Accelerating Large-Scale Graph Neural Network Training on Crossbar Diet

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

***Abstract*—Resistive random-access memory (ReRAM)-based manycore architectures enable acceleration of graph neural network (GNN) inference and training. GNNs exhibit characteristics of both DNNs and graph analytics. Hence, GNN training/inferencing on ReRAM-based manycore architectures give rise to both computation and on-chip communication challenges. In this work, we leverage model pruning and efficient graph storage to reduce the computation and communication bottlenecks associated with GNN training on ReRAM-based manycore accelerators. However, traditional pruning techniques are either targeted for inferencing only, or they are not crossbar-aware. In this work, we propose a GNN pruning technique called DietGNN. DietGNN is a crossbar-aware pruning technique that achieves high accuracy training and enables energy, area, and storage efficient computing on ReRAM-based manycore platforms. The DietGNN pruned model can be trained from scratch without any noticeable accuracy loss. Our experimental results show that when mapped on to a ReRAM-based manycore architecture, DietGNN can reduce the number of crossbars by over 90% and accelerate GNN training by ∼2.7× compared to its unpruned counterpart. In addition, DietGNN reduces energy consumption by more than ∼3.5× compared to the unpruned counterpart.**

***Index Terms*—Graph neural network (GNN), processing-in-memory (PIM), pruning, resistive random access memory (ReRAM).**

I. Introduction

GRAPH neural networks (GNNs) have recently become the mainstream approach for performing cognitive tasks, such as node classification, link-prediction, and visualization on graph-structured data [1]. Each GNN layer transforms graph nodes to a low-dimensional embedding space and aggregates the node features of its k-hop neighbors from previous layers [2]. The corresponding computational kernel is compute-intensive, as each GNN layer simultaneously performs matrix-vector multiplication (MVM) operations on its input vectors, neural layer weights, and graph adjacency matrices to produce output activations. These activations are then forwarded to other layers, which results in the high volume of on-chip traffic [3]. Overall, GNN training requires appropriate hardware support to address both the computation and communication challenges, which is often not possible on smaller edge/mobile platforms.

Training machine learning (ML) models at the edge (training on-chip or on embedded systems) can address many pressing challenges, including data privacy/security, increase the accessibility of ML applications to different parts of the world by reducing the dependence on the communication fabric and the cloud infrastructure, and meet the real-time requirements of AR/VR applications. Specifically, AR/VR applications require GNN training on embedded systems [4]. However, existing edge platforms do not have sufficient capabilities to support on-device training of GNNs. Moreover, it is estimated that training a single unpruned neural network on conventional compute platforms, such as GPUs, can cost over $10 000 and emit as much carbon as five cars over their lifetimes [5].

Resistive random-access memory (ReRAM)-based processing-in-memory (PIM) architectures can be used to address this problem. ReRAM-based PIM systems have been proposed to accelerate GNN computation [3]. The crossbar structure of ReRAM-based architectures enables efficient MVM operations, which are ubiquitous in modern ML tasks including GNN training [6], [7], [8]. In addition, the design of suitable Network-on-Chip (NoC) along with Dropout and DropEdge-based regularization can improve the communication throughput [9]. As a result, ReRAM-based architectures outperform GPUs significantly in terms of energy efficiency and execution time speedup for GNN training.

Despite these promising developments, ReRAM-based architectures are not scalable with the size of GNNs and the size of input graphs. GNNs can have multiple layers and each layer can have thousands of weights. Similarly, real-world graphs can have several thousand/millions of nodes/edges. In a ReRAM-based architecture, the execution of all neural layers happens in parallel [7]. This necessitates a very high amount of GNN weights, and the huge graph adjacency matrices, to be stored on-chip at the same time. Storing all these data requires many ReRAM crossbars along with the associated peripheral

Manuscript received 17 July 2022; accepted 26 July 2022. Date of publication 9 August 2022; date of current version 24 October 2022. This work was supported in part by the U.S. National Science Foundation (NSF) under Grant CNS-1955353 and Grant CNS-1955196. The work of Biresh Kumar Joardar was supported in part by NSF to the Computing Research Association for the CIFellows Project under Grant 2030859. This article was presented in the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES) 2022 and appears as part of the ESWEEK-TCAD special issue. This article was recommended by Associate Editor A. K. Coskun. *(Corresponding author: Partha Pratim Pande.)*

Chukwufumnanya Ogbogu, Aqeeb Iqbal Arka, Janardhan Rao Doppa, and Partha Pratim Pande are with the Department of Electrical Engineering and Computer Science, Washington State University, Pullman, WA 99164 USA (e-mail: c.ogbogu@wsu.edu; aqeebiqbal.arka@wsu.edu; jana.doppa@wsu.edu; pande@wsu.edu).

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

Digital Object Identifier 10.1109/TCAD.2022.3197342

circuits, which incurs considerable energy and area overhead. For example, training an unpruned GNN with over 9 million parameters on the Amazon2M graph dataset (which consists of over 2 million nodes and 11 million edges) requires over 36 GB of memory. Such a large amount of memory is not available in smaller hand-held devices. Hence, existing ReRAM-based PIM architectures for GNN training are not scalable with the size of GNNs and addressing this critical challenge is the primary focus of this work.

Pruning is a popular technique for reducing the number of parameters of any deep neural network (including GNNs) by making some of the weights zero. These pruned weights need not be stored on the chip as multiplying with zero always results in a zero; hence, there is no need to multiply [10], [11]. This reduces the storage and computational requirements, which makes pruning an attractive choice for enabling energy- and storage- efficient GNN computation, that is, *training on a crossbar diet without compromising accuracy*. State-of-the-art pruning methods can trim more than 90% of the model parameters with negligible accuracy loss [12]. Recent work has proposed pruning ideas specific to GNNs, e.g., joint weight and graph sparsification, to reduce multiply-and-accumulate (MAC) operations [13]. However, these pruning approaches are oblivious of hardware and not optimized for crossbar-based architectures. We show in this work that these approaches do not result in significant crossbar savings and incur significant accuracy loss after training. Crossbar-aware pruning techniques are necessary to address this challenge [14], [15], [16], [17], [18]. However, existing crossbar-aware methods target only inferencing and often do not result in significant amount of pruning for the purpose of training. Moreover, existing crossbar-aware pruning methods do not take into account the overall crossbar structure and only focus on row- or column-wise pruning. We show later that such pruning does not result in overall hardware savings. In this article, our goal is to enable the training of pruned GNNs from scratch without significant accuracy loss while reducing the overall hardware requirements considerably.

We propose a crossbar-aware pruning technique called DietGNN (GNN pruning on a crossbar diet) to address the storage, computation, and communication challenges of ReRAM-based GNN accelerators. DietGNN is motivated by the recently proposed lottery ticket pruning (LTP) hypothesis [12], which states that "dense, randomly initialized, networks contain subnetworks (winning tickets) that-when trained from scratch can reach the test accuracy of the unpruned network." In DietGNN, we integrate key insights from LTP with the ReRAM crossbar structure and mapping strategy to produce hardware-friendly sparse GNNs (also know as, hardware-friendly lottery tickets) that can be trained with negligible accuracy loss. To complement weight pruning, DietGNN adopts an efficient zero-storage mechanism to reduce the crossbar requirement for storing graph adjacency matrices on-chip. We train the pruned GNN model generated by DietGNN on a ReRAM-based 3-D manycore PIM architecture. Our DietGNN-enabled manycore architecture achieves low energy- and storage-efficient GNN computation. The key contributions of this work are summarized as follows.

1) We demonstrate that it is possible to prune more than 90% of GNN weights for diverse GNNs and real-world graph datasets.
2) The pruned GNNs enable significant hardware savings (that is, crossbar diet) and performance improvements without sacrificing accuracy.
3) The experimental results demonstrate that training the DietGNN-enabled pruned GNN model on the ReRAM-based manycore architecture achieves $\sim 2.7\times$ speedup compared to the unpruned version.

To the best of our knowledge, this is first attempt to develop and apply a crossbar-aware pruning technique for GNN training on massive real-world graphs. The remainder of this article is organized as follows. Section II discusses relevant prior work related to pruning, GNNs, and existing ReRAM-based accelerators. Section III presents an overview of GNNs, demonstrates the importance of pruning in relation to crossbar architectures and describes the DietGNN pruning technique. Section IV discusses the training of the DietGNN-enabled model on the manycore architecture. We present comprehensive results on a variety of real-world datasets in Section V, and finally, conclude our findings in Section VI.

II. Related Prior Work

In this section, we discuss prior work on GNN training and inferencing on ReRAM-based architectures and relevant GNN pruning methods.

A. ReRAM-Based Accelerators

ReRAM crossbar arrays can be used to perform MVM operations, which are predominant in deep neural network training and inferencing [6], [7], [19]. Recent work has proposed state-of-the-art ReRAM-based architectures that can accelerate training [7], [20] and inferencing [6], [19] of convolutional neural networks (CNNs). However, both training and inferencing are affected by the nonideal nature of ReRAM cells [21], [22]. ReRAM-based systems have low precision and limited write endurance that can affect the accuracy of the ML model. These challenges can, however, be addressed easily, as shown in recent work. For example, in [23], a low-rank training (LRT) algorithm was used to address the write endurance problem of ReRAM-based architectures. Adopting this methodology results in an overall lifetime of $\sim$10 years for the proposed ReRAM-based architecture for CNN training. Similarly, accuracy loss due to low precision can be avoided by using stochastic rounding [24]. However, all these existing architectures have been focused only on CNNs.

Unlike CNNs, the GNN training involves graph and weight-matrix computations, both of which require high storage and heavy data movement [3]. The significant amount of data movement resulting from GNN training/inferencing workloads pose a unique communication challenge for ReRAM-based manycore architectures [3]. ReRAM-based accelerators for GNN computation have been proposed [3], [9], [25], [26]. To address the communication bottleneck and hardware requirements during GNN training, reduced-precision representation can be used [3]. However,

training under low precision can lead to high accuracy loss in GNNs. Arka *et al.* [9] used a regularization method called "DropLayer" to reduce activations, thereby reducing the volume of data movement without any loss in accuracy. However, all existing ReRAM-based GNN accelerators suffer from a major shortcoming: they do not address the challenge of high crossbar requirement as all weights and graphs still need to be stored on-chip during training. Therefore, to address the storage requirement, we need to explore novel pruning techniques suitable for ReRAM crossbar-based manycore platforms.

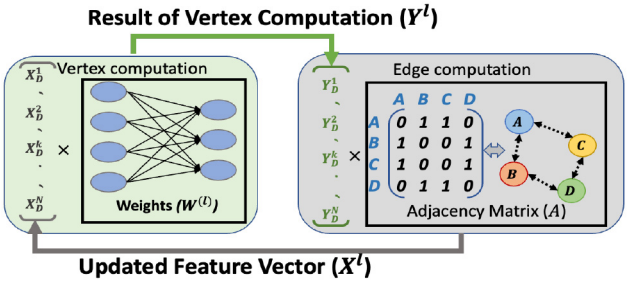

Fig. 1. Two phases of GNN computation kernel.

B. Graph Neural Network Pruning

Several pruning methods for deep neural networks have been proposed in [27], [28], [29]. However, these methods are targeted for inferencing purposes. These pruned models fail to match the accuracy of their unpruned counterparts when training is carried out from scratch. LTP is a recently proposed pruning technique for the purpose of training. LTP uses iterative magnitude pruning of the neural layer weights to obtain highly sparse models [12]. These pruned models can be trained from scratch with negligible accuracy loss. However, the LTP hypothesis is focused only on CNNs. Recently, a unified GNN sparsification (UGS) methodology has been proposed, which jointly prunes GNN weights and graph adjacency matrices using trainable masks to reduce the number of MAC operations associated with GNN training [13]. However, pruning the graph causes information loss and leads to significant accuracy degradation [10]. Moreover, since UGS does not consider the crossbar structure, it is not suitable for ReRAM crossbar-based platforms. Hence, crossbar-aware pruning methods for GNN models are necessary.

Existing crossbar-aware pruning methods leverage the idea of structured pruning to reduce the storage and energy of ReRAM-based architectures [14], [15], [16], [17], [18]. However, existing structured pruning methods principally adopt either row- or column- wise pruning. As we show later in our experiments, this does not lead to significant crossbar savings. Instead, implementing some of these existing pruning methods gives rise to very high hardware overhead [16]. The sparsity of graph structured data poses another significant challenge in efficient computation and storage due to the presence of redundant zeros. As mentioned earlier, these zeros are redundant as multiplication/addition with zero has a deterministic outcome; hence, such computations are not necessary.

In this work, we propose an LTP-inspired GNN weight pruning method to lower the overall hardware requirements during GNN training. In addition, we design an efficient mechanism to avoid/reduce the storage of zeros for the graph adjacency matrices to further reduce the required number of crossbars. By leveraging crossbar-aware pruning for the GNN model and reducing zero-storage for graph adjacency matrices, DietGNN can achieve high training accuracy with significantly lower storage, energy, and performance overheads compared to state-of-the-art crossbar-aware methods.

III. Crossbar-Aware Pruning of GNNs

In this section, we discuss the computation and communication patterns during GNN training. Next, we explain the DietGNN pruning technique and how it improves GNN training performance on a ReRAM-based manycore architecture.

A. Preliminaries of Graph Neural Network

Graph Data: A graph consists of nodes and edges represented by an adjacency matrix. The graph adjacency matrix is an $N \times N$ sparse binary matrix, where N is the total number of nodes in the graph. An edge between two nodes is represented by "1" in the adjacency matrix. Each kth node of the graph incorporates a D-dimensional feature vector X_D^k (e.g., attributes of products in recommendation system application). The matrix $\mathbf{X}$ ($\mathbf{X} \in \mathbb{R}^{N \times D}$) consists of the features of all the graph nodes.

GNNs: A GNN has multiple neural layers and a final output layer for classification. For example, an L-layer GNN has a weight matrix $W^{(l)}$ associated with each layer l, and an adjacency matrix (A). The computations in the lth GNN layer are depicted in Fig. 1. As shown in Fig. 1, the computation in each neural layer with weights $W^{(l)}$ occurs in two phases.

1) *Vertex Computation:* An MVM operation between the weights and node features ($Y_l = X_{l-1}.W^{(l)}$), where Y_l is the result of the vertex computation of neural layer l. This computation is represented by the green box in Fig. 1.
2) *Edge Computation:* A node-feature propagation and aggregation process that occurs in graph analytics to capture the relational structures ($X_l = A.Y_l$), where X_l is the updated node features and Y_l is the output of the lth neural layer. This feature propagation operation can be denoted as the MVM of the adjacency matrix (A) and the result of the vertex computation phase (Y_l) as depicted in the gray portion of Fig. 1. Note that the same adjacency matrix is used for edge computations associated with all the L GNN layers.

During GNN training, the output of one neural layer acts as the input to the next layer. The amount of communication is proportional to the number of nodes and edges in the graphs. Since real-world graphs can have millions of nodes and edges, this process results in a significant amount of on-chip data exchange in a ReRAM-based manycore architecture [10]. This heavy traffic can overwhelm the on-chip communication infrastructure and needs to be addressed as well [3].

B. Overview of Lottery Ticket Pruning

In this section, we first present some of the key features of LTP and highlight its shortcomings when the pruned model is trained on a ReRAM-based system. We use these insights to develop the DietGNN framework discussed in the next section.

It is well known that neural networks are often over-parameterized. Pruning is an effective way to reduce the number of parameters. The Lottery Ticket hypothesis states that there are many sparse subnetworks which can achieve the same accuracy as a given over-parameterized network. LTP proposes an iterative magnitude-based pruning method, which prunes these over-parameterized networks to find such a sparse subnetwork for the purpose of training. The pruned subnetworks (or pruned models) can then be trained in isolation and can reach an ?accuracy level comparable to its unpruned counterpart in a similar number of training epochs. LTP finds the pruned neural network model using the following steps sequentially.

1) *Step 1:* Randomly initialize the network with weights $W_0^{(l)}$ at $t = 0$.
2) *Step 2:* Train $W_0^{(l)}$ for E epochs to obtain trained weights $W_E^{(l)}$.
3) *Step 3:* Prune $p\%$ of the smallest magnitude weights in $W_E^{(l)}$.
4) *Step 4:* Reset the remaining weights to their original values in $W_0^{(l)}$ and repeat steps 2–4.

The pruning ratio (p) and the number of iterations (n) can be varied by the user to achieve the desired level of sparsity in the winning ticket. Overall, by repeating these steps in an iterative manner, LTP can obtain models (referred as "winning tickets") that are more than 90% sparse. These sparse models [with their remaining weights reset to $W_0^{(l)}$], can be trained from scratch to achieve similar accuracy as the unpruned counterparts.

Recent work has shown that LTP pruned models can generalize across a variety of datasets within an application domain as well as with different optimizers [12], [30]. Prior work has tried to explain these observations intuitively as follows: 1) model behavior is often transferable between datasets. This idea is similar to transfer learning, where a model trained on one dataset can be reused with slight changes for another dataset. The LTP pruned networks also exhibit this transferability property [30]; 2) The transferred tickets act as a regularizer and prevents overfitting while training [30]; and 3) winning tickets learn generic inductive biases which improve training. Hence, DietGNN-based models (which are based on LTP) can also be used with other datasets. However, note that there will be some nonzero accuracy loss when a pruned model is transferred between datasets [30]. Studying the transferability of pruned model between datasets is beyond the scope of our current work. We plan to investigate this in future work.

UGS is a pruning method that generalizes LTP to GNNs [13]. This technique jointly prunes the GNN model weights and graph adjacency matrices over multiple iterations to find the winning ticket. Following this approach, UGS can achieve a reduction in the number of MAC operations, which is the predominant computational step in GNN. However, both LTP and UGS are oblivious to the ReRAM crossbar structure and the mapping of neural layer weights to crossbars. This makes them unsuitable for pruning networks that are to be trained on ReRAM-based architectures.

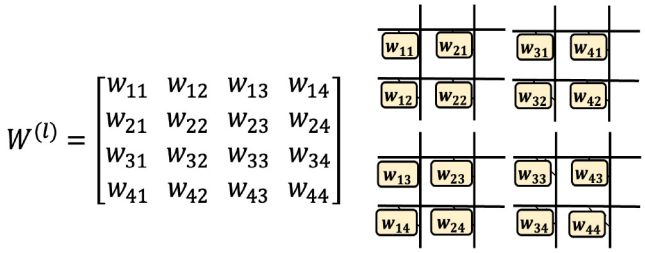

Fig. 2. Mapping the weights of a GNN layer to ReRAM crossbars.

Fig. 2 shows an example of the mapping of 16 weights of a GNN layer ($W^{(l)}$) to four 2×2 crossbars. Both LTP and UGS prune weights based on magnitude after each round of training without the mapping information. Moreover, UGS uses a trainable mask in pruning. In a ReRAM crossbar, each input activates the entire row while each column activates the entire column. Even if only one cell in a row/column is nonzero, that entire row/column will need to be activated. Hence, conventional magnitude pruning methods that are unaware of this fact (such as UGS and LTP) will not result in significant amount of area/power savings. Fig. 3(a) shows the outcome after the mapping of an example pruned GNN model to ReRAM crossbars; here, we prune weights randomly to illustrate the problem associated with crossbar-unaware methods such as LTP and UGS. As seen from Fig. 3(a), even though 50% of the weights are pruned, each crossbar contains at least one nonzero weight in all the rows and columns. As a result, none of the rows/columns can be power-gated to reduce energy consumption. Fig 3(a) shows that due to the crossbar-unaware nature of UGS and LTP, the winning ticket requires four crossbars, just like in the unpruned case in Fig. 2. Hence, there is no hardware or energy savings despite having a 50% pruned network for both UGS and LTP.

Additionally, UGS also prunes the input graph for achieving higher sparsity. However, pruning the graph adjacency matrix can result in loss of information as some critical edges are deleted. This affects the edge computation part of the GNN computation kernel, leading to lower predictive accuracy. Hence, it is important to address this problem to ensure high accuracy and high sparsity of the winning ticket.

C. DietGNN Framework

In this section, we introduce a crossbar-aware pruning technique, referred to as *DietGNN*. Crossbar-aware pruning methods for ReRAM-based architectures have been proposed in prior work [14], [15], [16], [31]. These methods focus on pruning entire rows or columns of ReRAM crossbars. If an entire row/column of a ReRAM crossbar is pruned, we can power-gate the row/column. In Fig. 3(b), we illustrate how a GNN layer's weights ($W^{(l)}$) pruned using existing crossbar-aware methods map to four 2×2 crossbars. In this example, we can see that at least one row or column in each of the four crossbars have been pruned; hence, we can power gate these rows/columns. However, selectively pruning *some* of

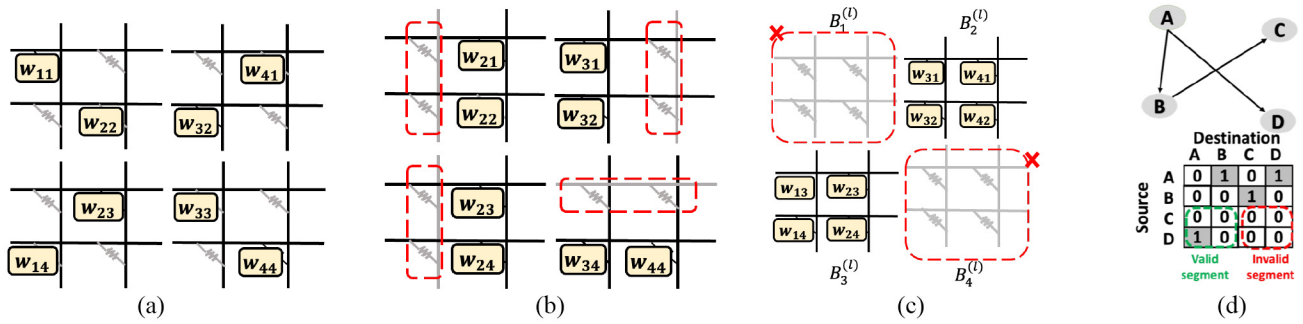

Fig. 3. Mapping weights to ReRAM crossbars after using (a) Crossbar-unaware pruning (LTP and UGS). (b) Traditional crossbar-aware pruning. (c) DietGNN pruning. (d) optimized graph adjacency matrix storage technique.

the rows/columns of a ReRAM crossbar is insufficient as it yields only marginal reduction in energy consumption. This is because peripheral circuits associated with partially pruned or unpruned rows and columns such as DACs (digital-to-analog converters) and S+H (sample-and-hold circuits), ADCs (analog-to-digital converters), in the crossbar must remain "on." For instance, if one row of weights in a crossbar is pruned out, the ADC associated with the crossbar must still be active to process the output of the other columns. Note that ADCs are shared among multiple crossbar columns [6]. From prior work, it is known that the ADCs account for ∼60% of the overall energy consumption of a processing element (PE) [6]. To tackle this, more efficient crossbar-aware techniques need to be explored.

DietGNN solves this problem by proposing a crossbar-aware pruning technique for ReRAM-based architectures. DietGNN has two phases: 1) pruning of the GNN model based on the crossbar knowledge to produce a winning ticket and 2) training of the winning ticket on a ReRAM-based platform. DietGNN incorporates key hardware characteristics such as the crossbar size ($c \times c$) and resolution (bits-per-cell) of the crossbar array to produce a *hardware-friendly winning ticket*.

Unlike existing LTP approaches (such as UGS), which iteratively prune individual weight values based on magnitudes, DietGNN prunes blocks of weights that are mapped to the same crossbar based on their average magnitudes. Typically, weights are mapped in a distributed manner on ReRAM systems [6], [7] . As a result, a $c \times c$ crossbar will have a group of $c \times (c * b/B)$ weights mapped onto it, where b represents the number of bits stored on each cell and B denotes the weight precision. Typical values of B and b are 16 and 2 in a ReRAM-based architecture [6], [32]. Unlike existing crossbar-aware pruning methods that prune individual rows/columns only, DietGNN prunes the entire $c \times (c * b/B)$ block of weights. As a result, the entire crossbar becomes inactive, including its peripheral circuits such as ADC, DAC, and S+H. Hence, we can not only power gate the crossbar, but also the peripherals (or avoid using that crossbar entirely). This would result in significantly higher area/power savings as we show later.

Similar to LTP, DietGNN is an iterative magnitude-based pruning method. To achieve crossbar-awareness, DietGNN divides the GNN model weights in each layer into $c \times (c * b/B)$ sized blocks. At the end of each pruning iteration, DietGNN considers the average magnitude of each block and prunes the lowest $p\%$ of the remaining blocks of weights. The crossbar-awareness of DietGNN directly translates to area/power savings and better performance.

In addition to weight pruning, DietGNN also utilizes an efficient graph adjacency matrix storage technique [10]. Note that adjacency matrices are binary in nature (unlike weight matrices). Hence, DietGNN uses a nonoverlapping *sliding-window*-based method to reduce zero storage. The size of the sliding window is determined by the crossbar size ($c \times c$) to decompose the $N \times N$ graph adjacency matrix into "valid" and "invalid" segments [as shown in Fig. 3(d)] for storing on ReRAM crossbar arrays. Any $c \times c$ segment where all c^2 entries are zero, are referred to as "invalid segment" [Fig. 3(d) in red]. The remaining segments that include at least one edge are defined as "valid segment" [Fig. 3(d) in green] and must be stored on ReRAM crossbars. The invalid segments (and the corresponding crossbars) can be safely discarded, as computations with zeros are redundant. This method helps retain the graph connectivity information while reducing the number of crossbars required to store large graphs.

Algorithm 1 summarizes the overall training process using DietGNN. The inputs to the DietGNN algorithm are an unpruned GNN model, the crossbar structure (size and cell resolution), the target pruning percentage per iteration ($p\%$), and the number of iterations (n). The value of n and the number of epochs (E) are user defined hyper-parameters. The output of the algorithm is a hardware-friendly pruned GNN model, which is referred to as the winning ticket. The DietGNN-enabled winning ticket can then be trained on a ReRAM-based manycore architecture any number of times with other datasets and hyper-parameters, which is typical in graph analytics-based applications. We start by initializing the GNN model weights using common initialization schemes, e.g., Xavier, Kaiming, etc. Next, we partition the weights into blocks based on the crossbar structure (line 2). In each pruning iteration, we execute the following steps in order: 1) train the GNN for E epochs (line 4). Note that, this training for pruning the model is separate from the in-field training of the pruned GNN model; 2) prune $p\%$ of blocks with the lowest average magnitude (line 5); and 3) reinitialize the remaining blocks of weights to their original values (line 6). Finally, the pruned GNN model known as the winning ticket is returned.

The DietGNN method (pruning phase) can be implemented on CPU/GPU-based platforms. This process is not executed on ReRAM-based architectures as the GNN model

Algorithm 1: Pruning With DietGNN

Input: GNN model, crossbar structure, prune percentage p

Output: Pruned GNN model or winning ticket

Algorithm: Algorithm

1: **Initialize**: $W^l \leftarrow W_{initial}$;
2: **Partition** W^l into blocks (B^l) of size $c \times \left(c * \frac{b}{B}\right)$
3: **while** *itr* $<$ *n:*
4: **Train** for E epochs
5: **Prune** $p\%$ of B^l based on average magnitude
6: **Reinitialize** remaining **weights** with $W_{initial}$
7: **Return** *Pruned Model (Hardware-friendly winning ticket)*

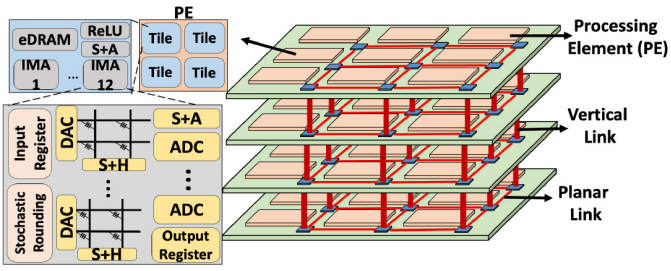

Fig. 4. Illustration of a ReRAM-based manycore architecture.

is dynamically pruned at runtime, which will require remapping of the weights on the fly after every iteration; this is difficult to achieve on ReRAMs. Once the model is pruned, the DietGNN-enabled winning ticket is then deployed for in-field training using ReRAM-based manycore architectures. The pruned model can be trained on the ReRAM architecture multiple times from scratch using different hyper-parameters and graph dataset. Note that, the pruning phase is a one-time preprocessing step. The cost of pruning is amortized over multiple training runs.

IV. Training DietGNN Model on ReRAM Platform

In this section, we present the details of the ReRAM-based manycore architecture (illustrated in Fig. 4) used to train the DietGNN-based GNN model. Next, we discuss how DietGNN improves computation and reduces on-chip communication latency in a ReRAM-based manycore system.

A. Overall ReRAM-Based Architecture

Fig. 4 illustrates the ReRAM-based manycore architecture for accelerating the in-field training of the DietGNN winning ticket. The architecture consists of multiple PEs, where each PE has many ReRAM crossbars of size $c \times c$. The crossbars adopt a b bits/cell resolution; b is typically 2–4 bits [32]. Each tile contains various peripheral circuits, such as ADCs, DACs, Shift, and Add (S+A) circuits, etc. Each tile also has an eDRAM buffer that is used to input activations and outputs of the MVM computation. Fig. 4 shows the top-level hierarchical organization of the manycore architecture.

In order to effectively utilize the high-throughput computation provided by ReRAM-based PEs, the manycore architecture needs to be supported with a high-performance and efficient communication backbone. GNN training generates an enormous volume of traffic, which can bottleneck performance [3]. In this architecture, we utilize a multicast-enabled 3-D mesh network on chip (NoC) as the interconnection backbone for communicating between PEs during GNN training. The 3-D ReRAM-based manycore architecture stacks planar tiers that are connected to each other using through-silicon-via (TSV)-based vertical links. The vertical links act as logical shortcuts and result in more efficient communication, which is crucial for GNN training [3]. In addition, we adopt both Dropout and DropEdge regularization methods to further reduce the amount of communication [9]. These regularization methods randomly drop output activations and graph edges, which results in both improved GNN accuracy and lower communication traffic [9].

Here, we emphasize that both 3-D and manycore architectures have been demonstrated to be commercially viable. For instance, Intel's Xeon is a manycore architecture with 16–56 cores. Micron's high bandwidth memories (HBMs) incorporate 3-D integration [33]. Intel's Lakefield architecture is a 3-D penta-core system. In addition to this, recent prototypes from CEA-Leti have established the feasibility of 3-D ReRAMs [34]. Hence, we expect that ReRAM-based 3-D manycore systems will be adopted for neural network training in the near future.

However, it is well known that 3-D architectures have relatively higher temperature than equivalent 2-D systems. To ensure that the 3-D manycore architecture (shown in Fig. 4) is viable, we have performed thermal simulations using the 3-D-ICE simulator for the training of the pruned GNN model [35]. Our experimental results show that the maximum temperature of the 3-D ReRAM-based manycore architecture is 90°C. For comparison, Nvidia's Tesla GPU can tolerate up to 105°C. Hence, the proposed 3-D manycore architecture is feasible from a thermal perspective.

Finally, it is also important to note that ReRAM crossbar arrays often exhibit nonideal behavior that may negatively impact training of GNNs. Some of these challenges include the use of low-precision weights (16-bit fixed point in our case), thermal noise, low write endurance, etc. [36]. To address the low-precision problem, we use stochastic rounding [24]. Stochastic rounding is an unbiased rounding scheme that achieves $\sim$0% rounding error and introduces negligible area and energy overhead [24], [37]. The problem of thermal noise can be resolved using a reference cell [38]. To solve the low write endurance problem, an LRT method or a threshold-based weight update can be used [23]. LRT can reduce the number of weight updates by $\sim$283$\times$ [23]. Prior work has shown that ReRAM write endurance is typically between 10^6–10^{12} writes [23], [39]. As an example, to train on the Reddit dataset for 200 epochs, we need 30k weight updates. Hence, even if we assume a worst-case scenario of 10^6 writes, the use of the LRT technique will enable us to train the winning ticket for up to 10 000 times. The use of ECC and weight clipping can further reduce the impact of ReRAM nonidealities in training [22]. Hence, even if ReRAMs are nonideal, we can train the DietGNN-pruned GNNs. However, since targeting

ReRAM reliability issues is not the focus of this work, we assume ideal ReRAM behavior.

B. DietGNN on ReRAM-Based Manycore Architecture

As mentioned earlier, the GNN weights and the graph adjacency matrices are mapped to the ReRAM-based PEs (see Fig. 2). The execution of the GNN training process on ReRAM-based manycore systems is divided into two stages: 1) computation on the crossbars and 2) inter-PE communication *via* the NoC. Both these stages need to be accelerated for overall performance gain.

DietGNN-enabled pruning results in multiple blocks of size $c \times (c * b/B)$ being pruned, that is, these blocks of weights are all zeros. Similarly, the sliding window method of storing graph adjacency matrices results in multiple "invalid segments" that include all zero entries [Fig. 3(d)]. As multiplication with zero is redundant, DietGNN-enabled pruning reduces the number of MAC operations significantly. The ReRAM crossbars responsible for storing these weights are unnecessary and can be power gated or reused for other purposes, as shown in Fig. 3(c); this will lead to area and power savings. Alternatively, we can use these unused crossbars to improve execution time. The remaining nonzero GNN weights and adjacency matrices can be duplicated on these crossbars; this will increase the amount of parallelism associated with the GNN computation [7]. For instance, by duplicating a set of weights on two crossbars, we can process inputs twice as fast compared to using only one crossbar, reducing the execution time by approximately half. This additional duplication capability is not available with the unpruned GNNs as duplicating the weights requires double the number of ReRAM crossbars, which may not be available in a resource-constrained edge platform. Overall, DietGNN enables GNN computation in a more area/power efficient manner, while also improving computation time.

Moreover, by reducing the number of crossbars required for training, we automatically reduce the amount of communication. The output of one crossbar is used as input to another crossbar [7]. If all the entries in a crossbar are zero, then the outputs of the MVM operations are also going to be zero. Hence, there is no need to send these data over the NoC (unlike in the unpruned training scenario). This reduces the amount of on-chip. communication associated with GNN training.

To summarize, training the DietGNN-enabled winning ticket on the proposed ReRAM-based manycore architecture (shown in Fig. 4) results in both reduced computation and communication latencies. In summary, the key features of the DietGNN framework are as follows.

1) The DietGNN-enabled ticket helps reduce the number of crossbars required, while also enabling faster computation by parallelizing computation via the duplication of nonzero weights and adjacency matrices on crossbars.
2) The hardware-friendly winning ticket also lowers on-chip communication traffic volume during training.
3) The winning ticket achieves high energy-efficiency and lower execution time when trained on the ReRAM-based manycore architecture, compared to the unpruned version, as demonstrated in our experiments.

TABLE I
ARCHITECTURAL SPECIFICATIONS

4 planar tiers, 9 cores per tier, 4 tiles per core	
ReRAM Tile	96-ADCs (8-bits), 12x128x8 DACs (1-bit), 96 crossbars, 128x128 crossbar size, 10MHz, 2-bit resolution

V. EXPERIMENTAL RESULTS

In this section, we present comprehensive experimental results for DietGNN when it is implemented on the ReRAM-based manycore architecture. First, we describe the experimental setup used to evaluate the performance of DietGNN. Next, we compare DietGNN in terms of model sparsity, accuracy, hardware area, and energy savings with respect to other existing pruning techniques.

A. Experimental Setup

The DietGNN pruning (as shown in Algorithm 1) is executed on an NVIDIA Quadro GPU with 24 GB of memory to generate the winning ticket. The DietGNN-enabled winning ticket (with weights reset to their untrained values) is then mapped to a ReRAM-based 3-D manycore architecture shown in Fig. 4 for in-field training. Recall that this pruned GNN model can be used to train on different graph datasets and/or with different hyper-parameters, a common scenario in the real-world. In this work, the ReRAM-based manycore architecture used for training the DietGNN-enabled winning ticket consists of 36 ReRAM-based PEs (cores) spread across four vertically stacked planar tiers to be commensurate with existing work [3].

We follow the ReRAM configuration presented in [6]. The ReRAM-based PEs consist of multiple morphable subarrays that can be configured for both storage and computation. Each PE includes eDRAM buffers, *in-situ* multiply accumulate (IMA) units, output registers, along with shift-and-add, ReLU, and max-pool units. Each IMA consists of multiple crossbars and associated peripheral circuits such as ADCs, DACs, S+H, and S+A, as well as memory buffers connected with a shared bus [6], [19]. Following prior work, 16-bit fixed-point precision is used for the computations on the ReRAM crossbars [6], [7]. We use a crossbar size of 128×128 with a 2 bit/cell resolution. Hence, 8 consecutive ReRAM cells (that is, 16/2) are required to store each GNN weight for computation on the crossbar array. The choice of the crossbar size has an impact on the overall throughput, power, and area. Overall, each ReRAM tile occupies an area of 0.37 mm^2 and consumes 0.40 W of power [6], [19]. The hardware specifications of the ReRAM tile used in this work is provided in Table I. We evaluate the training performance of the DietGNN-enabled winning ticket on this ReRAM-based architecture using NeuroSim v2.1 [40]. NeuroSim v2.1 incorporates cycle-accurate analytical tools (NVSim & CACTI) for the performance evaluation of Neural Network training on ReRAM-based manycore architectures. In this work, we modify NeuroSim v2.1 to support the on-chip training of GNNs. To evaluate the performance of the NoC, we use the cycle-accurate Garnet simulator [41].

TABLE II
GNN DATASET STATISTICS

Dataset	# of Nodes	# of Edges	# of GNN layers	# of Features
PPI	56,944	818,716	5	50
Reddit	232,965	11,606,919	4	602
Amazon2M	2,449,029	61,859,140	4	100
Flickr	89,250	899,756	3	500
Yelp	716,847	13,945819	3	300

For evaluating DietGNN, we choose five benchmark real-world graph datasets: 1) PPI; 2) Reddit; 3) Amazon2M; 4) Flickr; and 5) Yelp for the performance evaluation. We use the popular cluster-GCN (graph convolutional network) algorithm for training GNNs [2]. Cluster-GCN leverages graph partitioning, which reduces memory overhead and enables the training of GNNs on resource-constrained platforms. We train the GNNs for 200 epochs with a learning rate of 0.01 noting that we observed convergence within the maximum training iterations. At the beginning of pruning, the model weights are initialized with the Xavier initialization scheme. Table II provides details about the graph datasets and the GNNs used for training.

For a thorough evaluation, we compare DietGNN with three baseline pruning techniques. We choose LTP and UGS as the representative crossbar-unaware pruning techniques. As discussed in Section III, LTP is a recently proposed pruning technique that can remove more than 90% weights for the purpose of training [12]. UGS implements the LTP strategy using trainable masks specifically for GNNs [13]. We also employ a recently proposed crossbar-aware pruning (referred as "CAP") technique [16]. CAP utilizes a multi-group LASSO algorithm to prune groups of weights that would otherwise be mapped along a column in a ReRAM crossbar. Similar to DietGNN, we incorporate the optimized zero-storage mechanism [10] in CAP. Here, it should be noted that CAP achieves similar levels of pruning as other methods (such as [14], [15], [31]); hence, we choose CAP as a representative crossbar-aware pruning technique to evaluate the effectiveness of DietGNN. We implement iterative pruning in all the methods to ensure maximum sparsity that can be achieved without significant accuracy loss.

B. Training Performance of DietGNN

In this section, we present the performance of DietGNN compared to the baseline pruning methods.

Crossbar Configuration: First, we must choose the right crossbar size to establish sparsity-area-energy tradeoff. A smaller crossbar size can ensure higher sparsity due to more fine-grain pruning; recall that we prune a block of weights of shape $(c \times (c * b/B))$, where c is the crossbar size). However, smaller crossbars are inefficient in terms of overall area/energy, and *vice versa* [10]. Hence, the choice of the crossbar size is a crucial element in our design. We vary the crossbar size from 8×8 to 256×256. Fig. 5 shows the area (measured as the number of crossbars required), energy consumption, and sparsity of the winning ticket at different crossbar configurations, when trained using the Amazon2M dataset. Here, we show the results for Amazon2M only for brevity, noting that other datasets exhibit similar trends. As shown in Fig. 5, the number of crossbars and energy consumption reduces as the crossbar size increases. This happens as less number of larger crossbars would require fewer peripheral circuits (such as ADCs and DACs) in total. The peripheral area/energy in smaller crossbars dominate the overall area and energy consumption. However, the 256×256 crossbar configuration has higher overall energy consumption than its 128×128 counterpart. This is because 256×256 crossbars require high resolution ADCs (9-bits in our case) which consume significantly more power and area than the ADCs for 128×128 crossbars.

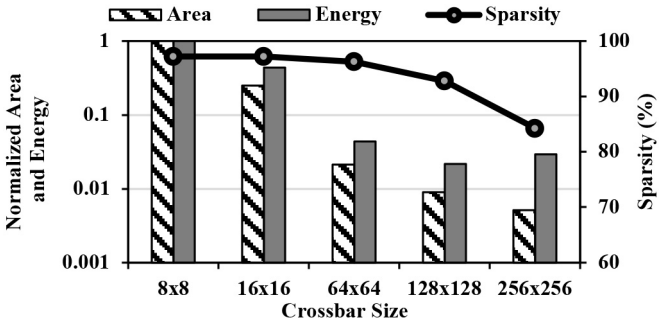

Fig. 5. Sparsity level, area, and energy versus crossbar size for the DietGNN winning ticket trained on the Amazon2M (all values are normalized with respect to 8×8).

In addition, Fig. 5 shows the maximum sparsity that can be achieved with minimum accuracy loss (we set it to an accuracy drop of no more than 1% with respect to the unpruned case). As shown in Fig. 5, the achievable sparsity decreases as the crossbar size increases. This happens as larger blocks of weights must be pruned in case of bigger crossbars. Pruning too many weights at a time leads to greater accuracy loss [16]. From Fig. 5, we can see that the 128×128 crossbar configuration achieves the sweet spot in the sparsity-area-energy tradeoff. The 128×128 configuration achieves an overall sparsity of 90% in the winning ticket and consumes the least amount of energy. Hence, we use 128×128 sized crossbars for all further analysis. As a result, considering 16-bit weight representation and 2 bits/cell resolution, the block size for pruning is computed to be 128×16 (following $(c \times (c * b/B))$, where $c = 128$, $b = 2$, and $B = 16$). Finally, it should be noted that the crossbar size is not part of the pruning process. We do not vary the crossbar size dynamically during pruning. The crossbar size is chosen by the hardware designer and will remain fixed once the device has been manufactured. The results from Fig. 5 are only meant as a guideline for the hardware designer to select the most suitable crossbar size based on their requirements. However, it should be noted that the proposed method is applicable to any crossbar size. Next, we compare DietGNN with the three different baseline methods, namely LTP, UGS, CAP.

DietGNN Pruning Phase: Fig. 6 shows the accuracy achieved by all four pruning techniques on the Amazon2M dataset compared to the unpruned scenario in terms of model prediction accuracy and sparsity. We use a pruning percentage

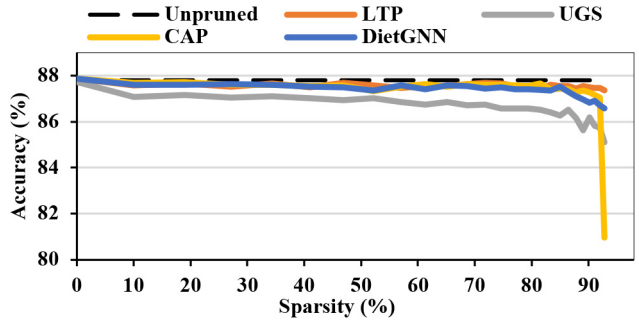

Fig. 6. Accuracy versus Sparsity for different GNN pruning methods trained on the Amazon2M graph dataset.

(p) of 10%, that is, in each pruning iteration, we prune 10% of the remaining blocks of weights. The value of p is a hyper-parameter, which can be chosen based on the desired level of sparsity. In Fig. 6, we can see that as the GNN model becomes sparse, the model prediction accuracy reduces for all pruning methods. However, UGS suffers from significant accuracy loss when compared to the unpruned GNN as shown in Fig. 6. This happens because unlike the other three methods, UGS prunes the input graph as well. This has the following drawbacks: 1) loss of information when the graph is pruned and 2) this method is input-specific (graph-specific in this case). Overall, by pruning graphs in addition to weights, UGS results in worse accuracy than the other three methods. As shown in Fig. 6, when 90% of the GNN weights are pruned, the model accuracy drop is less than 1% for the DietGNN, CAP, and LTP methods. However, at the same sparsity level, the accuracy drop of UGS is more than 5%. For all further analysis, we choose the maximally pruned model that can be trained from scratch to achieve similar accuracy as the unpruned model (that is, less than 1% accuracy loss). Under this constraint, LTP, UGS, CAP, and DietGNN achieve a maximum sparsity level of 97.9%, 61.3%, 92.0%, and 92.8%, respectively, for the Amazon2M dataset as shown in Fig. 7(a) as an example.

Figs. 7(a) and (b) show the accuracy and sparsity of the winning tickets, respectively, for all the datasets considered here. In Fig. 7(a), we can observe that LTP prunes up to 96.30% of GNN weights on average. DietGNN can prune 93.23% of weights on average for all five GNNs and datasets considered here. UGS and CAP prune 64.3% and 91.7% of weights on average, respectively. As mentioned earlier, UGS is unable to prune a lot of weights without experiencing accuracy loss due to additional graph pruning. Clearly, LTP achieves the highest sparsity for all four pruning methods in each dataset. However, we show later that LTP fails to result in area savings commensurate with the levels of pruning.

Once we obtain the winning tickets (pruned models with untrained weights), we load them on the ReRAMs for in-field training from scratch. In Fig. 7(b) we shows the accuracy that the winning tickets achieve when trained on the ReRAM-based manycore architecture in comparison with their unpruned counterpart. As shown in Fig. 7(b), all the pruned models, including the DietGNN-enabled winning ticket, achieve comparable accuracy with the unpruned model in all five datasets. For example, the DietGNN enabled winning ticket trained on the PPI, Reddit, Amazon2M, Flickr, and Yelp graph dataset achieves a model accuracy of 90.244%, 90.346%, 86.903%, 50.80%, and 61.72%, respectively, which is comparable to the accuracy of the unpruned models. Here, we observe that the accuracy of the GNN models trained on the Flickr and Yelp dataset is relatively low. This happens because there is a severe class imbalance in these two datasets. For instance, just one class in Flickr constitutes 43% of the entire dataset, while the remaining six classes make up the rest. As a result, the GNN model encounters difficulties in learning to predict these minority classes. This is a well-known problem in ML theory [42]. However, we emphasize that improving the accuracy of unpruned GNN models in the presence of class imbalance is not the focus of this work. Our aim here is to show the effectiveness of different pruning techniques. Hence, we focus on the relative accuracy trends between unpruned and pruned models only. Moreover, similar observations were made in state-of-the-art GNN implementations as well [43]. For instance, the unpruned GNNs trained using Flickr and Yelp datasets achieve 51.40% and 61.08% accuracy in [43]. Overall, Fig. 7 shows that except for UGS, all the pruned models are extremely sparse, and they can be trained from scratch with very minimal accuracy loss compared to their unpruned counterparts. However, as we show next, only DietGNN enables significant amount of area and energy savings during training.

Hardware Savings From Pruning: As shown in Fig. 3(a)–(c), rows/columns/crossbars that have been pruned can be either power-gated ("turned-off") or reused for other purposes. It is important to note that the number of ReRAM crossbars used can be varied depending on the desired level of computation parallelism adopted for accelerating GNN training. During GNN training, the weights of layers with higher computation latency are replicated using unused ReRAM crossbars to accelerate the overall training process. For example, if the weights in a layer have been duplicated, the computation latency of that layer reduces by half [7]. To fairly compare the different pruning methods in terms of crossbar requirements (hardware savings) and energy consumption, we chose an "iso-performance" setting, that is, equal parallelism (hence, equal training time) for all four pruning techniques.

Figs. 8(a) and (b) shows the area and energy required for training, respectively, using different pruning methods for all five datasets. From Fig. 8(a), we observe that the DietGNN-enabled winning ticket achieves the highest reduction in area. As shown in Figs. 8(a) and (b), respectively, DietGNN requires on average only 5.674% of area and consumes 5.812% of energy compared to its unpruned counterpart for the five datasets. The area savings come from both crossbar-aware pruning of GNN model parameters and reduction in redundant zero storage for the graph adjacency matrices. This is because DietGNN is a crossbar-aware pruning technique that removes all weights that would otherwise be mapped to a specific ReRAM crossbar array. Hence, we can power gate these ReRAMs as shown in Fig. 3(c). The CAP method achieves lower area and energy savings compared to DietGNN as it prunes column-wise, which only allows us to power-gate the crossbar columns. It is well known that the

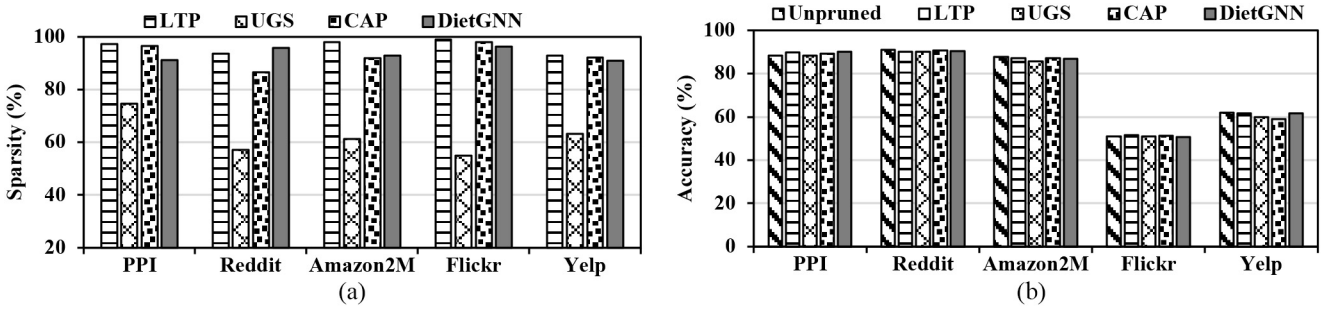

Fig. 7. (a) Sparsity and (b) Accuracy of pruned GNN models (winning tickets) obtained using different methods. All models are trained on ReRAM crossbars.

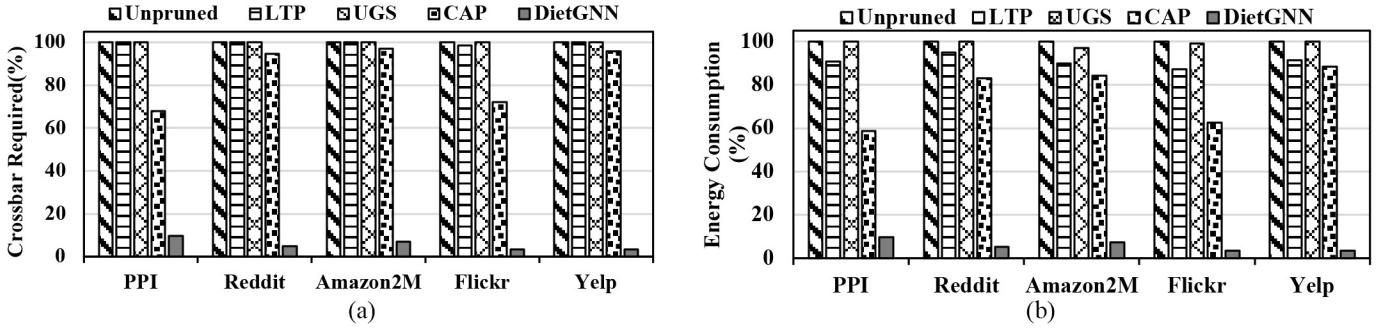

Fig. 8. Crossbar requirements and (b) energy consumption of pruned GNN models (winning tickets) obtained using different methods. All models are trained on ReRAM crossbars.

crossbar area and energy are relatively insignificant compared to the peripheral circuit area and energy. As peripheral circuits (such as ADC, buffers, etc.) are often shared by multiple row/columns of a crossbar [6], we cannot power-gate these circuits unless all these row/columns have been pruned together. The LTP winning ticket does not save ReRAM crossbar arrays as shown in Fig. 8(a) due to its crossbar-unaware nature. The UGS winning ticket has the lowest sparsity compared to other methods as shown in Fig. 7(a). Hence, it does not lead to any significant crossbar and energy savings as seen in Fig. 8(a) and (b), respectively. Overall, the winning ticket obtained via DietGNN pruning requires the least area when compared with the other methods and 92% less area than the unpruned case. Hence, DietGNN offers the best solution among these methods for enabling GNN training on resource-constrained hardware platforms. As shown in Figs. 8(a) and (b), neither LTP nor UGS results in significant area and energy savings. As a result, we exclude these two pruning methods from all further analyses.

C. Computation and Communication Analysis

Each GNN layer involves both computation (MVM operations) and inter-PE communication. The overall execution time for GNN training is determined by both the computation and communication stage delay. In this section, we evaluate the impact of DietGNN on the computation and communication latencies of GNN training when it is implemented on the ReRAM-based manycore architecture. We assume an iso-area setting for this analysis that is, the number of ReRAM crossbars available is the same for training all the pruned and unpruned cases. Fig. 9 shows the worst-case computation

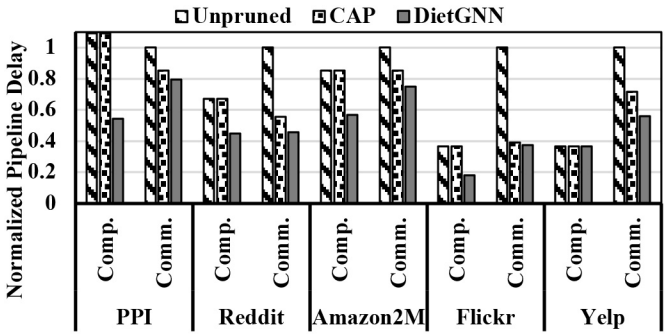

Fig. 9. Computation and communication delay for unpruned model, and pruned models obtained using CAP and DietGNN; all delays are normalized with respect to the communication delay of the unpruned model.

and communication latencies when we train the unpruned, CAP-, and DietGNN-pruned models on ReRAM-based architectures. Note that all three methods incorporate Dropout and DropEdge-based regularization techniques.

Computation Delay: Pruning leads to smaller models, which can be implemented on fewer crossbars (compared to the unpruned model). Hence, in an iso-area setting, it is possible to accelerate GNN computation further by utilizing the remaining crossbars. We can replicate the weights of the neural layers on these additional crossbars and parallelize the computation further [7]. This is not possible with the unpruned model as it has more weights that need to be mapped first. Hence, when the unpruned model is mapped on the crossbars, there is no crossbar left to replicate the weights and accelerate computation. Pruning enables further replication of weights as these models require fewer number of crossbars. However, as shown in Fig. 8(a), pruning using the CAP method does not result in

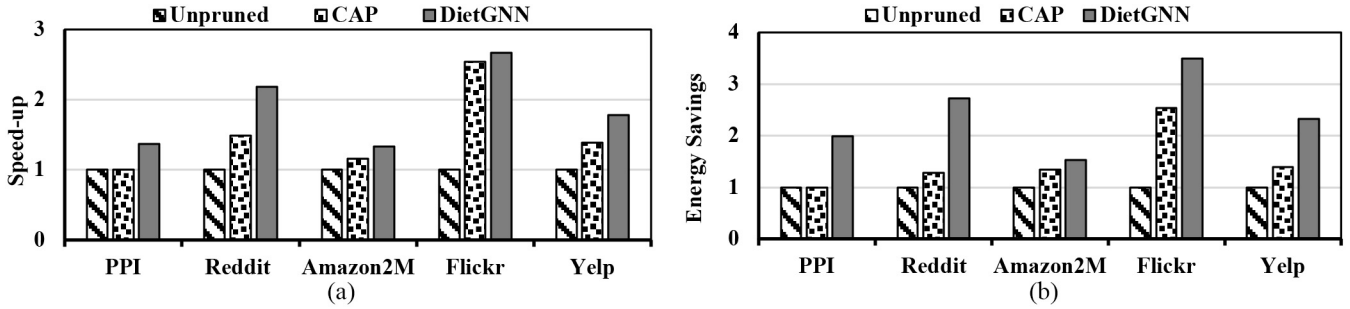

Fig. 10. (a) Execution time speed-up, and (b) energy savings of DietGNN-enabled pruned GNN model compared to Unpruned, and model pruned using CAP (normalized with respect to the execution of the unpruned model on the ReRAM-based manycore architecture).

high crossbar savings (only 14% fewer crossbars on average compared to the unpruned case). As a result, we cannot speed-up all the layers (as in the case of DietGNN) due to the lack of a sufficient number of ReRAM crossbars; recall that we are comparing in an iso-area setting. Thus, the overall computation latency improvement for the CAP method compared to the unpruned case is negligible. DietGNN considers the overall crossbar structure when pruning the GNN model. As a result, the pruned model can be implemented using fewer crossbars than other mechanisms, as shown in Fig. 8(a). The additional free crossbars can now be used to replicate the pruned weights in order to reduce the computation delay. Hence, DietGNN results in an average improvement of 58% in the computation latency as shown in Fig. 9. This results in faster GNN training on ReRAM-based architectures.

Communication Delay: Communication has a significant influence on the execution time of GNN training [3]. The data traffic in GNN training is generated from the MVM operations during the vertex and edge computation phases as discussed earlier in Section III-A. Pruning reduces the amount of data exchanges, thus alleviating the communication bottleneck.

If one full row of the weight matrix is pruned, the result of the MVM operation involving that row results in a zero. We can avoid sending "zero" data over the NoC. Hence, if a large number of row/columns are pruned, we can reduce the amount of communication significantly. However, partially pruned rows/column may result in nonzero outputs that must be communicated. Both CAP and DietGNN pruning results in multiple rows of zeros, which reduces the overall number of messages generated from the crossbars. Overall, CAP results in a communication delay improvement of 15%, 45%, 14%, 61%, and 28% for PPI, Reddit, Amazon2M, Flickr, and Yelp, respectively, when compared to the unpruned model. DietGNN results in a reduction of the communication delay by 21%, 55%, 25%, 62.5%, and 44% for PPI, Reddit, Amazon2M, Flickr, and Yelp, respectively. The improvement is higher for PPI, Reddit, and Flickr as they have relatively larger number of features amongst all the datasets under consideration here. A larger number of features results in more information being exchanged in each layer that is, more communication. Many of these features are redundant. As a result, we can prune more rows of weights, which results in larger improvements.

Overall, the end-to-end execution time is bottlenecked by the slower among the computation and the inter-PE communication stages. From Fig. 9, we can see that Reddit and Amazon2M are bottlenecked by communication whereas computation is the bottleneck in PPI. This happens as we do not have sufficient number of crossbars to accelerate computation in the case of PPI. However, DietGNN accelerates both computation and inter-PE communication significantly when compared to the unpruned case.

D. Full System Evaluation

In this section, we compare the end-to-end speed-up enabled by the different pruning techniques compared to their unpruned counterpart on the same ReRAM-based architecture. Here, we do not show any performance comparison with respect to GPUs because prior work has already demonstrated that ReRAMs significantly outperform GPUs in terms of execution time and energy efficiency for GNN training [3], [25]. Fig. 10(a) and (b) show the execution time speed-up and energy savings (normalized with respect to unpruned) for GNN training, respectively, for Unpruned, CAP, and DietGNN. DietGNN achieves 87% and 52% speed-up in overall execution time on average compared to Unpruned and CAP, respectively. The improvement in speed-up is enabled by the reduction in both computation and communication delay as shown in Fig. 9. Note that CAP performs significantly worse compared to DietGNN because the overall execution time is always bottlenecked by the computation delay. Fig. 10(b) shows that the DietGNN-enabled model reduces energy consumption by 2.4× and 1.6× compared to the Unpruned and CAP models running on the same ReRAM-based architecture, respectively. Overall, DietGNN can accelerate training by up to 2.7× while consuming up to 3.5× less energy than the unpruned implementation (Unpruned) on an iso-area ReRAM-based manycore architecture.

VI. Conclusion

GNN training on ReRAM-based manycore accelerators is both compute- and data-intensive. Moreover, the need to store DNN model weights and graph adjacency matrices give rise to very high storage requirements. Training a pruned GNN model from scratch can mitigate these challenges. However, existing pruned models that use crossbar-aware techniques cannot be trained from scratch as they are targeted only for inferencing. We have presented a crossbar-aware pruning technique called DietGNN, which can be trained from

scratch, achieves high sparsity, and enables significant reduction in energy consumption and area overhead. In addition to reducing GNN model parameters, DietGNN enables efficient storage of graph adjacency matrices by removing redundant zeros. DietGNN achieves $\sim 2.7\times$ speedup while being $3.5\times$ more energy efficient when compared to its unpruned version on a ReRAM-based manycore platform. DietGNN also significantly outperforms other state-of-the-art crossbar-aware pruning methods in terms of achievable sparsity, execution time, and hardware savings.

REFERENCES

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

[2] 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. ACM SIGKDD Int. Conf. Knowl. Discov. Data Min.*, 2019, pp. 257–266.

[3] A. I. Arka, B. K. Joardar, J. R. Doppa, P. P. Pande, and K. Chakrabarty, "Performance and accuracy tradeoffs for training graph neural networks on ReRAM-based architectures," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 29, no. 10, pp. 1743–1756, Oct. 2021.

[4] D. Xu *et al.*, "Edge intelligence: Architectures, challenges, and applications," 2020, *arXiv:2003.12172*.

[5] E. Strubell, A. Ganesh, and A. McCallum, "Energy and policy considerations for modern deep learning research," in *Proc. 34th AAAI Conf. Artif. Intell.*, 2020, pp. 13693–13696.

[6] A. Shafiee *et al.*, "ISAAC: A Convolutional neural network accelerator with in-situ analog arithmetic in crossbars," in *Proc. Int. Symp. Comput. Archit. (ISCA)*, 2016, pp. 14–26.

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

[8] K. Roy, I. Chakraborty, M. Ali, A. Ankit, and A. Agrawal, "In-memory computing in emerging memory technologies for machine learning: An overview," in *Proc. IEEE Des. Autom. Conf. (DAC)*, 2020, pp. 1–6.

[9] A. I. Arka, B. K. Joardar, J. R. Doppa, P. P. Pande, and K. Chakrabarty, "DARe: DropLayer-aware manycore ReRAM architecture for training graph neural networks," in *Proc. Int. Conf. Comput.-Aided Des. (ICCAD)*, 2021, pp. 1–9.

[10] 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)*, 2018, pp. 531–543.

[11] 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. Asia South Pacific Des. Autom. Conf. (ASPDAC)*, 2019, pp. 120–126.

[12] J. Frankle and M. Carbin, "The lottery ticket hypothesis: Finding sparse, trainable neural networks," in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2019, pp. 1–42.

[13] T. Chen, Y. Sui, X. Chen, A. Zhang, and Z. Wang, "A unified lottery ticket hypothesis for graph neural networks tianlong," in *Proc. Int. Conf. Mach. Learn. (ICML)*, 2021, pp. 1695–1706.

[14] L. Liang *et al.*, "Crossbar-aware neural network pruning," *IEEE Access*, vol. 6, pp. 58324–58337, 2018.

[15] J. Lin, Z. Zhu, Y. Wang, and Y. Xie, "Learning the sparsity for ReRAM: Mapping and pruning sparse neural network for ReRAM based accelerator," in *Proc. Asia South Pacific Des. Autom. Conf. (ASPDAC)*, 2019, pp. 639–644.

[16] J. Meng, L. Yang, X. Peng, S. Yu, D. Fan, and J.-S. Seo, "Structured pruning of RRAM crossbars for efficient in-memory computing acceleration of deep neural networks," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 68, no. 5, pp. 1576–1580, May 2021.

[17] S. Yang, W. Chen, X. Zhang, S. He, Y. Yin, and X.-H. Sun, "Auto-prune: Automated DNN pruning and mapping for ReRAM-based accelerator," in *Proc. Int. Conf. Supercomput. (ICS)*, New York, NY, USA, 2021, pp. 304–315.

[18] C. Chu *et al.*, "PIM-prune: Fine-grain DCNN pruning for crossbar-based process-in-memory architecture," in *Proc. IEEE Des. Autom. Conf.*, 2020, pp. 1–6.

[19] G. Yuan *et al.*, "FORMS: Fine-grained polarized ReRAM-based in-situ computation for mixed-signal DNN accelerator," in *Proc. Int. Symp. Comput. Archit. (ISCA)*, 2021, pp. 265–278.

[20] P. Chi *et al.*, "PRIME: A novel processing-in-memory architecture for neural network computation in ReRAM-based main memory," in *Proc. Int. Symp. Comput. Archit. (ISCA)*, 2016, pp. 27–39.

[21] A. Chaudhuri and K. Chakrabarty, "Analysis of process variations, defects, and design-induced coupling in memristors," in *Proc. IEEE Int. Test Conf. (ITC)*, 2019, pp. 1–10.

[22] B. Feinberg, S. Wang, and E. Ipek, "Making memristive neural network accelerators reliable," in *Proc. IEEE Int. Symp. High-Perform. Comput. Archit. (HPCA)*, 2018, pp. 52–65.

[23] K. Prabhu *et al.*, "CHIMERA: A 0.92-TOPS, 2.2-TOPS/W edge AI accelerator with 2-MByte on-chip foundry resistive RAM for efficient training and inference," *IEEE J. Solid-State Circuits*, vol. 57, no. 4, pp. 1013–1026, Apr. 2022.

[24] T. Na, J. H. Ko, J. Kung, and S. Mukhopadhyay, "On-chip training of recurrent neural networks with limited numerical precision," in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, 2017, pp. 3716–3723.

[25] Z. Wang *et al.*, "GNN-PIM: A processing-in-memory architecture for graph neural networks," in *Proc. 13th Conf. Adv. Comput. Archit.*, Kunming, China, 2020, pp. 73–86.

[26] Y. He, Y. Wang, C. Liu, H. Li, and X. Li, "TARe: Task-adaptive in-situ ReRAM computing for graph learning," in *Proc. IEEE Des. Autom. Conf. (DAC)*, 2021, pp. 577–582.

[27] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural networks," in *Advances in Neural Information Processing Systems (NeurIPS)*. Red Hook, NY, USA: Curran, 2015, pp. 1135–1143.

[28] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient ConvNets," in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2017, pp. 1–13.

[29] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding," in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2016, pp. 1–14.

[30] A. S. Morcos, Y. Haonan, M. Paganini, and Y. Tian, "One ticket to win them all: Generalizing lottery ticket initializations across datasets and optimizers," in *Advances in Neural Information Processing Systems*. Red Hook, NY, USA: Curran, 2019.

[31] X. Ma *et al.*, "Tiny but accurate: A pruned, quantized and optimized memristor crossbar framework for ultra efficient DNN implementation," in *Proc. Asia South Pacific Des. Autom. Conf. (ASPDAC)*, 2020, pp. 301–306.

[32] B. Q. Le *et al.*, "Resistive RAM with multiple bits per cell: Array-level demonstration of 3 bits per cell," *IEEE Trans. Electron Devices*, vol. 66, no. 1, pp. 641–646, Jan. 2019.

[33] D. U. Lee *et al.*, "25.2 A 1.2V 8Gb 8-channel 128GB/s high-bandwidth memory (HBM) stacked DRAM with effective microbump I/O test methods using 29nm process and TSV," in *Proc. IEEE Solid-State Circuits Conf.*, 2014, pp. 423–433.

[34] "CEA-Leti Research Team Proposes New Approach for Next-Generation Memories With RRAM Energy-Storage Breakthrough." CEA-Leti. 2021. [Online]. Available: https://www.leti-cea.com/cea-tech/leti

[35] A. Sridhar, A. Vincenzi, M. Ruggiero, T. Brunschwiler, and D. Atienza, "3D-ICE: Fast compact transient thermal modeling for 3D ICs with inter-tier liquid cooling," in *Proc. ICCAD*, 2010, pp. 463–470.

[36] S. Narang *et al.*, "Mixed precision training," in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2018, pp. 1–12.

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

[38] Y. H. Lin *et al.*, "Device instability of ReRAM and a novel reference cell design for wide temperature range operation," *IEEE Electron Device Lett.*, vol. 38, no. 9, pp. 1224–1227, Sep. 2017.

[39] W. Wen, Y. Zhang, and J. Yang, "ReNEW: Enhancing lifetime for ReRAM crossbar based neural network accelerators," in *Proc. IEEE Int. Conf. Comput. Des. (ICCD)*, 2019, pp. 487–496.

[40] X. Peng, S. Huang, H. Jiang, A. Lu, and S. Yu, "DNN+NeuroSim V2.0: An end-to-end benchmarking framework for compute-in-memory accelerators for on-chip training," 2020, *arXiv:2003.06471*.

[41] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha, "GARNET: A detailed on-chip network model inside a full-system simulator," in *Proc. Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, 2009, pp. 33–42.

[42] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "SMOTE: Synthetic minority over-sampling technique," *J. Artif. Intell. Res.*, vol. 16, no. 1, pp. 321–357, 2002.

[43] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, and V. Prasanna, "GraphSAINT: Graph sampling based inductive learning method," in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2020, pp. 1–19.

[44] A. Auten, M. Tomei, and R. Kumar, "Hardware acceleration of graph neural networks," in *Proc. IEEE Des. Autom. Conf. (DAC)*, 2020, pp. 1–6.

[45] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning structured sparsity in deep neural networks," in *Advances in Neural Information Processing Systems (NeurIPS)*. Red Hook, NY, USA: Curran, 2016, pp. 2082–2090.