Accelerating Graph Neural Network Training on ReRAM-based PIM Architectures via Graph and Model Pruning

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

***Abstract*— Graph Neural Networks (GNNs) are used for predictive analytics on graph-structured data, and they have become very popular in diverse real-world applications. Resistive random-access memory (ReRAM)-based PIM architectures can accelerate GNN training. However, GNN training on ReRAM-based architectures is both compute- and data-intensive in nature. In this work, we propose a framework called *SlimGNN* that synergistically combines both graph and model pruning to accelerate GNN training on ReRAM-based architectures. The proposed framework reduces the amount of redundant information in both the GNN model and input graph(s) to streamline the overall training process. This enables fast and energy-efficient GNN training on ReRAM-based architectures. Experimental results demonstrate that using this framework, we can accelerate GNN training by up to $4.5\times$ while using $6.6\times$ less energy compared to the unpruned counterparts.**

***Index Terms*—Graph Neural Network, ReRAM, PIM, Data Compression, Pruning**

I. INTRODUCTION

GRAPH neural networks (GNNs) have become increasingly popular in diverse real-world industrial and scientific applications [1] [2]. GNNs enable predictive analytics over graph data through iterative feature aggregation over neighborhood vertices. GNN computation occurs in two stages: 1) The vertex feature update stage involves matrix multiplication of trainable weights and vertex-level feature vectors similar to conventional deep neural networks (DNNs), and 2) The feature aggregation stage involves accumulating feature information from neighborhood vertices [3]. Hence, GNN training exhibits characteristics of both DNN training and graph computation simultaneously. Hence, it is both compute- and data-intensive in nature. The high amount of data movement required by GNN computation poses a challenge to conventional von-Neuman architectures (such as CPUs and GPUs) as they have limited memory bandwidth [4]. Hence, there is a need for new hardware architectures that are suitable for GNN training/inference using massive real-world graphs.

Training machine learning (ML) models on the edge (on-chip or on embedded systems) has become necessary due to data privacy concerns. Moreover, several applications such as AR/VR require GNN training on the edge [5]. Emerging Resistive random-access memory (ReRAM)-based architectures have been proposed as promising candidates for accelerating GNN training and inferencing in an on-chip environment [6]. The crossbar structure of ReRAMs makes them well-suited for performing matrix-vector multiplication (MVM) operations, which is the predominant computation kernel in both GNN training and inferencing. Hence, a PIM architecture consisting of multiple ReRAM-based processing elements (PEs) is suitable for large-scale GNN training.

It is well known that ReRAM writes are slow. Hence, to reduce the number of writes, a pipelined training strategy is employed [7] [8]. Figure 1 shows an illustration of the pipelined GNN training. The training occurs in a pipelined manner where all layers of the GNN are computed in parallel to improve the overall system throughput. Note that, to enable pipelined training, a large monolithic graph is first divided into multiple smaller subgraphs using graph partitioning [9]. Each subgraph is sent sequentially as input to the pipeline [6]. Figure 1(a)

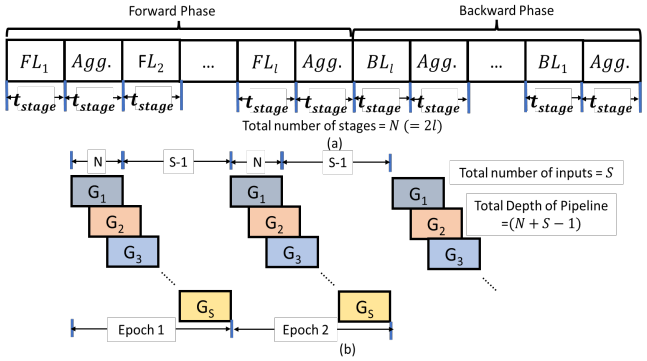

Fig. 1: (a) Pipeline stages during GNN training; each stage is a GNN layer. (b) Pipelined execution of a GNN with S input subgraphs.

This work was supported, in part by the US National Science Foundation (NSF) under grants CNS-1955353, and CNS-1955196. Biresh Kumar Joardar was also supported in part by NSF Grant # 2030859 to the Computing Research Association for the CIFellows Project.

Chukwufumnanya Ogbogu, Aqeeb Iqbal Arka, Janardhan Rao Doppa, and Partha Pratim Pande are with Washington State University, Pullman, WA, 99164. Email: {c.ogbogu, aqeebiqbal.arka, jana.doppa, pande}@wsu.edu .

Lukas Pfromm is with Oregon State University, Corvallis, OR, 97331. Email: lukaspfromm@gmail.com.

Biresh Kumar Joardar is with University of Houston, Houston TX, 77004. Email: bjoardar@Central.UH.EDU.

Krishnendu Chakrabarty is with the Department of Electrical and Computer Engineering, Duke University, Durham, NC, 27708. Email: krish@duke.edu

shows the end-to-end pipeline for training a l-layer deep GNN on one input subgraph. Here, FL_i and BL_i denote the forward and backward phase computation of the i^{th} layer of GNN training, respectively. The worst-case delay associated with a pipeline stage is denoted as t_{stage}. As shown in Figure 1(b), the end-to-end pipeline depth is $(N + S - 1)$, where $N\ (= 2l)$ is the total number of pipeline stages, S is the total number of input subgraphs, and G_n is the n^{th} input subgraph.

During GNN training, the overall execution consists of computation (i.e., MVM operations) on PEs and inter-PE communication (the data of one layer must be sent to the next layer). Each computation phase also has one stage for feature aggregation over graph edges ($Agg.$) associated with it. The overall execution time is determined by the *end-to-end pipeline depth* $(N + S - 1)$ and the *delay of each pipeline stage* (t_{stage}). The overall execution time is given by: $(N + S - 1) \times t_{stage}$. To improve the execution time, we must improve the pipeline depth and/or the pipeline stage delay. The depth of the pipeline is governed by the input dataset, i.e., number of subgraphs (S) used for training, while the pipeline stage latency is governed by the amount of computation and communication in a GNN layer. In this work, we improve the performance of ReRAM-based systems for GNN training by reducing both the pipeline depth and the stage delay.

The end-to-end pipeline depth can be shortened by reducing the number of input subgraphs needed for GNN training. Recent work has shown that, it is possible to prune the input data for training convolutional neural networks without sacrificing accuracy [10]. We generalize this approach to prune subgraphs (thereby reducing pipeline depth) to improve the performance of GNN training on ReRAM-based PIM architectures. We identify the necessary subgraphs for GNN training very early in the training process similar to [10]. The key insight behind pruning the sub-graphs is to exploit the redundancies in terms of nodes with similar features, labels, and neighborhood; and the fact that some node classification examples are easy to learn. These input subgraphs are identified very early in training and discarded without sacrificing accuracy. This reduces the number of input subgraphs S, resulting in reduced pipeline depth, which leads to lower execution time. We refer to this subgraph pruning methodology as GraphDiet, which enables the GNN to train using a small fraction of the input graph.

The performance of the ReRAM-based PIM architecture can be further enhanced by reducing the pipeline stage delay. Model weight pruning is a popular technique for reducing the number of parameters and redundant computations in a GNN by making some of the weights zero. These pruned weights need not be stored on chip as an MVM operation involving a zero will always result in a zero; hence, such computation is unnecessary [11]. Thus, model weight pruning helps to reduce redundant MVM operations. Also, weight pruning results in the reduction of on-chip traffic, which in turn reduces the inter-PE communication delay as well. Thus, weight pruning can help in reducing the pipeline stage delay (the parameter t_{stage}) mentioned above (Figure 1), which ultimately results in lower execution time.

In this work, we propose the incorporation of GraphDiet and model pruning methods to reduce both the overall pipeline depth and pipeline stage delay for executing GNN training on ReRAM-based PIM architectures. We refer to the synergistic combination of these two techniques as the SlimGNN framework. This framework produces a lightweight GNN model, which needs fewer input subgraphs compared to the unpruned graph to achieve high accuracy. We train the SlimGNN-enabled model on an ReRAM-based PIM architecture. SlimGNN achieves high performance while being energy- and storage efficient. The key contributions of this work are summarized as follows:

- We leverage GraphDiet and GNN model pruning as two synergistic methods to achieve high performance and energy efficient training of GNNs on an ReRAM-based PIM architecture without sacrificing accuracy.
- Using GraphDiet, we reduce the number of input subgraphs required for GNN training very early. Hence, it significantly reduces the end-to-end pipeline depth during GNN training on ReRAM-based PIM architectures.
- Complimentary to GraphDiet, by pruning the GNN model weights, we achieve significant hardware area and energy savings while improving the pipeline stage delay associated with GNN training on ReRAM architectures.
- Experimental results show that the SlimGNN-enabled model trained on the ReRAM-based PIM architecture outperforms state-of-the-art ReRAM-based counterparts by up to 4.5 × in terms of the execution time for training GNNs.

To the best of our knowledge, this is the first work that incorporates GraphDiet to *design a high-performance and energy-efficient ReRAM-based PIM system for scalable training of GNNs on massive graphs.* The rest of the paper is organized as follows. Section 2 describes relevant prior work. In Section 3, we discuss principles of GNN computations on ReRAM-based architectures. In Section 4, we present the SlimGNN framework, the effect of pruning on the pipeline stage delay and the impact of GraphDiet on pipeline depth and GNN training on the proposed architecture. Section 5 presents experimental results. Finally, Section 6 concludes the paper by summarizing our key findings.

II. Related Prior Work

In this section, we discuss relevant prior work, focusing on ReRAM-based architectures, as well as data pruning and model pruning for neural networks, especially GNNs.

ReRAM-based architectures: ReRAMs enable processing-in-memory, which allows for fast and efficient in-situ MVM [11]. Both DNN and graph computation rely heavily on such MVM operations. This makes ReRAM-based architectures excellent candidates for DNN training/inferencing and graph analytics [7] [8] [12] [13] [14] [15]. GNNs exhibit characteristics of both DNNs and graph computation. Hence,

they also benefit from ReRAM-based systems [16]. However, on-chip communication is a major performance bottleneck for GNN training on ReRAM-based PIM architectures. The communication bottleneck can be addressed by using a 3D network-on-chip (NoC) [6]. However, all these accelerators use an unpruned neural network model, which leads to unnecessary high area and power overheads.

Model and Data Pruning in Neural Networks: Model pruning for neural networks helps reduce redundant computations during DNN computation [17] [18] [19]. Several crossbar-aware model pruning (CAP) techniques have also been proposed to exploit the ReRAM crossbar structure while pruning to reduce area and improve energy efficiency without compromising accuracy [20] [21] [22]. However, all these methods prune pre-trained DNN models for inferencing purposes, hence they are not suited for training, which is the focus of this work. Moreover, these CAP methods prune weights row/column-wise only, which leads to only marginal energy and area savings as we elaborate later. Recently, an iterative model pruning method called Lottery Ticket Pruning (LTP) has been proposed to obtain highly sparse DNN models for training [23]. An LTP-inspired pruning method has also been used to prune GNN models and graph adjacency matrices [24]. However, LTP-based methods don't take the ReRAM crossbar structure into consideration. As a result, there is no significant area or power savings. In this work, we propose an LTP-inspired CAP method for GNN training on ReRAM-based architectures. Our method enhances existing CAP methods by taking the overall crossbar structure into consideration and implements the pruning in an iterative manner to ensure that the pruned GNN models can be trained from scratch. By pruning in a crossbar-aware fashion, we can also reduce the pipeline stage delay (the variable t_{stage} in Figure 1), which leads to better performance.

On top of model pruning, we can further prune the input data to reduce the end-to-end pipeline depth (the variable S in Figure 1) for GNN training on ReRAM-based architectures. Large training datasets pose a challenge for resource-constrained devices (such as edge devices) as they require high memory and processing power [25]. Recent work has proposed methods to reduce the amount of data required for training, generally referred to as data pruning [26]. An importance score-based data pruning methodology for CNNs was proposed to greatly reduce the amount of input data [10]. Graph Early Bird (GEB) prunes the edges of the input graphs to reduce MVM operations for GNN training very early during the training process [27]. Thus, the size of the input feature vector remains unchanged as only graph edges are pruned. In contrast to existing data pruning techniques for GNNs, our method intelligently prunes large portions of an input graph (both nodes and edges) to reduce the end-to-end pipeline depth of GNN training on ReRAM-based architectures.

III. GNN Computation on ReRAM Platforms

In this section, we discuss the preliminaries of GNN computations on ReRAM-based architectures. First, we introduce the GNN computation kernel. Next, we present the ReRAM-based PIM architecture under consideration.

A. Preliminaries of GNN Computation

The GNN computation kernel primarily consists of MVM operations involving model weights and graph adjacency matrices. To perform GNN training on memory constrained environments, the large monolithic graph is broken into S subgraphs $(G_1, G_2, \dots, G_s)$. Each subgraph G_i includes: (a) a total of V_i vertices, where each vertex v has a m-dimensional feature vector $f_{v,i}$ associated with it. (b) The graph adjacency matrix Adj_i, which is a sparse matrix of size $V_i \times V_i$. The GNN model consists of multiple back-to-back neural layers. Each layer (l) has a weight matrix W_l associated with it. The forward phase computation involved in the GNN layer l corresponding to the input subgraph G_i is shown below:

$$\text{Forward Phase: } f_{v,i}^{l} = \sigma(MVM(Adj_i, MVM(f_{v,i}^{l-1}, W^{l}))) \quad (1)$$

First, we implement an MVM operation involving feature vector $(f_{v,i}^{l-1})$ and the dense weight matrix (W^l). Next, we accomplish the feature aggregation over edges, which involves MVM with adjacency matrix Adj_i as shown in Equation 1. The input to each neural layer l is the output of the previous neural layer $l-1$. Here, we set the activation function $\sigma(.)$ to be ReLU [25]. Similarly, the error and gradient calculations are done during the backward phase computation. Each of them consists of two separate MVM operations. The backward phase computation is shown in Equations 2, 3, and 4.

$$\text{Error: } \begin{cases} err_{v,i}^{l} = MVM((W^{l+1})^{T}, (\delta_{v,i}^{l+1} \cdot f_{v,i}^{l+1})) & (2) \\ \delta_{v,i}^{l} = MVM(Adj_i, err_{v,i}^{l}) & (3) \end{cases}$$

$$\text{Gradient: } z_{v,i}^{l} = MVM\left((f_{v,i}^{l-1})^{T}, MVM(Adj_s, \delta_{v,i}^{l})\right) \quad (4)$$

Equations (2)-(3) show the error calculation ("Error") during the backward phase of GNN computation for neural layer l. Equation (2) shows the multiplication of features $(f_{v,i}^{l+1})$ and error $(\delta_{v,i}^{l+1})$ with the transpose of the weight matrix (W^{l+1}) to calculate the intermediate error, $err_{v,i}^{l}$. Equation (3) involves the MVM of the adjacency matrix Adj_i and $err_{v,i}^{l}$ for the computation of the final error $err_{v,i}^{l}$ for one vertex (v) of an input subgraph G_i. Similarly, Equation (4) is used to compute the gradient, $z_{v,s}^{l}$ where the differentiation operation can be decomposed into two separate MVM operations [28]. All these operations are executed for S subgraphs that make up the full input graph dataset. Moreover, the result of all MVM computations of layer l, i.e., $f_{v,i}^{l}$, $z_{v,i}^{l}$, and $\delta_{v,i}^{l}$, are used as inputs to the next GNN layer $l+1$. During the backward phase, the data flows from layer $l+1$ to layer l. The amount of data generated in every layer is also dependent on the number of weights in each layer. In this work, we leverage model/weight pruning to decrease the number of weights, which helps in reducing the number of MVMs and the associated data movement. As the GNN kernel computations must be repeated for every subgraph G_i, decreasing the number of subgraphs (S) needed for training will help reduce the end-to-end pipeline depth.

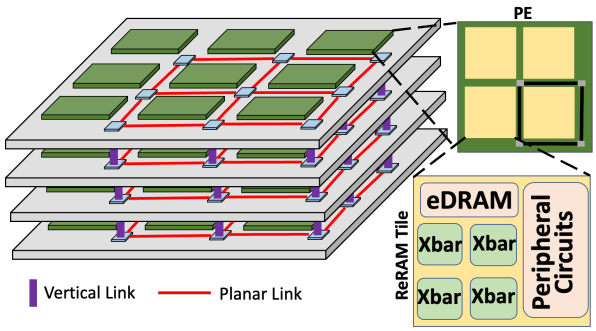

Fig. 2: Illustration of ReRAM-based 3D PIM Architecture.

B. ReRAM-based 3D PIM Architecture

In this work, we consider a 3D PIM architecture consisting of multiple ReRAM-based processing elements (PEs) for accelerating GNN training. We adopt a 3D architecture to enable high degree of integration. This architecture consists of multiple PEs stacked vertically across four layers as shown in Figure 2. To effectively utilize the high-throughput computation provided by ReRAM-based PEs, the overall architecture needs to be supported with a high-performance and efficient communication backbone. In this architecture, we utilize a 3D mesh network-on-chip (NoC) as the interconnection backbone for communication between PEs during GNN training. Each PE contains multiple ReRAM crossbar arrays (xbar) for executing MVM operations as shown in Figure 2. The exact PE configuration used in this work is described in the Section V. We use a crossbar size of 128×128, with 2 bits/cell configuration, as it provides the best performance-area-energy trade-off [6]. Each PE contains various peripheral circuits including ADCs, DACs, and Shift and Add (S+A), etc. The detailed specifications of each PE is shown later in Section V, Table I.

IV. Proposed SlimGNN Framework

In this section, we describe the SlimGNN framework. As shown in Figure 3, SlimGNN is divided into two phases: (1) GNN model offline pruning (pre-processing), and (2) in-field on-chip training of the pruned model using only critical input subgraphs on the ReRAM-based architecture. Here, our overall goal is to be able to train a pruned GNN model from scratch in-field with little to no loss in accuracy compared to its unpruned counterpart. For this purpose, we incorporate a crossbar aware LTP-inspired iterative model-pruning technique, referred to as LT-CAP. The model pruning step using LT-CAP is done offline once as a pre-processing step (Figure 3). The offline pruning step requires training the GNN model iteratively, and can be done using any hardware (CPU, GPU, TPU, etc.). However, this training is different from in-field training. The pruned model is mapped to ReRAM crossbars and can be used to train from scratch any number of times with different hyper-parameters and graph datasets. Hence, the cost of pre-processing is amortized over multiple in-field training instances. We note that all our experimental results show a comparison between the pruned and unpruned models for the in-field training phase.

However, we do not implement data pruning (we refer this step as GraphDiet) during model pruning in the preprocessing stage. As shown in Figure 3, GraphDiet is an online process which implemented during the in-field training process. We follow this algorithmic procedure for two reasons. First, the pruned models are general and can be used to train on any graph dataset. Second, even though the data pruning methodology is general and applicable to any graph dataset, the pruning steps rely on both model and graph data to decide which parts of the graph are redundant. The user may choose to train on different graph datasets in-field. Hence, the data pruning strategy should be executed in-field on whatever dataset the user wants to train on. To address this problem, we adopt a computationally inexpensive process, executed once and very-early (within few-epochs) during the GNN training process on the ReRAM-based architecture, referred as GraphDiet. The GraphDiet method allows us to prune the large graph dataset during in-field pruning. Hence, the computational cost for obtaining the pruned

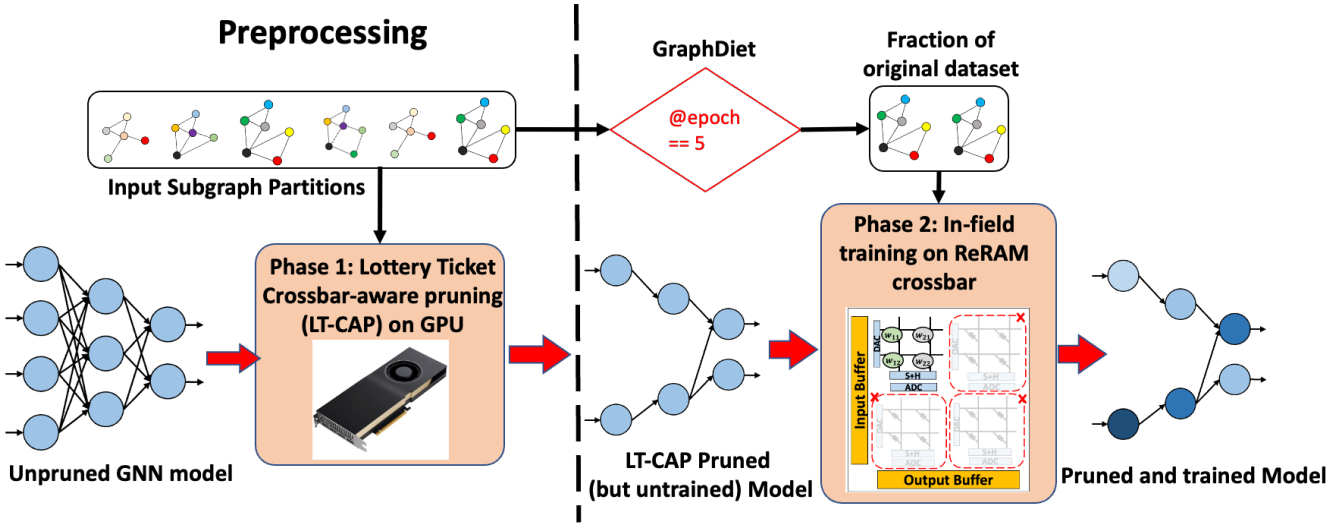

Fig. 3: Overview of the SlimGNN framework. It consists of the offline Preprocessing (Phase 1) to obtain the LT-CAP pruned model, and the in-field training (Phase 2) of the pruned model.

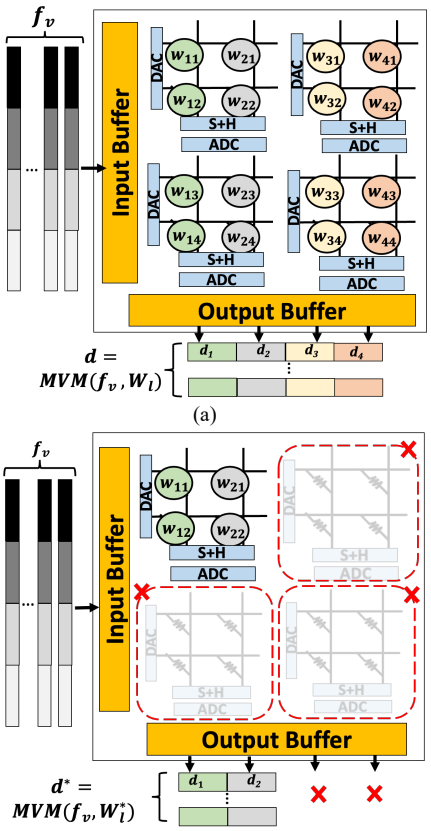

Fig. 4: Weight mapping and Inter-PE communication for (a) Unpruned GNN model and (b) LT-CAP enabled GNN model.

graph is negligible. We present details of the SlimGNN framework in the following sub-sections.

A. Model Pruning in SlimGNN

Model pruning techniques suited for ReRAM crossbars have been proposed in the literature [20] [21] [22]. These crossbar-aware pruning (CAP) methods prune few crossbar rows, columns, or small blocks (portion of a crossbar) to achieve high model sparsity while maintaining acceptable accuracy for inferencing purposes [21]. However, CAP methods lead to only a marginal energy reduction as rows/columns/portions of crossbars consume negligible energy compared to the crossbar's peripherals such the Analog-to-Digital Converter (ADC) [21]. Crossbar peripherals such as the ADCs (which accounts for ~60% of the total energy and area overhead) must remain active as the unpruned weights in crossbars must still be processed. Clearly, existing crossbar-aware pruning methods are not suited for ReRAM-based architectures as they only prune part of a crossbar.

SlimGNN utilizes a block pruning method that considers the entire structure of the ReRAM crossbar (i.e., size and bit/cell configuration) when pruning the model weights. The pruning block size is defined based on the crossbar size and its bits-per-cell configuration. For example, if the crossbar size is $c \times c$, with a r bit/cell configuration, the weight pruning block is of shape $c \times (c * r/B)$, where B is the precision of each weight value. In this work, the values of c, r, and B are 128, 2 and 16, respectively [7]. Hence, the block size for pruning becomes 128×16. Unlike existing CAP methods, we prune entire blocks of weights, which would have otherwise been mapped to an entire crossbar. This enables us to power gate or turn 'off' both the crossbar and its associated peripherals, which results in higher energy savings. Figure 4(a) and Figure 4(b) illustrate how unpruned weights and the pruned weights of a given GNN layer are mapped to ReRAM crossbar arrays. In Figure 4(a), the final output of the MVM operation is represented by d. Here, the output generated from the i^{th} column of the weight matrix is d_i where $d_i \subseteq d$. As shown in Figure 4(b), the crossbars outlined by the red boxes do not have any weights mapped on them due to pruning. Hence, they can be turned off alongside with their peripherals or they can be used for other purposes. As a result, those crossbars do not contribute to the final output (d^*). As demonstrated in Figure 4(b), columns 3 and 4 of the weight matrix are completely pruned away whereas columns 1 and 2 are partially pruned. As a result, we generate d_1 and d_2, but d_3 and d_4 are not generated. Hence, we do not need to send d_3 and d_4; this reduces inter-PE traffic.

Existing CAP methods prune pre-trained weights for inferencing purposes only. Most of these existing CAP methods utilize a multi-group LASSO algorithm to prune groups of weights that would otherwise be mapped along a row/column in an ReRAM crossbar. However, the pruned models fail to train from scratch. In the SlimGNN framework, we prune GNN model weights iteratively for the purpose of training from scratch as shown in Figure 3. To achieve this goal, we leverage a state-of-the-art LTP pruning method to enhance the existing CAP techniques so that we can train the pruned models from scratch without any noticeable loss in accuracy [23]. We refer to this pruning method as LT-CAP. The LT-CAP method iteratively prunes weights of a model based on their magnitude to find a pruned network that can be trained from scratch without accuracy loss. As shown in Figure 3, LT-CAP incorporates LTP by repeatedly pruning $p\%$ of $c \times (c * r/B)$-sized blocks of weights with the lowest average magnitude on each pruning iteration. The variable p is a user-defined parameter that can be varied based on the desired level of sparsity. We mention the value of p used in our experiments in Section 5. This step is done offline once per GNN model. The SlimGNN-enabled pruned model is then mapped to the ReRAM crossbars for in-field training. Model pruning accelerates GNN training in multiple ways: (a) it reduces the number of MVM operations necessary for training. This reduces the computation time for the GNN layers; (b) it reduces inter-PE communication. If all the weights in one column of a weight matrix are pruned, then the outputs are not generated, which reduces traffic hotspots. Both (a) and (b) lead to a reduction in the pipeline stage latency t_{stage} (Figure 1), which results in faster training; (c) The pruned models can be implemented using fewer crossbars, which leads to area and energy savings; and (d) The use of fewer crossbars also provides us the opportunity to duplicate model parameters to further parallelize GNN computations. This allows us to

process multiple input subgraphs simultaneously, thereby leading to better performance.

B. GraphDiet in SlimGNN

As discussed earlier, it is challenging to train GNNs on large-scale graphs. Data pruning can address this problem, but it has to be implemented during the in-field training phase. We address this problem by systematically reducing the number of subgraphs *very early* (within a few training epochs) during the in-field GNN training process on a given graph dataset. Prior work has shown that graphs used for GNN training contain redundant information [27]. In addition to this, recent work has proposed a method of finding important inputs for training DNNs without any accuracy loss [10]. Based on these findings, we hypothesize that all input subgraphs that are generated after the large monolithic graph is partitioned are not required for successful training. We validate this hypothesis experimentally in Section 5.

Scoring Function for Pruning: In this work, the unimportant subgraphs are identified and removed in a one-shot manner very early during GNN training. The key idea is to use the loss gradient norm of individual subgraphs to identify critical subgraphs to retain for the rest of the GNN training process. The importance of each subgraph (G_i) is determined by a scoring function ($Score$) described below:

$$\text{Score}(G_i) = \sum_l | z_v^l(G_i)\Delta W_l | \quad (5)$$

This scoring function shown in Equation (5) is the summation of the expected loss gradient norm for all neural layers, which bounds the expected change in loss for an arbitrary subgraph caused by removing the subgraph G_i. In other words, subgraphs with a small score are expected to have a limited influence on learning how to make predictions for the rest of the training subgraphs. The above scoring function ranks each subgraph based on the magnitude of the product of the gradient with respect to the input subgraph G_i (i.e., $z_v^l(G_i)$) and the change in the weights of the GNN layers (ΔW_l). The parameter z_v^l is the gradient update as mentioned earlier in Equation (4). This score indicates the contribution of each subgraph (G_i) to the change in the overall training loss function. We first start GNN training (in-field) using the entire graph (all the subgraphs). We compute the score of each subgraph during training on the ReRAM-based architecture and then remove g% of subgraphs with the lowest score ($Score$) at the e^{th} epoch very early in the training process. Here, g is a user-defined parameter determined by the tolerable accuracy drop.

Reasons for effective subgraph pruning: There are multiple synergistic reasons why we can prune subgraphs after a small number of training epochs with little to no accuracy loss: 1) Graphs are relational in nature and there are redundancies in terms of nodes with similar features, labels, and (approximate) relational structure. This leads to a "less is more" phenomenon; 2) The predictions for a large fraction of nodes can be made easily using the information from a small neighborhood. The subgraphs obtained by graph clustering automatically fall under easy (can be pruned) and hard cases (cannot be pruned); 3) The pruning step based on the scoring function does not require absolute scores (i.e., expected change in loss for an arbitrary subgraph caused by removing the subgraph G_i). As long as the relative ordering of subgraphs is preserved based on the scores, the pruning step will be effective.

Implementation details: At epoch e, Equation (5) (an MVM operation) is executed on the ReRAM crossbars and the result (the score) is transferred to the host to remove the unimportant subgraphs. This score computation is done only once throughout the training process and the timing overhead is included in the overall execution time for SlimGNN enabled training. After the eth epoch, only subgraphs with high scores are used for GNN training. As we show later, this process leads to trained GNN models with little to no accuracy loss when compared to training with the full graph. In Section 5, we show the predictive performance of the GNN model with GraphDiet for different values of $g \in [0,1]$ (fraction of pruned subgraphs). GraphDiet accelerates GNN training by reducing the size of the graph dataset. Instead of training on S subgraphs, the GNN model will now train on $((1-g) \times S)$ subgraphs (where $g < 1$). This reduces the pipeline depth as illustrated in Figure 1, which automatically speeds up training as there are fewer input subgraphs (hence fewer MVMs).

In summary, to accelerate GNN training on ReRAM-based PIM architectures, we leverage GraphDiet to reduce the input graph data and incorporate crossbar-aware model pruning to reduce the number of MVMs per layer. These two synergistic methods achieve high performance and improve the end-to-end execution time for training GNNs on the ReRAM-based system.

V. Experimental Results

In this section, we evaluate the performance of the proposed SlimGNN framework with ReRAM-based accelerator using various GNN models and graph datasets.

A. Experimental Setup

Datasets and GNN models (Workloads): In this work, we choose three diverse GNN models, namely: Graph Convolution Networks (GCN), Graph Attention Networks (GAT), and Graph sample and aggregate (SAGE); and four large-scale graph datasets – Reddit, Amazon2M, PPI, and Open Graph Benchmarks-Proteins (ogbn_p) for the purpose of evaluation [29]. The unique combination of these models and datasets constitutes the workloads for the ReRAM-based PIM architecture. More details about these GNN models and datasets are provided in Table II and Table III. For the sake of simplicity, we will refer to each GNN model and dataset workload configuration as Ci (as shown in Table III). For instance, C1 represents the setting where we train GCN with PPI dataset.

The datasets consist of several thousand/millions of nodes and edges and are representative of a wide range of real-world applications of predictive graph analytics. All GNN models considered in this work utilize mini-batch based training, which divides a large graph into multiple smaller sub-graphs. This allows us to train GNNs on memory-constrained settings such as in a ReRAM-based PIM architecture. This enables pipelined training of GNNs as mentioned earlier. All configurations (C1-C6) are trained for 200 epochs to ensure their convergence and to prevent overfitting. We experimented with different learning

Table I. Hardware Specifications

Preprocessing Hardware:	OS: Linux CentOS GPU: NVIDIA V100, 32GB CPU: Intel Xeon Gold 5222 @ 3.8GHz, 16 cores, 32K L1 cache and 1024K L2 cache.
In-field Training Hardware:	ReRAM-based 3D PIM architecture: 4 planar tiers, 9 PEs per tier, 4 ReRAM tiles per PE.
ReRAM Tile:	96-ADCs (8-bits), 12×128×8 DACs (1-bit), 96 crossbars, 128×128 crossbar size, 10MHz, 2-bit/cell resolution, 0.34W, 0.38 mm^2 [7]

Table II. Graph Datasets Statistics

Datasets	# Nodes	#Edges	FLR
PPI	56,944	818,716	0.41
Reddit	232,965	11,606,919	14.68
Amazon2M	2,449,029	61,859,140	2.13
Ogbn-proteins (ogbn-p)	132,534	39,561,252	4

Table III. Training Workload Configuration (lr: learning rate, b_size: batch size, S: number of graph partitions)

Datasets	Model	Wrkl. Config.	Hyperparameters
PPI	GCN	C1	lr=0.01, b_size=2, S=250
	GAT	C2	lr=0.005, b_size=2, S=250
Reddit	GCN	C3	lr=0.01, b_size=10, S=1,500
	SAGE	C4	lr=0.01, b_size=256, S=1,500
Amazon2M	GCN	C5	lr=0.01, b_size=20, S=10,000
Ogbn_p	SAGE	C6	lr=0.005, b_size=1, S=100

rates (lr), and choose the one that works best for each configuration. We tune the hyperparameters (lr, b_size and S) for each configuration to achieve the best possible accuracy. For graph partitioning, we employ METIS, which is a fast and scalable graph clustering tool [25][37]. The graph clustering algorithm only takes a small portion of preprocessing time, and allows us to train on large datasets. In phase 2 (in Fig R3), the graph partitions (subgraphs) are loaded from main-memory onto the ReRAM architecture for the in-field training process.

SlimGNN Configuration: As shown in Figure 3, there are two stages in the SlimGNN framework. The first stage involving LT-CAP (as shown in Figure 3) is executed on the preprocessing hardware (as shown in Table I) to generate the pruned GNN model. Then, the LT-CAP enabled pruned GNN model (with weights reset to their untrained values) is mapped to the ReRAM-based 3D PIM architecture (described in Table I) for in-field training with different graph datasets as shown in Figure 3. The graph pruning (GraphDiet) is done during the in-field training based on the importance of each input subgraph. The importance score for each input subgraph is determined at the 5th epoch of training on the ReRAM-based PIM architecture. Our experiments show pruning earlier than the 5th epoch leads to accuracy loss. This happens as the model does not get sufficient opportunity to explore and score the input subgraphs reliably before five epochs. The accuracy loss was negligible when the pruning was done earliest at the 5th epoch and this behavior was consistent for different datasets. Subsequently, only the identified critical subgraphs are used for the rest of the in-field training process. For all in-field training,

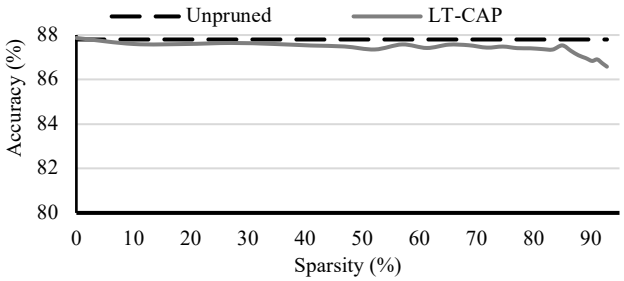

Fig. 5: Accuracy vs sparsity for LT-CAP pruning technique on the Amazon2M dataset in C5.

we compare the unpruned GNN model with its pruned counterpart running on the same ReRAM-based architecture as tabulated in Table I. This provides a fair comparison between the pruned and the unpruned models for each GNN workload configuration.

ReRAM-based Architecture: The architecture considered in this work consists of 36 ReRAM-based processing elements (PEs) distributed over four planar tiers connected using through silicon via (TSV)-based vertical links. Table I summarizes of the specifications of the ReRAM-based PIM architecture used in this work. In this work, the GNN model weights are stored on the ReRAM crossbar array using 16-bit fixed point precision. We evaluate the overall system performance of the SlimGNN framework using the NeuroSim v2.1 simulator [30]. NeuroSim models the performance of ReRAM-based PIM accelerators. We used NeuroSim to obtain the model prediction accuracy and determine the area and energy of the ReRAM architecture. The NeuroSim software runs on a GPU and CPU as described in Table I.

GNN training generates an enormous volume of traffic, which can bottleneck performance [6]. In this architecture, we utilize a multicast-enabled 3D mesh network on chip (NoC) as the interconnection backbone for communicating between the processing elements during GNN training [6]. The 3D ReRAM-based 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. The 3D mesh NoC architecture also efficiently handles the increased communication demand due to duplicated nodes in subgraphs and reduces the overall latency. In addition to this, we adopt an optimized mapping strategy that maps GNN layers to ReRAM tiles to further improve the communication latency [6]. The NoC performance has been evaluated using Garnet, and NeuroSim was used to calculate the injection rate needed for the cycle-accurate NoC simulator [31].

Training in the presence of non-idealities: To prevent accuracy loss due to low precision, we incorporate stochastic rounding [32]. Stochastic rounding enables high accuracy even at lower precision settings [33]. The implementation of the stochastic rounding unit introduces less than 1% area and energy overhead. In addition, ReRAMs suffer from multiple non-ideal effects such as noise and faults. However, recent work has shown that these issues can be addressed using simple techniques such as error-correction or weight clipping [34]. Hence, we can still train GNNs despite these non-ideal effects.

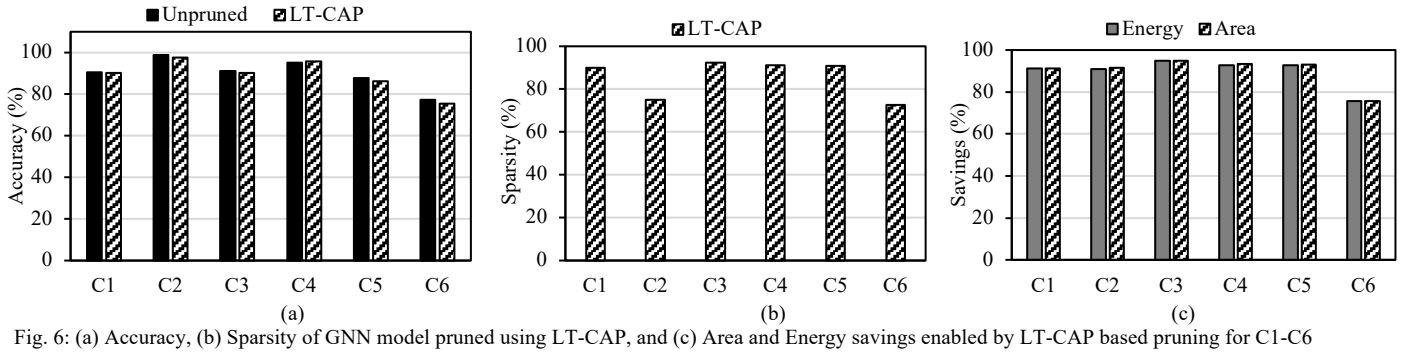

Fig. 6: (a) Accuracy, (b) Sparsity of GNN model pruned using LT-CAP, and (c) Area and Energy savings enabled by LT-CAP based pruning for C1-C6

However, for this work, we will assume ideal ReRAM behavior to demonstrate the effectiveness of the proposed model and graph pruning methodology.

B. Training Performance of SlimGNN

In this subsection, we present a detailed performance evaluation of the SlimGNN framework used for GNN training on the ReRAM-based 3D PIM architecture.

Results for model pruning: First, we present the accuracy and sparsity achieved by the model pruning technique (LT-CAP) and compare it with the unpruned counterpart [6]. Figure 5 shows the effectiveness of the LT-CAP technique on the C5 configuration (GCN with Amazon2M dataset) as an example compared to its unpruned version at different levels of sparsity. In this case, we choose a pruning percentage (p) of 10% in every iteration, i.e., we prune 10% of the weights remaining in each iteration. The value of p is a hyperparameter, which can be modified based on the required level of sparsity. As shown in Figure 5, as the degree of sparsity increases, the model accuracy decreases. Here, we choose the maximally pruned model which can be trained from scratch and achieves less than 0.5% accuracy loss compared to its unpruned counterpart. Given this constraint, the LT-CAP technique achieves up to 90.2% sparsity for C5. Similar accuracy trends were also observed for other configurations.

Next, Figure 6(a) and Figure 6(b) present the model accuracy and sparsity for the LT-CAP model for configurations C1 - C6 (from Table III). As shown in Figure 6(a) and Figure 6(b), under the proposed accuracy constraint (0.5% accuracy loss), LT-CAP can achieve up to 90%, 75%, 92.3%, 91.3%, 90.2%, and 72.6% sparsity for configurations C1-C6, respectively. Loading the LT-CAP pruned GNN models on the ReRAM-based PIM architecture enables us to power-gate or re-use the available crossbars as shown in Figure 4. This results in reduction in crossbar area and energy consumption. Figure 6(c) demonstrates the area and energy savings enabled by LT-CAP at iso-performance. The savings are compared to the unpruned model mapped on the ReRAM crossbars. For a fair comparison with the unpruned model, we consider an iso-performance scenario (equal training time) to estimate the hardware savings achieved by the LT-CAP enabled model. As seen from Figure 6(c), the proposed LT-CAP based model requires 90% less area on average when compared to the unpruned model. Commensurate with this, power gating the unused crossbars reduces energy consumption by 89.6% on average as well. This is because, the pruning technique incorporated in LT-CAP removes all weights that could have otherwise been stored on the crossbar array. We do not require the peripherals (such as ADC, DAC, S+A, etc.) associated with these empty crossbars either, resulting in huge area and energy savings. Hence, LT-CAP can achieve high sparsity while being extremely area and energy efficient.

Results for graph pruning (GraphDiet): While training the LT-CAP model, we incorporate GraphDiet introduced in Section 4. Figure 7 demonstrates the effect of GraphDiet on the accuracy of the LT-CAP GNN models for configurations C1, C3 and C5, respectively. Note that similar accuracy trends were also observed for other configurations. To determine how much of the input graph can be pruned, we vary the amount of graph pruning (g, the percentage of pruned subgraphs) from 10% to 90%. For all configurations, we observed that the GraphDiet decision made at the 5th epoch is sufficient to preserve the GNN model accuracy. As seen from Figure 7, as we reduce the number of input subgraphs used in training, the model accuracy starts degrading. Note that, we allow an additional 0.5% drop in accuracy when implementing GraphDiet on top of the LT-CAP based pruned GNN model. Overall, we allow a maximum of 1% drop in model accuracy when compared to the Unpruned GNN model with no GraphDiet.

The amount of achievable graph pruning through GraphDiet within this limit is governed by the difficulty of the prediction task and the amount of redundant information present in a given graph dataset. One of the key properties that characterizes this behavior is the number of features associated with each vertex (node) of the graph. Graphs with high number of features have more discriminative and redundant information with respect to the prediction task. However, this is not the only property, which determines the possible amount of allowable graph pruning using GraphDiet.

Table II presents a parameter "Feature to Label Ratio" (FLR), which captures the amount of available information to approximately characterize the difficulty of node classification task for any given graph dataset. A dataset with high FLR has more features to learn from, but also has relatively lower number of class labels resulting in a simpler classification task. The dataset with a low value of FLR has relatively less redundant information to learn from. Hence, it requires higher number of input subgraphs to achieve high accuracy. As seen from Figure 7, we can remove 60% of the input subgraphs from training for the Reddit dataset in configuration C3 given the constraint of less than 1% overall accuracy drop. Under this

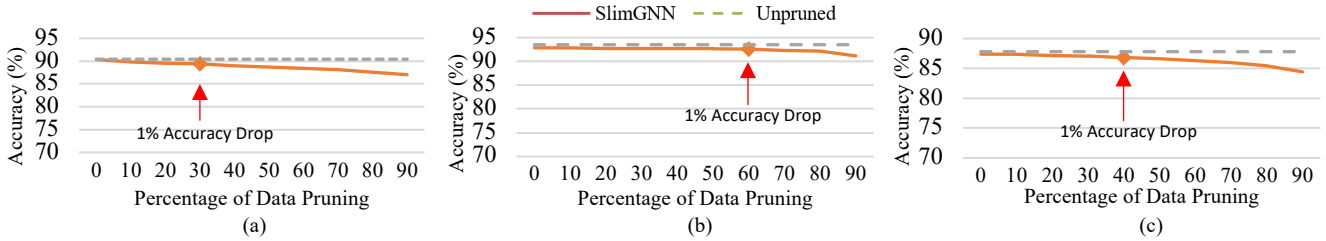

Fig. 7: Accuracy of *GraphDiet* based data pruning on LT-CAP based Pruned models for (a) C1, (b) C3 and (c) C5

same constraint, Amazon2M and PPI in C5 and C1 respectively achieves reduction in the number of input subgraphs by 40% and 30% respectively. These numbers are consistent with the FLR for these datasets. Reddit has the highest FLR of 14.68, which is ~7 ×, ~35 ×, and ~3.7 × larger than the FLR values for Amazon2M, PPI, and ogbn_p respectively. Thus, Reddit has the highest amount of redundant information and thus, a relatively larger portion of its graph can be removed from the training process. PPI has the lowest FLR value of 0.41 and this makes it very hard to achieve high accuracy without large number of input subgraphs. Thus, for PPI, the amount of achievable subgraph pruning incorporating GraphDiet is significantly lower. However, for all six configurations (C1-C6), we observed that we can still safely remove a large number of input subgraphs in the early part of the training process. This enables improvement in the performance of the SlimGNN-enabled ReRAM-based architecture. Overall, we can prune 90%, 75%, 92.3%, 91.3%, 91%, and 72.6% of the weights and 30%, 30%, 60%, 60%, 40%, and 40% of the input sub-graphs for all six configurations (C1-C6), respectively. Next, we present the overall performance of the ReRAM-based PIM architecture under consideration.

Granularity of graph partitioning for GraphDiet: In this work, we aim to determine an appropriate number of graph partitions or input subgraphs (S) that is compatible with the SlimGNN framework. It is intuitive that the identification of unimportant subgraphs becomes more difficult in scenarios where the subgraphs are relatively bigger. Given an input graph, we can partition it into S subgraphs using the METIS partitioning tool [9] [25]. A large S implies a larger number of subgraph partitions, but each subgraph is small and has less information; hence, it is easier to prune. Smaller S implies fewer partitions, but each partition (input subgraph) is large and contains more information; hence it is difficult to prune. However, note that, extremely large values of S are not desirable as it requires many crossbars to store each sub-graph, which may not be available. Similarly, a very small S is also undesirable as it unnecessarily increases the execution time as the subgraphs are too many in number.

In Figure 8(a) and Figure 8(b), we show the accuracy of GraphDiet (with pruning percentages of 60% and 40% respectively) compared to the baseline GNN training for different number of partitions. Here, we refer to the baseline as the corresponding unpruned GNN model trained using all subgraphs. As an illustration, we consider the Reddit and Amazon2M datasets trained on a Graph Convolution Network as in configurations C3 and C5, respectively. As expected, Figure 8 shows that as the number of partitions increases, we can prune more as there are many subgraphs with less information; hence, these subgraphs are not needed. Moreover, the GCN trained using the GraphDiet technique at large S achieves comparable accuracy with the baseline. In contrast, when the number of subgraphs is less, we are not able to prune many subgraphs and there is an accuracy drop. This happens as large sub-graphs contain useful information and pruning them results in poor training accuracy. In this work, we choose the number of partitions (S) to be 100, 250, 1500, and 10,000 for ogbn-proteins, PPI, Reddit, and Amazon2M datasets, respectively, as our analysis shows that these numbers provide a good trade-off between execution time and storage requirements.

C. Overall Performance Evaluation of SlimGNN

The overall execution time of GNN training on the ReRAM-based architecture is governed by the delay of each pipeline stage and the overall end-to-end pipeline depth. As each GNN layer contributes to one stage of the pipeline, the delays associated with GNN layers' computation (MVM operations), and inter-PE communication determine the overall stage delay

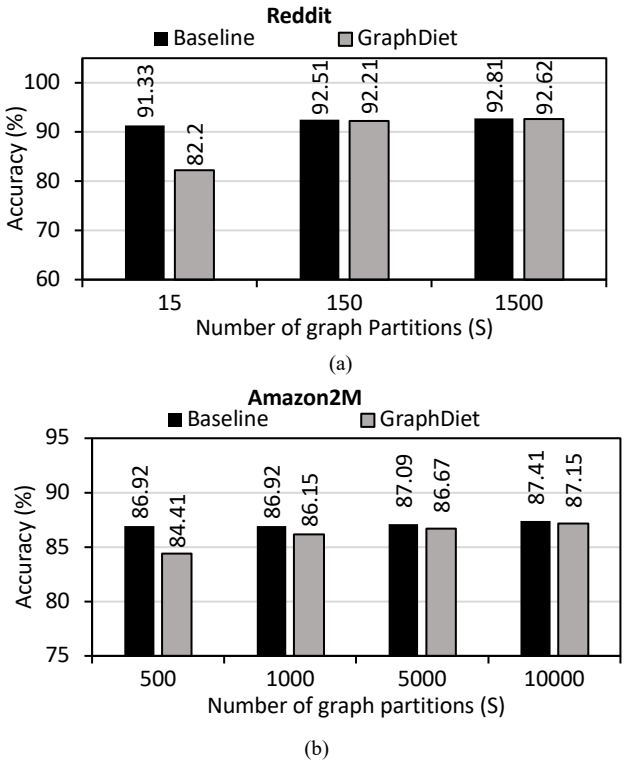

Fig. 8: Performance of GraphDiet for (a) Reddit and (b) Amazon2M using different number of graph partitions (subgraphs). As the number of partitions increase on the x-axis, the size of the subgraphs reduces.

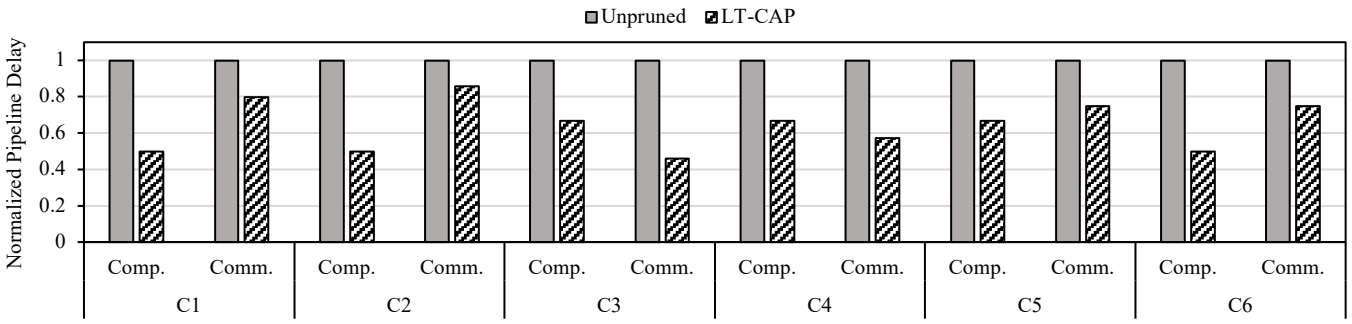

Fig. 9: Normalized computation (comp.) and communication (comm.) delays for the unpruned and pruned (using LT-CAP) models for all configurations (C1-C6). All computation and comunication delays of LT-CAP are normalized with the unpruned model's computation and communication delays respectively.

(t_{stage}). In this sub-section, we first evaluate the impact of LT-CAP on the computation and communication latencies of GNN training implemented on the ReRAM-based PIM architecture. Next, we demonstrate the effects of LT-CAP and GraphDiet (which together make up the SlimGNN framework) on the overall execution time. Finally, we present the overall performance evaluation and energy consumption of the SlimGNN-enabled ReRAM-based PIM architecture. We assume an iso-area scenario for this analysis, i.e., the number of ReRAM crossbars available is the same for all the cases.

Moreover, we compare the overall performance of SlimGNN with respect to a recently proposed unified GNN sparsification method (UGS) [24]. UGS jointly prunes GNN weights and graph adjacency matrices using trainable masks to reduce the number of MVM operations associated with GNN training [24]. However, the graph pruning in UGS only removes the edges from the graph adjacency matrix and thus the overall size of the input (i.e., the number of input subgraphs) remains unchanged. Moreover, the weight pruning method proposed in UGS is crossbar-unaware. UGS does not guarantee pruning all weights belonging to the same row/column/crossbar. Hence, it does not reduce crossbar requirement for ReRAM-based architectures. We show later via experiments that our proposed SlimGNN method outperforms UGS by a significant margin for training GNNs.

Effect of model pruning: Figure 9 shows the worst-case computation and communication latencies when we train the unpruned and LT-CAP pruned GNN models on ReRAM-based architectures. Here, all latencies are normalized with respect to the communication delay of the unpruned model. 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 duplicating the weights on multiple crossbars. Each crossbar would then process a different input subgraph in parallel. For instance, by duplicating the weights on two sets of crossbars, we can reduce the execution time by half and so on. The number of times the weights can be duplicated depends on the number of ReRAM crossbars available in the overall architecture. The unpruned model cannot achieve a similar level of parallelism as these models are large. Hence, the architecture will have fewer crossbars available to replicate its weights. As shown in Figure 6(c), pruning using the LT-CAP method results in high crossbar savings (over 90% fewer crossbars on average compared to the unpruned case). As a result, we can speed-up all the layers using the available ReRAM crossbars. The available crossbars can now be used to replicate the unpruned weights to reduce the computation delay. Hence, LT-CAP results in an average improvement of 39% in the computation latency as shown in Figure 9. This results in faster GNN training.

However, inter-PE communication also has a significant influence on the execution time of GNN training [6]. The data traffic in GNN training is generated from the MVM operations as discussed earlier in Section 3. Pruning reduces the amount of data being exchanged, thus alleviating the communication bottleneck as shown in Figure 4(b). Hence, if many rows/columns are pruned, we can reduce the amount of communication significantly. However, partially pruned rows/columns may still result in non-zero outputs that must be communicated. LT-CAP results in multiple rows/columns of the full weight matrix becoming zero, which reduces the overall number of useful messages generated from the weight matrix of any given layer. As shown in Figure 9, this results in a reduction of the communication delay varying between 14% to 54% depending on the configurations (C1-C6). The improvement is much higher for Reddit (in configurations C3 and C4) as it has the largest number of features (~75 ×, 12 ×, and 6 × more features than ogbn-proteins, PPI, and Amazon2M respectively) amongst all the datasets under consideration here. A larger number of features results in more redundant information being exchanged in each layer, i.e., more communication. LT-CAP helps in reducing redundant communication. Thus, we see relatively larger improvements for Reddit after pruning.

Overall, the pipeline stage delay is bottlenecked by the slowest among the computation and the inter-PE communication stage delays. From Figure 9, we can see that the execution time of GNN training for configurations are predominantly bottlenecked by communication. However, LT-CAP accelerates both computation and inter-PE communication significantly when compared to the unpruned case resulting in lower pipeline stage delay t_{stage}.

Performance Evaluation with LT-CAP and GraphDiet: The second factor that governs the overall execution time of GNN training on ReRAM-based architectures is the end-to-end pipeline depth. The depth of the pipeline is determined by the number of input subgraphs; this number can be reduced significantly due to GraphDiet on the ReRAM-based

architecture. As shown in Figure 7, we can reduce up to 30%, 60%, and 40% of the input subgraphs for configurations C1, C3, and C5 respectively. Figure 10(a) and Figure 10(b) demonstrates the normalized execution time and energy consumption respectively for; the unpruned case, UGS, an existing crossbar-aware column-pruning method referred as Col-P, the standalone LT-CAP method without GraphDiet (LT-CAP), and the SlimGNN framework (LT-CAP + GraphDiet) for C1-C6.

UGS is a crossbar-unaware pruning method. As a result, it cannot prune entire 128×16 block of weights. Hence, it is unable to improve the computation time (duplication of blocks of weights is not possible) and inter-PE communication time as shown in Figure 4. Thus, in an iso-area setting, UGS cannot improve the overall execution time of GNN training in a ReRAM-based PIM architecture. Furthermore, the Col-P method achieves similar levels of sparsity as other methods (such as [20] and [22]); hence, we choose Col-P as a representative crossbar-aware pruning technique to evaluate the effectiveness of LT-CAP.

As shown in Figure 10(a), the LT-CAP based pruned model is able to reduce the execution time by 27.1%, 21.4%, 50%, 38.6%, 25%, and 25% for configurations C1-C6 respectively compared to their unpruned versions. This improvement is determined by the reduction in pipeline stage delay for each configuration. Enabling GraphDiet on the pruned model results in a further execution time reduction of 55%, 33%, 76%, 71%, 53.3%, and 53% for configurations C1-C6 respectively. The performance improvement achieved on top of the LT-CAP based pruned model is due to the reduction in end-to-end pipeline depth.

Overall, as shown in Figure 10(a), the SlimGNN-enabled training improves execution time by ~57% on average compared to the unpruned model running on an iso-area ReRAM-based PIM architecture. Figure 10(b) depicts the energy consumption (normalized with respect to unpruned) for GNN training, respectively, for Unpruned, UGS, Col-P, LT-CAP, and SlimGNN. Figure 10(b) shows that the SlimGNN-enabled model reduces energy consumption by 62%, 54.21%, 95%, 87.5%, 68.8%, and 68% on average for C1-C6, compared to the Unpruned, UGS, and LT-CAP models running on the same ReRAM-based architecture, respectively. Reddit dataset in configurations C3 and C4 has the highest reduction in energy and execution time as it achieves high sparsity and has significantly more node-level features compared to other datasets.

Also, as shown in Figure 10(b), our proposed standalone LT-CAP method and the SlimGNN framework (LT-CAP + GraphDiet) consumes 27.16% and 46.55% less energy on average compared to the Col-P method, respectively. This happens because Col-P prunes only individual rows and/or columns of the crossbar which leads to only a marginal energy reduction as rows/ columns/portions of crossbars consume negligible energy compared to the crossbar's peripherals such the Analog-to-Digital Converter (ADC). Unlike existing Col-P methods, LT-CAP prunes entire blocks of weights, which would have otherwise been mapped to an entire crossbar. This enables us to power gate or turn 'off' both the crossbar and its associated peripherals, which results in higher energy savings.

In general, the reduced energy consumption results from the improved execution time and reduced crossbar requirements enabled by SlimGNN in all configurations. Additionally, SlimGNN-enabled model on the ReRAM-based architecture is $\sim 12.8\times$ faster in terms of execution time compared to a Nvidia V100 GPU on an average for the GNN models considered in this work. This happens as ReRAMs are significantly faster than GPUs for large scale MVM operations [7] [11]. Overall, SlimGNN can accelerate training by up to $4.5\times$ while consuming up to $6.6\times$ less energy than the unpruned implementation (Unpruned) on an iso-area ReRAM-based PIM architecture.

Lifetime of SlimGNN-enabled ReRAM Architectures: It is well known that ReRAM crossbar arrays suffer from low write endurance [35]. Prior work has proposed Low-rank Training (LRT) and optimized weight update techniques to address the low-write endurance challenge of ReRAM-based architectures and prolong their overall lifetime [35] [36]. Incorporating SlimGNN improves the endurance of the ReRAM-based architecture further. As GraphDiet reduces the number of input subgraphs, it also helps to reduce the number of weight updates occurring during training. ReRAM-write endurance has been shown to be between $10^6 - 10^{12}$ writes [35] [36]. Hence, for our analysis we consider a worst-case scenario of 10^6 writes. By incorporating LRT, the write endurance is enhanced by a factor of ~300 [35]. As an example, to train on the Reddit dataset (in C3 and C4) for 200 epochs, we need ~12k weight updates for the SlimGNN-enabled model. Thus, by incorporating SlimGNN and LRT, we can train the model for

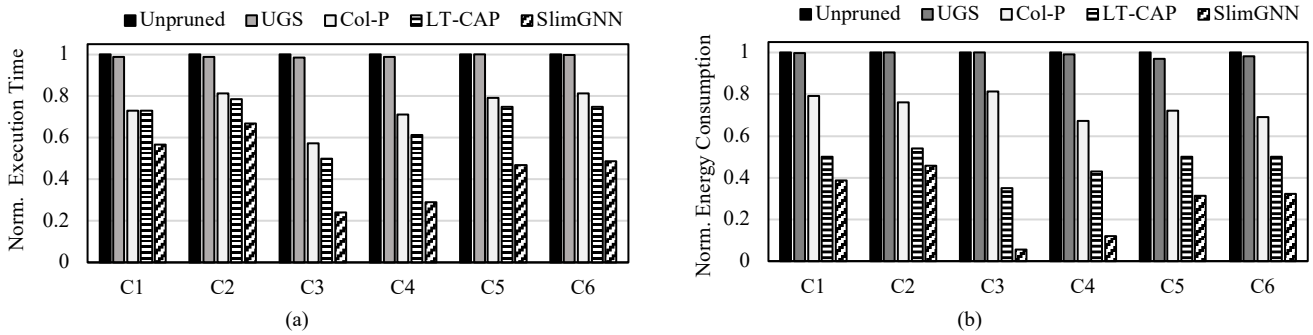

Fig. 10: Normalized (a) execution time and (b) energy consumption for the unpruned, UGS, Col-P, LT-CAP and SlimGNN-enabled GNN (normalized with respect to the execution of the unpruned model on the ReRAM-based PIM architecture).

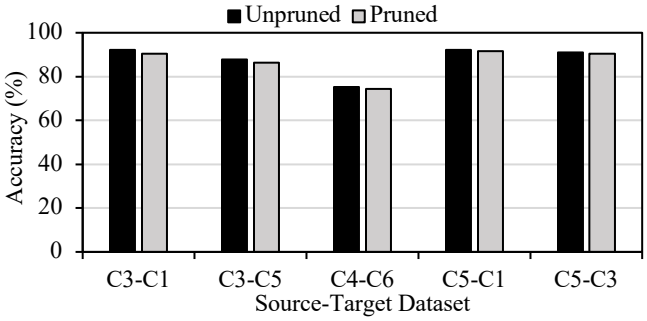

Fig. 11: Transferability of pruned models generated using a dataset in a configuration to other datasets.

up to 22,000 times on the proposed ReRAM-based system. Similar trends were also observed for the other two datasets. Thus, SlimGNN-enabled ReRAM based architectures have adequately high lifetime.

D. Transferability of Pruned Model to other Datasets

In this work, we also demonstrate that the pruned model obtained using one dataset in the pre-processing phase can be used to train in-field with another dataset with negligible accuracy loss. Figure 11 shows the result of this experiment. In Figure 11, the model is pre-processed with a '***source***' dataset and then the pruned model is used to train on a '***target***' dataset. The combinations are referred as Ci-Cj in Figure 11, where Ci represents the configuration for pre-processing (with 'source' dataset) and Cj represents the configuration used for in-field training (with 'target' dataset). Both Ci and Cj are elaborated in Table III of the manuscript. For the sake of brevity, in Figure 11 we show the results of training the offline pruned model generated using the Reddit dataset as the source (configurations C3 and C4) on the PPI, Amazon2M and ogbn_proteins datasets as targets (configurations C1, C5 and C6) for in-field training. Similarly, we have also used a pruned model generated using the Amazon2M dataset as the source (configuration C5) to train the PPI and Reddit datasets (in C1 and C3) as shown in Figure 11. As shown in Figure 11, the pruned models are transferrable across datasets. For example, the crossbar-aware pruned (LT-CAP) GCN model generated with the Reddit dataset in the pre-processing step can be used for in-field training with the Amazon2M dataset (referred as C3-C5 in Figure 11) without noticeable accuracy loss.

The transferability of Lottery-ticket pruned (LTP) models across datasets happens because: (a) 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 [37]; (b) The transferred tickets act as a regularizer and prevent overfitting while training [37]; and (c) winning tickets learn generic inductive biases which improve training. Hence, LT-CAP-based models (which are based on LTP) can also be used with other datasets. Overall, we observe less than 1% accuracy drop on average for the pruned model during in-field training compared to their respective unpruned versions for all source-target dataset combinations considered in this work. This further demonstrates that the offline pruned model generated using one dataset can be transferred to another dataset for in-field training with negligible accuracy drop compared to their unpruned versions.

VI. Conclusion

ReRAMs enable the design of high-performance and energy-efficient architectures for accelerating GNN training. The end-to-end execution time for GNN training on ReRAM-based PIM architectures is determined by the number of subgraphs (size of input workload) and the model parameters. In this work, we propose incorporating GraphDiet on ReRAM-based PIM architectures to reduce the number of input subgraphs without sacrificing the model accuracy. To compliment GraphDiet, we leverage a block-based GNN model pruning, which helps to further improve the performance and energy efficiency. GraphDiet and GNN model pruning together constitute the SlimGNN framework, which enables high performance GNN training on ReRAM-based PIM architectures. Overall, SlimGNN accelerates training by up to 4.5 × while using 6.6 × less energy when compared to its unpruned counterpart on ReRAM-based 3D PIM platforms.

References

[1] R. Ying et al., "Graph Convolutional Neural Networks for Web-Scale Recommender Systems," in *ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, London, 2018.

[2] W. Fan et al., "Graph Neural Networks for Social Recommendation," in *The World Wide Web Conference*, San Fransisco, CA, 2019.

[3] T. N. Kipf and M. Welling, "Semi-Supervised Classification with Graph Convolutional Networks," in *International Conference on Learning Representations (ICLR)*, Toulon, 2017.

[4] T. Geng et al, "AWB-GCN: A Graph Convolutional Network Accelerator with Runtime Workload Rebalancing," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020.

[5] D. Xu et al., "Edge Intelligence: Architectures, Challenges, and Applications," in *arXiv:2003.12172v2*.

[6] A. I. Arka et al., "Performance and Accuracy Tradeoffs for Training Graph Neural Networks on ReRAM-Based Architectures," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems,* vol. 29, no. 10, pp. 1743-1756, 2021.

[7] A. Shafiee et al, "ISAAC: a convolutional neural network accelerator with in-situ analog arithmetic in crossbars.," in *International Symposium on Computer Architecture (ISCA)*, Seoul, Korea, 2016.

[8] L. Song et al, "PipeLayer: A Pipelined ReRAM-Based Accelerator for Deep Learning," in *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, Austin, TX, 2017.

[9] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs.," *SIAM Journal on Scientific Computing,* 1998.

[10] M. Paul, S. Ganguli and G. K. Dziugaite, "Deep Learning on a Data Diet: Finding Important Examples Early in Training," in *Advances in Neural Information Processing Systems 34 (NeurIPS 2021)*, 2021.

[11] D. Fujiki, S. Mahlke and R. Das, "In-memory Data Flow Processor," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Portland, OR, 2017.

[12] B. K. Joardar et al., "AccuReD: High Accuracy Training of CNNs on ReRAM/GPU Heterogeneous 3D Architecture," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* 2020.

[13] L. Song et al., "GraphR: Accelerating Graph Processing Using ReRAM," in *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2018.

[14] L. Zheng et al, "Spara: An Energy-Efficient ReRAM-Based Accelerator for Sparse Graph Analytics Applications," in *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, New Orleans, LA, 2020.

[15] P. Chi et al., "PRIME: A Novel Processing-in-Memory Architecture for Neural Network Computation in ReRAM-Based Main Memory," in *International Symposium on Computer Architecture (ISCA)*, South Korea, 2016.

[16] T. Yang et al., "PIMGCN: A ReRAM-Based PIM Design for Graph Convolutional Network Acceleration," in *Design Automation Conference (DAC)*, 2021.

[17] S. Han, J. Pool, J. Tran and W. Dally, "Learning both weights and connections for efficient neural networks," *Advances in Neural Information Processing Systems (NeurIPS),* pp. 1135-1143, 2015.

[18] H. Li et al., "Pruning Filters for Efficient Convnets," in *International Conference on Learning Representations (ICLR)*, 2017.

[19] W. Wen et al., "Learning Structured Sparsity in Deep Neural Networks," *Advances in Neural Information Processing Systems (NeurIPS)),* pp. 2082-2090, 2016.

[20] L. Liang et al., "Crossbar-Aware Neural Network Pruning," *IEEE Access,* vol. 6, pp. 58324-58337, 2018.

[21] J. Meng et al., "Structured Pruning of RRAM Crossbars for Efficient In-Memory Computing Acceleration of Deep Neural Networks," *IEEE Transactions on Circuits and Systems II: Express Briefs,* vol. 68, no. 5, p. 15.

[22] C. Chu et al., "PIM-Prune: Fine-Grain DCNN Pruning for Crossbar-Based Process-In-Memory Architecture," in *IEEE Design & Automation Conference*, 2020.

[23] J. Frankle and M. Carbin, "The lottery ticket hypothesis: Finding sparse, trainable neural networks," in *International Conference on Learning Representations (ICLR)*, 2019.

[24] T. Chen et al., "A Unified Lottery Ticket Hypothesis for Graph Neural Networks," in *International Conference on Machine Learning (ICML)*, 2021.

[25] W. L. Chiang et al., "Cluster-GCN: An Efficient Algorithm for Training Deep and Large Graph Convolutional Networks," in *ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, Anchorage, AK, 2019.

[26] C. Coleman et al., "Selection via Proxy: Efficient Data Selection for Deep Learning," in *International Conference on Learning Representations (ICLR)*, 2020.

[27] H. You, Z. Lu, Z. Zhou, Y. Fu and Y. Lin, "Early-Bird GCNs: Graph-Network Co-Optimization Towards More Efficient GCN Training and Inference via Drawing Early-Bird Lottery Tickets," in *AAAI Conference on Artificial Intelligence*, 2022.

[28] A. Tripathy, K. Yelick and A. Buluc, "Reducing Communication in Graph Neural Network Training," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020.

[29] W. Hu et al., "Open Graph Benchmark: Datasets for Machine Learning on Graphs Weihua," in *Neural Information Processing Systems (NeurIPS)*, 2020.

[30] P. Chen, X. Peng and S. Yu, "NeuroSim: A Circuit-Level Macro Model for Benchmarking Neuro-Inspired Architectures in Online Learning," *EEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 37, no. 12, pp. 3067-3080, 2018.

[31] N. Agarwal, T. Krishna, L. Peh and N. Jha, "GARNET: A detailed on-chip network model inside a full-system simulator," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2009.

[32] S. Gupta, A. Agrawal, K. Gopalakrishnan and P. Narayanan, "Deep learning with limited numerical precision," in *International Conference on Machine Learning*, 2015.

[33] T. Na, J. Ko, J. Kung and S. Mukhopadhyay, "On-chip training of recurrent neural networks with limited numerical precision," *Proceedings of the International Joint Conference on Neural Networks,* pp. 3716-3723, 2017.

[34] B. K. Joardar et al., "Learning to Train CNNs on Faulty ReRAM-based Manycore Accelerators," *ACM Transactions on Embedded Computing Systems,* pp. 1-23, 2021.

[35] 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 Journal of Solid-State Circuits (JSSC),* vol. 1, pp. 9-10, 2022.

[36] W. Wen, Y. Zhang and J. Yang, "ReNEW: Enhancing Lifetime for ReRAM Crossbar Based Neural Network Accelerators," in *International Conference on Computer Design (ICCD)*, 2019.

[37] A. Morcos, Y. Haonan, M. Paganini and Y. Tian, "One ticket to win them all: Generalizing lottery ticket initializations across datasets and optimizers," *Advances in Neural Information Processing Systems,* vol. 32, no. NeurIPS, 2019.

Chukwufumnanya O. Ogbogu (Graduate Student Member, IEEE) received his B.S. degree in Electrical and Electronic Engineering from Obafemi Awolowo University, Nigeria, in 2019. He is currently pursuing the Ph.D. degree with the School of Electrical Engineering and Computer Science, Washington State University, Pullman, WA, USA. His current research interests include; low-power VLSI design, machine learning, hardware accelerators for deep learning, emerging non-volatile memories.

Aqeeb Iqbal Arka (Graduate Student Member, IEEE) received his B.Sc. degree in Electrical and Electronics Engineering from Bangladesh University of Engineering and Technology in 2016 and his Ph.D. in Electrical and Computer Engineering from Washington State University in 2022. His research interests include design of manycore accelerators for machine learning and HPC workloads and novel 3D integration techniques for designing high-performance manycore systems.

Lukas Pfromm (Student Member, IEEE) is an undergraduate student currently pursuing a B.Sc. degree in Electrical Engineering at Oregon State University, Corvallis, OR, USA. He has taken special interest in the area of digital circuit design and hopes to continue his education at graduate level.

Biresh Kumar Joardar (M'20) is an assistant professor at University of Houston. He finished his PhD from Washington State University in 2020. Following that, he was a Post-doctoral Computing Innovation Fellow (CI-Fellow) at the Department of Electrical and Computer Engineering at Duke University. His current research interests include machine learning accelerators, and hardware reliability. He received the 'Outstanding Graduate Student Researcher Award' at

Washington State University in 2019. His works have been nominated for Best Paper Awards at prestigious conferences such as DATE and NOCS. He is a member of the IEEE.

Janardhan Rao Doppa (Senior Member, IEEE) is the Huie-Rogers Endowed Chair Associate Professor at Washington State University. He received his PhD in computer science from Oregon State University. His research interests are at the intersection of machine learning and computing systems design. He won NSF CAREER award, Outstanding Paper Award from AAAI conference (2013), Best Paper Award from ACM Transactions on Design Automation of Electronic Systems (2021), IJCAI Early Career Award (2021), Best Paper Award from Embedded Systems Week Conference (2022), Outstanding Junior Faculty in Research Award (2020) and Reid-Miller Teaching Excellence Award (2018) from the College of Engineering, Washington State University.

Krishnendu Chakrabarty received the Ph.D. degrees from the University of Michigan, Ann Arbor, in 1995. He is now the John Cocke Distinguished Professor and Department Chair of Electrical and Computer Engineering (ECE) at Duke University. His current research projects include: design-for-testability of integrated circuits and systems (especially 3D integration and system-on-chip); AI accelerators; microfluidic biochips; hardware security; machine learning for healthcare; neuromorphic computing systems. He is a Fellow of ACM, IEEE, and AAAS, and a Golden Core Member of the IEEE Computer Society.

Partha Pratim Pande is a professor and holder of the Boeing Centennial Chair in computer engineering at the school of Electrical Engineering and Computer Science, Washington State University, Pullman, USA. He is currently the director of the school. His current research interests are novel interconnect architectures for manycore chips, on-chip wireless communication networks, heterogeneous architectures, and ML for EDA. Dr. Pande currently serves as the Editor-in-Chief (EIC) of IEEE Design and Test (D&T). He is on the editorial boards of IEEE Transactions on VLSI (TVLSI) and ACM Journal of Emerging Technologies in Computing Systems (JETC) and IEEE Embedded Systems letters. He was the technical program committee chair of IEEE/ACM Network-on-Chip Symposium 2015 and CASES (2019-2020). He also serves on the program committees of many reputed international conferences. He has won the NSF CAREER award in 2009. He is the winner of the Anjan Bose outstanding researcher award from the college of engineering, Washington State University in 2013. He is a fellow of IEEE