, Zhitao Ying, and Jure Leskovec 2017. Inductive Representation Learning on Large Graphs. *NIPS* 30, 2017.

[50] Mingxuan He, Choungki Song, Ilkon Kim, Chunseok Jeong, Seho Kim, Il Park, Mithuna Thottethodi, and T. N. Vijaykumar 2020. Newton: A DRAM-Maker's Accelerator-in-Memory (AiM) Architecture for Machine Learning, In *MICRO*.

[51] Guseul Heo, Sangyeop Lee, Jaehong Cho, Hyunmin Choi, Sanghyeon Lee, Hyungkyu Ham, Gwangsun Kim, Divya Mahajan, and Jongse Park 2024. NeuPIMs: NPU-PIM Heterogeneous Acceleration for Batched LLM Inferencing, In *ASPLOS*.

[52] Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and P. Sadayappan 2019. Adaptive Sparse Tiling for Sparse Matrix Multiplication, In *PpopP*.

[53] Kevin Hsieh, Samira Khan, Nandita Vijaykumar, Kevin Chang, Amirali Boroumand, Saugata Ghose, and Onur Mutlu 2016. Accelerating Pointer Chasing in 3D-stacked Memory: Challenges, Mechanisms, Evaluation, In *ICCD*.

[54] Kezhao Huang, Jidong Zhai, Zhen Zheng, Youngmin Yi, and Xipeng Shen 2021. Understanding and Bridging the Gaps in Current GNN Performance Optimizations, In *PpopP*.

[55] Bongjoon Hyun, Taehun Kim, Dongjae Lee, and Minsoo Rhu 2024. Pathfinding Future PIM Architectures by Demystifying a Commercial PIM Technology, In *HPCA*.

[56] Eun-Jin Im and Katherine A. Yelick 1999. Optimizing Sparse Matrix Vector Multiplication on SMP, In *PPSC*.

[57] Maurus Item, Geraldo F. Oliveira, Juan Gómez-Luna, Mohammad Sadrosadati, Yuxin Guo, and Onur Mutlu 2023. TransPimLib: Efficient Transcendental Functions for Processing-in-Memory Systems, In *ISPASS*.

[58] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell 2014. Caffe: Convolutional Architecture for Fast Feature Embedding, In *Proceedings of the 22nd ACM International Conference on Multimedia*.

[59] Zhihao Jia, Sina Lin, Mingyu Gao, Matei A. Zaharia, and Alexander Aiken 2020. Improving the Accuracy, Scalability, and Performance of Graph Neural Networks with Roc, In *MLSys*.

[60] Muhammad Attahir Jibril, Hani Al-Sayeh, and Kai-Uwe Sattler 2024. Accelerating Aggregation Using a Real Processing-in-Memory System, In *ICDE*.

[61] Gilbert Jonatan, Haeyoon Cho, Hyojun Son, Xiangyu Wu, Neal Livesay, Evelio Mora, Kaustubh Shivdikar, José L. Abellán, Ajay Joshi, David Kaeli, and John Kim 2024. Scalability Limitations of Processing-in-Memory Using Real System Evaluations. *Proc. ACM Meas. Anal. Comput. Syst.*, 2024.

[62] Konstantinos Kanellopoulos, Nandita Vijaykumar, Christina Giannoula, Roknoddin Azizi, Skanda Koppula, Nika Mansouri Ghiasi, Taha Shahroodi, Juan Gomez Luna, and Onur Mutlu 2019. Smash: Co-Designing Software Compression and Hardware-Accelerated Indexing for Efficient Sparse Matrix Operations, In *MICRO*.

[63] Sam Kaufman, Phitchaya Phothilimthana, Yanqi Zhou, Charith Mendis, Sudip Roy, Amit Sabne, and Mike Burrows 2021. A Learned Performance Model for Tensor Processing Units. *MLSys*, 2021.

[64] Liu Ke, Udit Gupta, Carole-Jean Wu, Benjamin Youngjae Cho, Mark Hempstead, Brandon Reagen, Xuan Zhang, David Brooks, Vikas Chandra, Utku Diril, et al. 2020. RecNMP: Accelerating Personalized Recommendation with Near-Memory Processing, In *ISCA*.

[65] Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K Nurminen, and Zhonghong Ou 2018. Rapl in Action: Experiences in Using RAPL for Power Measurements, In *TOMPECS*.

[66] Kevin Kiningham, Philip Levis, and Christopher Ré 2022. GRIP: A Graph Neural Network Accelerator Architecture. *IEEE Trans. Comput.*, 2022.

[67] Thomas N Kipf and Max Welling 2016. Semi-Supervised Classification with Graph Convolutional Networks. *arXiv*, 2016.

[68] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe 2017. The Tensor Algebra Compiler. *Proc. ACM Program. Lang.*, 2017.

[69] Penporn Koanantakool, Ariful Azad, Aydin Buluç, Dmitriy Morozov, Sang-Yun Oh, Leonid Oliker, and Katherine Yelick 2016. Communication-Avoiding Parallel Sparse-Dense Matrix-Matrix Multiplication, In *IPDPS*.

[70] Youngeun Kwon, Yunjae Lee, and Minsoo Rhu 2019. TensorDIMM: A Practical Near-Memory Processing Architecture for Embeddings and Tensor Operations in Deep Learning, In *MICRO*.

[71] Youngeun Kwon, Yunjae Lee, and Minsoo Rhu 2019. Tensordimm: A Practical Near-Memory Processing Architecture for Embeddings and Tensor Operations in Deep Learning, In *MICRO*.

[72] Young-Cheon Kwon, Suk Han Lee, Jaehoon Lee, Sang-Hyuk Kwon, Je Min Ryu, Jong-Pil Son, O Seongil, Hak-Soo Yu, Haesuk Lee, Soo Young Kim, Youngmin Cho, Jin Guk Kim, Jongyoon Choi, Hyun-Sung Shin, Jin Kim, BengSeng Phuah, HyoungMin Kim, Myeong Jun Song, Ahn Choi, Daeho Kim, SooYoung Kim, Eun-Bong Kim, David Wang, Shinhaeng Kang, Yuhwan Ro, Seungwoo Seo, JoonHo Song, Jaeyoun Youn, Kyomin Sohn, and Nam Sung Kim 2021. 25.4 A 20nm 6GB Function-In-Memory DRAM, Based on HBM2 with a 1.2TFLOPS Programmable Computing Unit Using Bank-Level Parallelism, for Machine Learning Applications, In *ISSCC*.

[73] Daniel Langr and Pavel Tvrdík 2016. Evaluation Criteria for Sparse Matrix Storage Formats, In *TPDS*.

[74] Sukhan Lee, Shin-haeng Kang, Jaehoon Lee, Hyeonsu Kim, Eojin Lee, Seungwoo Seo, Hosang Yoon, Seungwon Lee, Kyounghwan Lim, Hyunsung Shin, Jinhyun Kim, O Seongil, Anand Iyer, David Wang, Kyomin Sohn, and Nam Sung Kim 2021. Hardware Architecture and Software Stack for PIM Based on Commercial DRAM Technology: Industrial Product, In *ISCA*.

[75] Seongju Lee, Kyuyoung Kim, Sanghoon Oh, Joonhong Park, Gimoon Hong, Dongyoon Ka, Kyudong Hwang, Jeongje Park, Kyeongpil Kang, Jungyeon Kim, et al. 2022. A 1ynm 1.25 V 8GB, 16GB/s/Pin GDDR6-Based Accelerator-in-Memory Supporting 1Tflops MAC Operation and Various Activation Functions for Deep-Learning Applications, In *ISSCC*.

[76] Yunjae Lee, Jinha Chung, and Minsoo Rhu 2022. SmartSAGE: Training Large-Scale Graph Neural Networks Using In-Storage Processing Architectures, In *ISCA*.

[77] Damitha Lenadora, Vimarsh Sathia, Gerasimos Gerogiannis, Serif Yesil, Josep Torrellas, and Charith Mendis 2024. SENSEi: Input-Sensitive Compilation for Accelerating GNNs, In *arXiv*.

[78] Cangyuan Li, Ying Wang, Cheng Liu, Shengwen Liang, Huawei Li, and Xiaowei Li 2021. GLIST: Towards In-Storage Graph Learning, In *ATC*.

[79] Cong Li, Zhe Zhou, Xingchen Li, Guangyu Sun, and Dimin Niu 2023. NMExplorer: An Efficient Exploration Framework for DIMM-Based Near-Memory Tensor Reduction, In *DAC*.

[80] Cong Li, Zhe Zhou, Yang Wang, Fan Yang, Ting Cao, Mao Yang, Yun Liang, and Guangyu Sun 2024. PIM-DL: Expanding the Applicability of Commodity DRAM-PIMs for Deep Learning via Algorithm-System Co-Optimization, In *ASPLOS*.
[81] Cong Li, Zhe Zhou, Size Zheng, Jiaxi Zhang, Yun Liang, and Guangyu Sun 2024. SpecPIM: Accelerating Speculative Inference on PIM-Enabled System via Architecture-Dataflow Co-Exploration, In *ASPLOS*.
[82] Jiajun Li, Ahmed Louri, Avinash Karanth, and Razvan Bunescu 2021. GCNAX: A Flexible and Energy-Efficient Accelerator for Graph Convolutional Neural Networks, In *HPCA*.
[83] Shengwen Liang, Ying Wang, Cheng Liu, Lei He, LI Huawei, Dawen Xu, and Xiaowei Li 2020. Engn: A High-Throughput and Energy-Efficient Accelerator for Large Graph Neural Networks. *IEEE Trans. Comput.*, 2020.
[84] Chaemin Lim, Suhyun Lee, Jinwoo Choi, Jounghoo Lee, Seongyeon Park, Hanjun Kim, Jinho Lee, and Youngsok Kim 2023. Design and Analysis of a Processing-in-DIMM Join Algorithm: A Case Study with UPMEM DIMMs. *Proc. ACM Manag. Data*, 2023.
[85] Y. Lin and V. Prasanna 2023. HyScale-GNN: A Scalable Hybrid GNN Training System on Single-Node Heterogeneous Architecture, In *IPDPS*.
[86] Jiawen Liu, Hengyu Zhao, Matheus Almeida Ogleari, Dong Li, and Jishen Zhao 2018. Processing-in-Memory for Energy-Efficient Neural Network Training: A Heterogeneous Approach, In *MICRO*.
[87] Tianfeng Liu, Yangrui Chen, Dan Li, Chuan Wu, Yibo Zhu, Jun He, Yanghua Peng, Hongzheng Chen, Hongzhi Chen, and Chuanxiong Guo 2023. BGL: GPU-Efficient GNN Training by Optimizing Graph Data I/O and Preprocessing, In *NSDI*.
[88] Weifeng Liu and Brian Vinter 2014. An Efficient GPU General Sparse Matrix-Matrix Multiplication for Irregular Data, In *IPDPS*.
[89] Zhiyu Liu, Irina Calciu, Maurice Herlihy, and Onur Mutlu 2017. Concurrent Data Structures for Near-Memory Computing, In *SPAA*.
[90] Dominik Marek Loroch, Norbert Wehn, Franz-Josef Pfreundt, and Janis Keuper 2017. TensorQuant - A Simulation Toolbox for Deep Neural Network Quantization, In *arXiv*.
[91] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai 2019. Neugraph: Parallel Deep Neural Network Computation on Large Graphs, In *ATC*.
[92] Vasimuddin Md, Sanchit Misra, Guixiang Ma, Ramanarayan Mohanty, Evangelos Georganas, Alexander Heinecke, Dhiraj Kalamkar, Nesreen K. Ahmed, and Sasikanth Avancha 2021. DistGNN: Scalable Distributed Training for Large-Scale Graph Neural Networks, In *SC*.
[93] Sameh K Mohamed, Vít Nováček, and Aayah Nounu 2020. Discovering Protein Drug Targets Using Knowledge Graph Embeddings. *Bioinformatics*, 2020.
[94] P. Mpakos, D. Galanopoulos, P. Anastasiadis, N. Papadopoulou, N. Koziris, and G. Goumas 2023. Feature-Based SpMV Performance Analysis on Contemporary Devices, In *IPDPS*.
[95] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun 2019. Processing Data Where It Makes Sense: Enabling In-Memory Computation, In *MICPRO*.
[96] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun 2021. A Modern Primer on Processing in Memory, In *Emerging Computing: From Devices to Systems - Looking Beyond Moore and Von Neumann*. https://arxiv.org/pdf/2012.03112.pdf
[97] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and R. Ausavarungnirun 2021. A Modern Primer on Processing in Memory. *Emerging Computing: From Devices to Systems - Looking Beyond Moore and Von Neumann*, 2021.
[98] R. Nair, S. F. Antao, C. Bertolli, P. Bose, J. R. Brunheroto, T. Chen, C.-Y. Cher, C. H. A. Costa, J. Doi, C. Evangelinos, and et al. 2015. Active Memory Cube: A Processing-in-Memory Architecture for Exascale Systems, In *IBM JRD*.
[99] Maxim Naumov, L Chien, Philippe Vandermersch, and Ujval Kapasi 2010. Cusparse Library, In *GPU Technology Conference*.
[100] Yuyao Niu, Zhengyang Lu, Haonan Ji, Shuhui Song, Zhou Jin, and Weifeng Liu 2022. TileSpGEMM: A Tiled Algorithm for Parallel Sparse General Matrix-Matrix Multiplication on GPUs, In *PpopP*.
[101] S. Noh, J. Hong, C. Lim, S. Park, J. Kim, H. Kim, Y. Kim, and J. Lee 2024. PID-Comm: A Fast and Flexible Collective Communication Framework for Commodity Processing-in-DIMM Devices, In *ISCA*.
[102] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok Kim, David Blaauw, Trevor Mudge, and Ronald Dreslinski 2018. Outerspace: An Outer Product Based Sparse Matrix Multiplication Accelerator, In *HPCA*.
[103] Jaehyun Park, Jaewan Choi, Kwanhee Kyung, Michael Jaemin Kim, Yongsuk Kwon, Nam Sung Kim, and Jung Ho Ahn 2024. AttAcc! Unleashing the Power of PIM for Batched Transformer-Based Generative Model Inference, In *ASPLOS*.
[104] Yeonhong Park, Sunhong Min, and Jae W. Lee 2022. Ginex: SSD-Enabled Billion-Scale Graph Neural Network Training on a Single Machine via Provably Optimal In-Memory Caching. *Proc. VLDB Endow.*, 2022.

[105] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An Imperative Style, High-Performance Deep Learning Library. *NIPS*, 2019.
[106] PeakPerf 2021. PeakPerf. https://github.com/Dr-Noob/peakperf.git
[107] Ali Pinar and Michael T. Heath 1999. Improving Performance of Sparse Matrix-Vector Multiplication, In *SC*.
[108] Udo W. Pooch and Al Nieder 1973. A Survey of Indexing Techniques for Sparse Matrices, In *ACM Comput. Surv.*
[109] PyG 2024. PyG Website. https://pyg.org/
[110] Guocheng Qian, Abdulellah Abualshour, Guohao Li, Ali Thabet, and Bernard Ghanem 2021. Pu-GCN: Point Cloud Upsampling Using Graph Convolutional Networks, In *CVPR*.
[111] Zheng Qu, Dimin Niu, Shuangchen Li, Hongzhong Zheng, and Yuan Xie 2023. TT-GNN: Efficient On-Chip Graph Neural Network Training via Embedding Reformation and Hardware Optimization, In *MICRO*.
[112] Steve Rhyner, Haocong Luo, Juan Gómez-Luna, Mohammad Sadrosadati, Jiawei Jiang, Ataberk Olgun, Harshita Gupta, Ce Zhang, and Onur Mutlu 2024. Analysis of Distributed Optimization Algorithms on a Real Processing-In-Memory System, In *PACT*.
[113] Oguz Selvitopi, Benjamin Brock, Israt Nisa, Alok Tripathy, Katherine Yelick, and Aydın Buluç 2021. Distributed-Memory Parallel Algorithms for Sparse Times Tall-Skinny-Dense Matrix Multiplication, In *ICS*.
[114] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens 2007. Scan Primitives for GPU Computing, In *GH*.
[115] Chao Shang, Jie Chen, and Jinbo Bi 2021. Discrete Graph Structure Learning for Forecasting Multiple Time Series, In *ICLR*.
[116] Yongwon Shin, Juseong Park, Sungjun Cho, and Hyojin Sung 2023. PIMFlow: Compiler and Runtime Support for CNN Models on Processing-in-Memory DRAM, In *CGO*.
[117] Linghao Song, Yuze Chi, Atefeh Sohrabizadeh, Young-kyu Choi, Jason Lau, and Jason Cong 2022. Sextans: A Streaming Accelerator for General-Purpose Sparse-Matrix Dense-Matrix Multiplication, In *SIGDA*.
[118] Xinkai Song, Tian Zhi, Zhe Fan, Zhenxing Zhang, Xi Zeng, Wei Li, Xing Hu, Zidong Du, Qi Guo, and Yunji Chen 2021. Cambricon-G: A Polyvalent Energy-Efficient Accelerator for Dynamic Graph Neural Networks. *TCAD*, 2021.
[119] Nitish Srivastava, Hanchen Jin, Jie Liu, David Albonesi, and Zhiru Zhang 2020. Matraptor: A Sparse-Sparse Matrix Multiplication Accelerator Based on Row-Wise Product, In *MICRO*.
[120] Jacob R Stevens, Dipankar Das, Sasikanth Avancha, Bharat Kaul, and Anand Raghunathan 2021. GNNerator: A Hardware/Software Framework for Accelerating Graph Neural Networks, In *DAC*.
[121] Jonathan M Stokes, Kevin Yang, Kyle Swanson, Wengong Jin, Andres Cubillos-Ruiz, Nina M Donghia, Craig R MacNair, Shawn French, Lindsey A Carfrae, Zohar Bloom-Ackermann, et al. 2020. A Deep Learning Approach to Antibiotic Discovery. *Cell*, 2020.
[122] Foteini Strati, Christina Giannoula, Dimitrios Siakavaras, Georgios Goumas, and Nectarios Koziris 2019. An Adaptive Concurrent Priority Queue for NUMA Architectures, In *International Conference on Computing Frontiers*.
[123] stream 2021. STREAM. https://github.com/jeffhammond/STREAM.git
[124] Damian Szklarczyk, Annika L Gable, David Lyon, Alexander Junge, Stefan Wyder, Jaime Huerta-Cepas, Milan Simonovic, Nadezhda T Doncheva, John H Morris, Peer Bork, et al. 2019. STRING v11: Protein–Protein Association Networks with Increased Coverage, Supporting Functional Discovery in Genome-Wide Experimental Datasets. *Nucleic Acids Research*, 2019.
[125] Wai Teng Tang, Ruizhe Zhao, Mian Lu, Yun Liang, Huynh Phung Huyng, Xibai Li, and Rick Siow Mong Goh 2015. Optimizing and Auto-Tuning Scale-Free Sparse Matrix-Vector Multiplication on Intel Xeon Phi, In *CGO*.
[126] John Thorpe, Yifan Qiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Jinliang Wei, Keval Vora, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu 2021. Dorylus: Affordable, Scalable, and Accurate GNN Training with Distributed CPU Servers and Serverless Threads, In *OSDI*.
[127] Boyu Tian, Yiwei Li, Li Jiang, Shuangyu Cai, and Mingyu Gao 2024. NDPBridge: Enabling Cross-Bank Coordination in Near-DRAM-Bank Processing Architectures, In *ISCA*.
[128] Teng Tian, Xiaotian Wang, Letian Zhao, Wei Wu, Xuecang Zhang, Fangmin Lu, Tianqi Wang, and Xi Jin 2022. G-NMP: Accelerating Graph Neural Networks with DIMM-Based Near-Memory Processing. *Journal of Systems Architecture*, 2022.
[129] Alok Tripathy, Katherine Yelick, and Aydın Buluç 2020. Reducing Communication in Graph Neural Network Training, In *SC*.
[130] UPMEM 2020. UPMEM Website. https://www.upmem.com
[131] Field G Van Zee, Tyler M Smith, Bryan Marker, Tze Meng Low, Robert A Van De Geijn, Francisco D Igual, Mikhail Smelyanskiy, Xianyi Zhang, Michael Kistler, Vernon Austel, et al. 2016. The BLIS Framework: Experiments in Portability. *TOMS*, 2016.

[132] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, Yoshua Bengio, et al. 2017. Graph Attention Networks. *Stat*, 2017.
[133] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang 2014. *Intel Math Kernel Library*.
[134] Shu Wu, Yuyuan Tang, Yanqiao Zhu, Liang Wang, Xing Xie, and Tieniu Tan 2019. Session-Based Recommendation with Graph Neural Networks, In *AAAI*.
[135] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka 2018. How Powerful Are Graph Neural Networks? *arXiv*, 2018.
[136] Carl Yang, Aydın Buluç, and John D. Owens 2018. Design Principles for Sparse Matrix Multiplication on the GPU, In *Euro-PAR*.
[137] Zhilin Yang, William Cohen, and Ruslan Salakhudinov 2016. Revisiting Semi-Supervised Learning with Graph Embeddings, In *ICML*.
[138] Zihao Ye, Ruihang Lai, Junru Shao, Tianqi Chen, and Luis Ceze 2023. SparseTIR: Composable Abstractions for Sparse Compilation in Deep Learning, In *ASPLOS*.
[139] Serif Yesil, José E Moreira, and Josep Torrellas 2022. Dense Dynamic Blocks: Optimizing SpMM for Processors with Vector and Matrix Units Using Machine Learning Techniques, In *ICS*.
[140] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec 2018. Graph Convolutional Neural Networks for Web-Scale Recommender Systems, In *SIGKDD*.
[141] Sungmin Yun, Hwayong Nam, Jaehyun Park, Byeongho Kim, Jung Ho Ahn, and Eojin Lee 2023. GraNDe: Efficient Near-Data Processing Architecture for Graph Neural Networks. *IEEE Trans. Comput.*, 2023.
[142] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna 2019. GraphSaint: Graph Sampling Based Inductive Learning Method. *arXiv*, 2019.
[143] Liekang Zeng, Xu Chen, Peng Huang, Ke Luo, Xiaoxi Zhang, and Zhi Zhou 2023. Serving Graph Neural Networks With Distributed Fog Servers for Smart IoT Services. *IEEE/ACM Transactions on Networking*, 2023.
[144] Yi Zhai, Yu Zhang, Shuo Liu, Xiaomeng Chu, Jie Peng, Jianmin Ji, and Yanyong Zhang 2023. TLP: A Deep Learning-Based Cost Model for Tensor Program Tuning, In *ASPLOS*.
[145] Mingxing Zhang, Youwei Zhuo, Chao Wang, Mingyu Gao, Yongwei Wu, Kang Chen, Christos Kozyrakis, and Xuehai Qian 2018. GraphP: Reducing Communication for PIM-Based Graph Processing with Efficient Data Partition, In *HPCA*.
[146] Tianyi Zhang, Zhiqiu Lin, Guandao Yang, and Christopher De Sa 2019. QPyTorch: A Low-Precision Arithmetic Simulation Framework, In *arXiv*.
[147] D. Zheng, C. Ma, M. Wang, J. Zhou, Q. Su, X. Song, Q. Gan, Z. Zhang, and G. Karypis 2020. DistDGL: Distributed Graph Neural Network Training for Billion-Scale Graphs, In *IA3*.
[148] Zhe Zhou, Cong Li, Xuechao Wei, Xiaoyang Wang, and Guangyu Sun 2022. Gnnear: Accelerating Full-Batch Training of Graph Neural Networks with Near-Memory Processing, In *PACT*.
[149] Youwei Zhuo, Chao Wang, Mingxing Zhang, Rui Wang, Dimin Niu, Yanzhi Wang, and Xuehai Qian 2019. GraphQ: Scalable PIM-based Graph Processing, In *MICRO*.

A Appendix

A.1 PyGim Tuner Efficiency

Fig. 16 evaluates the PyGim tuner efficiency for COO format by comparing the performance slowdown achieved by the predicted aggregation configuration (predicted) of tuner versus the oracle prediction using various datasets and hidden sizes. For the oracle prediction performance, we exhaustively iterate and collect the execution times of all possible aggregation configurations, then we select and present in Fig. 16 the best-performing execution time among them (oracle). The selected aggregation configuration by the tuner achieves only 1% worse performance over the oracle-predicted configuration on average across all datasets and hidden sizes, when using COO format for GNN aggregation. We conclude that PyGim tuner effectively tunes the aggregation configuration in GNN executions for both CSR (Fig. 12) and COO formats.

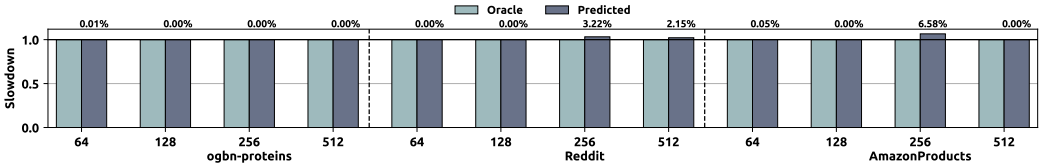

Fig. 16. Performance slowdown of the predicted COO aggregation configuration by tuner over oracle prediction.

A.2 GNN Aggregation Energy Consumption

Fig. 17 presents the energy consumption (in Joules) for all comparison points (See §4.1) in one GNN aggregation using int32 data type, and various datasets and hidden sizes. In PIM executions, we use 32 PIM devices, having in total 1992 cores, and we enable the PyGim's tuner. We use Intel RAPL [65] to measure energy in CPU execution parts, which are (i) the PyTorch scheme, and (ii) in PIM schemes, the load, retrieve, and merge steps of aggregation. For the kernel step of aggregation, we measure the energy consumed in PIM-enabled chips using the methodology suggested by the UPMEM PIM manufacturer, which is described in a recent paper [28]: the power of each PIM DIMM is 23.22W, thus the total energy of kernel time is calculated as $kernel\_time \times \#PIM\_DIMMs \times power$. PyGim provides higher energy efficiency by on average 4.08× and 1.39× over prior PIM-based and PyTorch schemes, respectively.

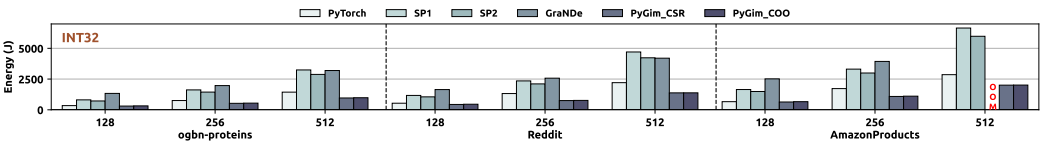

Fig. 17. Energy consumption of all comparison in the one aggregation, using various graph datasets and hidden sizes.

A.3 GNN Inference Performance

Figs. 18, 19 and 20 compare the performance of all comparison points (See §4.1) in the end-to-end GNN inference, using 32-bit float (**fp32**), 8-bit integer (**int8**) and 16-bit integer (**int16**) data types for data values, respectively. We evaluate various graph datasets and three different GNN models, where each model has 3 layers of 256 hidden size. In PIM executions, we use 32 PIM devices, having in total ~1992 cores. In PyGim, we evaluate both CSR and COO schemes and we enable the tuner to set the aggregation configuration.

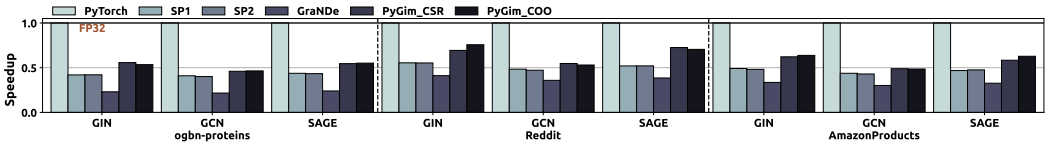

Fig. 18. Performance of all comparison points in the end-to-end GNN inference, using various graph datasets and GNN models for fp32 data type.

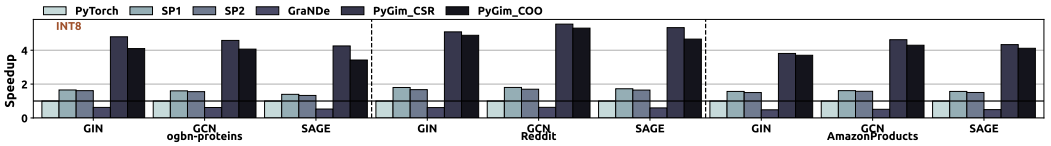

Fig. 19. Performance of all comparison points in the end-to-end GNN inference, using various graph datasets and GNN models for int8 data type.

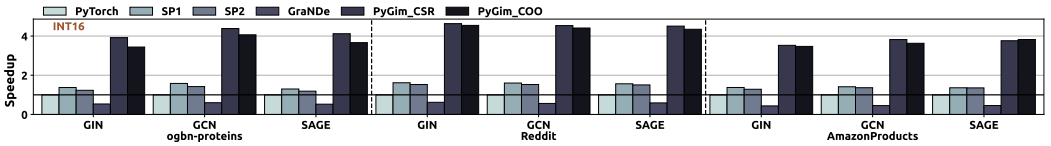

Fig. 20. Performance of all comparison points in the end-to-end GNN inference, using various graph datasets and GNN models for int16 data type.

We find that when using fp32 values, PIM GNN execution achieves low performance, being worse than that of PyTorch by on average 41.6%. This is because UPMEM PIM hardware does not support floating-point operations, which are software emulated, thus they incur high performance overheads. However, we expect that ML-oriented PIM systems will be available in the market (e.g., [74, 75]), and will hopefully support in hardware high precision data types. Instead, when using int8 and int16 data types, PyGim schemes significantly outperform PyTorch scheme by 4.49× (up to 5.54×) and 4.03× (up to 4.63×), respectively. Moreover, PyGim outperforms prior state-of-the-art PIM approaches by 3.59× (up to 9.89×) and 3.60× (up to 8.90×) for int8 and int16 data types, respectively. When arithmetic operations are natively supported by the PIM hardware, as it happens for the int8 and int16 data type in UPMEM PIM system, PyGim execution provides significant performance benefits over prior schemes.

A.4 GNN Inference Accuracy

Table 3 shows the test accuracy achieved in GNN inference with int32 and fp32 data types. We evaluate the accuracy of the experiments presented in Fig. 14 and Fig. 18. These models have been trained with fp32 data type, and then we run and evaluate inference using either int32 or fp32 data type. All the comparison points compared in Fig. 14 and Fig. 18, i.e., both the CPU-based scheme (PyTorch) and the PIM-based schemes (SP1, SP2, GraNDe, PyGim_CSR, PyGim_COO) achieve the same accuracy, shown in Table 3 for int32 and fp32 data types. The GNN models are relatively simple, having only 3 layers, thus int32 and fp32 data types provide the same accuracy.

	ogbn-proteins			Reddit			AmazonProducts		
	GIN	GCN	SAGE	GIN	GCN	SAGE	GIN	GCN	SAGE
INT32	79.89%	78.20%	73.38%	94.08%	91.90%	94.39%	26.04%	26.04%	26.04%
FP32	79.89%	78.20%	73.38%	94.08%	91.90%	94.39%	26.04%	26.04%	26.04%

Table 3. Inference accuracy achieved by all comparison points using various graph datasets and GNN models for int32 and fp32 data types.

A.5 GNN Training Performance

Table 4 shows the execution time of PyTorch (CPU) and PyGim CSR (UPMEM PIM) schemes, when training a 2-layer GCN model for 10 epochs with 256 hidden size, int32 data type and evaluating all three datasets. Note that UPMEM PIM does not support fp32 operations in hardware. Thus, for a fair comparison, we evaluate GNN training using int32 data type only, so that the data type used is fully supported by hardware in both evaluated systems. Table 5 shows the test accuracy achieved in the GCN model after the model is trained for 1000 epochs and learning rate of 0.01 with either PyTorch or PyGim CSR scheme for int32 data type. Both PyTorch and PyGim CSR schemes achieve the same accuracy in int32 data type. PyGim improves the training performance by on average 1.25× over the PyTorch scheme. Therefore, PyGim provides high performance benefits even for GNN training, without degrading accuracy.

	INT32 OGBN	**INT32 RDT**	**INT32 AMZ**
pytorch_sparse (Intel Xeon 4215 CPU)	203.982 s	339.023 s	440.899 s
PyGim (UPMEM PIM)	164.709 s	242.789 s	400.998 s

Table 4. Execution time of GCN training for 10 epochs using pytorch_sparse (CPU) and PyGim (PIM cores for aggregation) libraries in aggregation step for the ogbn-proteins (**OGBN**), Reddit (**RDT**), and AmazonProducts (**AMZ**) datasets.

	INT32 OGBN	**INT32 RDT**	**INT32 AMZ**
pytorch_sparse (Intel Xeon 4215 CPU)	70.34%	84.25%	26.04%
PyGim (UPMEM PIM)	70.34%	84.25%	26.04%

Table 5. Test accuracy achieved after training the GCN for 1000 epochs and a learning rate of 0.01 with either PyTorch or PyGim CSR scheme using int32 data type and the ogbn-proteins (**OGBN**), Reddit (**RDT**), and AmazonProducts (**AMZ**) datasets.

A.6 Evaluation of GNN Executions in GPU Systems

We present the performance and energy efficiency metrics to show the readers how much performance and energy efficiency the evaluated UPMEM PIM system can achieve when it is compared over commodity GPU systems. In Table 6, we present the performance in GNN aggregation comparing three GPU systems over the evaluated UPMEM PIM system. Similarly, in Table 7, we present the performance in GNN inference comparing two GPU systems over the evaluated UPMEM PIM system. For UPMEM PIM, we use PyGim library, and for GPU systems we use pytorch_sparse library that provides optimized CUDA implementations. Please note that these evaluation results are provided for completeness and *not* competition purposes, since real PIM systems are still in early manufacturing and design stages (especially compared to commercial CPU and GPU systems), and PyGim can be evaluated on other current and future real PIM systems with potentially better computation capabilities and energy efficiency than the evaluated UPMEM PIM system.

Dataset and data type	**OGBN INT32**	**RDT INT32**	**AMZ INT32**	**OGBN FP32**	**RDT FP32**	**AMZ FP32**
GPU GTX 1080 Ti over UPMEM PIM	17.7× | 8.0×	5.3× | 3.3×	6.9× | 3.8×	112.0× | 69.9×	35.9× | 28.1×	44.0× | 31.7×
GPU RTX 2080 Ti over UPMEM PIM	15.3× | 8.0×	7.3× | 3.8×	8.9× | 4.2×	102.5× | 66.4×	49.8× | 33.4×	56.3× | 35.9×
GPU RTX 3090 over UPMEM PIM	39.8× | 21.2×	19.3× | 6.9×	18.4× | 6.6×	240.4× | 200.1×	122.4× | 55.1×	111.3× | 46.1×

Table 6. Performance speedup (left number in each cell) and energy efficiency improvement (right number in each cell) of the three GPU generations over the UPMEM PIM system in GNN aggregation using INT32 and FP32 data types and the ogbn-proteins (**OGBN**), Reddit (**RDT**), and AmazonProducts (**AMZ**) datasets.

Dataset and GNN model	OGBN GIN	RDT GIN	AMZ GIN	OGBN GCN	RDT GCN	AMZ GCN
GPU RTX 2080 Ti over UPMEM PIM	17.4× | 88.1×	9.1× | 42.5×	11.2× | 45.3×	17.2× | 107.3×	8.5× | 51.8×	10.4× | 56.9×
GPU RTX 3090 over UPMEM PIM	37.2× | 188.7×	20.7× | 97.2×	20.8× | 84.4×	38.2× | 238.2×	19.9× | 121.2×	19.6× | 106.7×

Table 7. Performance speedup for int32 data type (left number in each cell) and for fp32 data type (right number in each cell) of two GPU generations over the UPMEM PIM system in the end-to-end GNN inference using GIN and GCN models and the ogbn-proteins (**OGBN**), Reddit (**RDT**), and AmazonProducts (**AMZ**) datasets.

A.7 Datasets

Sparse Matrices. We present the characteristics of the real-world matrices that we use in our experiments to evaluate PyGim when using one PIM core and when using one PIM cluster. The sparse matrices are taken from the Sparse Matrix Suite Collection [24]. Table 8 presents the number of rows, the number of non-zero elements (NNZs), the minimum number (min) of non-zero elements among rows, the maximum number (max) of non-zero elements among rows, the average number (avg) of non-zero elements among rows and standard deviation (std) of non-zero elements among rows.

Matrix Name	Rows	NNZ	Min NNZ	Max NNZ	Avg NNZ	Std NNZ
raefsky4	19779	1328611	18	177	67.17	15.96
wing_nodal	10937	150976	5	28	13.80	2.86
Dubcova2	65025	1030225	4	25	15.84	5.76
mosfet2	46994	1499460	4	162	31.91	11.71
poisson3Db	85623	2374949	6	145	27.74	14.71
smt	25710	3753184	52	414	145.98	47.52
av41092	41092	1683902	2	2135	40.98	167.04
Zd_Jac6	22835	1711983	1	1050	74.97	175.48
mycielskian15	24575	11111110	14	12287	452.13	664.17

Table 8. Sparse matrix dataset used for one PIM core and one PIM cluster analysis.

Graph Datasets. We present the characteristics of the real-world graph datasets that we use in our large-scale experiments to evaluate PyGim using multiple PIM DIMMs and devices, as well as to evaluate CPU and PIM schemes in aggregation operator and end-to-end GNN inference. The real-world graph datasets are taken from ogbn-proteins [124], Reddit [49] and AmazonProducts [142]. The original AmazonProducts dataset is too large to fit in a single machine, thus we split the dataset using cluster partition [17], and evaluate the largest subgraph in our experiments, its detailed characteristics are shown in Table 9. Specifically, Table 9 presents the number of vertices, the number of edges (EDGs), the minimum number (min) of edges among vertices, the maximum number (max) of edges among vertices, the average number (avg) of edges among vertices and standard deviation (std) of edges among vertices.

Graph Name	Vertices	EDGs	Min EDG	Max EDG	Avg EDG	Std EDG
ogbn-proteins	132534	79122504	1	7750	597.00	621.48
Reddit	232965	114615892	1	21657	492.00	799.82
AmazonProducts	403598	156149176	1	53864	386.89	1140.91

Table 9. Real-world graph datasets used for our large-scale experiments, when using multiple PIM DIMMs.