Performance and Accuracy Tradeoffs for Training Graph Neural Networks on ReRAM-Based Architectures

Aqeeb Iqbal Arka, *Graduate Student Member, IEEE*, Biresh Kumar Joardar, *Member, IEEE*, Janardhan Rao Doppa, *Member, IEEE*, Partha Pratim Pande, *Fellow, IEEE*, and Krishnendu Chakrabarty, *Fellow, IEEE*

***Abstract*— Graph neural network (GNN) is a variant of deep neural networks (DNNs) operating on graphs. However, GNNs are more complex compared with DNNs as they simultaneously exhibit attributes of both DNN and graph computations. In this work, we propose a ReRAM-based 3-D manycore processing-in-memory architecture called ReMaGN, tailored for on-chip training of GNNs. ReMaGN implements GNN training using reduced-precision representation to make the computation faster and reduce the load on the communication backbone. However, reduced precision can potentially compromise the accuracy of training. Hence, we undertake a study of performance and accuracy tradeoffs in such architectures. We demonstrate that ReMaGN outperforms conventional GPUs by up to 9.5× (on average 7.1×) in terms of execution time, while being up to 42× (on average 33.5×) more energy efficient without sacrificing accuracy.**

***Index Terms*— 3-D architectures, graph neural networks (GNNs), network-on-chip (NoC), reduced precision, ReRAM.**

I. Introduction

GRAPH neural networks (GNNs) enable comprehensive predictive analytics over graph structured data. As a result, they have become popular in diverse real-world applications such as social networks [1], recommendation systems [2], quantum chemistry [3], and many other applications [4]. A key challenge in facilitating such analytics is to learn good representations over nodes, edges, and graphs. Recent advances in GNNs have successfully addressed this challenge. Unlike traditional deep neural networks (DNNs), which work over regular structures (images or sequences), GNNs operate on graphs. This is achieved by performing a neighborhood aggregation operation, where each node aggregates the features of its *k-hop* neighbors to learn node representations with high predictive ability [5]. This in turn gives rise to repeated message-passing operations that can become very communication intensive. Moreover, the computations associated with GNN can be divided into two parts: 1) vertex-centric computations involving trainable weights, similar to conventional DNNs and 2) edge-centric computations, which involve accumulating information from neighboring vertices along the edges of the graphs [5], [6]. Hence, GNN training exhibits characteristics of both DNN training, which is compute-intensive, and graph computation that exhibits heavy data exchange. Conventional CPU- or GPU-based systems are not tailor-made for applications that exhibit such trait. This necessitates the development of new and efficient hardware architectures tailored for GNN training/inference.[1]

Both the vertex- and edge-centric computations in GNNs can be represented as multiply-and-accumulate (MAC) operations, which can be efficiently implemented using resistive random access memory or ReRAM-based architectures [6]–[9]. In addition, ReRAMs allow for processing in-memory, which helps reduce the amount of communication (data transfers) between computing cores and the main memory. This is particularly useful for GNN training as it involves repeated feature aggregation along the graph edges [6]. Feature aggregation (a.k.a. message passing) is a communication intensive task that generates significant data traffic [10]. The in-memory nature of ReRAM computation significantly reduces the on-chip traffic leading to better performance [11]. However, existing ReRAM-based architectures are designed to accelerate specifically either DNNs (see [12], [13]) or graph computations (see [14]). As GNN training exhibits characteristics of both DNNs and graph computations, these tailor-made architectures are not well suited for efficient GNN training. Hence, we design a novel ReRAM-based architecture that caters to the specific characteristics exhibited by GNN training and this is one of the main focuses of this work.

ReRAM-based accelerators for DNN training utilize relatively larger sized ReRAM crossbars (e.g., 128 × 128) [12], [13]. However, larger ReRAM crossbar sizes are not efficient for storing sparse matrices. On the other hand, traditional ReRAM-based architectures for accelerating graph computations use relatively smaller sized crossbars to avoid/reduce zero storage [14]–[16] as real-world graph data are often extremely sparse. These disparate design choices present a challenge as GNNs involve both DNN and graph computations. In this work, we show that even though larger ReRAM tiles are inefficient for storing sparse matrices, they have

Manuscript received March 17, 2021; revised June 16, 2021 and August 6, 2021; accepted September 2, 2021. Date of publication September 15, 2021; date of current version October 6, 2021. This work was supported in part by the U.S. National Science Foundation (NSF) under Grant CNS-1955353 and Grant CNS-1955196 and in part by the U.S. Army Research Office under Grant W911NF-17-1-0485. The work of Biresh Kumar Joardar was supported by NSF to the Computing Research Association for the CIFellows Project under Grant 2030859. *(Corresponding author: Partha Pratim Pande.)*

Aqeeb Iqbal Arka, Janardhan Rao Doppa, and Partha Pratim Pande are with the School of Electrical Engineering and Computer Science, Washington State University, Pullman, WA 99163 USA (e-mail: aqeebiqbal.arka@wsu.edu; jana.doppa@wsu.edu; pande@wsu.edu).

Biresh Kumar Joardar and Krishnendu Chakrabarty are with the Department of Electrical and Computer Engineering, Duke University, Durham, NC 27708 USA (e-mail: bireshkumar.joardar@duke.edu; krish@duke.edu).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TVLSI.2021.3110721.

Digital Object Identifier 10.1109/TVLSI.2021.3110721

better storage density per unit silicon area. As a result, larger ReRAM crossbars tend to be more efficient in terms of overall full-system area and energy consumption for storing sparse graphs.

Next, it is also important to ensure efficient communication between ReRAM-based processing elements (PEs) to achieve high performance. Traffic associated with GNN training exhibits many-to-one-to-many, multicast, and long-range characteristics. This data exchange pattern occurs due to the message-passing operations required to accumulate neighbor information in a recursive approach [10]. Traditional planar (2-D) architectures are not suitable for supporting high degree of long-range communication due to limited floor-planning choices [17] while also not being inefficient at handling multicast traffic [18]. Three-dimensional (3-D) manycore architectures are capable of addressing the performance limitations arising due to heavy long-range data exchange in addition to supporting efficient multicast [19], [20]. Prior work has shown that, by stacking one layer on top of another, it is possible to reduce overall communication latency in a system. This helps to design low latency and high-throughput communication architecture via a multicast enabled 3-D network-on-chip (NoC) [19], [20]. In addition, ReRAM-based accelerators rely on 16-bit fixed-point computation [12]. Employing a reduced-precision representation can help accelerate the computation and alleviate the heavy data exchange associated with GNN training. However, reduced precision can adversely affect the achievable accuracy. Hence, we explore performance and accuracy tradeoffs to fully realize the benefits of reduced-precision training on ReRAM-based architectures for GNN training. Hence, our hypothesis is that a multicast enabled 3-D architecture consisting of ReRAM-based PEs, using reduced-precision variable representation, will be suitable for GNN training.

In this article, we propose a ReRAM-based manycore architecture for training GNNs referred to as ReMaGN (pronounced as "reimagine"). The proposed ReMaGN architecture consists of: 1) ReRAM-based PEs; here, multiple ReRAM crossbars make up a tile and each PE consists of multiple tiles; these PEs are used to accelerate the large number of MAC operations for training GNNs; 2) stochastic rounding enabled reduced-precision operations to accelerate GNN training without sacrificing accuracy; and 3) a high-throughput 3-D NoC architecture as the communication backbone. The main contributions of this work include the following.

1) We demonstrate superior storage efficiency (bit/area) by using uniform ReRAM crossbar configuration for both graph and DNN computations without introducing any additional power overhead.
2) We design an energy-efficient and high-performance 3-D manycore architecture for accelerating GNN training. We map neural layers to PEs considering the on-chip traffic pattern, which enables high-throughput GNN training.
3) We propose a design methodology to train GNNs on ReMaGN architecture using reduced-precision representation, with minimal accuracy loss.
4) We demonstrate that ReMaGN outperforms traditional GPU-based designs for training GNNs on diverse real-world graphs with millions of nodes.

To the best of our knowledge, this is the first work that proposes a ReRAM-based manycore architecture with reduced-precision representation and enabled by a 3-D NoC for high-performance and energy-efficient training of GNNs. The rest of this article is organized as follows. Section II describes relevant prior work. In Section III, we discuss the salient features of GNNs, especially the traffic patterns that must be considered for NoC design. In Section IV, we introduce the proposed ReMaGN architecture, highlight the role of the 3-D NoC, and describe the GNN layer mapping strategy. Section V presents the experimental results and our analysis based on these results. The conclusion is provided in Section VI.

II. Related Prior Work

This work focuses on accelerating GNN training on ReRAM-based manycore architectures. In this section, we review related prior work on ReRAM-based architectures and different hardware platforms for accelerating GNN training/inference.

A. ReRAM-Based Architectures

ReRAMs can be used as memory and also to perform *in situ* MAC (IMA) operations [9], [11]. Both DNN and graph computation rely heavily on such MAC operations. This makes ReRAM-based architectures excellent candidates for DNN training/inference [12], [13], [21]. These architectures employ pipelined DNN training on ReRAM and have been shown to significantly outperform tradition CPU-/GPU-based system in terms of both execution time and energy efficiency. However, all of the ReRAM-based architectures mentioned above overlook the need for an appropriate communication backbone necessary in a manycore architecture. GPU-ReRAM-based heterogeneous architectures were proposed in [22] and [23] to improve the accuracy of trained models while also addressing the communication-related issues with an appropriate NoC design [20], [24]. ReRAM-based architectures have been used to accelerate recurrent neural network (RNN) training with similar success as well [25].

However, the sparsity of graph structured data poses a significant challenge in efficient computation and storage arising from the presence of redundant zeros due to sparse adjacency matrix for graphs. Note that these zeros are redundant as multiplication/addition with zero results in a zero and have no effect on the overall computation. Hence, storing these redundant zeros on ReRAM cells gives rise to unnecessary computation while requiring additional storage. There exists a significant body of work, which addresses this unnecessary zero storage issue. This in turn has made ReRAM-based graph accelerators faster and more energy efficient than traditional CPU-/GPU-based implementations [14]–[16].

All these ReRAM-based architectures are fine-tuned for either DNN training/inference or graph analytics. GNN training not only exhibits characteristics of both DNN computation and graph analytics but also involves heavy communication resulting from the neighborhood aggregation operation essential for GNN training. As a result, both computation and communication can become a performance bottleneck without suitable hardware support. Hence, an effective architecture for GNN training should address the requirements for

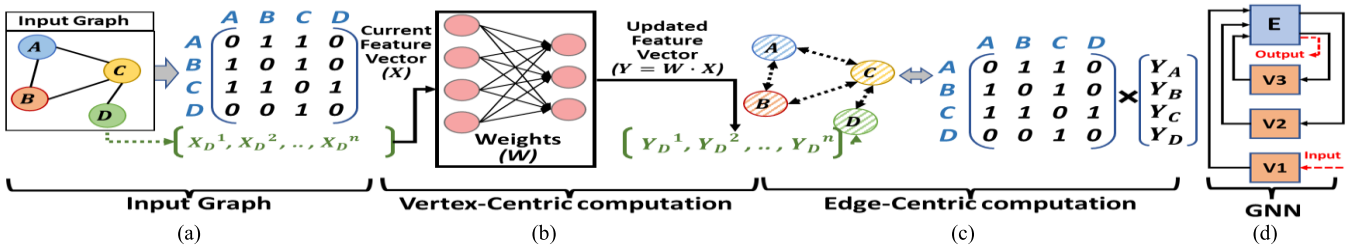

Fig. 1. Illustration of the computational components of a GNN. (a) Input graph represented as node features (X_i) and adjacency matrix (A), (b) vertex-centric computation layer (V), (c) edge-centric computation layer (E); both V- and E-layers together constitute a neural layer of GNN, and (d) overall GNN structure with three neural layers as an example. The arrows indicate the data communication pattern in a GNN. Note that each V-layer has its unique set of weights (like traditional DNNs) which need to be mapped to different PEs. However, the E-layer depends on A only which is fixed for a given input graph. This results in a many-to-one communication where all the PEs responsible for V-layer computations communicate with the same PEs storing the A matrix.

both DNN and graph computation while enabling efficient communication.

Moreover, ReRAM-based architectures cannot support 32-bit floating-point-based operations like GPUs. ReRAM-based architectures typically use 16-bit fixed-point representation [12], [26]. However, this can lead to unstable training or poor accuracy due to precision loss. Stochastic rounding can be used to preserve the accuracy of training CNNs on ReRAMs using 16-bit fixed-point representation [24], [27]. However, investigating the effect on performance and accuracy with even lower precision representation has not been thoroughly studied. It is important to note that stochastic rounding cannot be implemented on ReRAM without additional circuitry. In this work, we demonstrate the accuracy and performance tradeoffs when different bit precisions are used in conjunction with stochastic rounding for training GNNs.

B. Hardware for GNN Training

GNN training can be very memory intensive in the case of large-scale graphs. Graph partitioning helps to reduce the memory overhead associated with GNN training [28]. This "divide and conquer" approach allows for scalable GNN training over large graphs with high speed and accuracy [28]. GNN accelerators using commodity processors, field-programmable gate arrays (FPGAs), and custom ASICs have been proposed recently for GNN inference [6], [10], [29], [30]. However, all of these solutions focus only on GNN inference, which is significantly simpler than training. GNN training is considerably more challenging due to the heavy data exchange between the forward and backward phases. Furthermore, the architectures described in prior work focus on relatively small graphs (with few hundred thousand nodes only), which are less compute- and memory-intensive. In contrast, this article focuses on an accelerator for GNN training over large-scale graphs containing several millions of nodes.

In this work, we design a single-chip manycore architecture, referred to as ReMaGN, which is enabled by 3-D NoC uses ReRAM-based PEs to accelerate GNN training by leveraging the enhanced IMA capabilities of ReRAM. Next, we show that it is possible to improve performance even further by using reduced-precision variable representation. Finally, we demonstrate the efficacy and scalability of the ReMaGN architecture to accelerate GNN training over large-scale graphs containing millions of nodes.

III. GNN Computation Kernel

In this section, we introduce the salient characteristics of a GNN and discuss the computational and inter-PE communication characteristics observed during the training process. Fig. 1 shows different computational and communication characteristics of a GNN. A graph consists of: 1) vertices: each vertex V can be represented using a feature vector that characterizes the node (as shown in Fig. 1(a), X_V is the feature vector of vertex V) and 2) edges: represented by an adjacency matrix indicating the vertex connectivity (A). This node and edge information constitute the main element of the computational kernel for training GNNs.

A GNN consists of multiple back-to-back neural layers. Each neural layer involves two types of computations: 1) vertex computation (V-layer), which is similar to MAC operations in traditional DNNs, as shown in Fig. 1(b), and 2) edge computation (E-layer), which resembles the message-passing operation in graph analytics demonstrated in Fig. 1(c). The following equations illustrate the operations involved in a GNN (forward phase) and represent the vertex and edge computations of GNN layer l, respectively:

$$\text{GNN layer } l: \begin{cases} Y_V^l = X_V^{l-1} \times W^l & (1) \\ X_V^l = A \times Y_V^l. & (2) \end{cases}$$

A. Forward-Phase Computation

As mentioned above, vertex computation in (1) involves a set of learnable weights W^l for the lth layer (similar to traditional DNNs). As shown in Fig. 1(b), the feature vector of the nodes/vertices (X_V^{l-1}) is multiplied with the weight matrix (W^l) to calculate the updated feature vector Y_V^l. Edge computations (representing the message-passing operation in graph analytics [6]) require the aggregation of information from all one-hop neighbors. This is achieved by multiplying the updated feature vector Y_V^l with the adjacency matrix (A) as shown in (2). Note that A stores the edge information of the input graph. Hence, multiplying Y_V^l with A is equivalent to aggregating all the information from one-hop neighbors (message passing). This operation is illustrated in Fig. 1(c). As we can see from (1) and (2) and Fig. 1(b) and (c), the GNN layer forward-phase computations are essentially MAC operations that can be implemented using ReRAMs. Note that edge computation for GNNs is a sparse matrix-vector multiplication (SpMV) involving A (which is a sparse matrix) and the updated vertex feature vectors (Y^l).

B. Backward-Phase Computation

The backward-phase computations of GNN training consist of error and gradient calculation for weight update, both of which are also predominantly MAC operations [13], [31]. As mentioned earlier, GNN computation involves two layers, V-layer and E-layer. Equations (3) and (4) represent the error calculation part of the backward-phase computation in a GNN. Note that, in forward phase, data propagate from layer l to layer $l+1$, while in backward phase, errors propagate from layer $l+1$ to layer l. As a result, the errors do not need to be computed during the backward phase of the first layer of the GNN. The following equations represent the vertex and edge components during the error computation of GNN layer l, respectively:

$$\text{GNN layer } l: \begin{cases} \text{err}_V^l = \left(W^{l+1}\right)^T \times \delta_V^{l+1} \cdot X_V^{l+1} & (3) \\ \delta_V^l = A \times \text{err}_V^l. & (4) \end{cases}$$

Equation (3) determines the error for V-layer (err_V^l). Here, "$\times$" and "$\cdot$" represent matrix-vector multiplications and element-wise multiplication operations, respectively. The error, δ_V^{l+1}, is propagated from layer $l+1$ and err_V^l is the result of GNN backward-phase (V-layer) computation of neural layer l. The final error of layer l and δ_V^l, is computed in the E-layer following (4). Here, the error calculated in the V-layer (err_V^l) is accumulated over the graph edges via multiplication with the graph adjacency matrix (A) and the overall error, δ_V^l, is calculated for every node. The final error is then used to compute the partial derivatives of weights for every layer. The derivative is calculated by the following equations:

$$\text{GNN layer } l: \begin{cases} z_V^l = A \times \delta_V^l & (5) \\ \nabla W^l = \left(X_V^{l-1}\right)^T \cdot z_V^l. & (6) \end{cases}$$

Equations (5) and (6) represent the E-layer and V-layer computations for calculating the gradient of the weights for the lth GNN layer, respectively. Similar to other E-layer computations, the errors from each node are accumulated over the graph edges by multiplying δ_V^l with A (adjacency matrix) to calculate the accumulated error (z_V^l) following (5). The partial derivative of the weight matrix is the elementwise product of accumulated error (z_V^l) and feature vector from the previous layer (X_V^{l-1}). As can be seen from (3)–(6), the backward-phase computations are also simple MAC operations and can be efficiently implemented using ReRAM-based PEs similar to the forward phase. Hence, both forward and backward phases of GNN training are executed using ReRAM-based PEs in the proposed ReMaGN architecture.

C. Communication

Similar to traditional DNNs, the output of one neural layer of computation is the input of the next layer of computation (i.e., X_V^l from layer l is the input to layer $l+1$). The edge computation in (2) involves multiplication with the same adjacency matrix (A). Note that A stores the information of edges and is fixed for a given graph. This results in some interesting on-chip traffic patterns. First, each vertex computation has a unique set of weights (i.e., $W^l \neq W^{l+1}$), where W^l is the weight associated with the V-layer computation of layer l. Hence, as an example, for a three-layer GNN, three sets of weights—W^1, W^2, and W^3, need to be mapped/assigned to unique sets of PEs for forward-phase computation. Similarly, the adjacency matrix A is mapped to a separate set of PEs than W^1, W^2, and W^3. This is necessary as DNN training on ReRAMs follows a pipelined implementation, unlike in the case of traditional GPUs [12]. GPUs execute DNN layers one after another, whereas ReRAMs execute all the layers simultaneously on different inputs at any point of time (discussed in more detail in Section IV). As a result, all the necessary weights and adjacency matrices for GNN computation need to be stored in the PEs simultaneously to enable pipelined computation [13]. Hence, the same PE cannot be used to store the weights and the adjacency matrices.

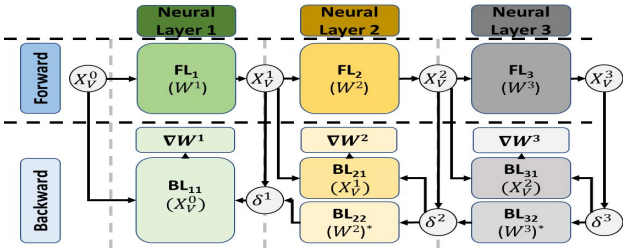

Fig. 2. Illustration of data movement between PEs performing forward- and backward-phase computations for a three-layer GNN.

Storing all weights and adjacency matrices on-chip (on ReRAM cells) also reduces off-chip communication. To accomplish GNN training on-chip, PEs storing W^1, W^2, and W^3 need to communicate their outputs with the same PEs storing A and vice versa [following (1) and (2)]. This communication pattern is illustrated in Fig. 1(d). Here, $V1$, $V2$, and $V3$ represent the set of PEs storing W^1, W^2, and W^3. At each PE, one V-layer computation occurs after which the result then moves on to the PE storing A as represented by E. The edge computation takes place at E. As evident in Fig. 1(d), this results in a many-to-one-to-many communication pattern as multiple PEs, i.e., $V1$, $V2$, and $V3$, storing the different sets of weights, communicate with the same PEs storing A and vice versa. In the absence of a suitable interconnection backbone, the inherent many-to-one-to-many communication can overwhelm the training process, resulting in a performance bottleneck.

Moreover, there is data movement between the forward and backward phases of GNN training. This data movement impacts the overall performance of the ReMaGN architecture. The feature vectors X_V^l are shared between both forward and backward phases. The output of layer l, X_V^l, is the input to the backward-phase computation of the same layer, as shown in (3). Furthermore, as shown in (1) and (6), the input to layer l, X_V^{l-1}, is shared between both forward and backward phases. Fig. 2 illustrates the data traffic between the forward and backward phases for a GNN with three neural layers. It should be noted that, each neural layer has V- and E-layer computations which are not shown explicitly in Fig. 2 for brevity. In Fig. 2, FL_l, BL_{l1}, and BL_{l2} denote the PEs storing forward-phase weights of layer l and PEs responsible for calculating backward-phase weight derivative and error of layer l, respectively. The parameters X_V^l and δ_V^l are feature vectors and error vectors, respectively, from layer l. All the data computed during the backward phase of one layer (l) need to be communicated to its preceding layer ($l-1$).

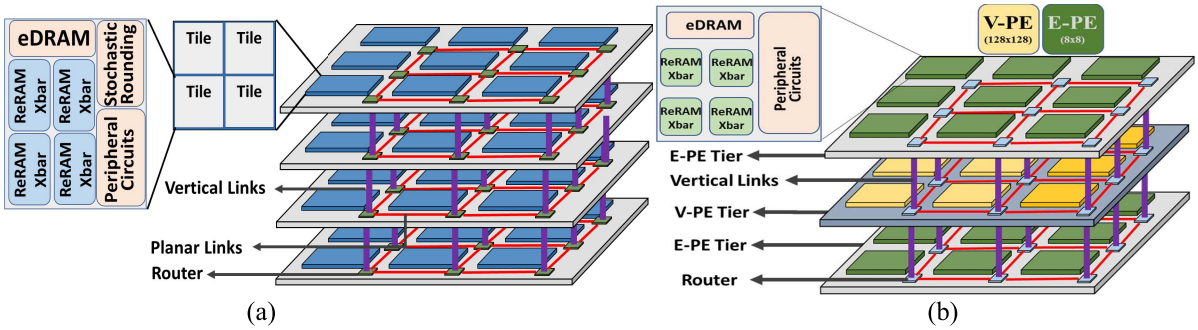

Fig. 3. Illustrative representation of (a) ReMaGN architecture and (b) ReGraphX architecture. Figures are for illustration purposes only.

It should be noted that the backward-phase computation in GNN also exhibits many-to-one-to-many traffic as different PEs responsible for the V-layer computation for backward phase communicate with the same PE storing the adjacency matrix, A, following (4) and (5). Hence, the manycore architecture needs to be supported by a suitable NoC for effective training of GNNs.

IV. ReMaGN Architecture

In this section, we present the key attributes of the ReMaGN architecture. First, we discuss the architecture of each ReRAM-based tile. Next, we present the salient characteristics of ReMaGN [see Fig. 3(a)] and discuss how it can be used to train GNNs over large graphs.

A. Basics of ReRAM-Based Accelerator Design

ReRAMs enable fast and efficient in-memory MAC operations. In ReRAM-based accelerators for GNN training, the neural weights are mapped to ReRAM cells following [12] and [13]. Fig. 4 shows an illustration of GNN weight mapping to the ReRAM crossbars. Here, we assume a V-layer (part of the GNN) with a weight matrix of size $M \times N$. Each element of the weight is stored on the crossbars in a 16-bit fixed-point format [12]. However, all 16-bits (2^{16} states) are not represented on a single ReRAM cell mainly due to the area and noise concerns [32]. As shown in Fig. 4, the 16 bits of each weight are distributed across multiple arrays/cells. In the ReMaGN architecture, each ReRAM cell has 2-bit resolution. As a result, a 16-bit fixed-point number requires eight ReRAM cells. For 12- and 8-bit representations, six and four ReRAM cells are needed to store each element of the weight matrix, respectively.

Each ReRAM cell stores information in the form of conductance. By applying a voltage into the word line (WL_i in Fig. 4) and sensing the resultant current along the bitline (I_j along BL_j in Fig. 4), we implement the dot product of the input voltage and the cell conductance. This dot product functionality is used to design accelerators for machine learning algorithms [12], [13]. A dot product is a sum of products. The sum is obtained through the current summation over the bitline. Each row computes a product by streaming in the multiplicand via the word-line digital-to-analog converter (DAC), as shown in Fig. 4. The MAC operation on ReRAMs is based on Ohm's and Kirchhoff's current laws. ReRAMs can perform N^2 multiplications in $O(1)$ time. Hence, it is fast and also energy efficient compared with traditional GPUs.

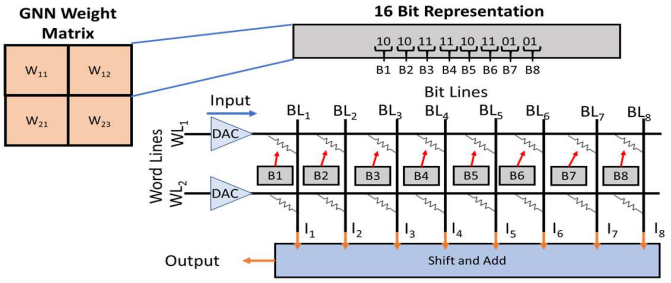

Fig. 4. Illustration of data mapping and MAC operations on ReRAM crossbars.

As mentioned in Section III, a GNN kernel principally involves MAC operations. We implement MAC operation in ReRAM crossbars in a pipelined manner [12]. Each bit of the input data is fed to the pipeline. The pipeline consists of seven stages: read from eDRAM (eDRAM is part of the ReRAM tile, as shown in Fig. 3), compute on crossbar, analog-to-digital conversion (ADC), bitwise shift-and-add (as shown in Fig. 4), global shift-and-add, and activation function, and then finally write output to memory. Hence, the total execution time to process a 1-bit data is seven cycles. Following the pipelined implementation, we can process a 16-bit number (i.e., multiply with the value stored on the ReRAM crossbars) in 22 cycles [12]. The overall execution time can be further lowered by duplicating the computations on multiple crossbars. Each crossbar would then process a different input in parallel [13]. For instance, by duplicating the weights on two crossbars, we can reduce the execution time by approximately half and so on. The number of times the weights can be duplicated depends on the amount of ReRAM crossbars available on the chip. As the computations on GNNs are primarily MAC operations (as described in Section III), we use this execution model to determine the execution time for each GNN layer and eventually the overall execution time of ReMaGN.

B. ReRAM Tile Configuration for ReMaGN

Traditionally, ReRAM-based architectures for DNN training are implemented using relatively large crossbars (e.g., 128 × 128 as shown in [12]) because weight matrices tend to be relatively dense. On the other hand, graph computations use relatively small crossbars as smaller crossbars avoid/reduce the storage of zeros (zeros are redundant in MAC operations) [15]. Graph computation generally involves sparse adjacency matrices. Hence, reduction in zero storage is key to improving graph computation on ReRAM-based architectures. This is achieved

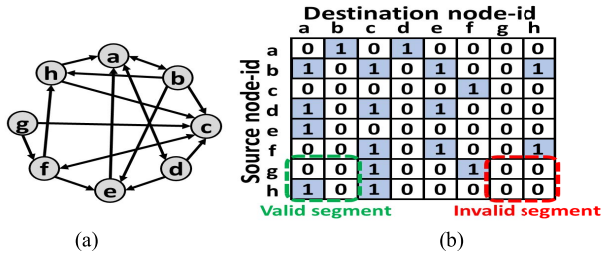

Fig. 5. (a) Example input graph and (b) adjacency matrix (0 means no edge and 1 means presence of an edge).

using a nonoverlapping sliding window operation where the adjacency matrix is decomposed into $N \times N$ segments to map on to $N \times N$-shaped ReRAM crossbars. This process is illustrated in Fig. 5. Fig. 5(a) and (b) shows the input graph and the adjacency matrix, respectively. Any $N \times N$ segment with N^2 zero entries (referred to as "invalid segment" in Fig. 5(b) in red) is discarded [14]. The remaining segments that include at least one edge (referred to as "valid segment" in Fig. 5(b) in green) are stored on ReRAM cells. This methodology reduces the number of zeros that need to be stored on-chip. For instance, the adjacency matrix in Fig. 5(b) has 14 valid segments and two invalid segments; the invalid segments can be safely discarded, which reduce redundant MAC operations involving zeros.

As elaborated in Section III, the training process of GNNs exhibits attributes of both DNNs and graph computations. This presents a challenge while choosing the suitable crossbar size for training GNNs. There are two possible choices while designing ReRAM-based accelerators for GNN training—heterogeneous and homogeneous. The heterogeneous architecture consists of different types of PE for DNN (V-layer) computation and graph (E-layer) computation. Such an architecture called ReGraphX has been proposed in [33]. In the heterogeneous ReGraphX architecture, we use 128 $\times$ 128 size crossbar for V-layer of the GNN kernel. These PEs are responsible for storing the DNN weights. In addition, we use 8 $\times$ 8 size crossbar for E-layer computation and sparse adjacency matrix storage. Note that smaller crossbars are more efficient for storing sparse data [14]. In the ReGraphX architecture, one planar layer of V-PEs is sandwiched between two layers E-PEs. The overall ReGraphX architecture is shown in Fig. 3(b).

It is also possible to design the architecture using a homogeneous crossbar configuration. From prior work, it is well-known that ReRAM tile area is dominated by peripheral circuits [12]. Hence, the ReRAM tile area and power do not vary significantly with the crossbar size. However, storage density (bits stored per unit area) varies significantly with the crossbar size. A crossbar of size $N \times N$ can store up to N^2 values. Hence, to match the storage capability of one 128 $\times$ 128 crossbar, we will need up to 256 ($=128^2/8^2$) 8 $\times$ 8 crossbars [12]. This is not desirable as having more tiles (and hence, more peripherals such as ADC, DAC, routers, etc.) can lead to higher full-system area and power consumption. The energy savings due to more efficient zero storage and smaller peripheral circuits are much less compared with the increase in power and area due to the larger number of peripherals.

To determine the suitable ReRAM crossbar configuration for training GNNs, we consider different ReRAM crossbar sizes varying from 8 $\times$ 8 to 256 $\times$ 256 for ReMaGN. We observe two important trends: 1) smaller crossbars lead to many tiles but store fewer zeros and 2) larger crossbars necessitate fewer tiles but store more zeros. The choice of crossbar size should be such that the overall area (and power consumption) is minimized. Our analysis, as shown in Section V, indicates that for all input graphs considered in this work for GNN training, 128 $\times$ 128 sized crossbars achieve the best storage–power–area tradeoffs. This happens because having smaller crossbars but many tiles is more expensive (than the other way around) in terms of both area and power. Hence, we use 128 $\times$ 128 sized crossbars for ReMaGN for storing both the relatively dense weights and the sparse adjacency matrices. The proposed ReMaGN architecture is comprised of multiple planar tiers of homogeneous ReRAM crossbar-based PEs stacked vertically on top of each other.

C. Communication Architecture for ReMaGN

In order to effectively utilize the computational benefits provided by ReRAM-based PEs, the ReMaGN architecture needs a high performance and efficient communication backbone. We leverage the benefits introduced by a 3-D NoC. Moreover, the performance of the ReMaGN architecture is further enhanced by reduced-precision representation.

1) High-Performance 3-D NoC: Efficient communication between PEs during GNN training is a key aspect to achieve high performance and energy efficiency in the ReMaGN architecture. The communication pattern in GNN training includes multiple many-to-one-to-many communication, as shown in Fig. 6(a). The overall communication process starts with the PEs storing the GNN V-layer weights sending the updated node features (Y in Fig. 1) to the PEs storing the adjacency matrix (A) for further processing. Then, the aggregated information is moved forward to the weights storing PE of the next neural layer. This results in the aforementioned many-to-one-to-many patterns, where multiple sets of PEs storing weights communicate with the same set of PEs storing the adjacency matrix (A) and vice versa. This results in the on-chip communication pattern illustrated in Fig. 6(b), where it is shown that the heavy many-to-one-to-many communication pattern can lead to a highly congested link, which in turn becomes the bottleneck due to heavy traffic. Moreover, training involves data sharing between the forward and backward phases of computations of each layer; often the backward-phase computations are implemented on separate set of ReRAMs, as described in [12] and [13], Overall, this results in the output of layer L_i being sent to: 1) PEs responsible for the next layer L_{i+1} and 2) the PEs responsible for the backward phase of layer L_i. As the same data are communicated between multiple sets of PEs, there is significant amount of multicast traffic, as shown in Fig. 6(c). Moreover, there can be additional multicast traffic due to the mapping of GNN layers (weights) to the ReRAM crossbars. For instance, if the weights of one neural layer are mapped across multiple PEs, the same data then have to be communicated to multiple destinations resulting in additional multicast traffic.

The amount of data communicated between different PEs is proportional to the total number of nodes present in the input graph, which can often be very high. For example, the smallest dataset considered for this work (PPI) has 1.6 million entries for one (sub)graph input (more details are provided in

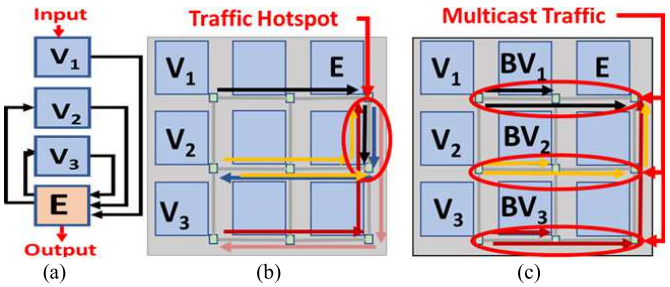

Fig. 6. Simplified illustration of the communication during GNN training in ReMaGN. (a) Communication pattern in GNN training for a 3-layer GNN, (b) many-to-few communication pattern and resulting traffic hotspots, and (c) multicast communication between different PEs.

Section V). When compared against datasets used to evaluate DNNs, the amount of data is orders of magnitude larger. As an example, one image in the popular ImageNet dataset only has 150k entries ($224 \times 224 \times 3$). This large number of nodes in each input graph results in a significant amount of multicast traffic on top of the many-to-one-to-many data exchange during GNN training. Therefore, a high performance and efficient communication backbone are necessary to enable accelerated GNN training in ReMaGN.

Traditional planar NoC architectures are not adequate for such heavy communication. The large physical distance between PEs in planar architectures imposes a significant amount of long-range communication. This can give rise to performance bottleneck in the presence of heavy many-to-one-to-many and multicast traffic. In addition, a planar-mesh NoC has a multihop nature, which leads to higher communication latencies [19]. This is not desirable for accelerating GNN training. A 3-D NoC architecture improves the latency compared with a planar counterpart. In a 3-D architecture, multiple planar tiers of PEs are stacked vertically on top of one another, resulting in lower physical distance between PEs [19]. This leads to lower latency and high-throughput communication. Moreover, a 3-D NoC architecture can support high-performance multicast. Combining these aspects of a 3-D NoC allows for high-performance GNN training in ReMaGN. In this work, we use a 3-D mesh NoC as the interconnection backbone, which also supports 3-D tree multicast. We consider tree multicast in this article as an example multicast scheme only, as any other multicast method can be used for ReMaGN.

Overall, ReMaGN has four planar tiers stacked on top of each other, each planar tier consisting of multiple ReRAM-based PEs with the same crossbar configuration. The PEs in different planar layers are connected with each other using TSV-based vertical links, as shown in Fig. 3(a). The 3-D NoC is further improved by a communication and multicast aware mapping policy that we elaborate later. Overall, the mapping policy complements the features of the 3-D NoC architecture and enables high-performance GNN training on the ReMaGN architecture. Note that, four tiers are used as an example only to show the efficacy of 3-D architectures for GNN training.

2) Role of Reduced Precision and Stochastic Rounding: Typically, ReRAMs compute using 16-bit fixed point, which has significantly less representation capability than 32-bit floating point used by traditional GPUs. However, our experimental analysis shows that a 16-bit fixed-point representation is sufficient for training GNNs and achieves similar accuracy as GPUs. In this work, we propose to use even lower-precision representations to reduce the heavy communication during GNN training. Lower precision representation reduces the traffic volume as the data are represented using fewer bits. This leads to higher throughput, which enables better performance [24]. In addition, reduced-precision representation also lowers the ReRAM requirement, which can lead to further speedup in computation as more weights can be duplicated to further parallelize in computation (as discussed in [12]). Both these characteristics are desirable for high-performance GNN training. However, this reduction of precision (lower than 16 bits) can often lead to unstable training or loss of prediction accuracy for GNNs.

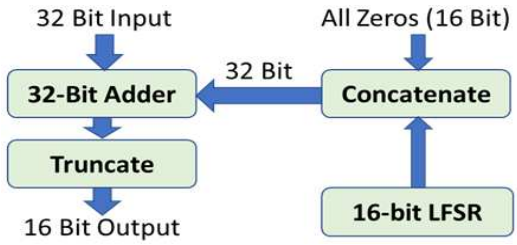

Fig. 7. Operation of the stochastic rounding unit (based on [27]).

Stochastic rounding is a probabilistic rounding scheme with a zero expected rounding error and is often used for reduced-precision training of DNNs [27]. Fig. 7 shows the operational principle of the stochastic rounding circuit used in ReMaGN. It consists of three parts: 1) LFSR: it generates pseudorandom 16-bit sequences; 2) Adder: it adds the 32-bit input with the 16-bit random number generated by the LFSR; and 3) Truncate: this truncates the result of addition to 16 bits after addressing overflow/underflow conditions. In this work, we show that both computation and communication can be made more efficient using lower precision computation for GNN training with the support of the stochastic rounding scheme. Hence, we propose to use stochastic rounding to improve the communication bottleneck in GNN training without sacrificing achievable accuracy. However, ReRAMs cannot inherently implement stochastic rounding and additional peripheral circuits are needed. We adopt the stochastic rounding circuit from [24], which adds minimal area overheads (less than 1% of total ReRAM PE area). Overall, the PEs considered in this work consist of ReRAM crossbars (128×128), necessary peripheral circuits (ADC, DAC, buffer, etc.), and an additional stochastic rounding unit, as shown in Fig. 3(a). We explore 16-, 12-, and 8-bit reduced-precision GNN training to establish the performance and accuracy tradeoffs.

D. Pipelined GNN Training

As mentioned in Section III, we employ a pipelined training methodology in ReMaGN (shown in Fig. 8). The adoption of a pipelined training allows for higher throughput in ReMaGN. However, pipelined training is not possible when training is carried out on a large monolithic graph. In addition, training GNNs on such a graph also require large memory footprint, which makes it impractical and inefficient. Graph partitioning is one of the approaches used to combat this high memory overhead. For instance, Chiang *et al*. implement GNN training with reduced memory requirements by using a

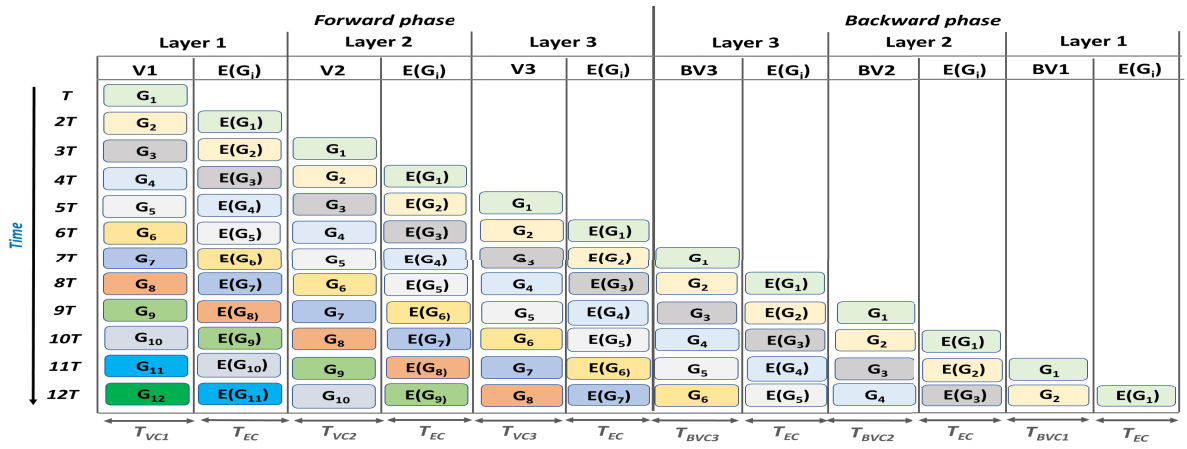

Fig. 8. Pipelined implementation of a layer deep GNN, with forward and backward phases. Each layer has two-sublayers–V-layer and E-layer.

graph partitioning tool (METIS [34]) to break a large graph into several smaller subgraphs, which allowed them to train GNNs over large graphs containing millions of nodes. Clustering/partitioning can be used to address this problem by introducing multiple subgraphs to be used as inputs for training. We can implement the pipelined training strategy for GNNs by dividing the graph into clusters. Fig. 8 shows an example of pipelined GNN training implementation on ReRAM-based architectures for a GNN with three neural layers. As discussed earlier, each neural layer can be further divided into two sublayers (E- and V-layers). Moreover, each E- and V-layers have a corresponding backward-phase computation layer (e.g., BV1 represents the backward-phase computation of $V1$ and so on). Here, V_i is the V-layer computation of the ith neural layer of the GNN. Overall, this results in a 12-stage training pipeline with two sublayers for each of the six neural layers, as shown in Fig. 8.

The GNN pipelined training works as follows: first, the large graph is partitioned into multiple smaller subgraphs where each subgraph is analogous to one input image in traditional DNNs. The size of each subgraph is chosen based on training time and end-to-end accuracy. At time T (Fig. 8), the first subgraph (G_1) is loaded for $V1$ layer computations. At the next timestamp $2T$, $G1$ advances for subsequent E-layer computation [$E(G_1)$], while the next subgraph G_2 is loaded for $V1$ layer computation and so on. Once the pipeline is filled (at time $12T$), all the PEs (corresponding to all forward and backward phases) are active all the time, leading to higher throughput and hardware utilization [12], [13], as shown in Fig. 8. The value of T (in Fig. 8) will depend on the maximum of computation/communication times for any given layer.

However, pipelined training also results in processing of multiple subgraphs simultaneously. For instance, at time $12T$ in Fig. 8, 12 subgraphs G_1–G_{12} are being processed. As mentioned earlier in Section III [and Fig. 1(d)], processing each subgraph results in a many-to-one-to-many traffic pattern. Hence, processing several subgraphs at the same time will result in multiple sets of many-to-one-to-many traffic patterns corresponding to each of the subgraphs present in the pipeline, resulting in high volume of data exchange which needs to be facilitated by the NoC. It is well known that in a pipeline, the slowest stage is the bottleneck and influences the overall execution time. Thus, improving the pipeline stage delay is essential in achieving high-performance GNN training in the proposed ReMaGN architecture.

E. Optimized Mapping of Neural Layers

The overall pipeline stage delay is also influenced by communication which depends on how CNN layers are mapped to ReRAM crossbars. For an efficient implementation of pipelined GNN training, all the neural layers need to be executed simultaneously (Fig. 8). This requires keeping all the neural weights and associated adjacency matrices on the chip. Hence, we also need to allocate (map) adequate resources (ReRAMs) to each neural layer and adjacency matrix based on the requirements. This mapping strategy is a key component in improving the communication stage delay in the pipeline. In addition, the mapping strategy is complementary to the NoC design as it influences on-chip traffic patterns. The aim of this mapping strategy is to reduce long-range traffic (as much as possible) while ensuring efficient multicast communication. The mapping of weights and the adjacency matrix to the PEs can be envisioned as a combinatorial optimization problem: given a total of P PEs, L layers (V-layers), and Adj adjacency matrices (E-layers), our goal is to distribute all computation layers such that the highly communicating layers are mapped to nearby PEs.

As mentioned earlier, the overall performance of GNN training on ReMaGN depends on the slowest stage of the training pipeline. It includes both the computation and communication latencies. While ReRAMs can provide sufficient resources to accelerate computation, it is necessary to have a complementary method to improve the communication latency as well. The mapping policy is used to ensure high-speed and low-latency communication between the PEs. Even if one link is overwhelmed by the large number of messages being sent, the overall execution time suffers. Hence, it is important to reduce the pipeline stage latency while reducing traffic hotspots. To achieve this goal, we use maximum link utilization as a metric to evenly distribute traffic across all the available links by mapping GNN neural layers and adjacency matrices suitably across the available ReRAM PEs. By lowering the maximum utilization, we avoid links becoming hotspots and prevent potential communication bottlenecks. Here, we follow the ReRAM performance models from [12], which results in a deterministic execution of GNN

Algorithm 1 Mapping GNN Layers on *ReMaGN*

Input: GNN architecture (L layers), Adjacency matrices (Adj), Hardware information – no. of available ReRAM PEs (P)
Output: Optimized mapping on *ReMaGN*
Algorithm:
1 *num-PE*[.] = No. ReRAM needed for given GNN
2 *Mapped*[.] = Random allocation of *num-PE*[.]
3 **Simulated Annealing (Repeat for Max Iterations)**:
4 *Mapped-New*[.] = ***Perturb***(*Mapped*[.])
5 ***Cost***(*Mapped-New*[.]) = *f*(*U*(*Mapped-New*[.]))
6 *If* **P**(***Cost***(*Mapped-New*[.]), *Annealing Temperature*)
7 *Mapped*[.] = *Mapped-new*[.]
8 **return** *Mapped*[.]

training on the proposed architecture. Therefore, it is possible to determine the on-chip traffic patterns for a given GNN mapping. By determining the traffic pattern, it is possible to redistribute the traffic over the chip following a simple simulated annealing-based optimization methodology elaborated in Algorithm 1 [20]. Overall, our objective is to find a suitable mapping of GNN layers on the ReMaGN architecture that prevents traffic hotspots to enable high-performance GNN training.

Algorithm 1 presents the optimization strategy used to determine the best mapping for ReMaGN. Assuming, we have a total of P PEs, L layers (V-layers), and Adj adjacency matrices where they are randomly mapped to the available PEs (Line 2). Next, we **Perturb** the candidate mapping solution to get a new mapping. A valid **Perturb** is defined as allocating all or part of the resources (PEs) required by a randomly chosen GNN layer or adjacency matrix to a different set of PEs than its current location. As in, moving the weights associated GNN layer $V1$ from PE_1 set of PEs to another set PE_2. Then, in order to evaluate the new mapping, we calculate the **Cost** of each mapping (Line 5), which is the max link utilization. Minimizing this cost helps in evenly distributing the traffic across all available links as mentioned before. We decide whether to discard/keep the new mapping in the archive based on the annealing temperature and **Cost** of both current and previous mapping solutions (Lines 6 and 7). Finally, we obtain the best possible mapping(s) (Line 8). We repeat the entire procedure (Algorithm 1) multiple times with different initial solutions and annealing schedules for a thorough exploration of the solution space of mapping policy. Note that this mapping optimization is a one-time offline process, and it does not add to overall run time of the system. By running the optimization once, we determine the optimized mapping for ReMaGN, which can then be used repeatedly. Thus, the optimization cost (time and energy) is amortized over multiple instantiations of ReMaGN.

Note that a similar mapping algorithm has been employed to get the best possible mapping for ReGraphX [33]. As mentioned earlier, ReGraphX consists of two different types of PEs, each designated for storing a specific layer. The V-layers can only be mapped to the larger crossbars. On the other hand, the adjacency matrices (E-layers) need to be mapped to the smaller crossbar-based PEs. This in turn restricts the possible optimized mapping scenarios. The rest of the mapping process in ReGraphX is the same as that for ReMaGN.

TABLE I
PARAMETERS OF THE REMAGN ARCHITECTURE

4 planar tiers, 9 PEs per tier, 4 tiles per PE	
ReRAM Tile (12 IMAs, In-situ Multiply -Accumlate Units)	1 IMA has: 8-ADCs (8-bits), 128x8 DACs (1-bit), 8 crossbars, 128x128 crossbar size, 10MHz, 2-bit resolution, 1 Stochastic Rounding Unit (SRU)

V. EXPERIMENTAL RESULTS

In this section, we first present the experimental setup to evaluate the performance of ReMaGN. We first explain our choice of the ReRAM crossbar configuration for ReMaGN. Next, we analyze the effect of various hyper-parameters when the GNN is trained on the proposed architecture. Then, we demonstrate the NoC performance and assess the performance and accuracy trade-offs of reduced-precision GNN training. Finally, we present the full-system performance evaluation of the proposed architecture and compare it against conventional GPUs.

A. Experimental Setup

The specific embodiment of the ReMaGN architecture considered in this work consists of 36 ReRAM-based PEs distributed over four planar tiers connected using through silicon via (TSV)-based vertical links. The ReRAM crossbar and tile configurations are shown in Table I. Note that the system size of 36 PEs was determined to provide sufficient storage capacity for all the weights and adjacency matrices considered in our experimental evaluation. The PEs communicate with each other via the 3-D NoC. To evaluate the characteristics of ReMaGN, we use the performance models from [12]. ReRAM arrays always execute instructions in-order and the instruction latencies are deterministic [12]. Hence, deterministic models have been used to evaluate ReRAM execution time, on-chip traffic, and so on [12]. The mapping of GNN layer weights and adjacency matrices on the tiles are determined offline. The traffic across the NoC is also statically determined to ensure conflict-free routing. We do not discuss the ReRAM execution models in detail for the sake of brevity and because they have been elaborated in [12]. To evaluate the performance of different NoCs, we use a cycle-accurate NoC simulator that can simulate any regular or irregular 3-D architecture [35]. To implement the different NoCs considered in this work, we followed the Garnet network structure [36]. As ReRAM crossbar arrays always execute instructions in-order and the instruction latencies are deterministic, we followed the deterministic ReRAM model proposed in [12] to calculate the injection rate needed for the cycle-accurate NoC simulator. The NoC simulator is used to obtain the communication latency for each NoC configuration. The deterministic ReRAM model explained in Section IV-A provides the execution time.

We also compare the performance of the proposed ReMaGN architecture with ReGraphX [33]. To the best of our knowledge, ReGraphX is the only existing ReRAM-based architecture for GNN training. As elaborated before, ReGraphX is a heterogeneous ReRAM-based architecture for GNN training. The fundamental difference between ReMaGN and ReGraphX is the ReRAM crossbar configuration and the number of PEs. ReGraphX includes two types of PEs, one for dense weight storage (128×128 crossbar) [12] and the other for sparse adjacency matrix storage (8×8 crossbars) [14]. The

PEs are distributed across three planar tiers connected using TSV-based vertical links. We compare the NoC and overall system performance of both these architectures in order to evaluate the efficacy of the proposed ReMaGN architecture.

As a representative GNN for performance evaluation, we use the popular graph convolutional network (GCN) algorithm [28] (implemented in TensorFlow). However, our findings and the proposed architecture are equally applicable to other GNNs that rely on the recursive message-passing scheme as all such GNNs share a similar computation and communication structures. The GCN configuration in our experiments uses graph partitioning to reduce memory overhead and to enable pipelined training.

Off-chip access is required for training GNNs on the ReMaGN architecture only when a new subgraph is loaded for processing. The time needed for this off-chip access must be lower than the latency of each stage of the ReMaGN pipeline to avoid execution stalls. As an example, a typical subgraph from the Amazon2M dataset (the largest dataset considered here) requires ∼35 MB of data. In our experimental setup, the off-chip memory access time required to load the data associated with each input subgraph of Amazon2M is ∼0.03 ms. This is much smaller than the pipeline stage delay, which is around 3 ms for the Amazon2M dataset. Similarly, for both PPI and Reddit, the off-chip access time is also significantly smaller than the on-chip pipeline stage delay. Hence, the execution is not bottlenecked by the off-chip memory access time here. For the evaluation of training GNNs on the ReMaGN architecture, we choose three popular graph datasets–PPI, Reddit, and Amazon2M (details provided in Table II). The GCN for each dataset consists of four neural layers. As Table II shows, the graph datasets are diverse in nature (in terms of size, partitioning, etc.). This allows comprehensive performance evaluation of ReMaGN.

B. Effect of GNN Hyperparameters on Performance

In this section, we explore how various GNN hyperparameters affect ReMaGN performance. The GCN configuration in our experiments employs the METIS graph partitioning tool [34] to reduce memory overhead and enable pipelined training. This allows us to evaluate GNNs for large-scale graphs which are otherwise impossible to process with limited memory, specifically in an on-chip environment. However, this can lead to the loss of important information (graph connections) that can cause unstable training due to biased gradients. To overcome this challenge, we use a stochastic multiclustering approach, which involves merging a random subset of subgraphs from all the partitions (NumParts) toward the goal of making the gradients unbiased over epochs [28]. The number of subgraphs that is merged back together to create an intermediate input subgraph is defined as batch size (β) in Table II. Hence, the number of effective input subgraphs (NumInput) for GNN training is obtained by dividing NumParts by β. Note that the notion of β in GNNs is not the same as in traditional DNNs. Batch size in DNNs refers to the total number of input images that need to be processed before DNN weights are updated.

Fig. 9 shows validation accuracy when different batch sizes (β) are used for the Reddit dataset as an example. In Fig. 9, we chose NumPart as 1500 (following [28]) while considering the values of β to be 1, 5, 10, and 20. From Fig. 9, it is clear that the choice of batch size does not affect the overall accuracy of the GNN significantly for the Reddit dataset. Similar observations were made for the other two datasets as well. However, it should be noted that smaller β affects convergence leading to unstable GNN training [28]. As a result, we choose a relatively larger β for stable GNN training.

However, larger β is costly in terms of hardware overhead as it leads to relatively larger intermediate input subgraphs. These larger subgraphs have a larger number of nodes and edges, resulting in an overall larger adjacency matrix. More ReRAM crossbars are needed to store larger adjacency matrices, thus increasing the requirement of overall hardware resources. Fig. 10 shows the effect of β on both the training time and hardware requirements (more specifically, ReRAM crossbar requirements) in ReMaGN. A smaller value of β results in smaller input subgraphs, which can be stored using fewer crossbars. However, it leads to higher NumInput (i.e., more input subgraphs) that need to be processed one-after-another, which causes the computational overhead for training to increase as a higher number of smaller subgraphs need to be processed. As shown in Fig. 10, this leads to higher training time. On the other hand, with increase in β, NumInput reduces, which in turn causes training time to decrease. However, higher β creates larger graphs leading to a drastic increase in ReRAM crossbar requirements. Interestingly, we note from Fig. 10 that the reduction in training time is relatively insignificant beyond $\beta = 10$, while crossbar requirements keep increasing steadily. From both Figs. 9 and 10, we note that larger β leads to faster and more stable training, which is desirable. However, it also necessitates more crossbars. Hence, we choose the maximum possible β whose crossbar requirements can be met by ReMaGN specifications (Table I). All the relevant parameters for GNN training on ReMaGN are listed in Table II.

TABLE II
GRAPH DATA STATISTICS AND GNN HYPERPARAMETERS

Datasets	No. of nodes	No. of edges	No. of partitions (NumPart)	Batch size (β)	No. of inputs (Num Input= Num Part/β)
PPI	56,944	818,716	250	5	50
Reddit	232,965	11,606,919	1500	10	150
Amazon2M	2,449,029	61,859,140	15000	10	1500

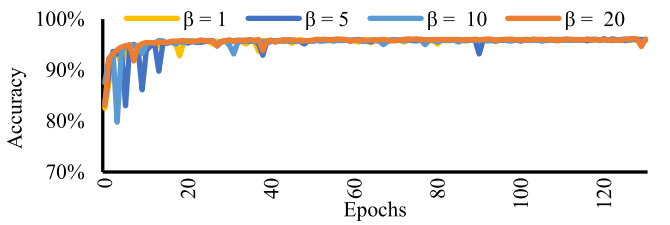

Fig. 9. GNN accuracy for different batch sizes for the Reddit dataset.

C. Crossbar Configuration

In this section, we first explain the choice of crossbar configuration for the ReMaGN architecture based on the storage efficiency, power, and area tradeoffs by varying the crossbar sizes from 8 × 8 to 256 × 256 [12], [37]. Fig. 11 shows the

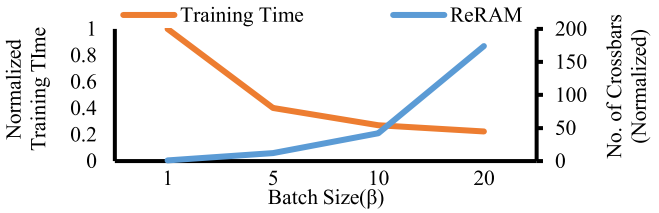

Fig. 10. Normalized training time and NumInputs for different batch size (β) for the Reddit dataset. All numbers have been normalized w.r.t. $\beta = 1$.

number of PEs required, area, and power needed to store one input subgraph of the Reddit dataset. It should be noted that similar trends were seen for other datasets as well. As shown in Fig. 11, 8 × 8 sized crossbars require 28× times more PEs than the 128 × 128 configuration to store the same amount of information. This happens as smaller crossbars need more PEs to store the same adjacency matrix, necessitating a larger number of peripheral circuits such as ADC, DAC, and routers. On the other hand, larger crossbars require fewer PEs, resulting in less peripheral circuits. This results in lower area and power overheads, in spite of the storage of more redundant zeros, as shown in Fig. 12. However, extremely large crossbars (beyond 128 × 128) require more area and power. This happens as peripheral circuits for such large crossbars are big. For instance, a 256 × 256 crossbar requires a 9-bit ADC [12], [37], which is not only difficult to design but is extremely area- and power-hungry, overshadowing any benefit of the larger crossbars (note that ADC is the most area- and power-hungry peripheral in an ReRAM tile [12]). The ADC resolution that is necessary for different crossbar sizes can be determined using the following equations [12]:

$$\text{Res} = \log(R) + \upsilon + w, \quad \text{if } \upsilon > 1 \text{ and } w > 1.$$

Here, Res is the ADC resolution, R is the number of rows in the ReRAM crossbar, υ represents the number of bits that is applied to the crossbar as input per cycle, and w represents the ReRAM cell resolution. It is clear from this equation that ADC resolution is proportional to the log of crossbar size (R); υ and w are fixed ($\upsilon = 1$ and $w = 2$ following [12]) for all crossbar settings in Fig. 9. Hence, larger crossbar sizes will require large ADCs. Based on our analysis in Fig. 11, a crossbar size of 128 × 128 is the effective choice in terms of area, power, and storage efficiency.

However, larger crossbars tend to be inefficient for storing sparse matrices that are mostly filled with zeros. Note that MAC operation with zeros is meaningless and should be avoided to reduce power consumption and improve performance. As shown in Fig. 12, 256 × 256 crossbars are the least efficient in this regard, while 8 × 8 is the best. Note that the number of zeros stored can be found using a sliding window operation, as shown in Fig. 5: any segment with at least one nonzero entry (defined as valid segment in Fig. 5) needs to be stored on ReRAM crossbars. A segment can only be discarded if all $N \times N$ entries are zeros (defined as invalid segment in Fig. 5), where N is the size of the crossbar. Larger crossbars tend to be inefficient as a result. For instance, a 256 × 256 crossbar requires consecutive 65 536 (=256 × 256) zero inputs in order to discard that segment which is very unlikely. Hence, smaller crossbars are preferred for storing sparse adjacency matrices. However, smaller crossbars necessitate a significantly larger number of

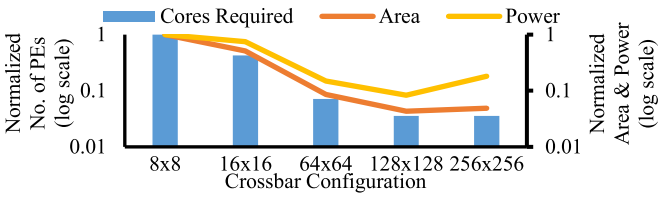

Fig. 11. Storage efficiency–power–area tradeoffs for different ReRAM crossbar configurations, normalized w.r.t. 8 × 8 crossbar configuration.

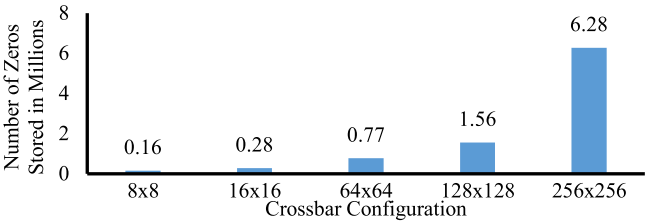

Fig. 12. Storage efficiency–power–area tradeoffs for different ReRAM crossbar configurations, normalized w.r.t. 8 × 8 crossbar configuration.

PEs to store the same input as when compared with larger crossbar configurations (Fig. 11).

This phenomenon presents an interesting trade-off between area, power, and storage efficiency. From both Figs. 11 and 12, we see that 256 × 256 sized crossbars are not only inefficient for storing sparse matrices, but also consume more area and power. ReGraphX uses 8 × 8 crossbars to efficiently store the sparse adjacency matrices. However, as shown in Fig. 11, this design necessitates many PEs, resulting in a much larger system size. The larger system size results in higher hop count in communication and it is also not efficient in terms of area and power. The 128 × 128 crossbar configuration is, therefore, the most suitable choice and is used in ReMaGN. We show later that ReMaGN outperforms ReGraphX in terms of both performance and power efficiency.

D. NoC Evaluation and Performance and Accuracy Tradeoff

We design the ReRAM-based PEs of the ReMaGN architecture using the aforementioned 128 × 128 crossbar configuration. The PEs communicate with each other using a 3-D NoC. Hence, we evaluate the performance of the 3-D NoC, which serves as the communication backbone for ReMaGN. We also explore the effect of reduced-precision representation on the computation and communication pipeline delays of ReMaGN. Following this, we assess the accuracy and performance tradeoffs for training GNNs using reduced-precision representation.

As discussed in Section IV, we implement the GNN training on ReMaGN in a pipelined fashion. It is well known that the slowest stage in the pipeline determines the overall achievable performance. Here, each stage of the pipeline represents either the vertex computation shown in (1) or edge computation shown in (2) of a GNN neural layer. Each stage involves a set of MAC operations and communication of subsequent outputs to the next stage. Hence, the overall time needed to accomplish each stage is determined by the slower of the computation and communication tasks. As discussed earlier, GNN training involves many-to-one-to-many and heavy multicast traffic that can bottleneck the overall performance. Moreover, reduced-precision training lowers NoC traffic and increases the overall

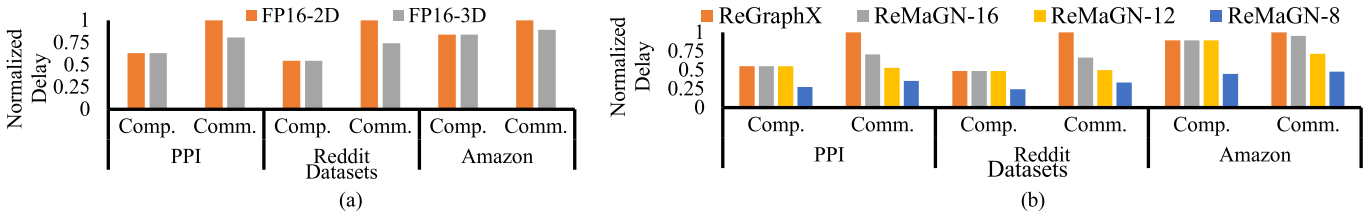

Fig. 13. Effect of (a) 3-D NoC on computation (comp.) and communication (comm.) delay, normalized w.r.t. FP16-2D comm. delay. (b) Reduced-precision representation on computation (comp.) and communication (comm.) delay for ReMaGN-P (P denotes the precision), normalized w.r.t. ReGraphX comm. delay.

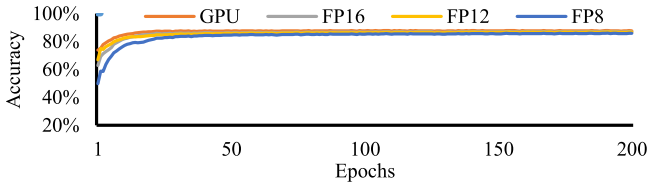

Fig. 14. Accuracy of GNN training with reduced-precision representation for the Amazon2M dataset.

throughput as fewer bits need to be communicated. Recall that the precision of the baseline ReMaGN configuration is 16-bit fixed point (FP16), in accordance with prior works [12], [24]. Hence, FP-16 can be considered as a representative of the existing architectures such as PipeLayer [13]and ISAAC [12].

First, we demonstrate the advantage of 3-D NoC by comparing the performance of a conventional multicast-enabled 2-D mesh NoC (i.e., FP16-2D) with respect to ReMaGN with 16-bit precision (FP16-3D). Fig. 13(a) shows the worst case computation and communication times observed among all the GNN layers (as pipeline latency is determined by the slowest stage) for different datasets with FP16-2D and FP16-3D. As shown in Fig. 13(a), a conventional multicast-enabled 2-D mesh NoC (i.e., FP16-2D) makes the communication time to dominate the computation time significantly. It is clear that the communication remains the bottleneck. Therefore, we design a multicast-enabled 3-D NoC as the communication backbone for ReMaGN. Going from 2-D to 3-D, there is no change in computation time as the number of ReRAMs remain unchanged. However, for FP16, the 3-D NoC (FP16-3D) lowers the communication delay in ReMaGN by up to 25%. This happens due to the fact that 3-D NoC lowers the overall hop count by bringing the PEs physically closer and increases the overall throughput [19]. Fig. 13(a) clearly demonstrates the benefit of the 3-D NoC architecture; hence, this architecture is subsequently used in conjunction with reduced-precision training in ReMaGN. It should be noted that the computation stage delay in both architectures is the same as it is not affected by the NoC. It is possible to accelerate the computation by further parallelizing the neural layers in ReRAM [13]. However, the communication latency is higher than the computation latency for all datasets. For instance, the communication stage delay of the FP16-3D architecture is 1.28× larger than the computation delay for the PPI dataset. Hence, the overall execution time will be bottlenecked by the communication delay and any acceleration of computation will fail to provide any benefit.

In Fig. 13(b), we compare the performance of ReMaGN with different precision settings with respect to ReGraphX [33]. The different configurations of ReMaGN are presented as ReMaGN-P, where P denotes the precision at which the GNN is trained. It should be noted that the ReGraphX implementation does not have any reduced-precision representation and operates with default 16-bit precision (FP16). ReMaGN-16 can be directly compared with ReGraphX as they both operate with 16-bit representation. ReMaGN-16 outperforms ReGraphX by up to 33.3% in terms of the communication delay, thus, reducing the overall pipeline stage delay by 33.3% as well. Moreover, reducing precision helps further improve the throughput of ReMaGN. As shown in Fig. 13(b), the use of reduced-precision in conjunction with the multicast-enabled 3-D mesh NoC in ReMaGN can achieve 39% (ReMaGN-12) and 59% (ReMaGN-8) better performance, on an average, in terms of communication compared with ReGraphX. This improvement in performance comes from the much lower number of hops between PEs and improved mapping in ReMaGN. Here, it should be noted that reduced precision does not affect the overall data (activations and gradients) that needs to be communicated during training. The same amount of data is conveyed by using a fewer number of bits, thus reducing the overall traffic. This results in improved NoC performance.

In addition, as mentioned in Section IV, reduced precision lowers ReRAM PE requirements. For instance, only 12 bits need to be stored in ReRAMs for ReMaGN-12 (which requires six ReRAM cells [12]) as opposed to 16 bits (which requires eight ReRAM cells [12]) in the default baseline setting. Each ReRAM cell stores only 2 bits. This in turn allows us to duplicate the weights on the unused ReRAMs, enabling higher performance. Note that the duplication of weights is a common strategy to accelerate computation on ReRAMs [13]. As shown in Fig. 13(b), computation time is reduced by 50% for the ReMaGN-8 configuration for all datasets. However, in the case of ReMaGN-12, we do not see any decrease in computation time. This happens because the reduction in ReRAM PE requirement in ReMaGN-12 is not sufficient to enable duplication of all the weights of the slower GNN layers. Note that in a pipelined implementation, the slowest stage dominates the overall execution time. Hence, unless all the slow GNN layers are accelerated, the worst case pipeline delay will remain unchanged as shown in Fig. 13(b). Overall, our results show that reduced precision in ReMaGN lowers both computation and communication delays compared with the ReGraphX leading to significantly better performance.

It is well known that reduced precision has lower representation capability, which can lead to accuracy loss or unstable training. Hence, it is important to evaluate the GNN training accuracy for various precision levels. Fig. 14 shows the GNN accuracy with varying levels of precision. As an example, we have chosen the Amazon2M dataset here noting that the

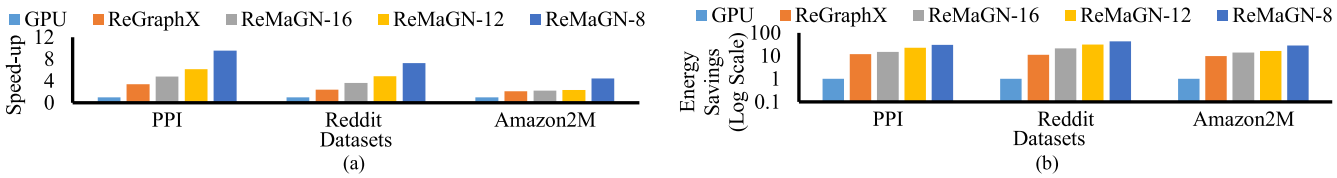

Fig. 15. ReMaGN-P (P denotes the precision). (a) Execution time speedup and (b) energy savings compared with GPU and ReGraphX (normalized w.r.t. GPU).

results are similar to other datasets. As shown in Fig. 14, FP16 (the default ReRAM configuration) achieves GPU-level accuracy (which uses 32-bit floating point). The accuracies of FP12 and FP8 are less than 1% and 2%, respectively, compared with the GPU-based counterpart. However, as shown later in Fig. 15, FP12 and FP8 outperform the GPU-based design by almost an order of magnitude. This shows the performance and accuracy tradeoffs associated with reduced precision. If an application has extremely stringent accuracy requirement, then only FP16-based architecture should be used. However, if 1%–2% accuracy loss can be tolerated, then FP12 and FP8 should be the preferred design due to the huge performance benefits. The best instantiation of ReMaGN architecture can be decided by the user based on the specific application requirements.

Note that training fails for both FP12 and FP8 when stochastic rounding is not used. However, below FP8, the accuracy drop was significant even with stochastic rounding. Overall, the use of stochastic rounding in ReMaGN allows us to achieve better performance and accuracy tradeoffs. From Figs. 13 and 14, we note that it is possible to train at lower precisions leading to higher performance without sacrificing prediction accuracy by the stochastic rounding approach.

E. Full System Evaluation

Next, we undertake a full system performance evaluation and compare the overall execution time and energy consumption of ReMaGN with respect to the ReGraphX and a conventional GPU-baseline (Nvidia V100 in this case). It should be noted that the training duration and energy consumption depend on the actual wall-clock time and not the epoch. As shown in Fig. 14, GPU is reaching convergence in a smaller number of epochs compared with ReRAM-based architectures. However, the actual time corresponding to each epoch in ReRAM-based architectures is much smaller than that of GPU. Fig. 15(a) and (b) shows the speedup and improvement in energy consumption, respectively, in ReMaGN and ReGraphX compared with the GPU. The different configurations of the ReMaGN architecture are denoted by ReMaGN-P, where P denotes the precision. Fig. 15(a) shows that ReGraphX and ReMaGN-16, using FP16 representation, outperform conventional GPU-based systems by 2.7× and 3.5×, respectively, in terms of execution time on an average considering all the datasets. Using reduced-precision representation, ReMaGN-12 and ReMaGN-8 achieve 4.4× and 7.1× speedup, respectively, on an average. The speedup of the ReRAM-based architectures comes from: 1) high-throughput PIM-based MAC operations on ReRAM PEs; 2) efficient communication through 3-D multicast-enabled NoC; and 3) faster computation and communication enabled by reduced-precision representation. In addition, ReMaGN achieves higher performance than ReGraphX due to the smaller number of PEs in the overall architecture and, hence, fewer hop counts resulting in efficient communication.

Moreover, both the ReRAM-based architectures also have lower energy consumption than the GPU-based system due to the energy efficiency of ReRAM-based crossbars. As shown in Fig. 15(b), ReGraphX and ReMaGN-16 are 11× and 16.7× more energy efficient than the GPU, respectively. Lowering precision further improves energy efficiency. ReMaGN-12 and ReMaGN-8 are 23.2× and 33.5× more energy efficient than the GPU baseline, respectively. This shows that the proposed ReMaGN architecture outperforms contemporary ReRAM-based GNN accelerators such as ReGraphX by a considerable margin while being more energy efficient.

As shown in Section V, the 128 × 128 crossbar configuration also outperforms 8 × 8 crossbars in terms of storage efficiency–power–area tradeoffs. As 8 × 8 crossbar-based PEs have lower storage efficiency, ReGraphX requires more PEs to store the same information compared with ReMaGN. As a result, the proposed ReMaGN architecture requires lower number of PEs compared with ReGraphX. The overall smaller system size results in lower energy consumption and reduces the average hop count needed for inter-PE communication in ReMaGN compared with ReGraphX. The lower hop count results in improved latency and overall higher performance of ReMaGN. Hence, the homogeneous system in ReMaGN has a better performance and energy efficiency than the heterogeneous ReGraphX architecture. Fig. 13(b) shows the improved communication latencies achieved by the ReMaGN NoC due to lower hop counts. Moreover, as shown in Fig. 15(a) and (b), all instantiations of ReMaGN have higher execution time speedup and energy savings when compared with the heterogeneous ReGraphX architecture. Overall, the ReMaGN architecture is significantly faster and more energy efficient than both GPUs and ReGraphX for training GNNs.

VI. Conclusion

GNNs have found applications in various domains such as social media, recommendation systems, and drug discovery. In this work, we have presented a novel ReRAM-based many-core PIM architecture: ReMaGN, specifically designed for accelerating GNN training over large-scale graphs. ReMaGN is supported by a multicast-enabled 3-D NoC architecture and robust reduced-precision training with stochastic rounding. Overall, ReMaGN outperforms conventional GPUs by up to 9.5× in terms of execution time while being up to 42× more energy efficient. These gains are derived from efficient MAC computation in ReRAM, high-throughput communication enabled by 3-D integration, and robust reduced-precision training process. Moreover, ReMaGN outperforms ReGraphX, a contemporary ReRAM-based GNN accelerator, by up to 2.8× in terms of execution time while using 3.8× less energy.

Finally, it should be noted that ReRAMs have reliability issues such as noise, hard faults (e.g., stuck-at-faults), and wear out due to limited write endurance [32], [38]. Such nonidealities can adversely affect the performance of ReRAM-based architectures such as ReMaGN. Techniques such as thermal reference cells [24] and weight clipping [39] are promising solutions toward robust GNN training on ReRAM-based architectures. However, studying the reliability and performance tradeoffs for GNN training is beyond the scope of this article and is the focus of our immediate future work.

REFERENCES

[1] W. Fan *et al.*, "Graph neural networks for social recommendation," in *Proc. World Wide Web Conf.*, San Francisco, CA, USA, May 2019, pp. 417–426.

[2] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, "Graph convolutional neural networks for web-scale recommender systems," in *Proc. 24th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, London, U.K., Jul. 2018, pp. 974–983.

[3] F. Ding. (Jul. 2019). *Graph Neural Networks for Quantum Chemistry*. [Online]. Available: https://github.com/ifding/graph-neural-networks

[4] J. Zhou *et al.*, "Graph neural networks: A review of methods," 2018, *arXiv:1812.08434*. [Online]. Available: https://arxiv.org/abs/1812.08434

[5] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *Proc. Int. Conf. Learn. Represent. (ICLR)*, Toulon, France, 2017, pp. 1–14.

[6] M. Zhang *et al.*, "An end-to-end deep learning architecture for graph classification," in *Proc. AAAI Conf. Artif. Intell.*, New Orleans, LA, USA, 2018, pp. 4438–4445.

[7] G. Murali, X. Sun, S. Yu, and S.-K. Lim, "Heterogeneous mixed-signal monolithic 3-D in-memory computing using resistive RAM," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 29, no. 2, pp. 386–396, Dec. 2021.

[8] Y. Chen, L. Lu, B. Kim, and T. T.-H. Kim, "Reconfigurable 2T2R ReRAM architecture for versatile data storage and computing in-memory," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 28, no. 12, pp. 2636–2649, Dec. 2020.

[9] D. Fujiki, S. Mahlke, and R. Das, "In-memory data flow processor," in *Proc. 26th Int. Conf. Parallel Architectures Compilation Techn. (PACT)*, Portland, OR, USA, Sep. 2017, p. 375.

[10] K. Kiningham, C. Re, and P. Levis, "GRIP: A graph neural network accelerator architecture," 2020, *arXiv:2007.13828*. [Online]. Available: http://arxiv.org/abs/2007.13828

[11] M. Hu *et al.*, "Dot-product engine for neuromorphic computing: Programming 1T1M crossbar to accelerate matrix-vector multiplication," in *Proc. 53rd Annu. Design Autom. Conf.*, Austin, TX, USA, Jun. 2016, pp. 1–6.

[12] A. Shafiee *et al.*, "ISAAC: A convolutional neural network accelerator with *in-situ* analog arithmetic in crossbars," in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit. (ISCA)*, Seoul, South Korea, Jun. 2016, pp. 14–26.

[13] L. Song, X. Qian, H. Li, and Y. Chen, "PipeLayer: A pipelined ReRAM-based accelerator for deep learning," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Austin, TX, USA, Feb. 2017, pp. 541–552.

[14] L. Song, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "GraphR: Accelerating graph processing using ReRAM," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Vienna, Austria, Feb. 2018, pp. 531–543.

[15] G. Dai, T. Huang, Y. Wang, H. Yang, and J. Wawrzynek, "GraphSAR: A sparsity-aware processing-in-memory architecture for large-scale graph processing on ReRAMs," in *Proc. 24th Asia South Pacific Design Autom. Conf.*, New York, NY, USA, Jan. 2019.

[16] L. Zheng *et al.*, "Spara: An energy-efficient ReRAM-based accelerator for sparse graph analytics applications," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, New Orleans, LA, USA, May 2020, pp. 696–707.

[17] B. K. Joardar, R. G. Kim, J. R. Doppa, P. P. Pande, D. Marculescu, and R. Marculescu, "Learning-based application-agnostic 3D NoC design for heterogeneous manycore systems," *IEEE Trans. Comput.*, vol. 68, no. 6, pp. 852–866, Jun. 2019.

[18] K. Duraisamy, Y. Xue, P. Bogdan, and P. P. Pande, "Multicast-aware high-performance wireless network-on-chip architectures," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 25, no. 3, pp. 1126–1139, Mar. 2017.

[19] B. S. Feero and P. P. Pande, "Networks-on-chip in a three-dimensional environment: A performance evaluation," *IEEE Trans. Comput.*, vol. 58, no. 1, pp. 32–45, Jan. 2009.

[20] B. K. Joardar, N. Kannappan Jayakodi, J. R. Doppa, H. Li, P. P. Pande, and K. Chakrabarty, "GRAMARCH: A GPU-ReRAM based heterogeneous architecture for neural image segmentation," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Grennoble, France, Mar. 2020, pp. 228–233.

[21] P. Chi *et al.*, "PRIME: A novel processing-in-memory architecture for neural network computation in ReRAM-based main memory," in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit. (ISCA)*, Seoul, South Korea, Jun. 2016, pp. 27–39.

[22] B. K. Joardar, B. Li, J. R. Doppa, H. Li, P. P. Pande, and K. Chakrabarty, "REGENT: A heterogeneous ReRAM/GPU-based architecture enabled by NoC for training CNNs," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Florence, Italy, Mar. 2019.

[23] B. Li, J. R. Doppa, P. P. Pande, K. Chakrabarty, J. X. Qiu, and H. Li, "3D-ReG: A 3D ReRAM-based heterogeneous architecture for training deep neural networks," *ACM J. Emerg. Technol. Comput. Syst.*, vol. 16, no. 2, pp. 1–24, Apr. 2020.

[24] B. K. Joardar, J. R. Doppa, P. P. Pande, H. Li, and K. Chakrabarty, "AccuReD: High accuracy training of CNNs on ReRAM/GPU heterogeneous 3-D architecture," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 40, no. 5, pp. 971–984, May 2021.

[25] Y. Long, T. Na, and S. Mukhopadhyay, "ReRAM-based processing-in-memory architecture for recurrent neural network acceleration," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 26, no. 12, pp. 2781–2794, Dec. 2018.

[26] P. Micikevicius, "Mixed precision training," in *Proc. Int. Conf. Learn. Represent. (ICLR)*, Vancouver, BC, Canada, 2018, pp. 1–12.

[27] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *Proc. Int. Conf. Mach. Learn.*, 2015, pp. 1737–1746.

[28] W.-L. Chiang, X. Liu, S. Si, Y. Li, S. Bengio, and C.-J. Hsieh, "Cluster-GCN: An efficient algorithm for training deep and large graph convolutional networks," in *Proc. 25th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, Anchorage, AK, USA, Jul. 2019, pp. 257–266.

[29] T. Geng *et al.*, "AWB-GCN: A graph convolutional network accelerator with runtime workload rebalancing," in *Proc. 53rd Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2020, pp. 922–936.

[30] A. Auten, M. Tomei, and R. Kumar, "Hardware acceleration of graph neural networks," in *Proc. 57th ACM/IEEE Design Autom. Conf. (DAC)*, San Francisco, CA, USA, Jul. 2020, pp. 1–6.

[31] A. Tripathy, K. Yelick, and A. Buluc, "Reducing communication in graph neural network training," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2020, pp. 1–14.

[32] Z. He, J. Lin, R. Ewetz, J.-S. Yuan, and D. Fan, "Noise injection adaption: End-to-end ReRAM crossbar non-ideal effect adaption for neural network mapping," in *Proc. 56th Annu. Design Autom. Conf.*, Jun. 2019, pp. 1–6.

[33] A. I. Arka *et al.*, "ReGraphX: NoC-enabled 3D heterogeneous ReRAM architecture for training graph neural networks," in *Proc. IEEE/ACM Design Automat. Test Eur. (DATE)*, 2021, pp. 1666–1672.

[34] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM J. Sci. Comput.*, vol. 20, no. 1, pp. 359–392, Aug. 1999.

[35] S. Das, J. R. Doppa, D. H. Kim, P. P. Pande, and K. Chakrabarty, "Optimizing 3D NoC design for energy efficiency: A machine learning approach," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, Nov. 2015, pp. 705–712.

[36] N. Agarwal, T. Krishna, L. Peh, and N. Jha, "GARNET: A detailed on-chip network model inside a full-system simulator," in *Proc. ISPASS*, Boston, MA, USA, 2009, pp. 33–42.

[37] B. Murmann. (2021). *ADC Performance Survey 1997-2020*. [Online]. Available: https://web.stanford.edu/~murmann/publications/ADCsurvey_rev20200802.xls

[38] A. Chaudhuri and K. Chakrabarty, "Analysis of process variations, defects, and design-induced coupling in memristors," in *Proc. IEEE Int. Test Conf. (ITC)*, Phoenix, AZ, USA, Oct. 2018, pp. 1–10.

[39] B. K. Joardar *et al.*, "Learning to train CNNs on faulty ReRAM-based manycore accelerators," *ACM Trans. Embedded Comput. Syst.*, 2021.