2025 IEEE International Symposium on High Performance Computer Architecture (HPCA) | 979-8-3315-0647-6/25/$31.00 ©2025 IEEE | DOI: 10.1109/HPCA61900.2025.00081

# I-DGNN: A Graph Dissimilarity-based Framework for Designing Scalable and Efficient DGNN Accelerators

Jiaqi Yang†, Hao Zheng‡, and Ahmed Louri†
George Washington University†, University of Central Florida‡
Yang_Jiaqi_Cute@gwu.edu, Hao.Zheng@ucf.edu, and louri@gwu.edu

***Abstract*—Dynamic Graph Neural Networks (DGNNs) have recently been used in numerous application domains, comprehending the intricate dynamics of time-evolving graph data. Despite their theoretical advancements, effectively implementing scalable DGNNs continues to be a formidable challenge due to the constantly evolving graph data and heterogeneous computation kernels. Recent efforts attempted to either exploit the graph data reuse to reduce memory access or eliminate the redundant computations between consecutive graph snapshots to scale the DGNN acceleration. These efforts are still falling short. In prior work, each graph snapshot, regardless of its size and connectivity, passes through the entire DGNN computation pipeline from layer to layer. Consequently, substantial intermediate data is generated throughout the DGNN computation, which leads to excessive off-chip memory access.**

**To address this crucial challenge, we argue that the computations between evolving graph snapshots should be decoupled from the DGNN execution pipeline. In this paper, we propose I-DGNN, a theoretical, architectural, and algorithmic framework with the aim of designing scalable and efficient accelerators for DGNN execution with improved performance and energy efficiency. On the theory side, the key idea is to identify essential computations between consecutive graph snapshots and encapsulate them as a separate kernel independent from the DGNN model. Specifically, the proposed one-pass DGNN computing model extracts the process of graph update as a chained matrix multiplication between evolving graphs through rigorous mathematical derivations. Consequently, consecutive snapshots utilize a one-pass computation kernel instead of passing through the entire DGNN execution pipeline, thereby eliminating the costly data movement of intermediate results across DGNN layers. On the architecture side, we propose a unified accelerator architecture that can be dynamically configured to support the computation characteristics of the proposed I-DGNN computing model with improved data and pipeline parallelism. On the algorithm side, we propose a new dataflow and mapping tailored for I-DGNN to further improve the data locality of inter-kernel data across the DGNN pipeline. Simulation results show that the proposed accelerator achieves 65.9%, 71.1%, and 58.8% reductions in execution time and 88.4%, 87.0%, and 85.9% improvements in energy efficiency on average across multiple DGNN datasets compared to state-of-the-art-accelerators [1]–[3].**

## I. Introduction

Dynamic Graph Neural Network (DGNN) models [4], [5] have recently emerged as a promising solution to understand the temporary and spatial relationship between entities in time-evolving graphs [6]–[9]. Theoretically, the success of DGNN models is driven by the combined use of graph neural networks (GNNs) and recurrent neural networks (RNNs), enabling the

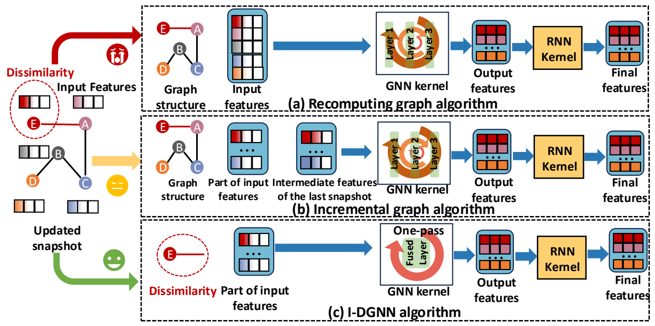


Fig. 1. The illustrative example of (a) Recomputing graph algorithm, (b) Incremental graph algorithm, and (c) I-DGNN algorithm.

learning capability of understanding both structural and temporal dynamics [10]. However, such a heterogeneous learning model, coupled with the evolving graph data, inevitably increases the computation complexity and memory overheads.

Recent efforts have proposed several accelerators to implement DGNNs [1], [2], [11]. However, the graph similarity between evolving graphs is not well exploited in these studies. For example, ReaDy [1] and DGNN-Booster [2] follow traditional GNN optimization techniques to exploit graph data reuse, thus reducing redundant computation and off-chip memory access. However, both ReaDy and DGNN-Booster process entire graph snapshots over time, despite the fact that graphs generally evolve gradually, with high similarity between consecutive snapshots [12]. This has led to redundant computations and off-chip memory access between graph snapshots.

To address the redundant computations between graph snapshots, recent work [3] proposed an incremental computing graph algorithm. This algorithm only processes the evolved components of the graph, such as changed vertices and edges, to understand the influence of temporal relationships between graph entities. Even though the proposed incremental computing approach can significantly reduce the redundant computations between snapshots, the updated graph components, regardless of the size and connectivity, still go through the entire DGNN execution pipeline from layer to layer. This comes with extra storage overhead and memory access due to the duplicated intermediate data. For instance, the intermediate data from consecutive snapshots should be simultaneously retained to update the final output of GNN kernels at each layer. Given that the size of the intermediate data is proportional to the graph size and the depth of the

GNN model, this duplication would significantly increase off-chip memory access.

In this paper, we argue that incremental computing should be decoupled from the DGNN execution pipeline due to the costly intermediate data overheads. Our key idea is to utilize classic graph theory, adjacency matrix powers, to identify the essential computations between graph snapshots in multi-layer DGNN models. This enables the precise encapsulation of the receptive field of GNNs affected by the evolving graphs. As shown in Figure1, the identified computations are developed as a one-pass computation model, as opposed to a multi-layer DGNN pipeline. Consequently, the evolved graph components no longer go through the entire DGNN pipeline. This can ensure the mathematical equivalence for capturing temporal relationships between consecutive snapshots while eliminating costly intermediate data duplication. Specifically, this paper makes the following contributions:

- On the theory side, we utilize graph theory, adjacency matrix powers, to develop a one-pass computation model that can efficiently capture the dynamics between graph snapshots without the need to execute the entire DGNN pipeline. The proposed model transforms a multi-layer GNN model to a computation kernel only involving the adjacency matrix through rigorous mathematical derivations. This approach can significantly reduce the computational complexity of analyzing evolving graphs in DGNNs.
- On the architecture side, we propose a unified accelerator architecture that can efficiently accelerate the proposed one-pass execution of the GNN kernel. The proposed accelerator architecture can be dynamically configured to support various computation characteristics desired by both the GNN and RNN kernels. In addition, we propose a scheduling policy to efficiently allocate hardware resources between GNN and RNN kernels with optimized pipeline parallelism, thereby significantly reducing execution time and energy consumption.
- On the algorithm side, we propose a dataflow and mapping for the proposed I-DGNN to parallelize the proposed dissimilarity computation and optimize the data locality of inter-kernel data between GNN and RNN kernels, eliminating costly data movement between processing elements and memory modules.
- We conduct a detailed performance and energy evaluation through simulation and show that the proposed accelerator achieves 65.9%, 71.1%, and 58.8% reductions in execution time and 88.4%, 87.0%, and 85.9% improvements in energy efficiency on average across multiple DGNN datasets compared to ReaDy [1], DGNN-Booster [2], and RACE [3], respectively.

## II. Background

### A. Dynamic Graph Representation

In real-world applications [5], [10], [13]–[17], graphs evolve over time, with vertices and edges being frequently added

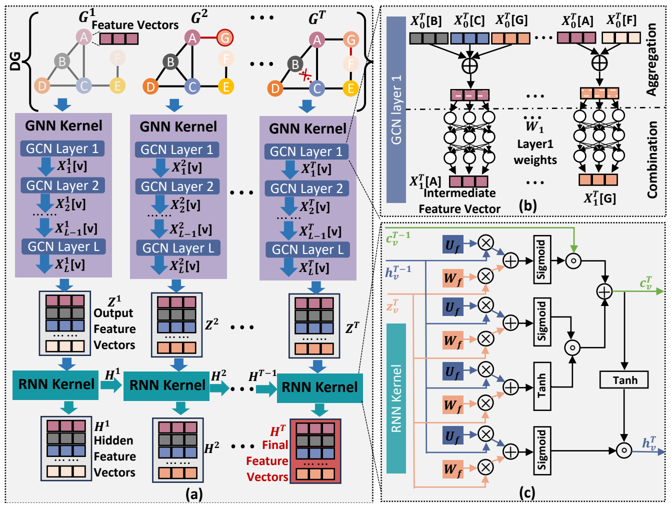


Fig. 2. (a)An example of a classic DGNN model, (b) a GCN computation kernel, and (c) an RNN computation kernel.

or removed. In general, there are two types of dynamic graphs [18] that have been used to record the temporal changes on the graphs: continuous-time dynamic graphs and discrete-time dynamic graphs. Continuous-time dynamic graphs are often described as a pair $<G, O>$, where $G$ represents the initial state of a static graph, and $O$ is a set of updates for vertices and edges. Discrete-time dynamic graphs are viewed as a sequence of discrete snapshots sampled at regular intervals illustrated in equation 1, where $G^t$ indicates a graph snapshot at the timestamp $t$. In this work, we design I-DGNN based on the discrete-time dynamic graph representation.

$$DG = \{G^1, G^2, ...., G^T\} \tag{1}$$

### B. Discrete-Time Dynamic Graph Neural Network

Discrete-time DGNN models [11] are designed for analyzing discrete-time dynamic graphs. These models can be classified into two groups: typical DGNN models and specialized DGNN models.

*1) Typical DGNN Models:* Typical DGNN models [11] integrate both conventional GNN and RNN kernels, as commonly observed in the DGNN algorithmic community [19]. For discrete-time dynamic graphs, the DGNN model sequentially processes each snapshot to identify the changes occurring in the graphs, as shown in Figure 2 (a). The GNN kernel takes a snapshot $G^t$ as the input, and it functions as a typical GNN model to learn the latent representation of graphs. The output feature vector $Z^t$ is then fed into the RNN kernel to generate a hidden state vector $H^t$, which contains both graph structure and temporal information. Consequently, the computations of DGNN can be formulated as equation 2.

$$\begin{aligned} Z^t &= GNN\{G^t\} \\ H^t &= RNN\{H^{t-1}, Z^t\} \end{aligned} \tag{2}$$

**The GNN kernel:** Figure 2 (b) shows a Graph Convolutional Network (GCN) layer. The GCN layer contains two computation phases, aggregation and combination. For the aggregation phase, each vertex $v$ collects the feature vectors from its connected vertices. During the combination phase,

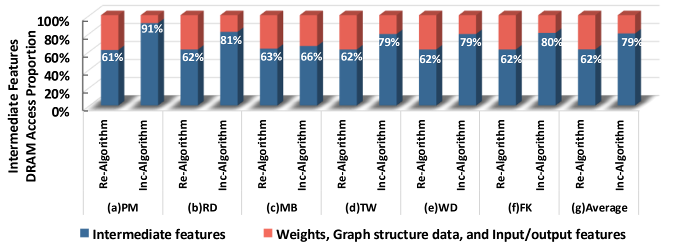


Fig. 3. DRAM access breakdown for intermediate, weight, graph, and feature vectors across different datasets with Recomputing Algorithm (Re-Algorithm) and Incremental Computing Algorithm (Inc-Algorithm).

the features are aggregated and multiplied by a weight matrix $W_l$.The computations of GCN can be defined as follows:

$$x_l^t[v] = Relu(A^t x_{l-1}^t[v] W_l), l \in (0, L] \tag{3}$$

where $A^t$ is the normalized Laplacian matrix over the adjacency matrix of the graph $G^t$, $x_{l-1}^t[v]$ is the initial feature vector of the vertex $v$ at $l_{th}$ GCN layer at the timestamp t, $x_l^t[v]$ is the updated feature vector of the vertex $v$ of $l_{th}$ GCN layer at the timestamp t, $L$ is the number of GCN layers, and $W_l$ is a weight matrix at $l_{th}$ GCN layer. It should be noted that the updated feature vector of the last GCN layer ($x_L^t[v]$) is defined as the output feature vector $z_v^t$. This will be used as the input for the RNN kernel to generate a hidden state vector $h_v^t$. While many GNN variants have been proposed such as GraphSAGE [20] and Graph Isomorphism Networks (GINs) [21], their key computations can be abstracted in the form of adjacency matrices.

**The RNN kernel:** As mentioned, the RNN kernel takes the output of GNN kernels as the input (i.e., $z_v^t$) to generate the hidden state whenever the snapshot arrives. This computation involves matrix multiplication, element-wise multiplication ($\circ$), addition, and activation functions (e.g., sigmoid, tanh). For example, Figure 2 (c) illustrates an example of RNN computations, where the most popular long short-term memory (LSTM) [22] is used. This work can also be efficiently applied to other RNN variants, such as gated recurrent units (GRUs). LSTM involves four input matrix multiplications by multiplying the input vector $z_v^t$ with four input weight matrices, $W_i$, $W_f$, $W_o$, and $W_c$, as shown in Equation 4.

$$\begin{gathered}
i_v^t = sigmoid(W_i z_v^t + U_i h_v^{t-1}) \\
f_v^t = sigmoid(W_f z_v^t + U_f h_v^{t-1}) \\
o_v^t = sigmoid(W_o z_v^t + U_o h_v^{t-1}) \\
c_v^t = f^t \circ c_v^{t-1} + i_v^t \circ tanh(W_c z_v^t + U_c h_v^{t-1}) \\
h_v^t = o_v^t \circ tanh(c_v^t)
\end{gathered} \tag{4}$$

Furthermore, the LSTM model includes four matrix multiplications by multiplying the hidden vector $h_v^{t-1}$ with four hidden weight matrices, $U_i$, $U_f$, $U_o$, and $U_c$, respectively. These eight matrix multiplications eventually produce the input gate $i^t$, forget gate $f^t$, output gate $o^t$, and cell state feature $c^t$ of vertex $v$.

*2) Specialized DGNN Models:* Several specialized DGNN models [23], [24] have been proposed to address specific applications. For example, structural-specific DGNN models like HDGNN [23] are designed for managing heterogeneous graphs, utilizing non-GNN methods for processing graph structural data while continuing to rely on RNNs for temporal time-series data processing. Similarly, temporal-specific DGNN models, such as STGNPP [25], are tailored for predictive tasks. STGNPP adopts GNNs for processing graph structural data and employs non-RNN methods, such as transformers and neural process priors (NPP), for handling temporal data, enabling effective traffic congestion time prediction. Additionally, some DGNN models, such as FiFrauD [26], handle dynamic graphs without relying on either GNN or RNN architectures. Instead, they use unsupervised and scalable approaches to detect suspicious traders and behavioral patterns. These specialized DGNN models are less commonly adopted because GNN or RNN models are replaced by application-dependent heuristics or unsupervised learning. For instance, the k-truss decomposition used in HDGNN is highly specialized for capturing multi-scale topological structures in heterogeneous graphs, limiting its generalizability to other types of graph data. Similarly, FiFrauD's unsupervised approach is tailored to identifying suspicious trading behaviors, making it less applicable to broader graph-based tasks. These application-specific techniques have limited applicability to various domains. Consequently, the focus of this paper remains on typical DGNN models.

## III. Motivation

### A. Pitfalls of Existing DGNN Acceleration Frameworks

Recent efforts [1], [2] have attempted to address the computation and data reuse concerns for DGNN execution. In general, prior work aims to eliminate redundant graph computations between consecutive snapshots or exploit the data reuse of graph datasets for reducing off-chip memory access. Nevertheless, all the prior work processes each graph snapshot, regardless of its size, through the entire DGNN pipeline. This requires substantial memory access and data movement to retrieve a full set of intermediate data and weight matrices from off-chip memory. According to our preliminary study as shown Figure 3, a considerable amount of off-chip memory accesses, ranging from 62% to 79%, are caused by moving the intermediate data even though only processing a limited set of evolved edges and vertices over snapshots. Specifically, prior work can be classified into two categories to optimize DGNN, namely recomputing and incremental computing approaches.

*1) Recomputing approach:* The recomputing approach [1], [2] is straightforward, in which each snapshot is processed individually through the entire DGNN model as shown in Figure 4 (a). The optimization strategy is very similar to traditional GNN acceleration which exploits the data reuse of graph datasets. For example, a set of vertices with high connectivity is grouped to reduce memory access. Despite significant performance improvement, a collection of vertices is computed repeatedly, as graph snapshots evolve slightly over time. Consequently, a significant amount of computations and memory access could be eliminated when processing consecutive snapshots.

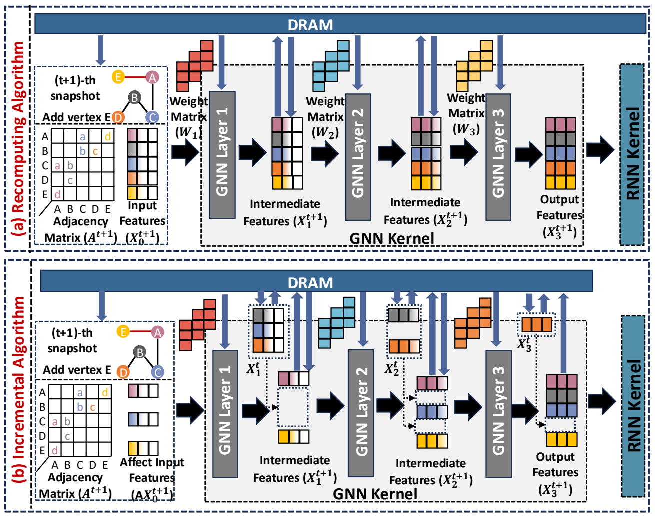


Fig. 4. (a) An example of recomputing algorithm execution. (b) an example of incremental algorithm execution.

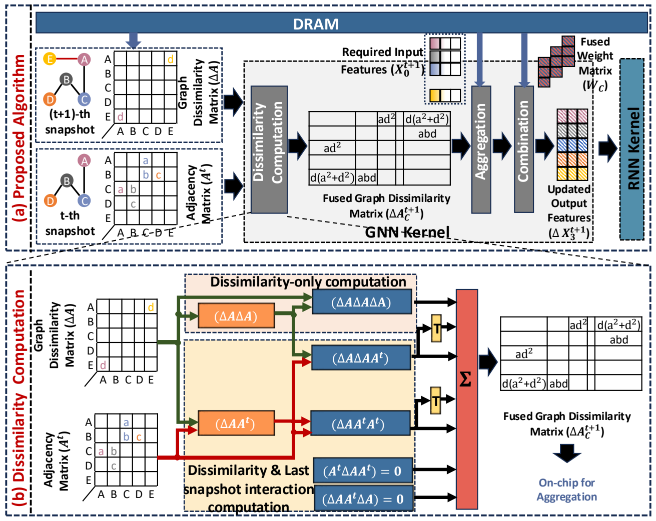


Fig. 5. (a) An example of the proposed I-DGNN execution algorithm, and (b) an example of the proposed one-pass computation kernel for graph update.

*2) Incremental Computing approach:* To address the redundant computations between snapshots, recent efforts [3] have proposed an incremental computing approach to accelerate DGNN, where the overlapped graph components between snapshots, such as vertices and edges, are removed for subsequent computations. However, such a computing paradigm, while only processing a limited amount of evolved graph data, goes through the entire DGNN execution. This still requires a full set of weight matrices and intermediate data, which leads to a significant amount of off-chip memory access. For example, two consecutive snapshots need to produce two sets of intermediate data that are eventually aggregated to update the feature vectors. Given the large dimension size of feature vectors, temporally duplicating intermediate data jeopardizes the effectual utilization of on-chip memory.

For simplicity, we use an example to illustrate the problem. In the first time epoch, all the input feature vectors are loaded and multiplied by the weight matrix, which produces the intermediate feature vectors for each vertex. Through a multi-layer of GNN, the intermediate features of the last layer of GNN, which we call output feature vectors in this paper, are forwarded to the RNN. All the intermediate data and the output features are stored for the following snapshots. In the next time epoch shown in Figure 4 (b), when a new snapshot arrives, only a part of the input features is computed through the entire execution pipeline. The new computations produce a new set of intermediate data which is augmented with the reusable intermediate data produced by the prior snapshot. As such, the intermediate data of both snapshots should be temporally stored on the chip. It should be noted that the size of intermediate data of the subsequent snapshot increases when GCN layers increase. This problem has not been well addressed by existing GNN and DGNN accelerators.

## IV. Proposed I-DGNN Algorithm

The objective of the proposed I-DGNN algorithm is to investigate a one-pass computation approach for processing evolved graphs, instead of going through the entire DGNN computing pipeline. This could serve as a basis to reduce the memory access for retrieving massive intermediate data. To achieve this, we postulate that a mathematical derivation of DGNN computations between snapshots is needed to understand the essential data and computation required for incremental computing. As shown in Figure 5 (a), the proposed I-DGNN algorithm consists of three components, including a dissimilarity-based computation kernel to capture all the vertices that require feature update (i.e., aggregation and combination) by the evolving graphs in multi-layer GNNs, a fused aggregation phase, and a fused combination phase. Consequently, multi-layer GNN architecture will be simplified as a dissimilarity computation kernel for subsequent snapshots as shown in Figure 5 (b). The dissimilarity computation only involves the evolved graph structure and the past snapshot, which will be discussed in what follows.

### A. DGNN Layer Fusion

Unlike traditional graph analytics, evolving graph components will impact other graph components in GNN models at both aggregation and combination phases due to the increased receptive field (e.g., the number of layers). Layer fusion has been well exploited in deep learning and GNNs to optimize data locality. Despite sharing a similar approach, we aim to leverage layer fusion to extract the essential computations incurred by the evolved graph components. For simplicity, following traditional layer fusion principles, we use an $L$-layer DGNN model to illustrate the key idea.

The computation of the $l_{th}$ GCN layer for the $t_{th}$ snapshot can be defined as equation 5, where $A^t$ is the adjacency matrix, $X^t_{l-1}$ is the feature vectors from the prior layer, and $W_l$ is the weight matrix.

$$X^t_l[v] = Relu(A^t X^t_{l-1}[v] W_l), l \in (0, L] \tag{5}$$

The final output of the GNN model for the $t_{th}$ snapshot can be represented by:

$$X^t_L[v] = Relu((A^t)^L X^t_0[v] \prod_{l=1}^{L} W_l) \tag{6}$$

We define the adjacency matrix of the fused GNN layer as equation 7, where the consecutive power of the adjacency matrix represents the adjacency matrix for different GNN layers.

$$A_C^t = (A^t)^L \tag{7}$$

Similarly, the weight matrix of the fused GNN layer can be represented by:

$$W_C = \prod_{l=1}^{L} W_l \tag{8}$$

Consequently, we can use equation 9 to represent the computation of fused GNN layers for a certain snapshot.

$$X_C^t[v] = Relu(A_C^t X_0^t[v] W_C) \tag{9}$$

As the weight matrices of the GNN layers are consistent across the continuous snapshots, we only compute the fused weight matrix ($W_C$) in the initial snapshot and reuse it in the following snapshots. On the other hand, the adjacency matrix of the GNN layers is dynamic between the continuous snapshots, so it should be recomputed at runtime which will be discussed in what follows.

## *B. Dissimilarity-based execution of DGNN Kernel*

After fusing DGNN layers, we need to understand how the evolved graph components affect the computations. The dissimilarity-based computation kernel is built upon a classic graph theory - adjacency matrix powers [27]. The power of adjacency matrices represents the distance between vertices in two matrices. This theory is used to measure the distance of feature propagation in multi-layer GNNs. For example, we use the equation 10 to understand the computation difference between two consecutive snapshots, $t$ and $t+1$. The difference between the output features of the $t_{th}$ and $(t+1)_{th}$ snapshot is defined by:

$$\begin{aligned}\Delta X_C^{t+1}[v] &= X_C^{t+1}[v] - X_C^t[v] \\ &= Relu\{(A_C^{t+1} X_0^{t+1}[v] - A_C^t X_0^t[v]) W_C\} \\ &= \underbrace{Relu\{\underbrace{(\Delta A_C^{t+1} X_0^{t+1}[v] + A_C^t \Delta X_0^{t+1}[v])}_{\text{Aggregation}} W_C\}}_{\text{Combination}}\end{aligned} \tag{10}$$

$\Delta X_0^{t+1}[v]$ is the updated input features of $(t+1)_{th}$ snapshot, defined by:

$$\Delta X_0^{t+1}[v] = X_0^{t+1}[v] - X_0^t[v] \tag{11}$$

$\Delta A_C^{t+1}$ is the fused graph dissimilarity matrix of the fused GNN layer between the $t_{th}$ and $(t+1)_{th}$ snapshot, which can be further defined by:

$$\begin{aligned}\Delta A_C^{t+1} &= A_C^{t+1} - A_C^t \\ &= (A^{t+1})^L - (A^t)^L\end{aligned} \tag{12}$$

Based on the principle of adjacency matrix powers, $(A^{t+1})^L$ and $(A^t)^L$ represent the receptive field of GNNs, where the exponent (i.e., the number of GNN layers) decides the longest distance of feature aggregation of each vertex in multi-layer GNN models.Furthermore, $\Delta A_C^{t+1}$ can be represented only related to the $t_{th}$ snapshot and the updated adjacency matrix $\Delta A$ as equation 13. The expansion represents the feature aggregation distance at each layer, in which all the vertices along the receptive field require feature update.

$$\begin{aligned}\Delta A_C^{t+1} &= (A^t + \Delta A)^L - (A^t)^L \\ &= \sum_{i=0}^{L-1} (A^t)^i \Delta A (A^t + \Delta A)^{L-1-i}\end{aligned} \tag{13}$$

Nevertheless, from the above derivation, we can also conclude that the feature aggregation phase is determined by four factors, the input features ($X_0^{t+1}[v]$), the updated input features ($\Delta X_0^{t+1}[v]$), the fused graph dissimilarity matrix ($\Delta A_C^{t+1}$), and the fused adjacency matrix of the previous snapshot ($A_C^t$), which already be calculated in the execution of the previous $t_{th}$ snapshot. $\Delta A$ is the matrix representation to record the evolved graph structure components, such as added or removed edges and vertices, which we call the graph dissimilarity matrix in this paper. $\Delta X_0^{t+1}[v]$ is the matrix representation to record the evolved input features, such as changed, added or removed features, which we call the updated input feature matrix in this paper. The combination requires one additional input - fused weight matrix ($W_C$). Please note that the weight matrix is relatively small in GNNs as compared to traditional deep neural networks. We use an example to illustrate the proposed algorithm, shown in Figure 5 (a). It is becoming evident that the computations for subsequent snapshots could be simplified as an equation that only contains the adjacency matrix, graph dissimilarity matrix, the involved input features of the latest snapshot, and a fused weight matrix. As such, it is not necessary to process each graph dissimilarity matrix through the entire execution pipeline.

## *C. Optimization for Dissimilarity Computations*

We have transformed the incremental computation into a simplified computation that only requires adjacency matrices and the involved input features. However, calculating the consecutive power of the adjacency matrix is still time-consuming. As such, we aim to leverage the symmetric characteristics of the adjacency matrix to use matrix transpose to replace a set of matrix multiplications. To better illustrate the idea, we take a three-layer GNN kernel as an example.

We expand the computation for the fused graph dissimilarity matrix as shown in equation 14. The third power of an adjacency matrix could be transformed into a set of chained matrix multiplications. Multiple chained matrix multiplications share common matrices, such as $A^t$ and $\Delta A$. The only difference is the sequence of those chained matrix multiplications. Given $A^t$ and $\Delta A$ are symmetric matrices, the results of chained matrix multiplications could be obtained by transposing those multiplications with an opposite order. For example, matrix transport of $\Delta A A^t A^t$ equals to $A^t A^t \Delta A$, and matrix transport of $\Delta A \Delta A A^t$ equals to $A^t \Delta A \Delta A$.

$$\begin{aligned}\Delta A_C^{t+1} &= (A^t + \Delta A)^3 - (A^t)^3 \\ &= \sum_{i=0}^{2} (A^t)^i \Delta A (A^t + \Delta A)^{2-i} \\ &= \Delta A A^t A^t + \Delta A A^t \Delta A + \Delta A \Delta A A^t \\ &+ \Delta A \Delta A \Delta A + A^t \Delta A A^t + A^t \Delta A \Delta A \\ &+ A^t A^t \Delta A\end{aligned} \tag{14}$$

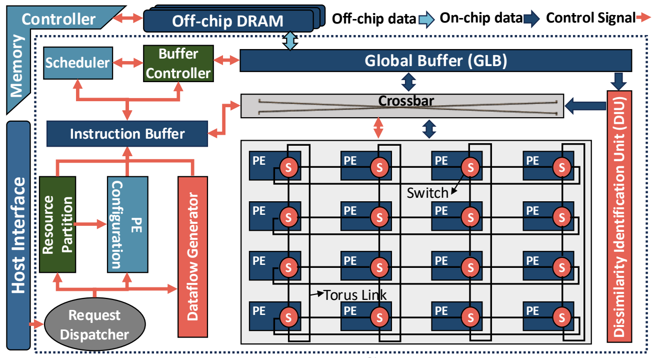


Fig. 6. Overview of the proposed **I-DGNN** accelerator architecture.

As such, the computation for $\Delta A_C^{t+1}[v]$ can be optimized as:

$$\begin{aligned}\Delta A_C^{t+1} =& A^t(\Delta AA^t) + \Delta AA^t\Delta A \\ &+ (\Delta A\Delta AA^t)(1 + Transpose) \\ &+ (\Delta AA^tA^t)(1 + Transpose) \\ &+ \Delta A\Delta A\Delta A\end{aligned} \tag{15}$$

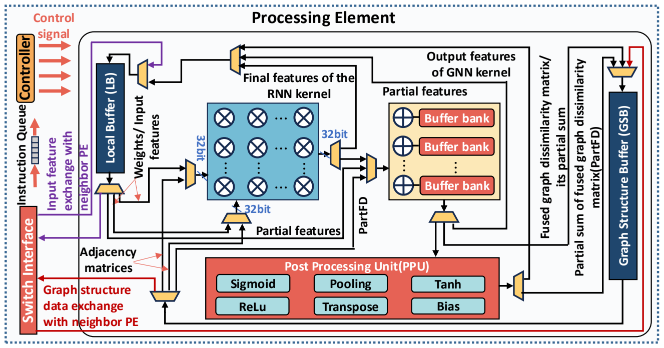
Eventually, as shown in Figure 5 (b), the computations of graph updates between snapshots are converted to a set of chained matrix multiplications and their transpose operations.

## V. I-DGNN Accelerator Architecture

While a significant number of accelerators have been proposed for GNNs and DGNNs, they mainly target the reuse of graph data. In this paper, our aim is to optimize the reuse of intermediate data for DGNN models. To achieve this goal, we have transformed the traditional DGNN computation into a set of chained matrix multiplications that are related to the graph adjacency matrix. The next challenge is to design an efficient architecture for such a computing paradigm while also supporting traditional DGNN executions. To address the mentioned challenge, the proposed I-DGNN accelerator features four unique components: a Dissimilarity Identification Unit (DIU), a reconfigurable PE architecture, a fine-grained pipeline scheduler, and a data locality-aware dataflow. The DIU generates the graph dissimilarity matrix and updated input feature vectors between consecutive snapshots. The reconfigurable PE design supports the distinct computation and communication characteristics of both GNN and RNN kernels. Moreover, I-DGNN's fine-grained scheduler can efficiently handle the pipeline parallelism among various GNN and RNN kernels with resource allocation. Lastly, the proposed data dataflow and mapping can reduce on-chip communication overheads by optimizing the locality of intermediate data between kernels.

### A. I-DGNN Accelerator Overview

As shown in Fig. 6, the proposed I-DGNN accelerator consists of a request dispatcher, an instruction buffer, a fine-grained pipeline scheduler, a dataflow generator, a PE configuration unit, a buffer controller, a global buffer (GLB), a Dissimilarity Identification Unit, and a PE array interconnected by a torus topology. The accelerator sends instructions to the fine-grained pipeline scheduler, which determines the resource allocation, dataflow, and mapping. The PE array is connected through a torus topology. The detailed microarchitecture of the PE will be described in the following sections.



Processing Element
Control signal
Controller
Instruction Queue
Input feature exchange with neighbor PE
Local Buffer (LB)
Weights/ Input features
Final features of the RNN kernel
Output features of GNN kernel
Partial features
Buffer bank
32bit
Partial features
PartFD
Fused graph dissimilarity matrix/ its partial sum
Partial sum of fused graph dissimilarity matrix(PartFD)
Graph Structure Buffer (GSB)
Switch Interface
Adjacency matrices
Graph structure data exchange with neighbor PE
Post Processing Unit(PPU)
Sigmoid
Pooling
Tanh
ReLu
Transpose
Bias

Fig. 7. Microarchitecture of the proposed **I-DGNN processing element**.

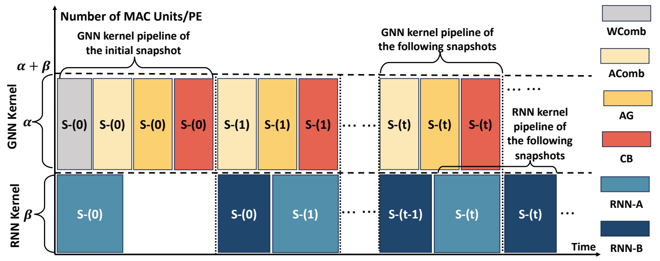
### B. Proposed PE Microarchitecture

The goal of the proposed PE is to support the distinct computation characteristics of GNN, RNN, and the one-pass computation kernel in one combined architecture. Such design can retain the inter-kernel data within PE architecture, reducing the data movement between PEs and memory modules.

Specifically, each PE consists of a switch interface, a controller, a Graph Structure Buffer (GSB), a Local Buffer (LB), a Multiplier Array (MA), an Adder Array (AA), and a Post Processing Unit (PPU), as shown in Fig. 7. Considering that the input matrices consist of both sparse and dense matrices stored in different formats, each PE incorporates separate input sparse (GSB) and dense buffers (LB) for matrices with distinct sparsity ratios. The buffers, including the GSB and LB, receive data through the switch interface. The PE controller generates control signals to guide PE execution, orchestrate resource partitioning, and configure MUX-DeMUXes to manage the data path. The GSB stores the adjacency matrix of the previous snapshot and the graph dissimilarity matrix in Compressed Sparse Row (CSR) format. The LB stores the weight matrices for the GNN and RNN kernels and the involved input features. The LB also holds the reused data, including RNN kernel output features of the previous snapshot, the product of the RNN weight matrices with the GNN kernel output features of the previous snapshot, and the cell state feature of the previous snapshot. After the RNN execution of the latest snapshot, all the reuse data stored in the LB will be updated. The PPU can perform nonlinear operations including ReLU, Sigmoid, Tanh, pooling, bias, and matrix transpose.

*1) Proposed Reconfigurable Datapath:* The proposed PE architecture can be configured to support several computing models desired by the GNN and RNN kernels.

**One-shot Computation:** The adjacency matrix of the previous snapshot and the graph dissimilarity matrix of consecutive snapshots are sent from the GSB to the MA. The AA then generates part of the fused graph dissimilarity matrix based on the MA's output. Each part of the fused graph dissimilarity matrix is accumulated at AA to produce the complete fused graph dissimilarity matrix, which is stored in the GSB. If needed, nonlinear matrix transformations are performed in the PPU before the partial accumulation. The PPU processes the nonlinear matrix transformations by exchanging the row and column index of the matrices.


Number of MAC Units/PE
GNN kernel pipeline of the initial snapshot
GNN kernel pipeline of the following snapshots
RNN kernel pipeline of the following snapshots
$\alpha+\beta$
$\alpha$
$\beta$
GNN Kernel
RNN Kernel
S-(0)
S-(1)
S-(t)
S-(t-1)
Time
WComb
AComb
AG
CB
RNN-A
RNN-B
Fig. 8. The proposed pipeline workflow, where $s_t$ means the relevant computations related to $t_{th}$ snapshot.

**GNN Aggregation:** The fused graph dissimilarity matrix and involved input features are sent from the GSB and LB to the MA, respectively. The AA then generates the aggregated partial features based on the MA's output. These aggregated partial features replace the original input features of the consecutive snapshots stored in the LB.

**GNN Combination:** The aggregated partial features and the combined weight matrix are sent from the LB to MA. The AA then generates the GNN updated output features based on the MA's output. If needed, nonlinear activation is performed in the PPU before writing the updated output features to LB. The updated output features of the consecutive snapshot replace the previously stored aggregated partial features in the LB.

**RNN:** The RNN output features of the previous snapshot, the GNN updated output features, the weight matrices of the RNN kernel, and the reused data are sent to MA from LB, respectively. The AA generates the output features of the RNN and writes it tothem to LB. If needed, nonlinear operations are performed in PPU before the output feature write back to LB.

### C. Proposed Analytical Model for Pipelining DGNN Kernels

Since we have simplified the computations for the GNN kernel, it is critical to balance the workload between the proposed GNN and RNN kernels. This can directly affect the hardware resources and memory access for storing and loading intermediate data between two models. If recalled, our proposed GNN execution consists of multiple phases: weight matrix fusion (WComb), adjacency matrix fusion (AComb), Aggregation (AG), and Combination. In addition, the RNN could be decomposed into two phases, which we call RNN-A and RNN-B. RNN-A refers to the execution phase of RNN kernels independent of GNN, so it can be executed in parallel. On the other hand, RNN-B depends on the output of the GNN kernel. For example, the computation of RNN-A phase can be described as equation 16.

$$A_{\alpha}^{t}[v] = U_{\alpha}h_{v}^{t-1}, \alpha \in [i, f, o, c] \tag{16}$$

On the other hand, the computation of the RNN-B phase can be described below:

$$\begin{aligned} i_v^t &= sigmoid(W_i z_v^t + A_i^t[v]) \\ f_v^t &= sigmoid(W_f z_v^t + A_f^t[v]) \\ o_v^t &= sigmoid(W_o z_v^t + A_o^t[v]) \\ c_v^t &= f^t \circ c_v^{t-1} + i_v^t \circ tanh(W_c z_v^t + A_c^t[v]) \\ h_v^t &= o_v^t \circ tanh(c_v^t) \end{aligned} \tag{17}$$

Our goal is to allocate adequate resources to each execution phase towards better pipeline parallelism as shown in Figure 8. Given this, we propose an analytical model to capture such dynamics. In general, the objective function is to equalize the execution time of both GNN and RNN which can be defined as $\min\left(\mid CompT_G^t - CompT_{RA}^{t-1} - CompT_{RB}^t \mid\right)$, where $CompT_G^t$ is the computation time of GNN kernel at time $t$, $CompT_{RA}^{t-1}$ is the computation time of RNN-A at time $t-1$, and $CompT_{RB}^t$ is the computation time of RNN-B at time $t$.

The computation time of the GNN kernel is further determined by multiple factors, such as the computation time of fusing adjacency matrix ($CompT_{AComb}^t$), the computation time of aggregation phase ($CompT_{AG}^t$), and the computation time of combination phase ($CompT_{CB}^t$).

The computation time of fusing adjacency matrix ($CompT_{AComb}^t$) is related to the sparsity of the adjacency matrix for last snapshot ($p^{t-1}$), the sparsity of the graph dissimilarity matrix ($s^t$), the number of vertex ($V^t$), the number of the PE ($M$), and the number of MAC units assigned to GNN ($\alpha$) per PE. The computation time of fusing adjacency matrix in a three-layer GNN kernel can be formulated as equation 18.

$$CompT_{AComb}^t = \{s^t(s^t + p^{t-1})(1 + 2p^{t-1})(V^t)^3\}/(M\alpha) \tag{18}$$

Similarly, the computation time of the aggregation phase is subject to the sparsity of the adjacency matrix for last snapshot ($p^{t-1}$), the sparsity of the graph dissimilarity matrix ($s^t$), the number of vertex ($V^t$), the number of the features for each vertex ($K^t$), the number of the PE ($M$), and the number of GNN MAC units per PE ($\alpha$). The computation time of the aggregation phase in a three-layer GNN kernel can be formulated as equation 19.

$$CompT_{AG}^t = \frac{\{3(s^t)^2 p^{t-1} + 3s^t(p^{t-1})^2 + (s^t)^3\}(V^t)^2 K^t}{M\alpha} \tag{19}$$

The computation time for the combination phase depends on the number of vertex ($V^t$), the number of the features for each vertex ($K^t$), the width of the GNN weight matrix ($C$), the number of the PE ($M$), and the number of GNN MAC units per PE ($\alpha$). The computation time of the combination phase in a three-layer GNN kernel can be formulated as equation 20.

$$CompT_{CB}^t = (V^t)K^tC/(M\alpha) \tag{20}$$

The computation time of the RNN-B phase is defined by:

$$CompT_{RB}^t = \{V^tR(4C + 3)\}/(M\beta) \tag{21}$$

$V^t$ represents the vertex number in the latest snapshot, $R$ represents the size of the weight and hidden matrices in RNN, and $\beta$ represents the number of RNN MAC units per PE.

Similarly, the computation time of the RNN-A phase ($CompT_{RA}^{t-1}$) is defined by:

$$CompT_{RA}^{t-1} = 4V^{t-1}CR/(M\beta) \tag{22}$$

### D. Proposed Dataflow and Mapping

In deep learning acceleration, both dataflow and mapping choices can affect the performance by exploiting the temporal and spatial data locality [28]–[30]. Given the limited dimension size of weight matrices in GNNs, it becomes difficult to

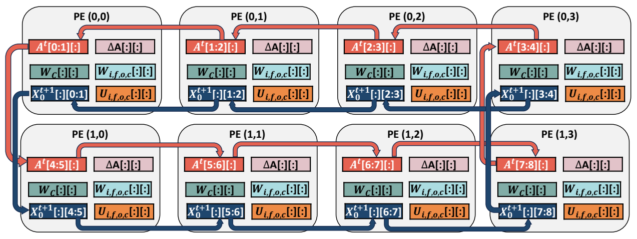


Fig. 9. An example of the proposed dataflow and mapping in a $2 \times 4$ PE array, where $\Delta A[:][:]$ means the graph dissimilarity matrix between consecutive snapshots, $W_C[:][:]$ means the fused GNN weight matrix, $A^t[\gamma][v]$ means the adjacency matrix, $X_0^{t+1}[f][v]$ means the $(f+1)_{th}$ input feature of vertex $v$, and $W_{i,f,o,c}[:][:]$ and $U_{i,f,o,c}[:][:]$ are weight matrices of the RNN kernel.

significantly improve data reuse by following traditional loop transformation. Recent efforts [31]–[33] have evidenced that graph reuse data, e.g., commonly-shared vertex, is the primary source to reduce off-chip memory access. Departing from prior work, we aim to propose a dataflow and mapping strategy that can efficiently support the proposed one-pass computation tailored for DGNNs. There are two major challenges, namely (1) parallelism strategy selection for computing fused graph dissimilarity matrix ($\Delta A_C^t$), and (2) dataflow and mapping for optimizing the data locality of intermediate data between GNN and RNN kernels.

**Dataflow and Mapping for GNN kernel:** As compared to existing dataflows that are optimized for input and weight reuse [32]–[34], the proposed dataflow is used to parallelize the dissimilarity computation and increase the intermediate data reuse between GNN and RNN kernels. Specifically, the weight matrix is relatively small in DGNN, and the graph changes slightly over snapshots. Consequently, both weight matrix and graph dissimilarity matrix could be duplicated at each PE. The major bottleneck is the adjacency matrix and feature vectors whose size is relatively larger. As such, we propose to distribute both adjacency matrix and feature vectors among PEs. To improve buffer utilization, we enabled inter-PE communications for sharing the distributed adjacency matrix and feature vectors. As shown in Figure 9, we distribute the adjacency matrix ($A^{t-1}$) and feature vectors ($X_0^t$) among PEs. The adjacency matrix is multiplied with graph dissimilarity to produce the fused graph dissimilarity matrix. The fused graph dissimilarity matrix is multiplied by feature vectors to produce the output data.

For example, at time step ❶, each PE is assigned a part of the adjacency matrix and feature vectors. For example, $A^{t-1}[0:1][:]$ and $X_0^t[:][0:1]$ are assigned to PE [0,0]. When all PEs complete the computation, both the adjacency matrix and feature vectors move from one to another. At time step ❷, after the first inter-PE movement, PE [0,0] receives $A^{t-1}[1:2][:]$ and $X_0^t[:][1:2]$ from PE [0,1], and so on and so forth. Similarly, in the following time steps, each partition of data is passed through all the PEs. Each PE generates part of the updated output features ($\Delta X_L^{t+1}$) and taking it as the input of the RNN kernel, when the the GNN execution is finished. For example, PE [0,0] generates $\Delta X_L^{t+1}[0:1][:]$, representing the $1_{th}$ and $2_{th}$ updated features of all the vertices.

**Dataflow and Mapping for RNN kernel:** We spatially duplicate the weight matrices of RNN kernels at each PE given their small size. Specifically, at each PE, the RNN-A phase takes the output hidden feature of the RNN-B phase at the last snapshot ($h_v^{t-1}$) as input. Each PE generates the output of the RNN-A phase by multiplying the hidden feature with four local RNN weight matrices, $U_i$, $U_f$, $U_o$, and $U_c$, respectively. Additionally, the RNN-B phase takes the output of GNN kernels ($\Delta X_L^{t+1}$) as input to generate the hidden features whenever a snapshot arrives. For each PE, the inputs ($\Delta X_L^{t+1}$) are multiplied with the other four local RNN weight matrices, $W_i$, $W_f$, $W_o$, and $W_c$. As such, the RNN kernel can consume the output features produced by the GNN kernel at each PE without incurring additional cross-PE data transfer.

TABLE I
DETAILS OF DATASETS USED FOR EVALUATION

| Datasets | Vertices | Edges | Features | Description |
|---|---|---|---|---|
| PubMed (PM) | 1,917 | 88,648 | 500 | Citation Graph |
| Reddit (RD) | 55,863 | 858,490 | 602 | Social Graph |
| Mobile (MB) | 340,751 | 2200203 | 362 | Citation Graph |
| Twitter (TW) | 8,861 | 119,872 | 768 | Sharing Graph |
| Wikipedia (WD) | 9,227 | 157,474 | 172 | Citation Graph |
| Flicker (FK) | 2,302,925 | 33,140,017 | 800 | Social Graph |

## VI. EVALUATION

### A. Evaluation Setup

**Accelerator Simulator :** We built a cycle-accurate simulator to measure the performance of the I-DGNN accelerator. In order to obtain execution time results, the simulator monitors the number of arithmetic operations and the number of accesses across the memory hierarchy. The I-DGNN accurately captures the dynamics of pipeline scheduling, dataflow, and distinct system configurations. The number of arithmetic operations is used to calculate the computation time, whereas the number of accesses of each memory hierarchy is used to calculate the communication time. The off-chip communication time is obtained from the DRAMSim2 simulator [35]. Ideally, the execution time of each system component, such as off-chip communication and processing time, can be executed in parallel and overlapped. In our simulation, we consider the detailed execution time of each component and their potential overlapping in the pipeline. The simulator counts the amount of on/off-chip communications and computations, which is used to estimate the related energy consumption according to the analytical model proposed in [36]. Additionally, to accurately estimate the area consumption, we used the Synopsys Design Compiler with the TSMC 45 nm standard library to synthesize and generate the waveform activity file to capture the dynamic switching activity of the logic gates. We use Cacti 6.0 [37] to estimate the area, power, and access latency of all types of on-chip buffers. Specifically, we analyzed most of the accelerator components, including PEs, the controller, on-chip buffers, NoCs, and other hardware components.

**Accelerator Modeling :** We implemented the I-DGNN including $32 \times 32$ PEs interconnected by a torus topology. Each PE consists of a dense local buffer, a sparse graph structure data buffer, a router interface, a post-processing

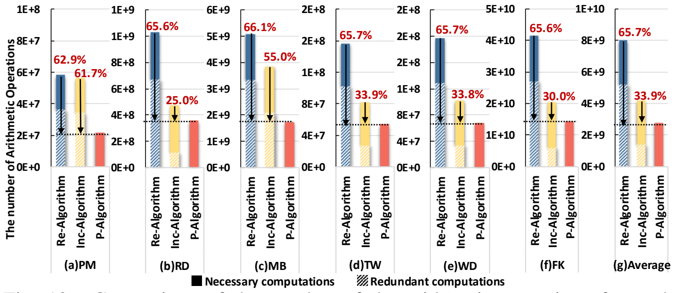


Fig. 10. Comparison of the number of the arithmetic operations for each dataset with Recomputing Algorithm (Re-Algorithm), Incremental Algorithm (Inc-Algorithm), and the proposed algorithm (P-Algorithm).

unit(PPU), a buffer controller, multipliers, adders, and required logic. Each PE includes a 4 × 4 multiplier array connected to an accumulation unit with 4 × 4 adders. The on-chip frequency of the I-DGNN is 700MHz. The global buffer capacity of the I-DGNN is 64MB. The sparse graph structure data buffer capacity of each PE is 128KB. The dense local buffer capacity of each PE is 100KB.

**Baselines :** We compare the I-DGNN with three GNN accelerators (ReaDy [1], DGNN-Booster [2], and RACE [3]). The baseline accelerators are scaled to be equipped with the same number of multipliers and off-chip/on-chip bandwidth as the I-DGNN. ReaDy uses a hierarchical architecture consisting of a mesh-based PE array for both the GNN kernel and RNN kernel and its computation resources are partitioned according to the workloads of the kernels. RACE uses an engine-based architecture consisting of a GNN engine for the GNN kernel and an RNN engine for the RNN kernel. The PEs are connected by a crossbar in each engine. Each PE contains a multiplier, an adder, and six MUXes. The computation resources are divided into two groups with the same number of PEs for the two engines according to the original configuration. We also resized the baseline accelerators to be equipped with the same on-chip storage capacity and frequency.

**Datasets and Benchmarks:** Table I illustrates six dynamic graphs used for evaluation in this paper [16], [38]–[42]. We consider one typical DGCN model [16], including GCN [43] and LSTM models [44]. The 32-bit floating-point representation is used in the evaluation, which proves to be sufficient for maintaining inference accuracy [45]–[47].

## B. Arithmetic Operation Analysis

Fig. 10 shows the breakdown of arithmetic operations for different algorithms. In order to examine the effectiveness of the proposed algorithm, we use breakdown to include both redundant computations (caused by both the data and model) and essential computations (for ensuring the correctness of graph update). We can observe that the proposed algorithm reduces the amount of redundant arithmetic operations by 65.7% and 33.9% for DGNN execution using a classic DGCN model [16], on average across multiple datasets, when compared to the baselines. The proposed algorithm outperforms previous approaches for several reasons. Firstly, similar to traditional incremental computing [1], the proposed accelerator exploits the graph dissimilarity between snapshots and only processes their evolved graph structures. In addition to prior work, the developed one-pass computing kernels can efficiently eliminate the need to recompute the entire GNN kernel, thus eliminating the computations at each GNN layer. In other words, the fused GNN model can avoid a significant amount of intermediate computation during the graph update stage. Given the proposed model, the output of GNN kernel is relatively smaller than prior work, as it only includes essential information related to a limited set of graph structures. As such, the RNN kernel also only processes a limited set of output features from GNN kernel, reducing the overall computation amount.

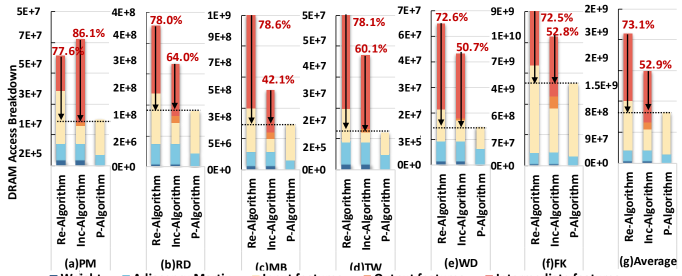


Fig. 11. Comparison of the DRAM access breakdown for each dataset with baseline and proposed algorithms.

## C. Off-chip DRAM Access Analysis

Fig. 11 illustrates the breakdown of the DRAM access volume for each dataset using three different algorithms. A lower value indicates a better performance. It is evident that the proposed algorithm consistently outperforms the baselines. The reduction in DRAM access varies across the datasets. On average, the proposed design achieves 73.1% and 52.9% reduction in DRAM access compared to the baselines. The DRAM access volume includes memory access for weights, adjacency matrix, input features, intermediate features, and output features.

Compared to the conventional execution approach, the proposed accelerator only preloads the weight matrices of the GNN kernel for the initial snapshot. This approach differs from the recomputing algorithm and the incremental algorithm, where the weight matrices need to be preloaded to the on-chip buffer for each snapshot. The recomputing algorithm accesses all the input features of the latest snapshot from the off-chip DRAM, writes back the intermediate features to the DRAM, and reads the intermediate features from the DRAM for the execution of the following GNN layers. The output features of the recomputing algorithm are retained on-chip for the RNN kernel execution. The incremental algorithm only fetches the affected input features from the DRAM, but it needs to write back the intermediate features and the output features of the latest snapshot and read the intermediate features or the output features from the DRAM for computation reuse. Our proposed execution approach only requires the updated adjacency matrix and the involved feature vectors related to the changes between the consecutive snapshots.

## D. Performance Analysis

Fig. 12 illustrates the execution time of the I-DGNN in comparison to previous works, measured in terms of the total

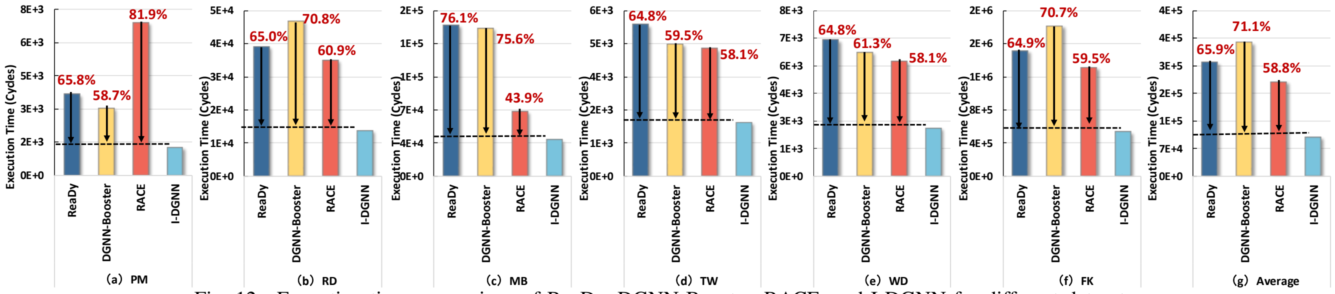


Fig. 12. Execution time comparison of ReaDy, DGNN-Booster, RACE, and I-DGNN for different datasets.

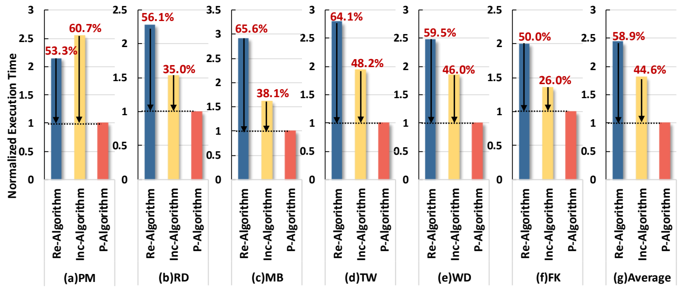


Fig. 13. Comparison of the normalized execution time for each dataset on same accelerator architecture with baseline and proposed algorithms.

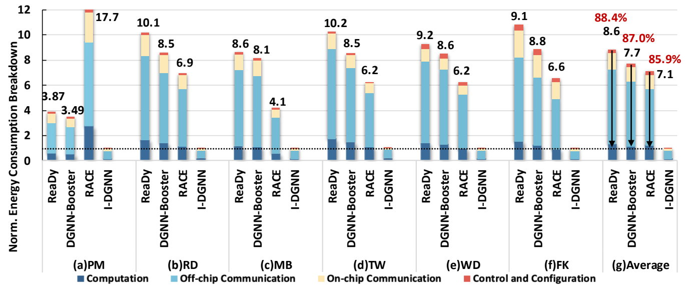


Fig. 14. Comparison of the Normalized Energy Consumption Breakdown for different datasets (Normalized to the energy consumption of the I-DGNN).

number of execution cycles. ReaDy and DGNN-Booster use the recomputing algorithm, whereas RACE uses the incremental computing algorithm. The I-DGNN achieves, on average, a 65.9%, 71.1%, and 58.8% reduction in execution time compared to baselines for multiple GNN datasets, respectively. RACE can reduce the DRAM access of the input features and the redundant computation by exploiting the intermediate features/output features of the previous snapshot. However, it introduces DRAM access for the intermediate features, which accounts for over 60% of the total DRAM access volume. The heterogeneous architecture of RACE also introduces hardware underutilization due to the imbalanced workload between GNN and RNN kernels. ReaDy and DGNN-booster can increase performance by providing snapshot-level and kernel-level pipeline workflow, aiming to increase parallelism. However, inter-kernel parallelism is not well exploited, and their recomputing execution approach introduces redundant computation and DRAM accesses.

Our proposed design can outperform baseline architectures due to three reasons: reduced computation, on-chip communication optimization, and off-chip communication optimization. Specifically, benefiting from the proposed execution algorithm, we eliminate redundant computation and reduce computational complexity. We also eliminate all off-chip communication unrelated to the graph update between the consecutive snapshots. Furthermore, the proposed pipeline workflow can fully utilize the hardware resources and reduce memory access latency for storing and loading intermediate data between GNN and RNN kernels. Fig. 13 illustrates the normalized execution time of the proposed algorithm in comparison to baseline algorithms with same hardware architecture. The proposed algorithm achieves, on average, 58.9% and 44.6% reduction in execution time compared to the baseline algorithms. Our proposed reconfigurable PE architecture also plays an important role in providing adequate communication patterns and balanced computation workload to support GNN and RNN kernels at the same time, resulting in improved parallelism of the overall execution and eliminating inter-kernel data communication. It should be noted that the intermediate data between kernels is much larger than weight matrices.

The proposed design performs 2.8-4.2× better than ReaDy, 2.4-4.1× better than DGNN-Booster, and 1.8-5.5× better than RACE. The performance gain on the PubMed dataset is more significant when compared to other datasets, because the vertex-to-edge ratio in PubMed is smaller than that of other datasets. This could result in a more significant workload imbalance between GNN (depending on vertex and edge count) and RNN (depending on vertex count only) kernels. In such a case, RACE, employing a heterogeneous architecture, suffer from such a significant workload imbalance issue.

## E. Energy Efficiency Analysis

In the energy analysis, it is important to highlight that the evaluation encompasses the energy consumption of the entire execution process. This includes energy consumption related to computation, on-chip communication, off-chip communication, and control & configuration. Figure 14 provides an overview of the breakdown analysis, showing the normalized energy consumption of the I-DGNN. As shown, the I-DGNN achieves an average reduction of 88.4%, 87.0%, and 85.9% in energy consumption for each dataset compared to the baselines. These values are normalized to the energy consumption of the I-DGNN. The primary drivers behind these energy savings can be attributed to several factors. These include reduced DRAM accesses, decreased computation, and minimized on-chip communication latency. These improvements are achieved through various strategies, such as reducing inter-kernel data on-chip communication, alleviating communication contention through the proposed dataflow and mapping, and reducing DRAM access and computation amounts as mentioned in detail previously. The energy consumption of control and

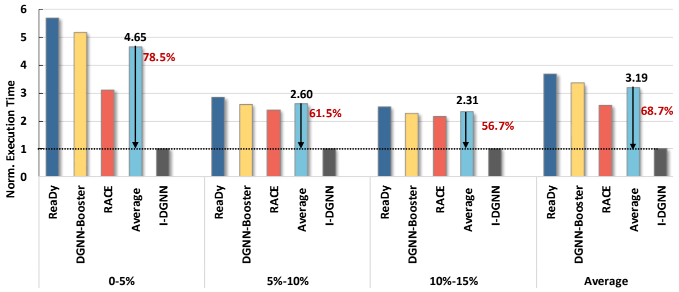


Fig. 15. The sensitivity study of graph dissimilarity proportion between consecutive snapshots for the proposed and baseline accelerators (Normalized to the execution time of the I-DGNN with same graph dissimilarity proportion).

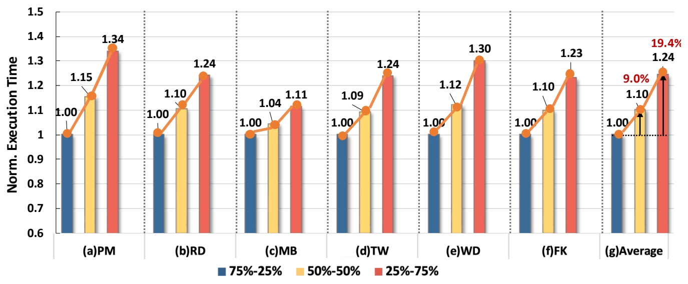


Fig. 16. The sensitivity study of addition and deletion operations ratio between consecutive snapshots on the I-DGNN for various datasets (Normalized to the execution time of ratio 75%-25% (75% addition and 25% deletion)).

configuration accounts for less than 3% of the overall energy consumption.

## F. Sensitivity Analysis

Since the proportion of dissimilarity between consecutive snapshots varies from 4.1% to 13.3% [3], we also changed the proportion to demonstrate that I-DGNN consistently outperforms baseline accelerators across different proportions. Figure 15 shows the normalized execution time when changing the proportion of dissimilarity between consecutive snapshots from 0% to 15% using the Wikipedia dataset. The execution time of the baseline accelerators is normalized to the execution time of the I-DGNN with the same graph dissimilarity ratio. The I-DGNN achieves 78.5%, 61.5%, and 56.7% reduction in execution time compared to baselines as the proportion of dissimilarity between consecutive snapshots changes from 0% to 15%. Even though the performance speedup of I-DGNN becomes smaller when dissimilarity decreases, the proposed one-pass kernel can still eliminate a large number of intermediate features and computations as compared to prior work. From this study, we conclude that the performance gains of our proposed algorithm diminish when dissimilarity and GNN layer count increase. The proportion of dissimilarity types between consecutive snapshots varies, so we also adjusted the ratio of addition and deletion operations to demonstrate that I-DGNN is robust across different proportions of dissimilarity types. Figure 16 shows the normalized execution time when changing the proportion of dissimilarity type between consecutive snapshots from 75%-25% (75% addition and 25% deletion) to 25%-75% (25% addition and 25% deletion) using the WD dataset. The execution time of the baseline accelerators is normalized to the execution time of the I-DGNN with the same dissimilarity types ratio. As depicted, the execution time of the I-DGNN could change under different addition and deletion ratios. The deletion operation is fairly time-consuming, and performing more deletions will lead to an increase in the total execution time. Nevertheless, our solution aims to reduce the total number of intermediate data used for DGNN, which is orthogonal to CommonGraph [12].

## G. Scalability Analysis

We evaluate the scalability of the proposed architecture by varying the dimension of the unified PE array. The array size is scaled from 32 to 4096. All performance metrics are measured in cycles and normalized to the performance of the

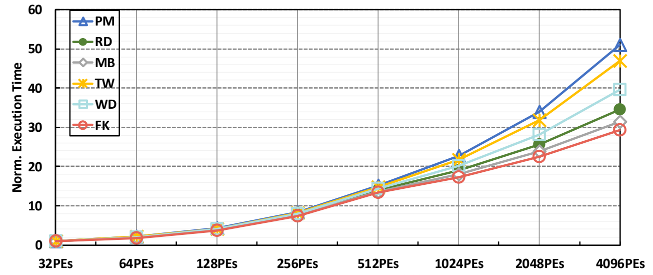


Fig. 17. Scalability study of I-DGNN across different datasets with different PE sizes (Normalized to the execution time with 32 PEs).

4×4 dimension. The execution time of the I-DGNN running at the same frequency with different PE counts is depicted in Figure 17. The I-DGNN can achieve nearly linear speedups when the number of PEs is smaller than 512, showing ideal scalability. When the number of PEs increases further from 512 to 4096, the performance of the I-DGNN can be improved by 1.4× on average once the number of PEs has a 2× increase, showing adequate scalability even though the off-chip memory bandwidth limits the performance.

## H. MACs and Buffer Utilization Analysis

We evaluate the MAC units and buffer utilization of the proposed architecture using the WD dataset. The utilization of the MAC units is shown in Figure18(a). The dynamic hardware configuration can be completed within 16 cycles to balance the workload among MAC units, allowing most of the execution to benefit from the proposed analytical model for pipelining DGNN kernels. The buffer capacity utilization is illustrated in Figure18(b). During the computation process, intermediate results are gradually generated and stored in the buffer. The buffer capacity is nearly fully utilized after 120 cycles.

## I. Area Consumption Analysis

Fig.19 (a) shows the breakdown of the overall area of the I-DGNN. This includes the PE, global buffer, interconnects, and logic components for control and configuration. The PE array accounts for 36.06% of the total chip area. The on-chip global buffer makes up 58.89% of the chip area. The torus interconnect uses 4.6% of the chip area. The controller's area consumption is negligible at 0.45% of the total chip area. Fig.19 (b) provides a breakdown of area consumption for the proposed PE. The MACs array accounts for 42.53% of the total PE area, while the sparse graph structure buffer and dense local buffer account for 25.51% and 31.89% respectively. The PE

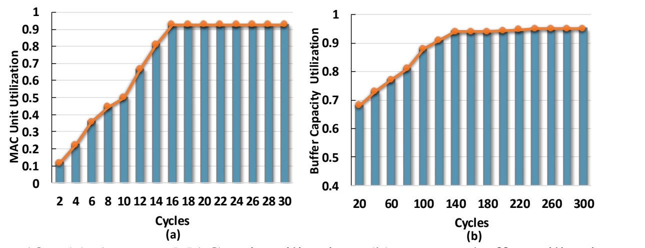


Fig. 18. (a) Average MAC unit utilization, (b) average buffer utilization.

reconfigurable Muxes and local control logic have a negligible area consumption of 0.07% of the total PE area.

## VII. Related Work

Despite significant efforts in dynamic graph analytics and DGNNs, the proposed I-DGNN framework is the first work that identifies the evolving graph components and their interactions with multi-GNN layers.

**Dynamic Graph Neural Network Accelerators:** To accelerate the computations of DGNNs, several customized accelerators [1]–[3] have been proposed. The prior research provided a comprehensive workload characterization of DGNNs and demonstrated the improved performance and energy efficiency of specialized architecture as compared to GPUs. DGNN-Booster [2], an FPGA DGNN inference accelerator, uses a message-passing mechanism for the GNN kernel, but follows the recomputation computing paradigm in which each snapshot is processed by GNNs individually. ReaDy [1], a ReRAM-based DGNN inference accelerator, implements redundancy-free data scheduling and inter-kernel pipelining to enhance efficiency, but it shares a similar concern with DGNN-Booster. RACE [3] employs a heterogeneous architecture for GNN and RNN kernel with incremental computing graph algorithm. This algorithm only processes the dissimilar graph components between snapthots, but introduces extra storage overhead and memory access due to the duplicated intermediate data. Neither Race [3] nor DGNN-Booster [2] fully reveals the dynamic interactions between snapshots and GNN models.

**Optimizations for Evolving Graph Analytics:** In evolving computing scheme, multiple snapshots are processed simultaneously when available. For instance, CommonGraph and MEGA [12], [48] identified the high computational overhead associated with graph deletion operations, and thus converting expensive deletion operations into addition operations by leveraging the mutually inclusive graph structure across snapshots. However, this evolving computing paradigm has limited applicability to DGNN executions with discrete graphs. First, evolving computing relies on pre-processing to identify a mutually inclusive subgraph among multiple snapshots, which is inefficient for handling real-time graphs. Additionally, the redundant computations involved in the GNN pipeline are not addressed. Our proposed method can be integrated with this evolving computing paradigm to overcome the aforementioned limitations, enabling efficient acceleration of DGNN execution with evolving graphs.

**Dynamic Graph Processing Accelerator:** On the other hand, a significant amount of accelerator architectures [49]–

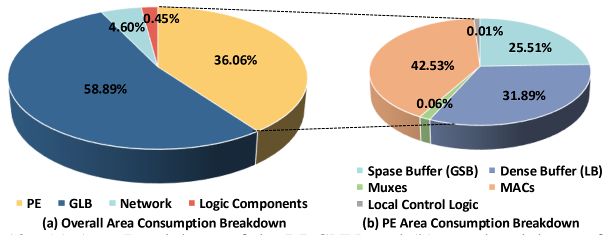


Fig. 19. (a) Area Breakdown of the I-DGNN, and (b) area breakdown of the proposed PE.

[56] have been proposed to optimize dynamic graph processing. For example, Coup [57] proposed a modified graph traversal algorithm tailored for dynamic graphs which can minimize read/write memory access. Additionally, Basak et al. [56] have proposed an accelerator that sorts streaming edges to improve data locality and execution speed on a conventional graph accelerator. However, those graph processing accelerators have not well considered the combined challenges from both GNN and RNN kernels. The proposed one-pass computation method can be efficiently applied to dynamic graph processing through a slight modification. It still can eliminate the repeated read/write memory access and computations.

## VIII. Conclusion

In this paper, we propose I-DGNN, a theoretical, architectural, and algorithmic framework with the aim of designing scalable and efficient accelerators for DGNN execution with improved performance and energy efficiency. On the theory side, the key idea is to identify essential computations between consecutive graph snapshots and encapsulate them as a separate kernel independent from the DGNN model. Specifically, the proposed one-pass DGNN computing model extracts the process of graph update as a chained matrix multiplication between evolving graphs through rigorous mathematical derivations. Consequently, consecutive snapshots utilize a one-pass computation kernel instead of passing through the entire DGNN execution pipeline, thereby eliminating the costly data movement of intermediate results across DGNN layers. On the architecture side, we propose a unified accelerator architecture that can be dynamically configured to support the computation characteristics of the proposed I-DGNN computing model with improved data and pipeline parallelism. On the algorithm side, we propose a new dataflow and mapping tailored for I-DGNN to further improve the data locality of inter-kernel data across the DGNN pipeline. Simulation results show that the proposed accelerator achieves 65.9%, 71.1%, and 58.8% reductions in execution time and 88.4%, 87.0%, and 85.9% improvements in energy efficiency on average across multiple DGNN datasets compared to state-of-the-art-accelerators [1]–[3].

## Acknowledgements

This research was partially supported by NSF grants CCF-1901165, CCF-195398, CCF-2131946, CCF-1901165, and CNS-2321224. We sincerely thank the anonymous reviewers for their excellent and constructive feedback.

## References

[1] Yu Huang, Long Zheng, Pengcheng Yao, Qinggang Wang, Haifeng Liu, Xiaofei Liao, Hai Jin, and Jingling Xue. Ready: A reram-based processing-in-memory accelerator for dynamic graph convolutional networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(11):3567–3578, 2022.

[2] Hanqiu Chen and Cong Hao. Dgnn-booster: A generic fpga accelerator framework for dynamic graph neural network inference. In *Proceedings of the IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 195–201, 2023.

[3] Hui Yu, Yu Zhang, Jin Zhao, Yujian Liao, Zhiying Huang, Donghao He, Lin Gu, Hai Jin, Xiaofei Liao, Haikun Liu, et al. Race: An efficient redundancy-aware accelerator for dynamic graph neural network. *ACM Transactions on Architecture and Code Optimization*, 20(4):1–26, 2023.

[4] Jinyin Chen, Xueke Wang, and Xuanheng Xu. Gc-lstm: Graph convolution embedded lstm for dynamic network link prediction. *Applied Intelligence*, pages 1–16, 2022.

[5] Kai Lei, Meng Qin, Bo Bai, Gong Zhang, and Min Yang. Gcn-gan: A non-linear temporal link prediction model for weighted dynamic networks. In *Proceedings of the IEEE conference on computer communications (INFOCOM)*, pages 388–396. IEEE, 2019.

[6] Pete Burnap, Omer F Rana, Nick Avis, Matthew Williams, William Housley, Adam Edwards, Jeffrey Morgan, and Luke Sloan. Detecting tension in online communities with computational twitter analysis. *Technological Forecasting and Social Change*, 95:96–108, 2015.

[7] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *VLDB Endowment*, 8(12):1804–1815, 2015.

[8] Lubos Takac and Michal Zabovsky. Data analysis in public social networks. In *Proceeding of the International scientific conference and international workshop present day trends of innovations*, volume 1, 2012.

[9] N Laptev and S Amizadeh. Yahoo anomaly detection dataset s5. *URL http://webscope. sandbox. yahoo. com/catalog. php*, 2015.

[10] Sherif Sakr, Angela Bonifati, Hannes Voigt, Alexandru Iosup, Khaled Ammar, Renzo Angles, Walid Aref, Marcelo Arenas, Maciej Besta, Peter A Boncz, et al. The future is big graphs: a community view on graph processing systems. *Communications of the ACM*, 64(9):62–71, 2021.

[11] Venkatesan T Chakaravarthy, Shivmaran S Pandian, Saurabh Raje, Yogish Sabharwal, Toyotaro Suzumura, and Shashanka Ubaru. Efficient scaling of dynamic graph neural networks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.

[12] Mahbod Afarin, Chao Gao, Shafiur Rahman, Nael Abu-Ghazaleh, and Rajiv Gupta. Commongraph: Graph analytics on evolving data. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, volume 2, pages 133–145, 2023.

[13] George Panagopoulos, Giannis Nikolentzos, and Michalis Vazirgiannis. Transfer graph neural networks for pandemic forecasting. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 4838–4845, 2021.

[14] Ling Zhao, Yujiao Song, Chao Zhang, Yu Liu, Pu Wang, Tao Lin, Min Deng, and Haifeng Li. T-gcn: A temporal graph convolutional network for traffic prediction. *IEEE transactions on intelligent transportation systems*, 21(9):3848–3858, 2019.

[15] Osman Asif Malik, Shashanka Ubaru, Lior Horesh, Misha E Kilmer, and Haim Avron. Dynamic graph convolutional networks using the tensor m-product. In *Proceedings of the SIAM international conference on data mining (SDM)*, pages 729–737. SIAM, 2021.

[16] Aldo Pareja, Giacomo Domeniconi, Jie Chen, Tengfei Ma, Toyotaro Suzumura, Hiroki Kanezashi, Tim Kaler, Tao Schardl, and Charles Leiserson. Evolvegcn: Evolving graph convolutional networks for dynamic graphs. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 5363–5370, 2020.

[17] Hongxia Yang. Aligraph: A comprehensive graph neural network platform. In *Proceedings of the 25th ACM SIGKDD International Conference on knowledge Discovery & Data Mining*, pages 3165–3166, 2019.

[18] Seyed Mehran Kazemi, Rishab Goel, Kshitij Jain, Ivan Kobyzev, Akshay Sethi, Peter Forsyth, and Pascal Poupart. Representation learning for dynamic graphs: A survey. *Journal of Machine Learning Research*, 21(70):1–73, 2020.

[19] ZhengZhao Feng, Rui Wang, TianXing Wang, Mingli Song, Sai Wu, and Shuibing He. A comprehensive survey of dynamic graph neural networks: Models, frameworks, benchmarks, experiments and challenges. *arXiv preprint arXiv:2405.00476*, 2024.

[20] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Proceedings of International Conference on Neural Information Processing Systems (NIPS)*, pages 1025–1035. ACM, 2017.

[21] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *arXiv preprint arXiv:1810.00826*, 2018.

[22] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[23] Hongxi Li, Zuxuan Zhang, Dengzhe Liang, and Yuncheng Jiang. K-truss based temporal graph convolutional network for dynamic graphs. In *Asian Conference on Machine Learning*, pages 739–754. PMLR, 2024.

[24] Fan Zhou, Xovee Xu, Ce Li, Goce Trajcevski, Ting Zhong, and Kunpeng Zhang. A heterogeneous dynamical graph neural networks approach to quantify scientific impact. *arXiv preprint arXiv:2003.12042*, 2020.

[25] Guangyin Jin, Lingbo Liu, Fuxian Li, and Jincai Huang. Spatio-temporal graph neural point process for traffic congestion event prediction. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 14268–14276, 2023.

[26] Samira Khodabandehlou and Alireza Hashemi Golpayegani. Fifraud: unsupervised financial fraud detection in dynamic graph streams. *ACM Transactions on Knowledge Discovery from Data*, 18(5):1–29, 2024.

[27] John Adrian Bondy and Uppaluri Siva Ramachandra Murty. *Graph theory*. Springer, 2008.

[28] Tayo Oguntebi and Kunle Olukotun. GraphOps: A dataflow library for graph analytics acceleration. In *Proceedings of ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM, 2016.

[29] Hyoukjun Kwon, Prasanth Chatarasi, Michael Pellauer, Angshuman Parashar, Vivek Sarkar, and Tushar Krishna. Understanding reuse, performance, and hardware cost of DNN dataflow: A data-centric approach. In *Proceedings of IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 754–768. IEEE, 2018.

[30] Raveesh Garg, Eric Qin, Francisco Munoz-Mart'inez, Robert Guirado, Akshay Jain, S. Abadal, Jos'e L. Abell'an, Manuel E. Acacio, Eduard Alarc'on, Sivasankaran Rajamanickam, and Tushar Krishna. Understanding the design space of sparse/dense multiphase dataflows for mapping graph neural networks on spatial accelerators. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 571–582. IEEE, 2021.

[31] Tong Geng, Chunshu Wu, Yongan Zhang, Cheng Tan, Chenhao Xie, Haoran You, Martin Herbordt, Yingyan Lin, and Ang Li. I-GCN: A graph convolutional network accelerator with runtime locality enhancement through islandization. In *Proceedings of IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1051–1063. IEEE, 2021.

[32] Jiajun Li, Ahmed Louri, Avinash Karanth, and Razvan Bunescu. Gcnax: A flexible and energy-efficient accelerator for graph convolutional neural networks. In *Proceedings of IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 775–788, 2021.

[33] Cen Chen, Kenli Li, Yangfan Li, and Xiaofeng Zou. Regnn: A redundancy-eliminated graph neural networks accelerator. In *Proceedings of IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 429–443. IEEE, 2022.

[34] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE journal of solid-state circuits*, 52(1):127–138, 2016.

[35] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. Dramsim2: A cycle accurate memory system simulator. *IEEE computer architecture letters*, 10(1):16–19, 2011.

[36] Mark Horowitz. Energy table for 45nm process. In *Stanford VLSI wiki*. 2014.

[37] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. Cacti 6.0: A tool to understand large caches. *University of Utah and Hewlett Packard Laboratories, Tech. Rep*, 147, 2009.

[38] Palash Goyal, Nitin Kamra, Xinran He, and Yan Liu. Dyngem: Deep embedding method for dynamic graphs. *arXiv preprint arXiv:1805.11273*, 2018.

[39] Ryan Rossi and Nesreen Ahmed. The network data repository with interactive graph analytics and visualization. In *Proceedings of the AAAI conference on artificial intelligence*, volume 29, 2015.

[40] Wikidata. Retrieved from https://github.com/mniepert/mmkb/tree/master/TemporalKGs/wikidata. Accessed: 2024.

[41] Flickr. Retrieved from https://www.kaggle.com/datasets/hsankesara/flickr-image-dataset. Accessed: 2024.

[42] Mobile. Retrieved from https://dblp.uni-trier.de/xml/. Accessed: 2024.

[43] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[44] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with lstm. *Neural computation*, 12(10):2451–2471, 2000.

[45] Seyed Mehran Kazemi, Rishab Goel, Kshitij Jain, Ivan Kobyzev, Akshay Sethi, Peter Forsyth, and Pascal Poupart. Representation learning for dynamic graphs: A survey. *Journal of Machine Learning Research*, 21(70):1–73, 2020.

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

[47] Youngjoo Seo, Michaël Defferrard, Pierre Vandergheynst, and Xavier Bresson. Structured sequence modeling with graph convolutional recurrent networks. In *Proceeding of 25th International Conference*, pages 362–373. Springer, 2018.

[48] Chao Gao, Mahbod Afarin, Shafiur Rahman, Nael Abu-Ghazaleh, and Rajiv Gupta. Mega evolving graph accelerator. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 310–323, 2023.

[49] Andrey Ayupov, Serif Yesil, Muhammet Mustafa Ozdal, Taemin Kim, Steven Burns, and Ozcan Ozturk. A template-based design methodology for graph-parallel hardware accelerators. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(2):420–430, 2017.

[50] Muhammet Mustafa Ozdal, Serif Yesil, Taemin Kim, Andrey Ayupov, John Greth, Steven Burns, and Ozcan Ozturk. Energy efficient architecture for graph analytics accelerators. *ACM SIGARCH Computer Architecture News*, 44(3):166–177, 2016.

[51] Mark C Jeffrey, Suvinay Subramanian, Cong Yan, Joel Emer, and Daniel Sanchez. A scalable architecture for ordered parallelism. In *Proceedings of the 48th international symposium on microarchitecture*, pages 228–241, 2015.

[52] Mark C Jeffrey, Suvinay Subramanian, Maleen Abeydeera, Joel Emer, and Daniel Sanchez. Data-centric execution of speculative parallel programs. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13. IEEE, 2016.

[53] Shafiur Rahman, Nael Abu-Ghazaleh, and Rajiv Gupta. Graphpulse: An event-driven hardware accelerator for asynchronous graph processing. In *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 908–921. IEEE, 2020.

[54] Shafiur Rahman, Mahbod Afarin, Nael Abu-Ghazaleh, and Rajiv Gupta. Jetstream: Graph analytics on streaming data with event-driven hardware accelerator. In *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1091–1105, 2021.

[55] Qinggang Wang, Long Zheng, Yu Huang, Pengcheng Yao, Chuangyi Gui, Xiaofei Liao, Hai Jin, Wenbin Jiang, and Fubing Mao. Grasu: A fast graph update library for fpga-based dynamic graph processing. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 149–159, 2021.

[56] Abanti Basak, Zheng Qu, Jilan Lin, Alaa R Alameldeen, Zeshan Chishti, Yufei Ding, and Yuan Xie. Improving streaming graph processing performance using input knowledge. In *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1036–1050, 2021.

[57] Guowei Zhang, Webb Horn, and Daniel Sanchez. Exploiting commutativity to reduce the cost of updates to shared data in cache-coherent systems. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 13–25, 2015.