NEM-GNN: DAC/ADC-less, Scalable, Reconfigurable, Graph and Sparsity-Aware Near-Memory Accelerator for Graph Neural Networks

SIDDHARTHA RAMAN SUNDARA RAMAN, The University of Texas at Austin, Austin, USA
LIZY JOHN, The University of Texas at Austin, Austin, USA
JAYDEEP P. KULKARNI, The University of Texas at Austin, Austin, USA

Graph neural networks (GNNs) are of great interest in real-life applications such as citation networks and drug discovery owing to GNN's ability to apply machine learning techniques on graphs. GNNs utilize a two-step approach to classify the nodes in a graph into pre-defined categories. The first step uses a combination kernel to perform data-intensive convolution operations with regular memory access patterns. The second step uses an aggregation kernel that operates on sparse data having irregular access patterns. These mixed data patterns render CPU/GPU-based compute energy-inefficient. Von Neumann based accelerators like AWB-GCN [7] suffer from increased data movement, as the data-intensive combination requires large data movement to/from memory to perform computations. ReFLIP [8] performs resistive random access memory based in-memory (PIM) compute to overcome data movement costs. However, ReFLIP suffers from increased area requirement due to dedicated accelerator arrangement, and reduced performance due to limited parallelism and energy due to fundamental issues in ReRAM-based compute. This article presents a scalable (non-exponential storage requirement), DAC/ADC-less PIM-based combination, with (i) early compute termination and (ii) pre-compute by reconfiguring SOC components. Graph and sparsity-aware near-memory aggregation using the proposed compute-as-soon-as-ready (CAR) broadcast approach improves performance and energy further. NEM-GNN achieves ∼80–230x, ∼80–300x, ∼850–1,134x, and ∼7–8x improvement over ReFLIP, in terms of performance, throughput, energy efficiency, and compute density.

CCS Concepts: • **Computer systems organization** → *Architectures*; *Other architectures*; *Neural networks;*

Additional Key Words and Phrases: Graph neural networks, L1 cache, processing in memory, compute-as-soon-as-ready, broadcast, early compute termination, pre-compute, sparsity-aware, graph-aware

ACM Reference Format:
Siddhartha Raman Sundara Raman, Lizy John, and Jaydeep P. Kulkarni. 2024. NEM-GNN: DAC/ADC-less, Scalable, Reconfigurable, Graph and Sparsity-Aware Near-Memory Accelerator for Graph Neural Networks. *ACM Trans. Arch. Code Optim.* 21, 2, Article 39 (May 2024), 26 pages. https://doi.org/10.1145/3652607

1 INTRODUCTION

Over the past decade, there has been widespread utilization of deep learning models such as convolutional neural networks and recommendation networks in diverse fields like image and video processing. These models primarily operate within the realm of Euclidean data, wherein data inputs conform to a structured and precisely defined n-dimensional space. However, these

Authors' address: S. R. Sundara Raman, L. John, and J. P. Kulkarni, The University of Texas at Austin, 2515 Speedway, Austin, TX 78712; e-mails: s.siddhartharaman@utexas.edu, ljohn@ece.utexas.edu, jaydeep@austin.utexas.edu.

ACM 1544-3973/2024/05-ART39
https://doi.org/10.1145/3652607

well-established models exhibit inefficiency when confronted with domains like citation networks, molecular chemistry, and electric grids [8, 9], characterized by non-Euclidean data structures such as graphs and manifold geometries (3D surface structures). In these cases, the data inputs deviate from structured paradigms, and the pursuit to encapsulate them within a strictly defined n-dimensional space leads to loss of valuable information. Consequently, geometric deep learning [2] has emerged as a solution tailored to the idiosyncrasies of non-Euclidean data relationships. One of the approaches geared toward processing graph-based data hinges on the utilization of **Graph Neural Network (GNN)** models. These models are adept at handling graph-based inputs, facilitating the classification of nodes in a graph into distinct categorical groups.

GNNs employ two fundamental mechanisms: (i) a combination kernel, akin to the convolution process in CNNs, is harnessed to encode the information within nodes, and (ii) aggregation kernels are then utilized to encode the information in edges, which helps to comprehend the relationship between graph nodes. Regarding computations, the operations associated with the combination kernel are notably data intensive, exhibiting regular memory access patterns. However, operations linked to the aggregation kernel are characterized by sparsity and irregularity. The mixed data pattern is a major limiter behind using CPU/GPU for GNN compute [28]. Various accelerator designs have been proposed to tackle this particular concern.

These accelerators can broadly be classified into traditional von Neumann based accelerators/processing in/near-memory designs. The proposed designs (both von Neumann/**Processing in Memory (PIM)**) are dedicated domain-specific accelerators that require periodic interaction with the host, resulting in energy overhead. The existing von Neumann based accelerator designs like AWB-GCN [7] and HyGCN [28] incur data movement cost for transferring data from the processor to memory. Since the combination kernel uses data-intensive operations, requiring periodic data movement costs, this architecture is energy-inefficient. The state-of-the-art PIM/near-memory architectures for solving traditional graph algorithms, and GNNs, like ReFLIP [8], exhibit reduction in data movement by enabling computations within memory [19]. These architectures use crossbar **Resistive Random Access Memory (ReRAM)** arrays, **Digital-to-Analog Converters (DACs)**, and **Analog-to-Digital Converters (ADCs)**. The problems specifically with ReFLIP are the following. First, this is a *dedicated accelerator* requiring periodic host-accelerator interaction, leading to energy inefficiency. Second, the presence of DACs/ADCs renders the architecture susceptible to process variations, causing a decline in *accuracy*. Third, exponential increase in storage requirement with increased bit-precision/resolution for GNN compute incurs *scalability* challenges. Fourth, aggregation is performed only upon the completion of combination operations, lacking any overlap between them, limiting *performance.* Fifth, sparse in-memory aggregation leads to *compute density* challenges. The major contributions of this work are the following:

— *Re-configurability*: SOC components like L1/L2 cache are reconfigured to realize NEM-GNN without requiring memory array modifications/dedicated accelerator arrangement, thereby improving energy of the design.
— *DAC/ADC-less*: Bit-serial PIM architectures for combination (NEM-C1, NEM-C2, NEM-C3) with early termination of compute and pre-compute strategies without DAC/ADC, achieving highly accurate compute.
— *Scalability*: NEM-GNN is scalable to high bit-precision compute without requiring exponential storage for the compute array.
— *Graph-aware*: Near-memory aggregation optimized specifically with **Compute-as-soon-as-Ready (CAR)** and "broadcast" approaches to hide aggregation latency, by overlapping combination and aggregation completely.

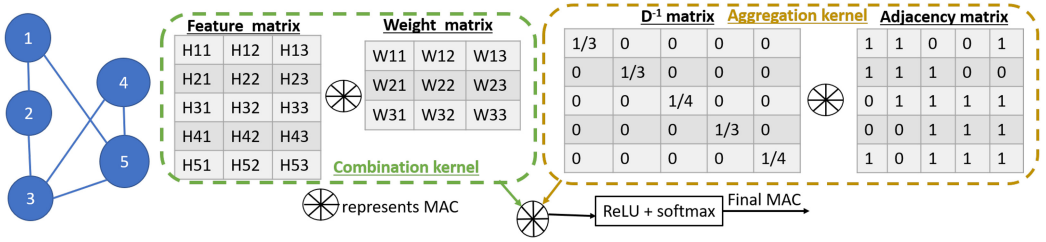


Fig. 1. Undirected, unweighted graph with five nodes and six edges passing through a one-layer GCN. Combination showing MAC between dense feature and weight matrices, and aggregation showing MAC between sparse D^{-1} and adjacency matrices to generate the final MAC before ReLU and softmax function.

— *Sparsity-aware*: Leveraging *sparsity* in kernels to minimize the unnecessary compute and utilizing the high-throughput of the design to improve compute density further.
— NEM-GNN outperforms ReFLIP by ~80–230x, ~80–300x, ~850–1,134x and ~7–8x in terms of performance, throughput, energy efficiency, and compute density.

2 BACKGROUND

2.1 Graph Neural Networks

Graphs, characterized by nodes and edges, have been a traditional and crucial data structure to represent unstructured data across a diverse range of real-world applications. In the pursuit of facilitating classification and clustering of graph nodes, considerable research effort has been devoted to GNNs. These networks are designed to extract distinctive features from graph structures and to enable an understanding of relationships among nodes within a graph. A variety of GNN variants, including **Graph Convolutional Networks (GCNs)**, **Graph Attention Networks (GATs)**, and GraphSage [24, 25] are being researched extensively. These explorations are geared toward unraveling specific attributes of interest in various domains such as social network modeling, molecular chemistry, and citation networks. A GNN model is composed of multiple layers. In each GNN layer, every node from the input graph undergoes processing by two kernels: the combination and the aggregation kernel.

The function of the combination kernel is to convert the feature vectors (H), which characterize individual nodes in the graph, into an equivalent vector. This process entails the **Multiplication and Accumulation (MAC)** of the H values associated with each node in the graph using a weight matrix, akin to what is seen in a fully connected layer of a traditional neural network. The formulation of this kernel remains constant across all GNNs within the l^{th} layer and is expressed generically for the n^{th} node as $\mathbf{H_{comb}[l][n] = H_{layer}[l\text{-}1][n]^{*}W[l]}$. When l = 1, the $H_{layer}[l\text{-}1]$ becomes the same as the input feature vector. For other layers, it is the final output of previous layer, similar to a traditional CNN. Across all nodes, feature vectors can be concatenated to form the feature matrix $H_{comb}[l] = \{H_{comb}[l][n]\}$.

The aggregation kernel combines attributes from neighboring nodes, represented by H, to gain insight into interactions with adjacent nodes and to mitigate data irregularities, by averaging across nodes [14]. This process varies across different GNN models. The aggregation mechanism for the l^{th} layer is denoted as $\mathbf{H_{agg}[l] = M^{*}H_{comb}[l]}$, with the matrix M being contingent upon the specific GNN model in use. The aggregation mechanisms associated with different GNN models are as follows.

For *GCN*, $M = D^{-1}{*}A$, where D is the degree matrix normalized over the node degree, typically represented as a diagonal matrix for easy matrix multiplication, and A is the adjacency matrix (Figure 1) across all neighbors. The adjacency matrix includes a self-loop, which results

in the diagonal elements being set to 1. Similarly, the degree matrix is a diagonal matrix where $D_{ii} = \Sigma A_i$.

As an example of GCN, in Figure 1, a depiction of a graph with five nodes in an unweighted, undirected setup is shown. Each node contains three features, resulting in a feature matrix size of 5*3. These features undergo a transformation using a weight matrix, akin to neural network convolutions. The weight matrix (combination kernel) is sized as 3*3, leading to a resultant combination matrix of 5*3. Subsequently, aggregation follows, employing kernels consisting of the degree and adjacency matrix. For example, the feature vectors of node 1 (H11, H12, H13) transform based on the weight matrix. Node 1 has three adjacent nodes, accounting for self-looping. Consequently, $D^{-1}{}_{11}$ becomes 1/3. The aggregated output for node 1 is a scaled sum of combination vectors of nodes 1, 2, and 5.

In the case of *GraphSage*, $M = D_{samp}{}^{-1} * A_{samp}$, where D_{samp} and A_{samp} denote the sampled version of the degree and adjacency matrices, respectively. In contrast to the approach of aggregating information from all neighbors, as seen in GCN, GraphSage exclusively employs a subset of chosen or pre-trained neighbors associated with a specific node for aggregation purposes. This selection process aims to ensure a consistent and predetermined count of neighbors for all nodes. Consequently, this strategy introduces a sense of relative preference among neighbors for aggregation.

For *GAT*, M = (Attn)*(A), where Attn is the attention matrix. Unlike GCN and GraphSage, Attn relies on the combination vector of a node to determine the weight corresponding to each neighboring node. The Attn vector for a particular node is obtained by (i) defining attention between the i^{th} and j^{th} neighboring node as $e^{a*(ReLU(H_{comb\text{-}i}||H_{comb\text{-}j}))}$, where a is a trained vector and || indicates concatenation of combination vector of both the nodes, and (ii) normalizing the attention across all elements in a row of the Attn matrix to refrain from numerical explosion.

ReLU is used as the activation function at the output of the aggregation layer to introduce nonlinearity. Finally, the softmax function is used for classification into different categories.

2.2 Processing in/Near-Memory

CPUs and GPUs have long served as the primary workhorses for various user applications, including GNNs. However, they face increased data movement costs due to their von Neumann architecture, requiring periodic data transfers between memory and computational units. PIM aims to embed computations within memory, mitigating these costs. The choice of memory technology in PIM designs is crucial, with many GNN accelerators favoring ReRAM. However, ReRAM-based accelerators are often dedicated solely to GNNs, leading to increased dead silicon area when integrated into existing SOCs crowded with multiple accelerators. Understanding ReRAM characteristics is essential to grasp the drawbacks of these designs.

2.3 ReRAM Bitcell

This non-volatile memory element operates by varying resistance, unlike traditional charge-based memory like **Static Random Access Memory (SRAM)**. It toggles between low-resistance (SET) and high-resistance (RESET) states, akin to '1' or '0' in SRAM/**Dynamic Random Access Memory (DRAM)**. The memory bitcell consists of 1T1R (1-Transistor 1-Resistor), wherein writing involves activating the corresponding **Word Line (WL)** and applying voltage to the **Bit Line (BL)** for SET/RESET. Reading relies on current flow, low for RESET and high for SET states. Despite their compactness, ReRAMs suffer from limited endurance, higher voltage requirements, latency, and susceptibility to process variations [13, 17, 19]. However, there are many drawbacks (which are overcome by the SRAM bitcell), and their compact nature is a driving factor [3, 16, 18].

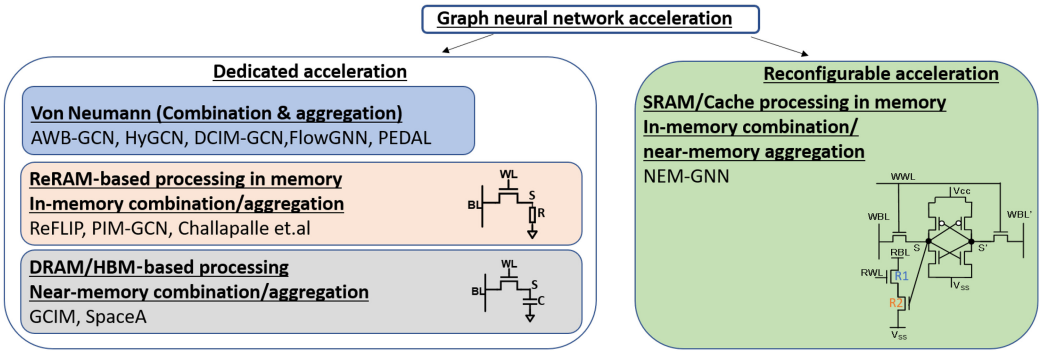


Fig. 2. Landscape of GNN-based acceleration. The prior works are predominantly dedicated accelerators requiring periodic host-accelerator interaction. These are further classified into von Neumann, ReRAM-based PIM, and DRAM/HBM-based PIM. The proposed accelerator is not dedicated and reuses cache in CPUs to perform GCNs. The bitcells for PIM designs are also shown.

2.4 SRAM Bitcell

SRAM stands out in cache/register file design for its ultra-low access latency (in the nanosecond range), surpassing other memory technologies [12]. Designs typically feature either a decoupled read/write port structure, seen in the 8T (8-Transistor) SRAM, or a shared read/write port structure, as seen in the 6T (6-Transistor) SRAM. The 8T SRAM employs the **Write Word Line (WWL)** for writing and the **Write Bit Line (WBL)** to store data in the storage node (S). During a read operation, the **Read Bit Line (RBL)** is precharged to a high voltage. If the S node holds '1', R2 is activated, discharging RBL through the R1-R2 stack. For a '0' in the bitcell, precharged RBL remains unchanged, aiding content distinction. The 8T SRAM performs efficient RAW (read-after-write) computations, unlike the typical 6T SRAM that necessitates write, precharge, and read commands, making RAW a three-cycle operation. For 8T SRAMs, precharge is overlapped with the write command, thereby making RAW a two-cycle operation, because RBL can be precharged, while a different row of bitcells is written using a combination of WWL/WBL. Additionally, these have lower write/read voltage/power as opposed to ReRAM, and resilience to process variations, thus giving performance and power advantage [23].

2.5 DRAM Bitcell

DRAM serves as the main memory storage due to its low cost and compact bitcell design. The 1T1C bitcell (Figure 2), utilizes a capacitor to store information. The write operation involves activating WL while storing data onto the 'S' node by driving BL. Precharging the BL to half of the operating voltage precedes the read operation, where the voltage at BL changes based on the bitcell contents. The advantages of DRAM lie in its compactness, cost-effectiveness, and high capacity. Additionally, its capacity can be augmented through 3D stacking, enabling **High-Bandwidth Memory (HBM)/High Memory Cube (HMC)** designs. DRAM faces challenges such as periodic refresh due to the capacitor's dynamic nature, resulting in performance/energy overhead.

2.6 Related Work

The existing GNN accelerators are classified into von Neumann/PIM-based dedicated accelerators. Von Neumann based dedicated architectures like AWB-GCN [7], HyGCN [28], DCIM-GCN [15], PEDAL [5], and FlowGNN [21] make use of a dedicated accelerator, specifically designed for GNN. The logic units are present outside the memory for performing combination and aggregation.

AWB-GCN introduces hardware-level enhancements aimed at addressing the challenge of workload imbalance, a consequence of the interaction between sparse input graphs and dense weight matrices. The proposed hybrid architecture employs distinct storage units for the combination and aggregation stages. This is complemented by specialized control logic that facilitates the seamless distribution of workloads across different processing engines. However, it is important to note that this strategy brings about an additional area overhead and offers only a modest assurance of optimal resource utilization. Moreover, the scalability of this solution is constrained by the number of processing elements. This is geared toward optimizing GCN predominantly.

HyGCN introduces an architectural approach and programming paradigm that harnesses both intra-vertex and inter-vertex parallelism during the combination and aggregation phases, thereby enhancing overall performance. Nevertheless, it is worth noting that despite these improvements, HyGCN lacks an inherent awareness of sparsity, leading to less efficient utilization of hardware resources and consequently resulting in an additional area overhead. Moreover, the inherent limitations of the von Neumann architecture become evident as graph sizes grow, contributing to the energy overhead caused by frequent data transfers between memory and computational units.

DCIM-GCN [15] employs memory as a storage component and supplements it with NOR gates/logic in proximity to the memory for near-memory combination. The aggregation, however, utilizes the von Neumann architecture and is specifically optimized for GCN.

PEDAL [5] introduces a power-efficient dataflow accelerator that optimizes dataflow based on the incoming graph's nature, enhancing efficiency and flexibility, by changing the order of execution between combination and aggregation. It employs a dedicated von Neumann architecture, necessitating periodic CPU-accelerator interaction.

FlowGNN [21] proposes a generic accelerator with parallelism ranging from node/edge, using multiple data queues and execution units along with scatter/gather mechanisms. The dataflow is general and can be applied to any graph-based model like GCN, GAT, and GraphSage. This architecture is realized on an FPGA, similar to other von Neumann accelerators and suffers from similar drawbacks.

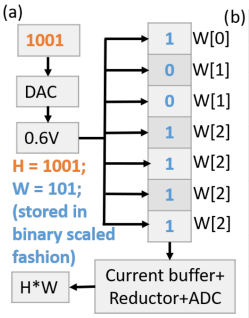
PIM-based accelerators predominantly have utilized ReRAM/DRAM for in-memory processing. ReFLIP, PIM-GCN [29], and Challapalle et al. [3] present a PIM accelerator that utilizes a crossbar ReRAM architecture. *ReFLIP* adopts a weight stationary approach for executing dot product operations, and it includes a peripheral DAC/ADC to perform analog compute. However, there are notable drawbacks associated with this approach, which are detailed in Section 3.

PIM-GCN uses a similar approach to ReFLIP for performing MAC for combination, except that the aggregation is performed using a two-step process with the first step being a **Content-Addressable Memory (CAM)** to identify neighbors, and the second step involves performing MAC. The proposed approach aims to schedule node computations in a way that the inter-node parallelism is maximal.

Challapalle et al. propose multiple engines with an architecture similar to PIM-GCN, except that the presence of separate engines potentially aids performance, at the cost of power/energy. However, the architecture is limited to performing GCN and cannot be extended to GAT/GraphSage.

DRAM/HBM-based near-memory accelerators make use of compute units closer to DRAM. *GCIM* [4] uses a novel data-aware mapping algorithm to efficiently utilize near-memory (3D-stacked HMC) MAC units. *SpaceA* [27] uses a near-memory CAM/MAC structure in processing engines to enable graph processing and perform workload balancing by mapping different sparse features to different banks, and is optimized for general graph processing and not for GCNs.

(a)



(b)

Block	ReFLIP	Proposed
DAC/ADC	NEEDED	NOT NEEDED
Current buffer+reductor	NEEDED	NOT NEEDED
Binary scaled storage	NEEDED	NOT NEEDED
Accuracy degradation	YES	NO
Reconfigurable	NO	YES
Compute density	LOW	HIGH

(c)

Problems	Solution
Scalability – **Exponential data replication requirement**	No data replication using pre-compute/ECT
Accuracy/Throughput degradation – **DAC/ADC**	Bit-serial PIM schemes without DAC/ADC
Compute density – **Sparse PIM aggregation**	Sparsity, graph aware near-memory aggregation
Performance/Energy – **Serial execution of combination, aggregation**	Effective overlap using CAR and broadcast approaches

Fig. 3. (a) ReRAM approaches (i) use DAC for incoming H conversion to an equivalent analog value, (ii) store weights of GNN in a binary scaled fashion, and (iii) utilize current buffer+reductor to perform current-based summation and ADC to generate H*W. (b) Qualitative comparison between ReRAM approaches and NEM-GNN. (c) A summary of the identified issues and the proposed solutions.

3 MOTIVATION

In this section, we motivate NEM-GNN by highlighting the issues in existing PIM designs. The combination phase, predominantly consisting of convolution between weights and feature vectors, is naturally suitable for PIM compute. Furthermore, the dense weights are reused across various feature vectors, establishing the PIM-based combination as a weight-stationary approach. Aggregation is not amenable to PIM compute, as detailed later.

3.1 Issues in ReRAM-Based PIM for Combination

ReRAM device limitations are discussed in Section 2.3 and also detailed in the work of Boppidi et al. [1]. This section covers disadvantages, with specific reference to compute, as the existing ReRAM-based accelerators follow the same strategy for performing in-memory dot product compute as that of ReFLIP. Figure 3(a) illustrates dot product compute between H = 1001 and W = 101. The weight bits are stored in a binary scaled fashion: the 0^{th} bit is stored once, the 1^{st} bit is replicated twice, and the 2^{nd} bit is replicated four times. In the combination phase, the corresponding analog value for H is mapped onto the WL, and the current flowing through the BL is aggregated using a current reductor. This is then fed into an ADC to derive the equivalent digital value. First, the weights are stored in binary scaled fashion in memory. The incoming H value of 1001 is converted into an equivalent analog voltage of 0.6V by using DAC. This voltage then activates multiple bitcells in the same column, by mapping H onto WL. The current summation across all ReRAM bitcells in a column (BL) is then performed by a combination of the current buffer and reductor. This is then converted into an equivalent voltage, and fed into ADC, which outputs an equivalent digital value of 101101. NEM-GNN overcomes the usage of analog blocks (summarized in Figure 3(b)). The disadvantages of the compute strategy are as follows. First, the presence of ADC and using ReRAM-based compute implies that the existing SOC components cannot be reconfigured to realize ReFLIP and need a dedicated accelerator arrangement. Second, the binary scaled storage requirement requires that 2^n-1 rows are needed for storing an n-bit weight. This implies that the area of the memory array scales exponentially as the precision of weight increases, limiting scalability. Third, ADC should be extremely precise, as an 8-bit weight multiplied by an 8-bit feature vector (H) requires a 16-bit ADC. The process variations inherent to ADCs pose serious implications on the compute accuracy (see Figure 3(b)). Fourth, throughput is restricted by the number of ADCs per bank (1 in the case of ReFLIP). Furthermore, the ADCs and current reductor affect the energy of the design with an additional area overhead arising from bulky ADCs.

Fifth, compute density, quantified as the ratio of operations performed within memory per bitcell, serves as an indicator for gauging the efficiency of hardware resource utilization. In ReFLIP, for the dot product between an m-bit feature vector and an n-bit weight, the compute density equals $(m^*n)/(2^n)$, which needs to be improved substantially. These issues are summarized in Figure 3(c).

3.2 Issues in ReRAM-Based PIM for Aggregation

ReRAM-based PIM approaches perform both combination and aggregation in memory. However, aggregation is not really suitable for PIM computation for the following reasons. First, realizing the exponential compute required for aggregation, like in GATs, is not suitable for PIM. Second, aggregation in memory can be achieved by two means. The first option involves the usage of a separate memory array to store the resultant dot product from the combination phase, and the second option involves the reuse of the existing combination memory array for aggregation as well. The first option incurs additional area overhead for the associated memory array resulting in ineffective utilization of hardware resources, degrading the array density, and energy for compute. The second option improves the utilization of hardware resources at the cost of performance and additional control circuitry. Furthermore, this adds a degree of serialization between combination and aggregation, as aggregation can be initiated only after combination is complete. This causes a performance bottleneck with no overlap between combination and aggregation. Therefore, both of these approaches are inefficient in terms of performance, energy, and area. ReFLIP and PIM-GCN uses the latter, whereas Challapalle et al. use the former. Finally, the sparse aggregation compute, if performed in-memory, degrades the compute density, as most PIM computes are insignificant.

3.3 Issues in DRAM-Based Near-Memory Compute for GNN

DRAMs, designed for cost efficiency and expanded storage, face constraints when integrating near-memory processing, leading to reduced storage capacity [20]. Given that DRAM is a single-chip package from manufacturers and directly integrated into existing SOC architectures, the fixed area allowance must accommodate additional near-DRAM logic. Notably, Samsung's near-DRAM chip for ML acceleration halved its storage capacity to include per-bank near-memory logic [10]. Consequently, traditional DRAM-hits turn into (DRAM+PIM)-misses, causing performance dips in storage-intensive workloads. This issue persists even in HBMs, with smaller storage capacities (e.g., recent HBM3 of 24 GB vs. DDR4 of 256 GB) and reduced storage density due to added bulky near-memory digital logic to DRAM bitcell arrays.

From GNNs' perspective, large datasets can push DRAM capacities to their limits. Storing basic node information (excluding edges/weights) with an 8-bit representation for features in PubMed/Reddit datasets alone demands 0.4 GB/0.5 GB. Preserving DRAM capacity becomes crucial to avoid the consequential power/energy expenses. Reduced DRAM capacity might limit concurrent CPU applications and performance degradation in traditional CPU workload execution.

Samsung's demonstration focused on ML, optimizing for MAC operations. However, GNNs differ as aggregation navigates nodes, utilizing CAM operations, distinct from mere arithmetic operations. There are two approaches: (i) employing CAMs via traditional CPU, causing GNN execution degradation due to CPU-DRAM interaction, different from ML applications, and (ii) including extra CAMs, which would further reduce storage capacity. Considering XNOR computation area vs. FP16 addition, this logic could trim DRAM by 10%. For instance, a reduced 3-GB HBM-based PIM (originally 6 GB, halved for ML in the work of Lee et al. [10]), now reduced to 2.7 GB, pushes DRAM limits, potentially causing misses and impacting performance/energy for datasets. Moreover, PIM's 3.5x ML performance gain would see a 10% reduction. Moreover, SpaceA augments HBM with specialized processing engines, expanding bit-width, incorporating CAMs, and load queues. This adaptation reduces DRAM capacity, impacting both traditional CPU tasks and GNN performance.

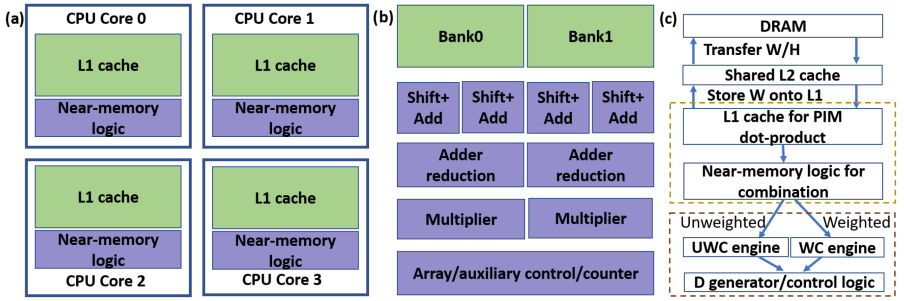


Fig. 4. (a) NEM-GNN is realized by repurposing the L1 cache for in-memory compute, with minimal near-memory peripheral logic added to each CPU core. (b) In an L1 cache, consisting of two banks, shift and add are present at a granularity of 1 per every eight columns per bank, with one adder reduction/multiplier per bank, and other dedicated logic shared across the entire cache. (c) DRAM is accessed to transfer weights/feature vectors into a shared L2 cache. The L1 cache stores weights. Combination datapath involves a PIM (L1 cache) dot product with near-memory logic. The combination result goes via the UWC/WC engine depending on the weighted nature of the underlying graph, followed by D-generator/control logic for aggregation. It is to be noted that UWC/WC engines use the added near-memory logic (multiplier/adder) and do not require dedicated hardware.

GCIM adopts 3D-stacked HMC, encountering issues akin to SpaceA, facing bandwidth constraints leading to Micron halting its production in 2018. To summarize, Compute-near-DRAM approaches reduce DRAM capacity, causing performance and energy bottlenecks with increased misses, particularly for high-capacity workloads.

4 NEM-GNN'S SALIENT FEATURES

4.1 Reconfigurability

The requirement of dedicated accelerator arrangement in previous works is overcome by reusing the L1 cache inside the CPU core for performing in-memory compute with additional near-memory logic. Figure 4(a) shows a multi-core CPU design with each core integrating dedicated minimal near-memory logic for performing combination/aggregation. Figure 4(b) shows the per-core additional near-memory logic for the L1 cache, assumed to be consisting of two banks for illustration. Shift and add are present at a granularity of 1 for every eight columns (assuming weights for 8 bits) for every bank. One adder reduction tree/multiplier per bank along with buffer/control/counter shared across the entire L1 cache is used to realize NEM-GNN.

The datapath is split into combination and aggregation. For the combination datapath, the compute array reuses the L1 cache to achieve PIM functionality with additional near-memory control logic for **Early Compute Termination (ECT)** and addition. For aggregation, we utilize a near-memory buffer for storing the adjacency matrix, a D generator for generating the corresponding degree (M) matrix, and **Unweighted Control (UWC)/Weighted Control (WC)** engines for graph/sparsity-aware aggregation. If the underlying graph is unweighted, the UWC engine is used for aggregation, whereas the WC engine is used for weighted graphs. It is to be noted that these engines reuse the added near-memory logic like multiplier/adder and do not require dedicated logic for their functionality. There is no requirement for ADC/DAC/specific memory technology, enabling integration of NEM-GNN into a traditional CPU pipeline. For GNNs that do not fit on-chip, DRAM is accessed to fetch data into L1/L2. The access latency of DRAM is amortized by prefetching data, as PIM accesses are deterministic.

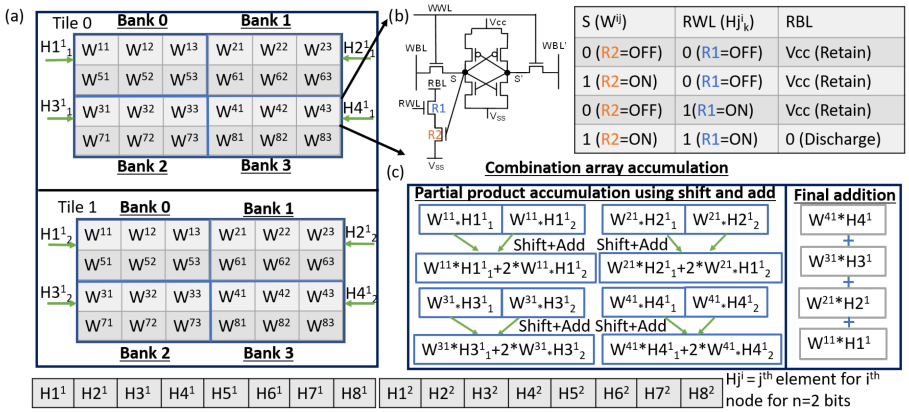


Fig. 5. (a) Compute array organization for NEM-C1: two tiles with four banks in each tile, with bit-serial PIM performed between H mapped onto RWL and W replicated across both tiles, are shown for illustration. Two-bit eight-element H and 1-bit 8*3 weight matrix are shown with $Hj^i{}_n$ indicating the n^{th} bit of the j^{th} element for the i^{th} node. (b) W is stored in an 8T SRAM bitcell in the L1 cache, and H is mapped onto RWL. RBL discharge is used as a measure of the dot product between W and H. RBL is initially precharged to Vcc and discharges only when W and H are '1.' (c) Shift and add across partial dot products from PIM compute, with the result of final addition written into a row of combination array.

4.2 Scalable L1 Cache Digital PIM Compute without DAC/ADC: Architecture to Circuit

Scalability is affected by exponential area requirement as the precision of weight increases. This is primarily because of the requirement for data replication across different banks. We propose three PIM approaches, which begin with n^2 data replication (NEM-C1), and further reduce it to no replication requirement (NEM-C2/C3), as detailed in the next section.

Our proposed designs implement a digital bit-serial PIM methodology for combination that eliminates the necessity for DAC/ADC. This setup proves robust against process variations, ensuring accurate computations. With the employment of bit-serial computation devoid of DAC/ADC, the design's throughput is freed from ADC-related constraints. The bit-serial computation, in conjunction with strategies like early computation termination or pre-computation, facilitates achieving high performance/throughput for NEM-GNN designs. Moreover, the absence of data replication combined with bit-serial computation contributes to an enhanced compute density, elaborated in Section 5. Furthermore, given that (i) ReRAMs suffer from scalability, compute density, and performance/energy issues and (ii) DRAMs suffer from reduced storage capacity, we propose L1 cache based PIM, which retains storage of DRAM, while offering improved performance/energy. From a memory organization standpoint, the L1 cache consists of multiple tiles, with each tile consisting of multiple banks, leveraging tile and bank-level parallelism (as shown in Figure 5(a)). These designs use decoupled read/write port structure (8T SRAM), with the decoupled read port transistors marked as R1 and R2 in Figure 5(b). The read port transistors are repurposed to perform dot product compute (bit-wise AND) for combination by mapping H bits onto RWL and storing weights in the bitcell. The compute can be described as follows. First, RBL is precharged to Vcc before compute. Second, only when both S and RWL are '1' does RBL discharge via the read-port transistors (R1/R2), implying that the computed value is 1. Third, RBL remains at Vcc for other combinations of H and W, because one of the read-port transistors is turned OFF. When W = '0,' R2 is turned OFF, as the S node is 0; when H = '0,' R1 is turned OFF, preventing discharge of RBL in either case. Using this PIM approach, we propose three different bit-serial scalable, performance, and compute

density optimized PIM approaches, namely (i) n^2 data replication, (ii) no data replication with ECT, and (iii) pre-compute, which are detailed in Section 5.

4.3 Graph and Sparsity-Aware Aggregation

We use a near-memory approach for aggregation, which enables performing exponential in-memory compute. The near-memory aggregation with CAR and broadcast approach helps achieve better performance in NEM-GNN, as aggregation latency is effectively hidden by computing as soon as the combination results are ready. This is done by effective broadcast of combination results, as soon as the combination outputs are available for aggregation.

Furthermore, the sparsity-aware approach helps alleviate insignificant computations while leveraging PIM's high throughput for performing significant computations. Graph connectivity-aware aggregation helps optimize aggregation based on different graphs, enabling achieving high performance, without area overhead, for graphs that fundamentally require less computation for aggregation. For instance, unweighted graphs require less computation as compared to weighted graphs, because of no requirement on multiplying with weight of the graph edge for aggregation.

4.4 No Impact on Traditional CPU Workloads

Since NEM-GNN is realized reusing the L1 cache, with additional near-memory logic, with no reduction in storage capacity, there is no impact on traditional CPU workloads. First, there are no SRAM memory array level modifications to realize NEM-GNN, which could potentially impact the execution of traditional CPU workloads. Second, there is minimal near-memory logic that is added to realize NEM-GNN, achieved through effective reuse of hardware. For instance, the UWC/WC engines share the same adder, D-generator shares the multiplier with UWC/WC engines, and softmax shares exponential compute, with aggregation for GAT. Third, the additional near-memory logic does not intervene in normal CPU operation and does not add additional power overhead.

5 NEM-C*: SCALABLE IN-MEMORY COMBINATION DESIGNS WITHOUT DAC/ADC

5.1 NEM-C1: Scalable Bit-Serial PIM with n^2 Data Replication

The exponential storage requirement (n-bit weight requiring 2^n bitcells, independent of H bit-width) in ReFLIP is combatted by bit-serial PIM, with n-bit weight requiring m*n bitcells for m-bit H, as detailed in the subsequent paragraph.

At the bank/tile level, the weights are kept stationary in the compute arrays, making the overall design a weight-stationary design. The weights are shown to be replicated for illustration of NEM-C1 design in Figure 5(a) (whereas other designs do not require data replication). An eight-element H vector each of 2-bits and 1-bit 8*3 weight matrix is shown with $Hj^i{}_n$ indicating the n^{th} bit of the j^{th} element for the i^{th} node. The observation is that during dot product compute between H for each node (feature vector) and weight matrix to output combination vector, the same H element ($H1^1$) is used across a row of weights (W^{11}, W^{12}, W^{13}). This reuse is used to map a row in the weight matrix onto a row in a compute bank, with H mapped onto the corresponding RWLs in a bit-serial fashion, enabling W^{xy} parallelism over the y-dimension ($H1^1{}_1$* W^{11} computed in parallel with $H1^1{}_1$* W^{12}). Inside a bank (e.g., in bank0), $H1^1{}_1$ and $H5^1{}_1$ are mapped in successive cycles onto the RWL as a row is computed per bank, per cycle. The weight matrix is split across banks in a tile, enabling $Hj^i{}_n$ parallelism over the j-dimension ($H1^1{}_1$*W^{11} computed in parallel with $H2^1{}_1$*W^{21}), and the weights are replicated across tiles for enabling $Hj^i{}_n$ parallelism over n-dimension ($H1^1{}_1$*W^{11} computed in parallel with $H1^1{}_2$*W^{11}). The replication of weights across multiple tiles ensures that the dot product between $Hj^i{}_n$ and W^{xy} is computed using $O(n^2)$ bitcells in a single cycle.

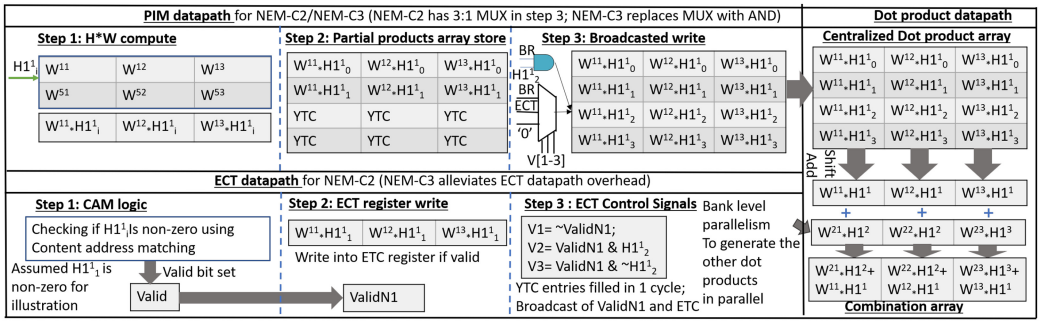


Fig. 6. *NEM-C2*: ECT occurs once one of the bit-serial H element bits is found to be 1, without a data replication requirement. The ECT datapath checks for a non-zero H bit in step 1 and writes the non-zero dot product into the ECT register in step 2. In parallel, the PIM datapath computes partial dot products in step 1 and subsequently stores them in the ECT register in step 2. This value is broadcasted to fill the other dot product entries in step 3 of the PIM datapath using ECT control signals generated in step 3 of the ECT datapath. A 3:1 MUX for writing into the partial products array: BR/ECT/'0' indicate bank read (computed value from memory)/read of ECT register/zero value, respectively. The dot product datapath fills the combination array with the result of combination. *NEM-C3* eliminates the ECT datapath by pre-computing for H bit being 0/1 and uses an AND gate (marked in blue) instead of 3:1 MUX in step 3, while using PIM and dot product datapaths, similar to NEM-C2.

This is illustrated by using an example in Figure 5. If the different bits of Hj^i are mapped onto RWLs of the same banks in different tiles, then bank0 of tile0 computes $W^{11}*H1^1{}_1$, whereas bank0 of tile1 computes $W^{11}*H1^1{}_2$. These two partial dot products can be shifted and added together to generate the dot product $W^{11}*H1^1$. Similarly, dot products obtained across different banks and tiles are shifted and added to obtain $W^{21}*H2^1$, $W^{31}*H3^1$, and $W^{41}*H4^1$. Finally, a full addition is performed to generate the MAC result for combination. A full adder accumulates the dot product from SA into a row of combination array (see Figure 5(c)).

The overall replication requirement is reduced to m*n for the dot product between m-bit H and n-bit W, resulting in $O(n^2)$ data replication. Furthermore, the compute density for the L1-cache compute is improved from $(m*n)/(2^n)$ in ReFLIP to $((m*n)/(m*n) = 1)$ in NEM-C1, indicating effective utilization of hardware resources. Moreover, the lower overhead of SAs compared to DAC/ADC, and the lesser number of RWLs to be turned ON (n^2 vs. 2^n) leads to reduced energy requirement.

5.2 NEM-C2: Scalable Bit-Serial PIM with ECT

NEM-C2 aims to eliminate the need for weight replication in the compute array by utilizing previously computed values. This aims to enhance the efficiency of bit-serial computation in terms of area, energy, and compute density. For example, when storing 3700*10 weights for a single layer of GCN in memory, where both feature vectors and weights are represented with 8-bit resolution, ReFLIP would demand about 1.2 MB of storage, NEM-C1 would need around 0.2 MB, and NEM-C2 would utilize only 4.6 KB for storing these weights.

Similar to NEM-C1, multi-bit weights are stored in a single row of the compute array, with elements of H mapped onto RWL in a bit-serial fashion, enabling W^{xy} parallelism over the y-dimension. Various banks serve to accommodate distinct weight rows, thereby permitting $Hj^i{}_n$ parallelism across the j-dimension. Furthermore, parallelism across the i-dimension extends across tiles. There is no requirement for data replication across banks, unlike NEM-C1.

Figure 6 illustrates the interplay among distinct data paths: PIM, ECT, and the dot product path. In step 1 of the PIM datapath, the "H*W compute" stage generates the dot product between a row

of weights and a bit of H element. Subsequently, in the second step of the PIM datapath, the resulting dot product between $H1^1{}_i$ and (W^{11}, W^{12}, W^{13}) is stored in the partial products array. It is notable that when one of the bits within the bit-serial H element is determined to be 1, it leads to the identification of both potential outcomes from multiplying a row by 0 and 1 (as multiplying by 0 would output 0). This knowledge can be leveraged to prematurely terminate the H*W computation, thus saving energy. The ECT datapath provides assistance to terminate compute early, by identifying the position of '1' in a bit-stream of H mapped onto RWL. The CAM logic allows for rapid searching based on content rather than memory addresses. This is utilized in step 1 of the ECT datapath in Figure 6 and is used to identify whether H-bit is '1.' A valid bit is set if a non-zero element is found. The associated dot products between the non-zero $H1^1{}_1$ and (W^{11}, W^{12}, W^{13}) are written into the ECT register in step 2 and the partial products array in parallel. Upon the completion of the ECT write operation, the remaining elements of the partial products array (denoted as YTC (Yet to Compute)) are promptly filled by employing a single-cycle, broadcasted write using the data contained within the ECT register. This broadcast begins with the generation of ECT control signals in step 3 of the ECT datapath, using the values of H elements and valid bit indication from CAM logic. Specifically, "BR" is chosen if the valid bit is '0,' "ECT" is chosen if both the valid bit and the corresponding H bit are '1,' and '0' is chosen if the valid bit is set while the corresponding H bit is '0.' The broadcast is done in step 3 of the PIM datapath, using three write-ported partial products array with three ports being bank read (when CAM logic has not detected a 1 in H element yet), ECT read (when ECT register is filled), and '0' (when the H element bit is 0). Finally, the partial dot products are accumulated from the dot product array, onto the combination array, which is shown in the dot product datapath.

The broadcasted write turns off the compute array earlier, thereby improving the energy of the system. No weight replication for a bit-serial PIM approach improves the PIM compute density to ((m*n)/n = m) along with reduced area requirements.

5.3 NEM-C3: Scalable Bit-Serial PIM with Pre-Compute

NEM-C2 introduces an ECT datapath, which can be mitigated by leveraging the fact that all H bits are available from the storage array read. NEM-C3 tries to eliminate ECT overhead and enhance performance, by virtue of increased pre-computation.

Similar to NEM-C2, multi-bit weights are stored in a single compute array row, with H mapped onto RWL in a bit-serial fashion. Bank/tile-level parallelism for parallelism across the j/i-dimension in $Hj^i{}_n$ is leveraged, without requiring data replication (as shown in Figure 6).

During the computation of dot products, NEM-C2 awaits the first occurrence of '1' in the H element before terminating the compute. In contrast, in NEM-C3, the computation terminates even earlier by pre-computing dot products corresponding to both '1' and '0' H-bits. This approach obviates the need for the ECT datapath while still utilizing the existing PIM and dot product datapaths to implement NEM-C3. The first two steps of the PIM datapath remain the same as NEM-C2. However, in step 3, both dot products are broadcasted to fill the partial-products array based on the H bit (shown as blue AND in Figure 6), eliminating the overhead of ECT. This helps reduce three write ports (3:1 MUX) on the partial products array to a simple logical AND operation between the bank read and the bit for the row in partial products array ($H1^1{}_2$ for the 2^{nd} row). This AND operation effectively signifies whether the array should be populated with 'zero' or the actual H value, depending on whether the H-bit mapped onto the RWL is '0' or '1.' The dot product datapath itself remains unchanged from NEM-C3.

This results in overall area/power reduction, improved compute density due to ECT elimination, and reduction in write ports. Furthermore, the pre-compute strategy helps with achieving improved performance and energy, as the compute can be terminated earlier than NEM-C2.

6 GRAPH AND SPARSITY-AWARE NEAR MEMORY AGGREGATION

Aggregation involves a two-step procedure, with the first being the multiplication of the resultant combination output with the adjacency (A) matrix, and the second being the multiplication of the outcome from the first step with the D matrix (assuming GCN for simplicity of explanation). In the context of the first step, which is carried out in the WC/UWC engine, the computation is both graph-aware and sparsity-aware. The graph structures are intelligently optimized and distributed across different engines based on connectivity patterns, while the computation is made aware of matrix sparsity to mitigate unnecessary operations involving the A matrix. Additionally, we introduce two approaches: the CAR strategy and "broadcast" technique. These techniques help overlap aggregation with combination, enabling resource reuse and concurrent processing. Moving on to the second step, this operation is executed within the D generator, and the process adopts a "sparsity-aware" approach. This technique strategically reduces the number of computations by accounting for the matrix's sparse characteristics.

6.1 Graph and Sparsity-Aware UWC Engine for Unweighted Graphs

While GCNs find application across various graph types, their most frequent utilization occurs with unweighted graphs, notably in citation networks like CiteSeer and Cora. We enhance the efficiency of these specific graphs by skillfully mapping them onto the UWC engine. This mapping capitalizes on inherent graph patterns and reusability, effectively minimizing the hardware demands.

Several key challenges are evident from prior approaches. First, in the context of ReRAM-PIM-based aggregation, the PIM array computes partial dot products, necessitating an additional array (referred to as the "aggregation array") to store these partial dot products before they are collectively accumulated. This requires increased area. Second, ReFLIP employs the same PIM array for both combination and aggregation, leading to a situation wherein aggregation only commences after the combination process concludes. Third, the process of writing the resultant combination array onto PIM array before aggregation introduces an added energy cost. Contrasting this, NEM-GNN adopts a distinct strategy. It does away with the requirement for a PIM array dedicated to aggregation, instead using the "aggregation array" to initiate aggregation as soon as a fresh entry is introduced into the combination array. This facilitates the concurrent execution of aggregation and combination, resulting in improved performance, while eliminating the need for power and area overhead associated with PIM array write and read operations.

For aggregation in undirected, unweighted graphs, we begin by reading the adjacency vector (stored in compressed sparse row format (CSR)) for the node undergoing combination (indicated by the "NodeProc" register) in Figure 7. In step 1, reading the adjacency vector highlights nodes that share an adjacency with the specific node in focus. The identified adjacent nodes, along with the "NodeProc" information, are stored within the Update Index register to serve as aggregation candidates. The aggregation array, with indexing based on the Update Index register content, undergoes immediate updates. This update involves the accumulation of the existing aggregation array values with the incoming combination vector through the utilization of a set of adders. This approach, characterized by simultaneous computation whenever data is ready, is known as the CAR strategy. The incoming combination vector is broadcasted to the aggregation candidates, allowing them to accumulate the combination vector concurrently. This mechanism facilitates "aggregation-vector" level parallelism. Figure 7 shows that node 5 is the NodeProc by the combination datapath. We identify that node 5 is connected to nodes 1 and 3 from the read-out of the adjacency matrix. Therefore, the Update Index register shows that nodes 5 (for self-loop), 1, and 3 are the aggregation candidates. Hence, the aggregation array entries of nodes 5, 1, and 3 are added with the incoming combination vector "C51, C52, C53" in step 2.

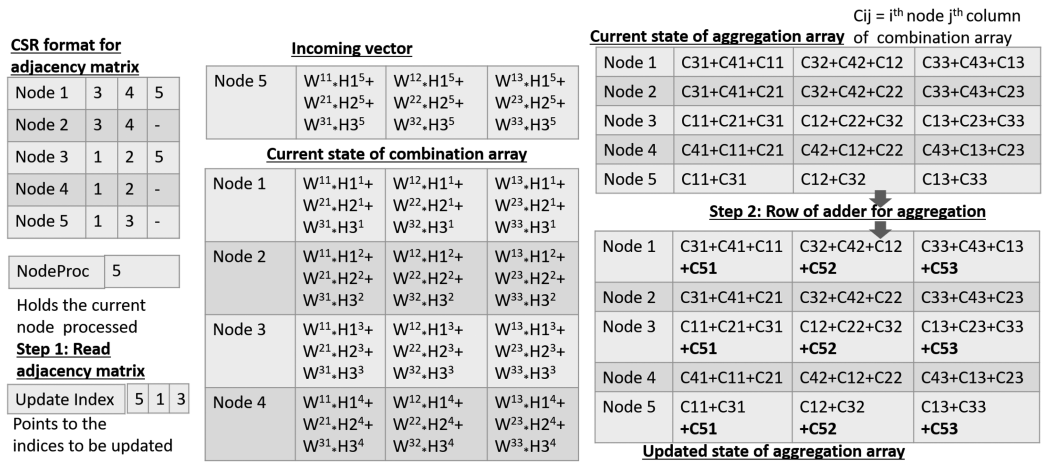


Fig. 7. Incoming graphs are mapped onto different engines based on graph connectivity (*graph-aware*) and read-out of the adjacency matrix (stored in Compressed Sparse Row Format) to eliminate unnecessary compute (*sparsity-aware*). *UWC engine*: Aggregation of unweighted graphs by reading the adjacency matrix and NodeProc register (indicating the node being processed by combination) to fill the update index register in step 1 and updating the aggregation array using adders in step 2. Node 5 (self-loop), node 1, and node 3 are aggregated with an incoming combination vector for node 5, by *broadcasting* the vector (C51) to the aggregation array. The aggregation array is updated immediately with C51 once the combination result for a particular node is available, enabling the CAR approach.

For the aggregation of directed, unweighted graphs, the adjacency matrix has an extra bit that shows the direction of interaction with the neighbors. '0'/'1' indicates an outgoing/incoming edge from/to a node. This is done so that only nodes having an outgoing edge from "NodeProc" are aggregated. Post-read-out of the adjacency matrix in step 1, CAM with the direction bit for the NodeProc is done to identify nodes that are to be aggregated in step 2 in Figure 8(a). We utilize "CAR" and "broadcast" to improve performance in NEM-C3 as well. When NodeProc = 5, since node 5 has an outgoing edge to node 3 alone, the combination vector for node 5 is added with entries corresponding to nodes 5 and 3 of the aggregation array, using adders in step 3.

This approach eliminates redundant multiplications between the incoming combination vector and the adjacency vector when nodes are not adjacent. This refinement makes the design sensitive to sparsity, meaning the aggregation process no longer includes aggregation with a "0" weight for non-neighboring nodes. In Figure 8(b), there is an evident overlap between aggregation and combination. This is achieved by concurrently reading the adjacency vector of the n^{th} node for aggregation while simultaneously calculating the dot product for the $(n+1)^{th}$ node's combination. This overlap continues with the generation of partial products and the filling of the combination array for the $(n+1)^{th}$ node, all while the aggregation process for the n^{th} node is under way. Due to the complete pipelining of both aggregation and combination, the latency introduced by aggregation is effectively hidden, resulting in improved performance.

6.2 Graph and Sparsity-Aware WC Engine for Weighted Graphs

For weighted graphs, the adjacency matrix (A) is repurposed to store the weight of interaction between two adjacent nodes. For compute of directed graphs, post-read-out of A in step 1, we find the outgoing nodes of the NodeProc in step 2 to find the update index and weights (indicated as Weight_G). Multipliers are used to perform the dot products between the incoming combination

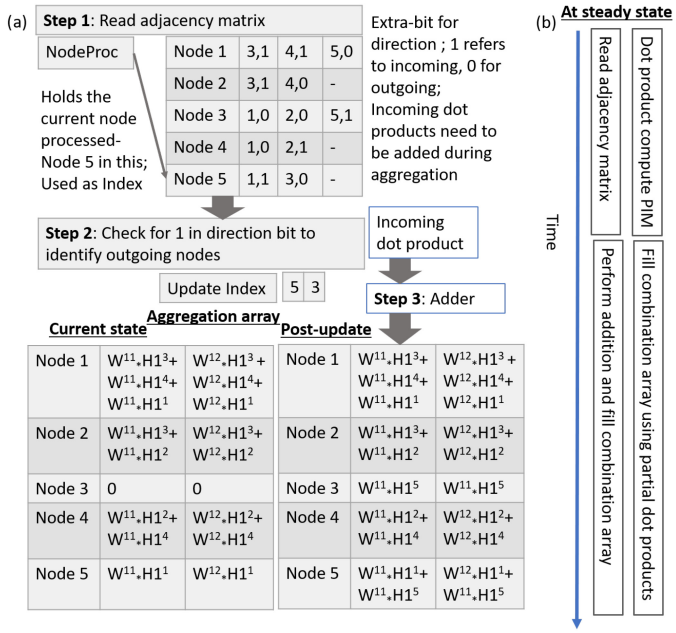


Fig. 8. (a) *UWC engine*: Aggregation for an unweighted, directed graph begins with reading the adjacency vector corresponding to Node Proc in step 1, identifying outgoing nodes in step 2, and storing in Update Index register, using adders to aggregate the incoming combination vector onto the nodes in Update Index register in step 3. Each adjacency matrix element is of the form (i,j), where i/j represents the neighboring node/direction of interaction with the neighbor. (b) Timing diagram to show that the dot product PIM for combination/read of the adjacency matrix of $(n+1)^{th}/(n)^{th}$ node overlap; filling of combination/aggregation array for $(n+1)^{th}/(n)^{th}$ node overlap, *hiding aggregation latency*.

vector and the weight corresponding to the edges. Similar to the unweighted graphs, the incoming combination vector is "broadcasted" to accumulate onto the aggregation array corresponding to the update indices using adders in step 3. Figure 9(a) shows that there is an outgoing edge of weight=7 from node 5 to 3, which is multiplied with the incoming combination vector and aggregated onto node 3. For compute of undirected graphs, the adjacency matrix has additional bits (see Figure 9(b)) for storing the weight of the edge with no peripheral direction detection logic. The adjacency matrix is read in step 1, to identify the adjacent nodes, which is followed by the "broadcasted" incoming combination vector using the "CAR" approach in step 2. Apart from the UWC engine's advantages, the WC engine consumes less power than ReFLIP, due to absence of power-hungry DACs/ADCs.

6.3 Sparsity-Aware D Generator and Control Logic

The key observation is that D^{-1} is sparse, and that multiplication of D^{-1} with aggregation array results in unnecessary computations. Therefore, we propose a *sparsity-aware approach* that consists of two main facets. First, the approach involves exclusively storing the non-zero elements, specifically the diagonal elements, of the degree matrix (D^{-1}). This curtails the required area by a factor of n. Second, multiplying the aggregation array with D^{-1} is realized as element-by-vector multiplication for every row, reducing the number of computations by a factor of 2n, for D^{-1} of n*n and the aggregation arrays of n*n. This is because when two arrays each of size n*n array are

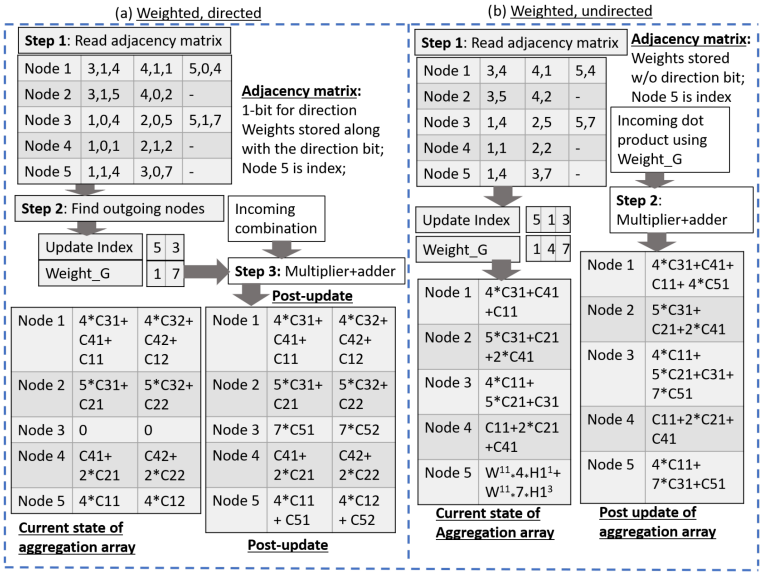


Fig. 9. (a) Weighted, directed aggregation, with the adjacency matrix storing the weights of graphs and the direction in the case of directed graphs. The direction is read out in step 1 to check for outgoing nodes in step 2, and aggregation with the incoming combination vector is achieved using near-memory multipliers and adders in step 3. (b) Weighted, undirected aggregation follows the same datapath as the directed one, but without the notion of direction, making it a two-step operation.

multiplied with each other, the total number of MAC operations would be ~ 2*n*n*n, assuming one operation each for multiplication and addition. However, the proposed approach necessitates only n*n multiplication operations without any accumulation operations. This is achieved by capitalizing on the characteristic of multiplication of a diagonal matrix (D^{-1}) with an adjacency matrix. This involves multiplying each row in the adjacency matrix with the corresponding non-zero element in the same row of D^{-1}, thereby converting a vector-by-vector product to element-by-vector multiplication. Figure 10 shows that $D^{-1}{}_{11}$ (1/3) (element) is multiplied with the first row of the aggregation array, instead of the first row of D^{-1} (like in ReFLIP). Furthermore, during compute, the degree matrix is generated in parallel with the first step of aggregation, followed by multipliers that perform multiplication between the non-zero D^{-1} element and result of aggregation.

The advantages of this approach are that (i) D matrix is reused across different layers and (ii) D-generator is power-gated, once D-matrix is computed the 1st time. This helps achieve improved power/performance while reducing the number of unnecessary sparse computations. The auxiliary *control logic* consists of (i) ReLU to identify the sign of elements and update the final aggregation array, and (ii) softmax control logic employing exponential/summation to generate classification output.

7 ISA SUPPORT

NEM-GNN's reconfigurability (L1/L2 cache used for compute/storage) helps repurpose CPU instructions like load/store to read/write data onto the L1 cache for compute. Since the L1 cache can operate in two modes (i.e., normal and compute mode), LCONF instruction configures L1 in compute mode by programming a special purpose register. Additional instructions for performing combination/aggregation are proposed. The instruction MACC Vx, VH, VW carries out an

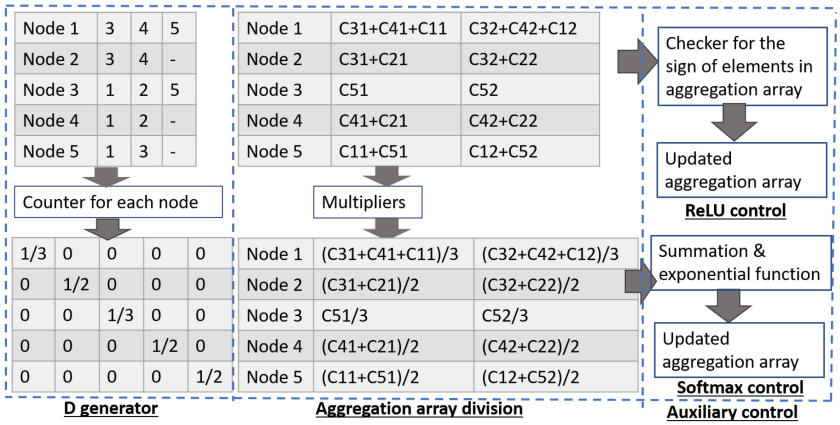


Fig. 10. *D-generator and control logic*: Degree matrix generator for generating D^{-1} using a *sparsity-aware* approach that (i) performs element-by-vector (instead of vector-by-vector) multiplication for every row and (ii) reduces the number of computations/area by a factor of 2n/n. Auxiliary control for ReLU and softmax is shown in the rightmost figure.

Software support (Datasets/ testing networks)	Dataset - #nodes/edges/features/ Storage requirement / Operations CS – Custom datasets with weights in graphs for testing WC engine	**Citeseer (CS)** –3,327/9,104/3,703/40MB/2.3M **Cora (CA)** – 2,708/10,556/1,433/16MB/1.3M **Pubmed(PD)** - 19,717/88,648/500/41MB/18.6M **Nell (NL)** – 65,755/251,550/5,415/1.6GB/782M CS1 – Weighted+directed CiteSeer CS2 – Weighted+undirected CiteSeer CS3 – Unweighted+directed CiteSeer
	GCN/GAT/GraphSage network	2-layer combination+aggregation; #hidden dimensions=128

Micro-architecture	Compute array	Tiles- 32; Banks – 256 Array Size – Upto 4KB
	Combination/ aggregation arrays	Combination – 128*32 flops Aggregation – Banked flop array (256*128*32) entries
	Area (mm^2)	Combination: SRAM – 4; Near-memory combi – 3, ECT – 1 Aggregation: UWC – 6, WC – 7.5; Aux logic – 2;

Fig. 11. *Benchmarks*: Datasets for GNNs, the number of nodes/edges/features in each of them, and the network used for GCN/GAT/GraphSage networks. *Micro-architecture* of NEM-GNN with the additional near-memory logic requiring 2% of AMD's Zen3 CPU per-core area.

in-memory dot product, followed by near-memory accumulation. This process accomplishes combination between a row of weights (VW) and a feature vector (VH), ultimately storing the outcome in Vx. MACA Vagg, Va, Vx performs near-memory aggregation using the incoming combination vector (Vx) and adjacency vector stored in Va to store the aggregation result in Vagg.

8 EXPERIMENTAL METHODOLOGY

Benchmarks. A GNN model consisting of two graph convolutional layers with 128 hidden dimensions, similar to that of ReFLIP/AWB-GCN [7, 8] is used as the underlying network to ensure a fair comparison between the proposed and existing designs. The quantization of GNN network weights, feature matrix, and weights of the input graphs (for weighted graphs) is done using PyTorch. The evaluated datasets and their properties are tabulated in Figure 11.

Microarchitecture. The compute array, repurposing the 8T SRAM L1 cache, is organized as 32 tiles/256 banks, matching the L1 cache size of an Intel Xeon E5-2680 v3. In the following, we integrate NEM-GNN's near-memory architecture to Intel Xeon E5-2680 v3's architecture (wherever available) to ensure fair comparison, with a banking strategy. The combination and aggregation arrays are implemented as registers, with the size of the combination array accommodating 128 W*H dot products each of 32 bits. The aggregation array is organized as banks, with banks differentiated by node number (e.g., Bank0: 0–255 nodes, Bank1: 256–511 nodes). The near-memory

logic like shifters/adders are present at 1 per eight columns and adder reduction/multiplier at 1 per bank.

Performance/Power/Area Evaluation. Microarchitecture of *digital components* is specified using System Verilog, and synthesized with FreePDK 45-nm [22] technology, operating voltage of 1 V, and clock latency of 2 ns to identify the power and area tradeoffs. For the SRAM compute array and its peripheral circuits, power and area are estimated using Cadence Virtuoso. *SRAM compute* energy is measured using an SRAM array of size 32*256, with the sense amplifier offset voltage set to 100 mV, RBL capacitance of 35 fF, as measured from layout extraction with read/write latency of SRAM ∼2 ns, and operating voltage of 1 V. We use a custom performance simulator that accounts for the microarchitecture and the workload, to quantify the performance tradeoffs for combination and aggregation. For large-sized graphs not fitting on-chip, the overheads of (i) write latency of compute array is amortized by leveraging separate decoding circuits for write and read logic in 8T SRAM on the periphery, allowing write onto the n^{th} row, when the m^{th} row is computed, and (ii) data movement from DRAM to storage/compute array is amortized by data prefetching. The data movement energy is 1 pJ/bit (∼800x addition energy) [11].

Comparison Methodology. NEM-GNN is compared against PyG-CPU and PyG-GPU (software optimized frameworks of PyG on CPU/GPU), AWB-GCN (non-PIM hardware accelerator), and ReFLIP (PIM hardware accelerator). *PyG-CPU* is PyG [6], a Python-based GCN-optimized library implementation of the GCN network on an Intel Xeon E5-2680 v3, with 12 cores per socket and operating frequency of 2.5 GHz, with an L1 cache size of 64 KB, an L2 cache size of 256 KB, and an L3 cache size of 2.5 GB, along with DDR4 capacity of 256 GB. Similarly, *PyG-GPU* is implemented on an NVIDIA Tesla v100, with 64 CUDA cores per streaming multiprocessor and an operating frequency of 1.5 GHz, with a 96-KB L1 cache per streaming multiprocessor, a 6-MB L2 cache, and a 16-GB HBM2. *AWB-GCN*'s performance is obtained from its implementation on an Intel D5005 FPGA with DRAM capacity of 32 GB and 4096 PEs, with frequency of 0.3 GHz [7]. The performance of *ReFLIP* for combination is dependent on (i) the write latency of ReRAM (50.88 ns [8]), (ii) the number of banks, and (iii) the number of dot products computed per bank (16,384), limited by the number of DACs/ADCs per bank—1 per bank in the work of Huang et al. [8]. Aggregation needs to wait until combination completes and is further split into (i) A and (ii) D^{-1} matrix. Multiplication with A and D^{-1} needs to be done serially in a PIM array in GCN/GAT, and multiplication with D^{-1} cannot be done in a PIM array for GraphSage because of the exponential function. The overall clock latency is limited by write latency of 50 ns and operating voltage of 1 V. DRAM prefetching is assumed for large-sized graphs, even in ReFLIP (for fairness). Power is obtained for (i) PyG-CPU using power-stat, (ii) PyG-GPU from NVIDIA's system management interface, (iii) using the same configuration mentioned in the work of Geng et al. [7] for AWB-GCN with rebalancing/distribution smoothing, and (iv) using the power estimated in the work of Huang et al. [8] for ReFLIP for DAC/ADC/ReRAM. The additional write costs onto PIM for aggregation, and data movement costs for large-sized graphs are also included. Energy is identified by multiplying the execution time with the power. For NEM-GNN, UWC/WC engine uses unweighted/weighted graphs for aggregation, and NEM-C1,NEM-C2,NEM-C3 are for combination.

9 RESULTS

Performance. The performance of NEM-GNN (Figure 12) is better for *GCN* than PyG-CPU because the cost of the increased data movement with increased graph size in CPUs is amortized by performing PIM. For PyG-GPU, the irregular memory accesses and limited on-chip memory restrict the performance of the GPUs for larger workloads. For smaller workloads, the ineffective utilization of GPUs due to sparsity limits the performance of GPUs. This is reflected by the increased speedups, as high as $\sim 10^4$ for PubMed (large graph) and $\sim 10^3$ for Cora (small graph). To

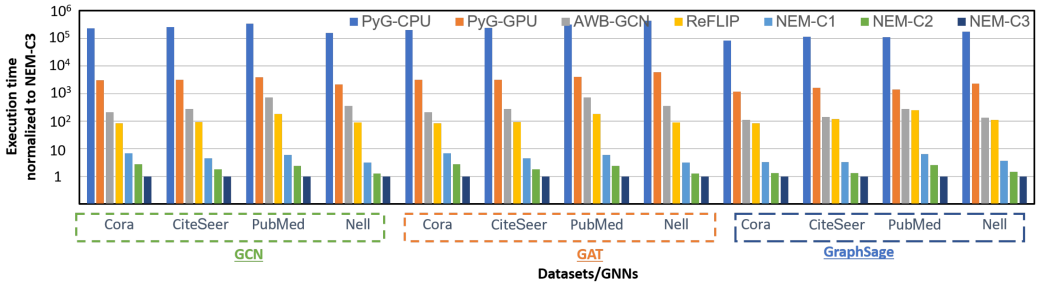


Fig. 12. Performance comparison normalized to NEM-C3 for GCN, GAT, and GraphSage. The UWC engine is used for aggregation, and NEM-C1, NEM-C2, and NEM-C3 are used for combination.

summarize, concerning CPUs/GPUs, apart from in/near-memory compute, ECT/pre-the compute strategy, along with hiding latency using CAR and broadcast approaches for aggregation, helps achieve improved performance for NEM-GNN. For AWB-GCN, performance is limited by stalls arising from dynamic adjustment of workloads among 4096 PEs (by performing runtime optimization) and growth in the number of off-chip memory accesses. Furthermore, the multi-stage network (omega) coupled with the control logic necessary for distribution smoothing leads to performance bottlenecks, as it limits the parallelism obtainable across PEs. In NEM-GNN, PIM enables fast memory accesses with high parallelism across all columns/banks, without additional control logic for distribution smoothing, resulting in a speedup of 10^4, compared to AWB-GCN in PubMed. In ReFLIP, the major performance limiters are (i) one DAC/ADC per bank limiting the number of dot products per bank to 1, (ii) serialization of combination and aggregation, and (iii) high access latencies of ReRAM limiting the cycle time for a single dot product compute. The major advantages of NEM-GNN are (i) for combination, the dot product between a row of weights in a bank and a single H element mapped onto the bank is computed in parallel; (ii) the CAR scheme hides the latency of aggregation completely, making the overall latency effectively the time taken to perform combination alone; and (iii) lower SRAM access latencies, resulting in speedups of ~80–200x in almost all workloads. In particular, NEM-GNN performs well when the ratio of the number of H to the number of nodes is low (observed in PubMed), as the parallelism across the number of Hs/nodes is dependent on the number of banks/tiles. Among NEM-C* designs, NEM-C1's performance is lower due to the limited parallelism across nodes, NEM-C2's performance is limited by the time to compute one node's combination array (as this is data dependent), and NEM-C3 combines the advantages of both NEM-C1 and NEM-C2 for better performance. For *GAT*, the number of computations is lower (assuming a fixed number of neighbors are predetermined), and NEM-GNN offers better performance (~75–140x) than ReFLIP. For *GraphSage*, NEM-GNN achieves speedups of ~230x.

Throughput. Throughput (see Figure 13) captures the scalability of different designs. PyG-CPU is limited by the number of execution units capable of performing dot products. PyG-GPU is limited by the ability to handle sparse data. AWB-GCN is limited by the number of processing elements and workload rebalancing. ReFLIP's scalability is limited by the number of DAC/ADCs per bank, serial nature of combination and aggregation, and data movement onto the PIM array for initiating aggregation. NEM-GNN overcomes these by performing bit-serial computation, complete use of available throughput from the different columns in a bank (parallel dot product between a row of weights and incoming H element), pre-compute/ECT, parallel combination and aggregation, and sparsity-aware compute leading to ~80–300x higher throughput than ReFLIP.

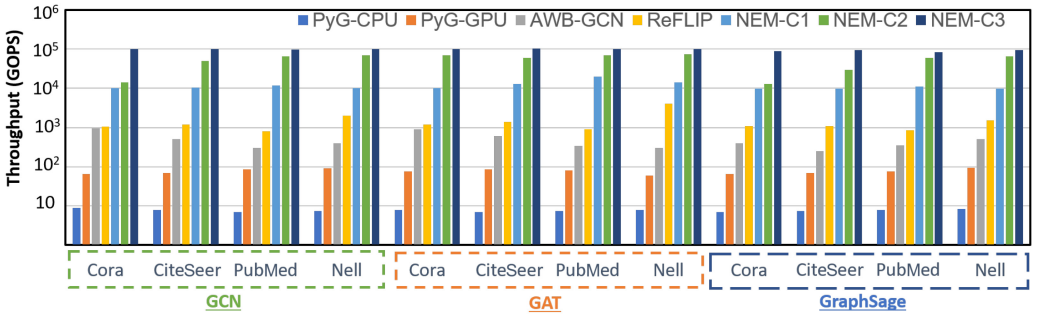


Fig. 13. Throughput comparison measured in GOPS for GCN, GAT, and GraphSage. The UWC engine is used for aggregation, and NEM-C1, NEM-C2, and NEM-C3 are used for combination.

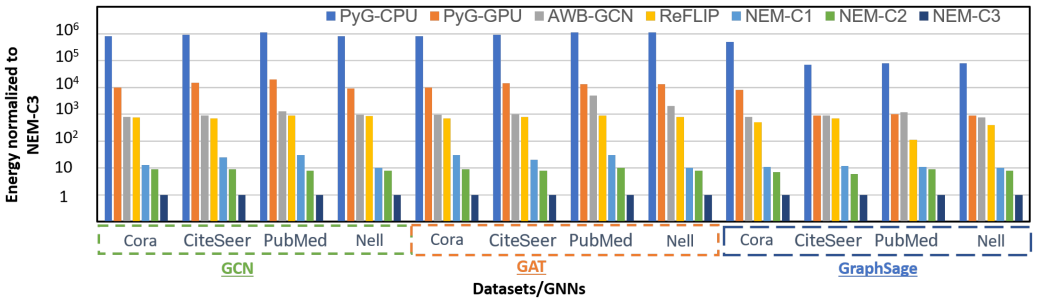


Fig. 14. Energy comparison for GCN, GAT, and GraphSage. UWC engine is used for aggregation, NEM-C1, NEM-C2, and NEM-C3 are used for combination.

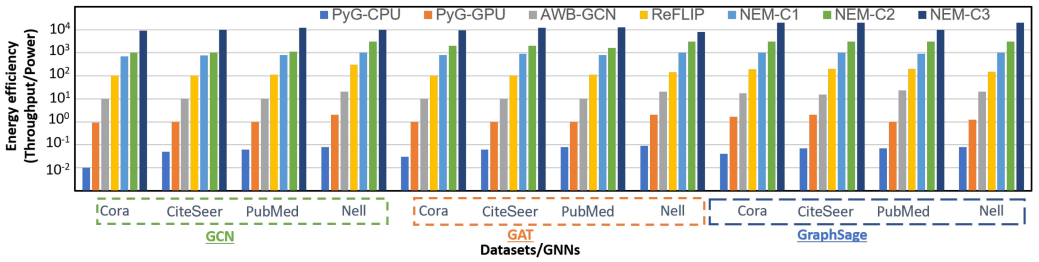


Fig. 15. Energy efficiency comparison for GCN, GAT, and GraphSage. The UWC engine is used for aggregation, and NEM-C1, NEM-C2, and NEM-C3 are used for combination.

Energy. The overall power of NEM-GNN is divided into the power for (i) combination control logic (ECT datapath, dot product datapath)/arrays (combination/dot product array), (ii) UWC engine (adder)/arrays (adjacency matrix, aggregation array), (iii) auxiliary control logic, (iv) SRAM power across banks/tiles, (v) dot product array and multipliers in the WC engine, and (vi) ECT datapath. The overall estimated power from (i) is ~3 W, (ii) is ~6 W, (iii) is ~1 W, and (iv) is ~2 W. Additionally, (v) and (vi) in NEM-C2 costs additional 1.5 W power (see Figure 12). Among NEM-GNN designs, NEM-C3 shows the least amount of energy (see Figure 12) because the improved performance compensates for the power overhead. NEM-C2 has a slightly higher energy because of the lower performance and increased power from the additional ECT datapath, as opposed to NEM-C3. Although NEM-C1 has the least power, the overall energy is higher because the decreased

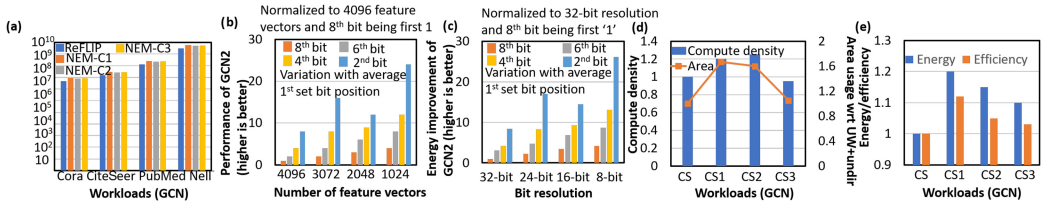


Fig. 16. (a) *Compute density* comparison across PIM designs. (b) *NEM-C2 performance* variation with number of Hs. (c) *NEM-C2 energy* variation with bit resolution and average bit-position for first '1.' (d) *Compute density* and *area* for CS1, CS2, and CS3. (e) *Energy* and *efficiency* for CS1, CS2, and CS3.

performance overcompensates for the lower power. In comparison to ReFLIP, NEM-GNN has the following advantages: (i) no power-hungry DAC/ADC requirementsn (ii) lower write/read voltages for SRAM than ReRAM, and (iii) no additional write required to store back into the compute array post combination resulting in energy improvements of 10^{2-3} in NEM-GNN (*GCN*). PyG-CPU, PyG-GPU, and AWB-GCN suffer from irregular memory accesses and frequent data movement, costing energy. Furthermore, there is no rebalancing of workloads like in AWB-GCN. In *GAT*, the energy improvement is higher because of the low power requirement in NEM-GNN, from lesser computations, along with the advantage from improved performance. In *GraphSage*, the power dissipated is higher (increased number of computations) with similar latency (latency of M matrix generation is completely hidden).

Efficiency. Efficiency is calculated by dividing the throughput by power (see Figure 12). CPUs have limited on-chip memory and throughput, with the least efficiency. GPUs have higher throughput with many on-chip processing cores and improved on-chip memory capacity, indicating better energy efficiency as opposed to CPUs. AWB-GCN's decreased throughput is compensated by better energy, improving the efficiency over CPUs. ReFLIP performs PIM-based compute with a higher degree of parallelism, improving throughput and energy. NEM-GNN has the highest amount of parallelism leading to increased throughput and decreased energy. Bit-serial PIM combination with pre-compute and near-memory aggregation results in ~850–1,134x improvement over ReFLIP.

Utilization. Figure 16(a) tracks the resource utilization efficiency in terms of the compute density, which is equal to the number of computations (dot products) per unit area. The overall additional near-memory logic area required per CPU core is tabulated in Figure 11 and sums to 0.47 mm^2, which is 2% of AMDs Zen3-CPU area. UWC and WC engines share most of their datapaths, thereby saving area. The NEM-C1+UWC/WC engine has an area requirement of 0.27 mm^2. The NEM-C2+UWC/WC engine has an additional 0.2 mm^2 due to the presence of the ECT datapath compared to NEM-C1-based design. NEM-C3 has negligible area increase due to the presence of an extra AND gate compared to NEM-C1-based design. The compute density is ~7–8x that of ReFLIP, due to the elimination of bulky DACs/ADCs, no data replication, and sparsity-aware compute.

Design Space Exploration. The performance of NEM-C2 varies roughly linearly with (i) the average position of the first '1' of the incoming H vector, and (ii) the number of Hs, as they determine the average time for combination per node and the required number of dot products (see Figure 16(b)). For energy (see Figure 16(c)), (i) lower bit-resolution causes fewer RBLs to discharge, implying less power (for a fixed number of nodes/Hs) and (ii) a decrease in the number of bits to obtain first '1,' implying better performance. Both of these factors combined show less energy for low resolution (e.g., 8-bit over 16-bit) and decreased number of bits to obtain the first '1' (e.g., 2-bit over 4-bit). The area (see Figure 16(d)) required for evaluating CS1, CS2, and CS3 is more than CS, because of the additional multipliers for processing weighted edges and the associated control logic for restricting the computation to outgoing edges. The compute density is the highest for

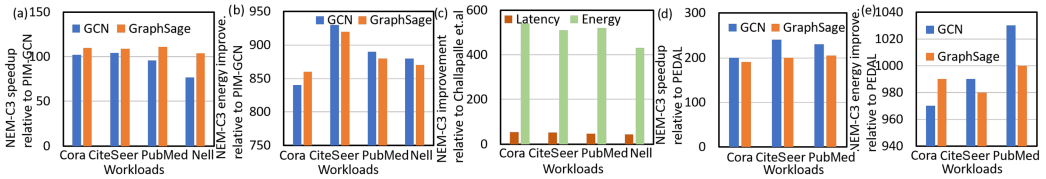


Fig. 17. (a, b) Performance/energy improvement of NEM-C3 relative to PIM-GCN. (c) Speedup/energy improvement relative to Challapalle et al. (d) Speedup. (e) Energy of NEM-C3 relative to PEDAL.

evaluating CS2, because increase in area is compensated by increased numbers of computations. Figure 16(e) shows that the energy (Power*Performance) for CS1 is the highest, as there is a power increase from the multiplier and control logic (even when MAC for weighted edges is performed in a single clock cycle). The energy efficiency is the highest for CS1, as the energy increase is compensated by the increased number of computations.

10 ADDITIONAL COMPARISON RESULTS

In this section, we compare NEM-GNN's design, utilizing NEM-C3 for combination and the UWC engine for aggregation, against other ReRAM-based PIM designs like PIM-GCN [29], Challapalle et al. [3], FlowGNN (state-of-the-art von Neumann accelerator), and PEDAL, focusing on GCN and GraphSage performance/energy metrics. As absolute throughput values are not provided, a direct energy efficiency/throughput comparison is not feasible, and GAT results are not included.

PIM-GCN demonstrates speedups compared to PyG-CPU running on an Intel Xeon E5-2680 v3 (the same hardware used in our simulations). Therefore, we employ the speedup/energy efficiency metrics reported in the work of Yang et al. [29] for comparison. Our focus is limited to GCN and GraphSage for PIM-GCN, as the execution methodology for GAT is not detailed in the article. While the disadvantages of ReFLIP are applicable to PIM-GCN, NEM-GNN designs achieve higher speedups. However, the achieved speedup is slightly greater compared to ReFLIP in smaller datasets like Cora/CiteSeer, mainly because PIM-GCN faces challenges in hiding additional latency for performing CAM to identify neighbors in the scheduling policy, whereas it performs better for larger datasets. This results in speedups of ~76–105x, as depicted in Figure 17(a). Similarly, in terms of energy efficiency, enhancements of ~840–940x are observed for GCN/GraphSage, as shown in Figure 17(b).

Challapalle's provided absolute performance/energy figures guide the comparison. Unlike ReFLIP, this architecture has distinct engines for traversal, combination, and aggregation, potentially enabling faster aggregation. However, NEM-GNN demonstrates speedups of ~45–53x and energy enhancements of ~430–570x, detailed in Figure 17(c). These gains stem mainly from high ReRAM write latency/power and challenges in completely concealing latency due to PIM compute. The combination engine must write results into the aggregation engine before in-memory computation, and all neighboring nodes' combination vectors must be available before initiating aggregation for a node. Otherwise, frequent data transfers between the main memory and ReRAM array incur power/latency costs. Moreover, lacking sparsity-aware compute, aside from CSR/CSC data representation, limits potential computational reductions, unlike NEM-GNN's approach, leading to increased energy consumption in the UWC engine. It is important to note that PIM-GCN and Challapalle et al.'s modeling does not take into account the additional data movement cost associated with movement from host CPU to ReRAM-based memory array, and that would further improve speedup/energy associated with NEM-GNN.

We compare using latency/energy data from FlowGNN. While FlowGNN introduces the capability to compute graphs with edge embeddings, its performance remains nearly on par with previous

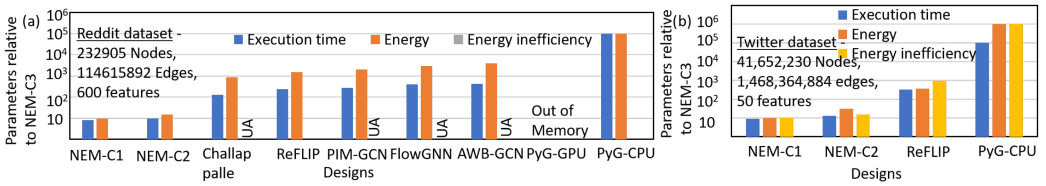


Fig. 18. Execution time/energy requirement/energy inefficiency of designs relative to NEM-C3 for the Reddit dataset (a) and the Twitter dataset (b). UA means unavailable.

accelerators for unweighted/undirected graphs when normalized to DSP usage. Hence, the energy efficiency improvements/speedups mirrored in NEM-GNN designs would resemble those seen in AWB-GCN, as depicted in Figure 12 and Figure 14 for GCN.

For PEDAL, we calculate latencies by multiplying workload cycles by clock frequency and determine energy by multiplying average power with latency. The highest performance across IP-AC, RW-AC, and RW-CA dataflows is reported. We observe ~200–230x/~190–205x performance improvement in GCN/GraphSage and ~960–1,030x/~980–1,000x energy improvement in GCN/GraphSage. These enhancements stem from factors like periodic host-accelerator interaction, constrained throughput from processing engines, and limited opportunities to conceal aggregation latency, owing to inherent von Neumann architecture constraints compared to PIM.

We compare prior works for the Reddit dataset in terms of execution time, energy, and energy inefficiency (wherever applicable). Similarly, comparison against the Twitter dataset is performed against ReFLIP and PyG-CPU, as other designs have not reported their values. PyG-GPU has out-of-memory errors for both of these datasets. Such large datasets show the scalability potential of NEM-GNN designs. The speedup reason for von Neumann architecture designs is the same as mentioned in Section 9. Comparing NEM-GNN designs with other ReRAM-based PIM designs, the advantages are as follows. In large datasets like Reddit or Twitter, because we rely on the combination result to broadcast the data to the aggregation array (instead of aggregation process involving search for combination vectors corresponding to neighboring nodes to be aggregated for every node, like in ReFLIP), the possibility of the combination vector getting broadcasted to more candidates increases. Since the adjacency matrix is read in parallel to dot product compute for combination, the broadcast approach can broadcast the combination vector to potentially larger numbers of aggregation candidates/nodes. Therefore, NEM-GNN is efficient for larger workloads as well. Furthermore, this is not possible in other PIM designs, as combination vector results need to be written before initiating aggregation. These features enable improved performance/energy/efficiency of NEM-GNN, as shown in Figure 18.

NEM-GNN can integrate memory-compiled 8T SRAM arrays, improving compute performance and reducing production cycle time. Scaled to 65 nm [26], our custom array is 1.5x larger with 1.4x slower read/write latency and 1.6x higher power consumption than compiled memory, attributed to optimized layout reducing BL/WL capacitance. This boosts performance and power efficiency, and reduces footprint, resulting in 1.3x, 1.33x, and 1.35x performance benefits in GCNs, GATs, and GraphSage for NEM-GNN compared to the custom array. Energy efficiency improves up to 1.4x, 1.45x, and 1.47x for GCNs, GATs, and GraphSage compared to the custom array NEM-GNN.

11 CONCLUSION

We propose a scalable, reconfigurable, DAC, ADC-less, energy-efficient, high-performance GNN accelerator that reconfigures the L1 cache to realize PIM architecture for performing combination and a near-memory approach for performing aggregation. For combination, bit-serial PIM

designs with ECT and pre-compute are proposed to make the design scalable, without requiring data replication and DAC/ADC. For aggregation, graph and sparsity-aware approaches leveraging the underlying graph connectivity and sparsity combined eith CAR and "broadcast" approaches hide the aggregation latency to improve performance.

REFERENCES

[1] Pavan Kumar Reddy Boppidi, S. Siddhartha Raman, H. Renuka, and Souvik Kundu. 2020. Pt/Cu:ZnO/Nb:STO memristive dual port for cache memory applications. *AIP Conference Proceedings* 2265, 1 (2020), 030212. https://doi.org/10.1063/5.0016597

[2] Michael M. Bronstein, Joan Bruna, Yann LeCun, Arthur Szlam, and Pierre Vandergheynst. 2017. Geometric deep learning: Going beyond euclidean data. *IEEE Signal Processing Magazine* 34, 4 (2017), 18–42.

[3] Nagadastagiri Challapalle, Karthik Swaminathan, Nandhini Chandramoorthy, and Vijaykrishnan Narayanan. 2021. Crossbar based processing in memory accelerator architecture for graph convolutional networks. In *Proceedings of the 2021 IEEE/ACM International Conference on Computer Aided Design (ICCAD '21)*. 1–9. https://doi.org/10.1109/ICCAD51958.2021.9643465

[4] Jiaxian Chen, Yiquan Lin, Kaoyi Sun, Jiexin Chen, Chenlin Ma, Rui Mao, and Yi Wang. 2022. GCIM: Toward efficient processing of graph convolutional networks in 3D-stacked memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 11 (2022), 3579–3590. https://doi.org/10.1109/TCAD.2022.3198320

[5] Yuhan Chen, Alireza Khadem, Xin He, Nishil Talati, Tanvir Ahmed Khan, and Trevor Mudge. 2023. PEDAL: A power efficient GCN accelerator with multiple dataflows. In *Proceedings of the 2023 Design, Automation, and Test in Europe Conference and Exhibition (DATE '23)*. 1–6. https://doi.org/10.23919/DATE56975.2023.10137240

[6] Matthias Fey and Jan Eric Lenssen. 2019. Fast graph representation learning with PyTorch Geometric. *arXiv preprint arXiv:1903.02428* (2019).

[7] Tong Geng, Ang Li, Runbin Shi, Chunshu Wu, Tianqi Wang, Yanfei Li, Pouya Haghi, Antonino Tumeo, Shuai Che, Steve Reinhardt, and Martin C. Herbordt. 2020. AWB-GCN: A graph convolutional network accelerator with runtime workload rebalancing. In *Proceedings of the 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '20)*. IEEE, 922–936.

[8] Yu Huang, Long Zheng, Pengcheng Yao, Qinggang Wang, Xiaofei Liao, Hai Jin, and Jingling Xue. 2022. Accelerating graph convolutional networks using crossbar-based processing-in-memory architectures. In *Proceedings of the 2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA '22)*. IEEE, 1029–1042.

[9] Chanwoo Jeong, Sion Jang, Eunjeong Park, and Sungchul Choi. 2020. A context-aware citation recommendation model with BERT and graph convolutional networks. *Scientometrics* 124, 3 (2020), 1907–1922.

[10] 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 *Proceedings of the 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA '21)*. 43–56. https://doi.org/10.1109/ISCA52012.2021.00013

[11] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun. 2020. A modern primer on processing in memory. *arXiv preprint arXiv:2012.03112* (2020).

[12] S. S. Teja Nibhanupudi, Siddhartha Raman Sundara Raman, and Jaydeep P. Kulkarni. 2021. Phase transition material-assisted low-power SRAM design. *IEEE Transactions on Electron Devices* 68, 5 (2021), 2281–2288. https://doi.org/10.1109/TED.2021.3067849

[13] S. S. Teja Nibhanupudi, Siddhartha Raman Sundara Raman, Mikaël Cassé, Louis Hutin, and Jaydeep P. Kulkarni. 2021. Ultra-low-voltage UTBB-SOI-based, pseudo-static storage circuits for cryogenic CMOS applications. *IEEE Journal on Exploratory Solid-State Computational Devices and Circuits* 7, 2 (2021), 201–208. https://doi.org/10.1109/JXCDC.2021.3130839

[14] Hongbin Pei, Bingzhe Wei, Kevin Chen-Chuan Chang, Yu Lei, and Bo Yang. 2020. Geom-GCN: Geometric graph convolutional networks. *arXiv preprint arXiv:2002.05287* (2020).

[15] Yikan Qiu, Yufei Ma, Wentao Zhao, Meng Wu, Le Ye, and Ru Huang. 2022. DCIM-GCN: Digital computing-in-memory to efficiently accelerate graph convolutional networks. In *Proceedings of the 2022 IEEE/ACM International Conference on Computer Aided Design (ICCAD '22)*. 1–9.

[16] Siddhartha Raman Sundara Raman. 2024. A review on non-volatile and volatile emerging memory technologies. In *Computer Memory and Data Storage*, Azam Seyedi (Ed.). IntechOpen, Rijeka, Chapter 3. https://doi.org/10.5772/intechopen.110617

[17] Siddhartha Raman Sundara Raman, S. S. Teja Nibhanupudi, Atanu K. Saha, Sumeet Gupta, and Jaydeep P. Kulkarni. 2021. Threshold selector and capacitive coupled assist techniques for write voltage reduction in

metal–ferroelectric–metal field-effect transistor. *IEEE Transactions on Electron Devices* 68, 12 (2021), 6132–6138. https://doi.org/10.1109/TED.2021.3121348

[18] Siddhartha Raman Sundara Raman, Feng Wen, Ravi Pillarisetty, Vivek De, and Jaydeep P. Kulkarni. 2021. High noise margin, digital logic design using Josephson junction field-effect transistors for cryogenic computing. *IEEE Transactions on Applied Superconductivity* 31, 5 (2021), 1–5. https://doi.org/10.1109/TASC.2021.3054347

[19] Siddhartha Raman Sundara Raman, Shanshan Xie, and Jaydeep P. Kulkarni. 2022. IGZO CIM: Enabling in-memory computations using multilevel capacitorless indium–gallium–zinc–oxide-based embedded DRAM technology. *IEEE Journal on Exploratory Solid-State Computational Devices and Circuits* 8, 1 (2022), 35–43.

[20] Siddhartha Raman Sundara Raman, Shanshan Xie, and Jaydeep P. Kulkarni. 2021. Compute-in-eDRAM with backend integrated indium gallium zinc oxide transistors. In *Proceedings of the 2021 IEEE International Symposium on Circuits and Systems (ISCAS '21)*. 1–5. https://doi.org/10.1109/ISCAS51556.2021.9401798

[21] Rishov Sarkar, Stefan Abi-Karam, Yuqi He, Lakshmi Sathidevi, and Cong Hao. 2023. FlowGNN: A dataflow architecture for real-time workload-agnostic graph neural network inference. In *Proceedings of the 2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA '23)*. 1099–1112. https://doi.org/10.1109/HPCA56546.2023.10071015

[22] James E. Stine, Ivan Castellanos, Michael Wood, Jeff Henson, Fred Love, W. Rhett Davis, Paul D. Franzon, Michael Bucher, Sunil Basavarajaiah, Julie Oh, and Ravi Jenkal. 2007. FreePDK: An open-source variation-aware design kit. In *Proceedings of the 2007 IEEE International Conference on Microelectronic Systems Education (MSE '07)*. IEEE, 173–174.

[23] Siddhartha Raman Sundara Raman, S. S. Teja Nibhanupudi, and Jaydeep P. Kulkarni. 2022. Enabling in-memory computations in non-volatile SRAM designs. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 12, 2 (2022), 557–568. https://doi.org/10.1109/JETCAS.2022.3174148

[24] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2017. Graph attention networks. *arXiv preprint arXiv:1710.10903* (2017).

[25] Hongwei Wang, Jialin Wang, Jia Wang, Miao Zhao, Weinan Zhang, Fuzheng Zhang, Wenjie Li, Xing Xie, and Minyi Guo. 2019. Learning graph representation with generative adversarial nets. *IEEE Transactions on Knowledge and Data Engineering* 33, 8 (2019), 3090–3103.

[26] Shanshan Xie, Siddhartha Raman Sundara Raman, Can Ni, Meizhi Wang, Mengtian Yang, and Jaydeep P. Kulkarni. 2022. Ising-CIM: A reconfigurable and scalable compute within memory analog Ising accelerator for solving combinatorial optimization problems. *IEEE Journal of Solid-State Circuits* 57, 11 (2022), 3453–3465. https://doi.org/10.1109/JSSC.2022.3176610

[27] Xinfeng Xie, Zheng Liang, Peng Gu, Abanti Basak, Lei Deng, Ling Liang, Xing Hu, and Yuan Xie. 2021. SpaceA: Sparse matrix vector multiplication on processing-in-memory accelerator. In *Proceedings of the 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA '21)*. IEEE, 570–583.

[28] Mingyu Yan, Lei Deng, Xing Hu, Ling Liang, Yujing Feng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie. 2020. HyGCN: A GCN accelerator with hybrid architecture. In *Proceedings of the 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA '20)*. IEEE, 15–29.

[29] Tao Yang, Dongyue Li, Yibo Han, Yilong Zhao, Fangxin Liu, Xiaoyao Liang, Zhezhi He, and Li Jiang. 2021. PIMGCN: A ReRAM-based PIM design for graph convolutional network acceleration. In *Proceedings of the 2021 58th ACM/IEEE Design Automation Conference (DAC '21)*. 583–588. https://doi.org/10.1109/DAC18074.2021.9586231

Received 6 September 2023; revised 1 March 2024; accepted 10 March 2024