DiTile-DGNN: An Efficient Accelerator for Distributed Dynamic Graph Neural Network Inference

Jiaqi Yang
The George Washington University
Washington DC, USA
Yang_Jiaqi_Cute@gwu.edu

Hao Zheng
University of Central Florida
Orlando, Florida, USA
Hao.Zheng@ucf.edu

Ahmed Louri
George Washington U.
Washington DC, USA
louri@gwu.edu

Abstract

Dynamic Graph Neural Networks (DGNNs) have recently emerged as a promising model for learning complex temporal and spatial relationships in evolving graphs. The performance of DGNNs is enabled by the simultaneous integration of both graph neural networks (GNNs) and recurrent neural networks (RNNs). Despite the theoretical advancements, the design space of such complex models has significantly exploded due to the combinatorial challenges of heterogeneous computation kernels and intricate data dependency (i.e., intra- and inter-snapshot data dependency). This makes the computations of DGNN hard to scale, posing significant challenges in parallelism, data reuse, and communication. To address this challenge, we propose DiTile-DGNN, an efficient accelerator for large-scale DGNN execution. The proposed DiTile-DGNN consists of a redundancy-free parallelism strategy, workload balance optimization, and a reconfigurable accelerator architecture. Specifically, we propose a redundancy-free framework that can efficiently find an efficient parallelism strategy that can fully eliminate the data redundancy between graph snapshots while minimizing the communication complexity. Additionally, we propose a workload balance optimization for DGNN models to enhance resource utilization and eliminate synchronization overhead between snapshots. Lastly, we propose a reconfigurable accelerator architecture, with a flexible interconnect, that can be dynamically configured in support of various DGNN dataflows. Our simulations demonstrate that DiTile-DGNN achieves 48.4%, 56.1%, 23.2%, and 36.1% reductions in execution time and 83.4%, 84.0%, 75.6%, and 71.4% improvements in energy efficiency compared to state-of-the-art accelerators, including ReaDy [20], DGNN-Booster [8], RACE [51], and MEGA [12], on average across multiple DGNN datasets.

ACM Reference Format:
Jiaqi Yang, Hao Zheng, and Ahmed Louri. 2025. DiTile-DGNN: An Efficient Accelerator for Distributed Dynamic Graph Neural Network Inference. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25), June 21–25, 2025, Tokyo, Japan.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3695053.3731017

ISCA '25, Tokyo, Japan

ACM ISBN 979-8-4007-1261-6/25/06
https://doi.org/10.1145/3695053.3731017

1 Introduction

Dynamic Graph Neural Networks (DGNNs) [9, 28] have gained significant attention due to their ability to model complex temporal and spatial relationships in evolving graphs [6, 11, 27, 41], making them suitable for applications such as social network analysis, recommendation systems, and traffic prediction [9, 28]. By combining Graph Neural Networks (GNNs) with Recurrent Neural Networks (RNNs), DGNNs can effectively capture both structural and temporal dynamics within graph data [38]. However, scaling DGNN computations faces a major challenge due to the computational and communication complexity incurred by the temporary and spatial graph dependency, as well as redundancy between consecutive snapshots.

Current DGNN accelerators are often optimized for a centralized architecture [8, 12, 20, 48, 51, 54], and few of them have considered the challenges of scaling DGNN in a large-scale accelerator with distributed buffers. The challenge is multi-fold, including the complex data dependency among or within snapshots, redundancy between consecutive snapshots, and unbalanced workload caused by the heterogeneous graph data sets and models. Unfortunately, existing machine learning accelerators [30, 31, 45, 47, 49, 50, 53], irrespective of deep learning and graph neural networks, are only optimized for a single model. Furthermore, several prior works [42] have attempted to accelerate multi-DNN models. However, they are inefficient in handling the distinct workload characteristics of RNNs and GNNs.

Recent research [7, 8, 12, 16, 20, 51, 56] attempted to optimize the parallelism efficiency of DGNNs, however, they only pursue data-level parallelism by distributing graph snapshots without considering their data dependency. Consequently, this leads to unbalanced workload and irregular communication patterns. For example, prior work [7, 8, 16, 20, 51] distribute each snapshot to individual computing tiles. While each snapshot executes independently in the GNN phase, RNN model needs to collect the intermediate data from each snapshot to comprehend the evolving graph components. This requires a global synchronization among computing tiles. On the other hand, MEGA and Aligraph [12, 56], following traditional GNN dataflows, partition all the snapshots among computing tiles to avoid the synchronization issue during the RNN phase. However, the distributed graph components incur irregular communications to aggregate vertex features at the GNN phase.

Furthermore, prior work has indicated that the variation of consecutive snapshots is negligible, with 86.7% to 95.9% of vertices remaining the same over time [51]. To leverage this opportunity, current DGNN accelerators proposed reusing repetitive computations and their intermediate results between consecutive snapshots.

However, parallelizing snapshots is inevitable to incur random communications as the repetitive computations are unpredictable. This further exacerbates the communication complexity in distributed DGNN inference.

Lastly, real-world dynamic graphs have skewed degree distributions, and therefore, evenly partitioning graphs is required in graph applications. For example, BNS-GCN [43] ensures workload balance by evenly distributing vertices. Graph Ladling [21] divides vertices into partitions without considering edge connection. This leads to increased synchronization overhead and inefficient resource utilization in distributed settings. Existing DGNN frameworks typically employ coarse-grained partitioning strategies, such as snapshot [12, 56] or vertex partitioning [7, 8, 16, 20, 51]. We observed that prior works only balance the workload for a particular model or execution phase without considering the DGNN as a whole. In this paper, we argue that the parallelism optimization of DGNNs should consider the combined effects of both GNN and RNN models. To this end, we mathematically analyze the intricate data dependency among and within snapshots, and fully understand their communication and computation characteristics at both GNN and DNN phases. Upon this theoretical analysis, we propose DiTile-DGNN, a novel high-performance and energy-efficient distributed-tiled accelerator for efficient large scale DGNN execution. Specifically, this paper makes the following contributions:

- We propose a parallelism strategy that can simultaneously balance the data reuse and parallelism efficiency of both GNN and RNN models. Specifically, we propose a new tiling algorithm that aims to reduce costly off-chip communication. The parallelism strategy comprehensively considers the communication and data reuse of compounded GNN and RNN kernels.
- We propose a workload optimization strategy to balance the workload among kernels, enabling even distributed workload at both GNN and RNN models. This significantly improves resource utilization, reduces synchronization costs, and enhances scalability.
- We propose a reconfigurable and distributed-tiled accelerator architecture to efficiently manage the diverse communication requirements of DGNNs. The architecture dynamically configures the interconnect to handle both regular communication for temporal dependencies and irregular communication for spatial dependencies. By combining dedicated paths for regular communication with flexible routing for irregular exchanges, it minimizes bottlenecks and ensures scalability for dense and evolving graph workloads. Additionally, it adapts to graph structure and workload dynamics, optimizing resource utilization and reducing communication overhead.

We conduct a detailed performance and energy evaluation through simulation and show that the proposed accelerator achieves 48.4%, 56.1%, 23.2%, and 36.1% reductions in execution time and 83.4%, 84.0%, 75.6%, and 71.4% improvements in energy efficiency on average across multiple DGNN datasets compared to ReaDy [20], DGNN-Booster [8], RACE [51] and MEGA [12], respectively.

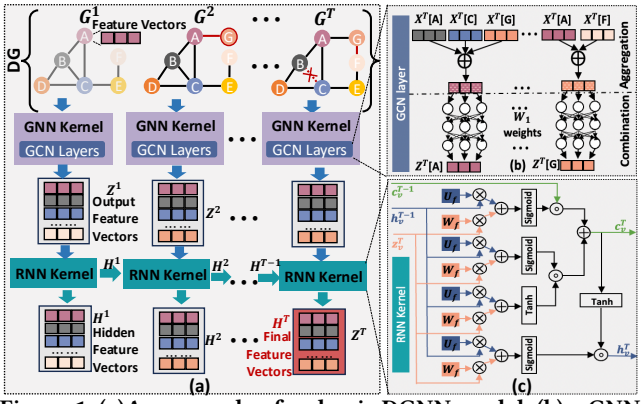


Figure 1: (a)An example of a classic DGNN model, (b) a GNN computation kernel, and (c) an RNN computation kernel.

2 Background

2.1 Dynamic Graph Representation

In real-world applications [28, 32, 34, 35, 38, 46, 52], graphs evolve over time, with vertices and edges being frequently added or removed. In general, there are two types of dynamic graphs [23] 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 DiTile-DGNN based on the discrete-time dynamic graph representation.

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

2.2 Discrete-Time Dynamic Graph Neural Network

Discrete-time DGNN models [7] are designed for analyzing discrete-time dynamic graphs. These models can be classified into two groups: typical DGNN models and specialized DGNN models.

Typical DGNN models [7, 22, 25, 29, 55] are composed of both conventional Graph Neural Network (GNN) and Recurrent Neural Network (RNN) kernels. For discrete-time dynamic graphs, the DGNN model sequentially processes each snapshot to identify the changes occurring in the graphs, as shown in Figure 1 (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 1 (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.

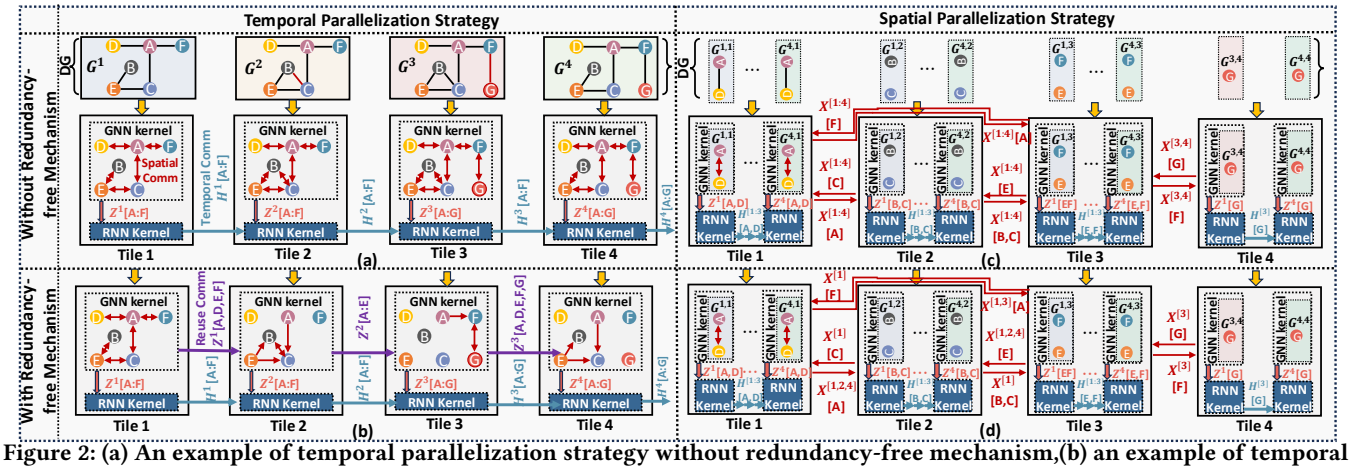


Figure 2: (a) An example of temporal parallelization strategy without redundancy-free mechanism,(b) an example of temporal parallelization strategy with redundancy-free mechanism,(c) an example of spatial parallelization strategy without redundancy-free mechanism,(d) an example of spatial parallelization strategy with redundancy-free mechanism (G^{α} is the α-th snapshot, $G^{\alpha,\beta}$ means β-th part of the α-th snapshot, $X^{[m:n]}[V]$ means the input feature of vertex V in the m-th to n-th snapshots, $Z^{\alpha}[V]$ means the output feature of vertex V in the α-th snapshot, $H^{\alpha}[V]$ is the hidden feature of vertex V in the α-th snapshot).

During the combination phase, 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[17] and Graph Isomorphism Networks (GINs)[44], 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 1 (c) illustrates an example of RNN computations, where the most popular long short-term memory (LSTM) [18] 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.

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

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.

3 Motivation

3.1 Pitfalls of Existing DGNN Parallelization Strategies

Despite advancements in the parallelization of Dynamic Graph Neural Networks (DGNN), current approaches face several critical limitations, especially in efficiently managing temporal and spatial dependencies inherent in dynamic graphs. Below, we outline the key challenges associated with conventional parallelization strategies, which result in suboptimal performance in large-scale DGNN applications.

3.1.1 Inefficiencies in Temporal Parallelization Strategies. The conventional temporal parallelism strategy [7, 8, 16, 20, 51], illustrated in Figure 2(a), assigns each snapshot of the dynamic graph to a separate distributed tile, allowing each tile to independently execute GNN kernels. The communication involved in aggregating vertex features within GNN kernels is referred to as spatial communication and is confined within each distributed tile. However, due to temporal dependencies across RNN kernels, vertex exchanges—referred to as temporal communication—are required between tiles to synchronize RNN computations. In DGNN applications with a large number of snapshots, this setup significantly increases the overhead of temporal communication, resulting in inefficiencies and high communication costs.

Most real-world dynamic graphs exhibit strong temporal similarity, meaning there are substantial overlaps in the input features and neighbors between consecutive snapshots. On average, unaffected vertices account for 86.7% to 95.9% of all vertices in various real-world dynamic graphs [51], as each batch of updates typically affects only a small portion of the graph. This high level of similarity allows us to reuse the final states of most vertices from the

previous snapshot to efficiently compute the updated states for the current snapshot. To minimize redundant communication, the redundancy-free temporal parallelism strategy (Figure 2(b)) leverages this overlap by reusing vertex states across distributed tiles, incrementally updating the states of vertices for the latest snapshot. This approach introduces what we term reuse communication. For example, the output features of vertices A, D, E, and F in the first snapshot remain the same in the second snapshot. As a result, spatial communication is only needed for aggregating the features of vertices B and C. Additionally, the output features of vertices A, D, E, and F in tile 1 can be reused and sent to tile 2 through reuse communication, thereby serving as the output features of these vertices in the second snapshot. While this strategy effectively reduces redundant spatial computations between consecutive snapshots, vertex exchanges for RNN computations are still required, leading to regular temporal communication overhead. Furthermore, when snapshots have significant overlap, the reuse mechanism can introduce irregular communication overhead, further exacerbating inefficiencies.

3.1.2 Inefficiencies in Spatial Parallelization Strategies. In spatial parallelization (depicted in Figure 2(c), the graph is partitioned, and each distributed tile processes a subset of vertices. While this allows local execution of RNN kernels, GNN kernel computations require remote spatial communication among tiles. When DGNN applications have dense snapshots, the need for spatial communication increases substantially, causing performance degradation due to high spatial communication overhead. The redundancy-free spatial parallelism strategy (Figure 2(d)) reduces overhead by reusing the output features of vertices locally within each tile, thereby avoiding redundant computations and communication. For instance, if the output features of vertex A in the first snapshot remains unchanged in subsequent snapshots, the intra-tile spatial communication required for aggregating vertex A (i.e., $X^{[2:4]}[D]$) and the inter-tile spatial communication (i.e., $X^{[2:4]}[C]$ and $X^{[2:4]}[F]$) can be replaced by more efficient intra-tile reuse communication ($Z^{[1:3]}[A]$). However, in cases where snapshots are dense with limited similarity, spatial communication costs can still be high, particularly when vertex distributions shift frequently between snapshots. This results in inefficiencies and challenges in maintaining scalability.

3.1.3 Inflexibility in Adapting to Diverse Workloads. A significant limitation of existing DGNN parallelization strategies is their static nature. These strategies are not optimized to adapt dynamically to varying workload characteristics, such as changes in the number of snapshots or the degree of similarity between consecutive snapshots. As a result, they either suffer from excessive temporal communication in temporal parallelism or incur high spatial communication costs in spatial parallelism. The inability to adapt to dynamic workloads results in increased computational and communication inefficiencies, particularly in large-scale DGNN scenarios.

To overcome these challenges, a more adaptable approach is required. The proposed redundancy-free dynamic parallelism strategy optimizes the parallelization process by dynamically selecting the most efficient level—temporal, spatial, or a combination—based on the specific workload characteristics. By minimizing redundant communication and computation at both temporal and spatial levels, as well as reducing the volume of costly inter-tile communication, this strategy effectively lowers communication overhead while ensuring efficient execution of GNN and RNN kernels. This adaptive approach enhances scalability and performance, making it highly suitable for diverse and dynamic DGNN workloads where graph characteristics frequently change.

Algorithm 1: Proposed Parallelism Optimization

Input : Application features:
The number of layers in the GNN kernel (L),
Total number of the snapshots (T),
The number of the vertices in each snapshot ($V_i, i \in (0, T]$),
The number of the edges in each snapshot ($E_i, i \in (0, T]$),
The dissimilar rate between $i-th$ and $(i-1)-th$ snapshots ($Dis_i, i \in (0, T]$).

Input : Hardware features:
Total number of available tiles ($TotalTiles$),
The capacity of the distributed buffer (C_{DB}).

Output : Efficient parallel factors: P_v and P_s.

```
/* Generate the size of subgraph with minimal DRAM Access */
Procedure Subgraph Tiling
    // Divide the dynamic graph into multiple subgraphs.
    SV_i = V_i/α;
    DA = Σ_{i=1}^{T} {V_i + α × [E_i × SV_i × (V_i − SV_i)]/(V_i)^2};
    // Choose tiling factor with minimal DRAM access
    for datavolume(SV_i) ∈ [0, C_DB] do
        α ← Minimal(DRAMaccess);
    end
/* Generate efficient parallel factors with minimal inter-tile
   communication amount */
Procedure Parallelization Optimization
    TotalComm = Tcomm() + RFScomm() + ReComm()
    for T/P_s ∈ (0, √TotalTiles], SV_i/P_v ∈ (0, √TotalTiles] do
        P_s, P_v ← Minimal(TotalComm);
    end
```

4 Redundancy-free Dynamic Parallelization Strategy for Distributed DGNN Acceleration

The objective of the proposed redundancy-free dynamic parallelization strategy is to generate an optimized parallelization strategy for distributed DGNN inference to reduce off-chip memory access and inter-tile communication. Specifically, the proposed parallelization strategy includes (1) a matrix tiling algorithm to partition graph snapshots into subgraphs to reduce DRAM access and (2) a parallelism optimization technique that reduces inter-tile communication and eliminates data and computation redundancy in both GNN and RNN kernels.

4.1 Proposed DGNN Matrix Tiling

Unlike traditional deep learning models, the size of dynamic graphs is substantial and typically dominates the on-chip memory. Consequently, it is imperative to partition each graph snapshot into multiple subgraphs. It should be noted that the proposed algorithm focuses on inference, but the proposed methodology can be applied to the training stage where gradient and embedding propagation

follow graph structure as well. Specifically, each snapshot is partitioned into multiple subgraphs regardless of their size, as shown in Algorithm 1 (Line 3-4). The number of subgraphs is referred to as the tiling factor α. The number of vertices in the subgraph of the i-th snapshot (SV_i) is defined by Equation 5, where V_i is the number of the vertices in the i-th snapshot.

$$SV_i = V_i/\alpha \tag{5}$$

The proposed tiling strategy selects an appropriate number of subgraphs to meet on-chip storage constraints while minimizing redundant graph components. The RNN kernels were not considered when estimating DRAM access, as the output of the GNN kernels serves as the input for the RNN kernels. This can be further optimized through dataflow design to enhance intermediate data reuse, which will be discussed later. Additionally, both GNN and RNN weight matrices are relatively small compared to the graph data. Specifically, the DRAM access of the GNN kernels for the input dynamic graph(DA) is represented by Equation 6 and shown in Algorithm 1 (Line 5), where E_i is the number of the edges in the i-th snapshot, and T is the total number of the snapshots.

$$DA = \sum_{i=1}^{T} \{V_i + \alpha \times [E_i \times SV_i \times (V_i - SV_i)]/(V_i)^2\} \tag{6}$$

The number of the vertices in the subgraph of the i-th snapshot (SV_i) related data is limited by the distributed buffer capacity(C_{DB}) shown in Algorithm 1 (Line 7). Then we can find the tiling factor (α) that can minimize the DRAM access with the distributed buffer capacity(C_{DB}) limitation, shown in Algorithm 1 (Line 8).

4.2 Proposed DGNN Parallelism Optimization

Spatial and temporary parallelism are required for deep learning applications to enhance parallelism and data reuse. Unlike deep learning models that rely on loop ordering, assigning graph partitions and snapshots is vital to ensure spatial and temporary parallelism for DGNN workloads. Specifically, the parallelism of DGNN is determined by (1) data dependency between snapshots (called temporal parallelism), (2) graph data parallelism (called spatial parallelism), and (3) redundant computations between snapshots (called reuse). Nevertheless, given the use of distributed buffers, communication is avoidable to eliminate data duplication. Consequently, we aim to design an analytical model to optimize the placement of snapshots and graph partitions, reducing the overall communication between processing tiles. Specifically, inter-tile temporal communication refers to the communication caused by temporal dependencies between consecutive snapshots in the RNN kernels across tiles. The inter-tile spatial communication describes inter-tile spatial dependencies within the graph structure for the GNN kernel in the same snapshot. Reuse communication is the communication between consecutive snapshots across tiles to reuse intermediate data between snapshots due to graph similarity. For example, Figure 3 shows the three types of communication patterns in DGNNs, where the number of snapshots assigned to each tile is referred to as the snapshot parallel factor, P_s, and the number of vertices assigned to each tile is the vertex parallel factor, P_v.

The inter-tile total communication amount ($TotalComm$) can be modeled by the Equation 7 and shown in Algorithm 1 (Line 13). $Tcomm$ is the inter-tile temporal communication amount. $Scomm$ is

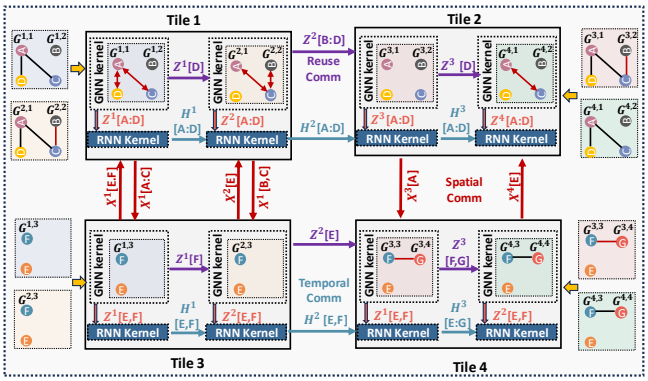


Figure 3: An example of redundancy-free and dynamic parallelization strategy. $G^{\alpha,\beta}$ means β-th part of the α-th snapshot, $X^{[m:n]}[V]$ means the input feature of vertex V in the m-th to n-th snapshots, $Z^{\alpha}[V]$ means the output feature of vertex V in the α-th snapshot, $H^{\alpha}[V]$ means the hidden feature of vertex V in the α-th snapshot.

the inter-tile spatial communication amount. $Recomm$ is the inter-tile reuse communication amount.

$$TotalComm = Tcomm + RFScomm + ReComm \tag{7}$$

The parallelism optimization aims to find the values of P_s and P_v that minimize $TotalComm$ (Algorithm 1, Line 15). The snapshot parallel factor P_s is constrained by the total number of snapshots (T) and the total number of distributed tiles ($TotalTiles$). Similarly, the vertex parallel factor P_v is limited by the number of vertices in the subgraph of the i-th snapshot (SV_i) and $TotalTiles$ (Algorithm 1, Line 14).

4.2.1 Inter-tile Temporal Communication Modeling. The inter-tile temporal communication addresses temporal dependencies of consecutive snapshots in the RNN kernels across tiles. The number of subgraphs is referred to as the tiling factor, α. The number of snapshots assigned to each tile is referred to as the snapshot parallel factor, P_s. $AvgSV$ is the average vertex number among all sub-snapshots. The inter-tile temporal communication amount of all subgraphs can be modeled by the Equation 8.

$$Tcomm = \alpha \times AvgSV \times (\lceil \frac{T}{P_s} \rceil - 1) \tag{8}$$

4.2.2 Inter-tile Spatial Communication Modeling. The inter-tile spatial communication manages inter-tile spatial dependencies within the graph structure for the GNN kernel in the same snapshot. The inter-tile redundant-free spatial communication amount ($RFScomm$) of all subgraphs can be modeled by the Equation 9, where $Scomm$ is the inter-tile spatial communication amount of all subgraphs without redundant-free mechanism and $RScomm$ is the inter-tile redundant spatial communication amount of all subgraphs.

$$RFScomm = Scomm - RScomm \tag{9}$$

The inter-tile spatial communication amount of all subgraphs without redundant-free mechanism can be represented by the Equation 10, where $TotalScomm$ is the total spatial communication amount of all subgraphs and $IntraTileScomm$ is the intra-tile spatial communication amount of all subgraphs.

$$Scomm = TotalScomm - IntraTileScomm \tag{10}$$

Algorithm 2: Balance-aware Workload Optimization

Input :
The number of layers in the GNN kernel (L),
Total number of the snapshots (T),
The average number of the vertices in sub-snapshots ($AvgSV$),
Total number of available tiles ($TotalTiles$),
The efficient parallel factors (P_v and P_s).

Output: Partition results

```
1  /* Calculate vertex workload for all layers and all snapshots
     */
2  for t ∈ [1, T] do
3      for l ∈ [1, L] do
4          for l' ∈ [1, l] do
5              vload[i] += N^{l'}(v_i^t)
6          end
7      end
8  end
9  /* get partition and balanced groups                    */
10 partition = RoundRobin(vload[i])
11 BalancedGroups = Split(partition[Tile_id], P_v, P_s)
12 return Partition results
```

The total spatial communication amount of all subgraphs without redundant-free mechanism can be represented by the Equation 11, where $AvgSE$ is the average edge number among all sub-snapshots and L is the GNN layer amount of the GNN kernel.

$$TotalScomm = \alpha \times L \times T \times AvgSE \tag{11}$$

The intra-tile spatial communication amount of all subgraphs without redundant-free mechanism is represented by the Equation 12.

$$IntraTileScomm = \alpha \times L \times T \times \frac{AvgSE}{(AvgSV)^2} \times \{{P_v}^2 \lfloor \frac{AvgSV}{P_v} \rfloor + Mod(\frac{AvgSV}{P_v})^2\} \tag{12}$$

The ratio of inter-tile redundant spatial communication amount and total redundant spatial communication amount is related to the ratio of inter-tile spatial communication amount and total spatial communication amount. The inter-tile redundant spatial communication amount of all subgraphs can be represented by the Equation 13, where $TotalRScomm$ is the total redundant spatial communication amount of all subgraphs.

$$RScomm = TotalRScomm \times Scomm/TotalScomm \tag{13}$$

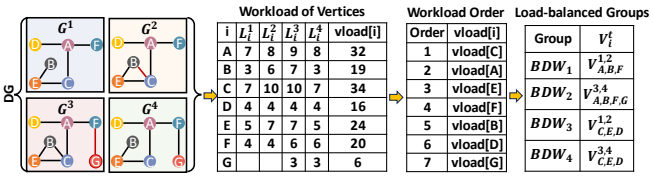
The total redundant spatial communication amount of all subgraphs can be represented by the Equation 14, where $VScomm$ represents the related spatial communication amount of each vertex within all L layers GNN kernel and Dis is the average dissimilarity rate of each vertex between consequent snapshots.

$$TotalRScomm = a \times T \times AvgSV \times (1 - Dis) \times VScomm \tag{14}$$

$$VScomm = \sum_{l=1}^{L} \sum_{l=1}^{l'} \left(\frac{AvgSE}{AvgSV}\right)^{l'} \tag{15}$$

4.2.3 Inter-tile Reuse Communication Modeling. Given the graph similarity between snapshots, a significant amount of intermediate data could be reused between consecutive snapshots. The intermediate data exchange between snapshots is defined as inter-tile reuse communication. The inter-tile reuse communication amount of all subgraphs can be modeled by the Equation 16. $\lceil \frac{T}{P_s} \rceil$ represents the number of groups into which the T snapshots are divided, where each group contains P_s snapshots.

$$Recomm = \alpha \times \{\lceil \frac{T}{P_s} \rceil - 1\} \times AvgSV \times (1 - Dis) \times VScomm \tag{16}$$



i	L_i^1	L_i^2	L_i^3	L_i^4	vload[i]
A	7	8	9	8	32
B	3	6	7	3	19
C	7	10	10	7	34
D	4	4	4	4	16
E	5	7	7	5	24
F	4	4	6	6	20
G			3	3	6

Order	vload[i]
1	vload[C]
2	vload[A]
3	vload[E]
4	vload[F]
5	vload[B]
6	vload[D]
7	vload[G]

Group	V_i^t
BDW_1	$V_{A,B,F}^{1,2}$
BDW_2	$V_{A,B,F,G}^{3,4}$
BDW_3	$V_{C,E,D}^{1,2}$
BDW_4	$V_{C,E,D}^{3,4}$

Figure 4: An example of workload balance optimization.

5 Proposed Workload Balance Optimization

In DGNNs, the workload is highly related to the vertex. Prior work [13, 30] simply estimates the workload of graph analytics based on the vertex or edge count. To analyze the complex DGNN workload, we propose a new analytical model that can estimate the multi-layer GNNs based on vertex structure. Specifically, the computation of GNNs depends on the number of L layers of neighbors associated with a given vertex. The workload (L_i^t) for a vertex ($v_i^t \in G^t$) in snapshot G^t is defined as the recursive sum of degrees of all L layers, as shown in Equation 17:

$$L_i^t = \sum_{l=1}^{L} \sum_{l'=1}^{l} (N^{l'}(v_i^t)) \tag{17}$$

Here, $N^{l'}(v_i^t)$ represents the set of l'-hop neighbors of vertex v_i^t in G^t. The computation cost for Recurrent Neural Networks (RNNs) is disregarded, as it is uniform for all vertices.

To estimate the vertex workload in DGNN, we propose a label aggregation technique that can record the total amount of workload related to a given vertex. Initially, each vertex is assigned a label of 1, representing a workload contribution of 1. Labels of vertices are propagated along edges during the GNN aggregation, while their values are progressively accumulated as they reach the destination vertex. This process is repeated for all L layers in an L-layer DGNN to determine the final workload for each vertex.

Based on the proposed labeling technique, the proposed workload optimization is summarized in Algorithm 2. We start initializing a data structure, *vload*, to store the cumulative workload of each vertex across all layers and snapshots. This data structure provides a comprehensive view of the computational demand associated with each vertex, including its multi-layer interactions and dynamic graph updates across snapshots.

The next step is to compute the workloads of all vertices by iterating through the L layers of the GNN and the T snapshots of the dynamic graph(line 2-8). This can reflect the influence of multi-hop dependencies in GNN computations. The cumulative vertex workloads are progressively updated in *vload* to capture the complete workload for each vertex across all layers and snapshots. For example, consider a GNN kernel with two layers ($L = 2$). The computational workload of vertex A in the first snapshot (L_A^1), as shown in Figure 4, is determined by its receptive field. Specifically, according to Equation 17, it can be formulated as $2 \times N^1(v_A^1) + N^2(v_A^1)$, where $N^1(v_A^1)$ represents the number of 1-hop neighbors, and $N^2(v_A^1)$ represents the number of 2-hop neighbors in the first snapshot. Given

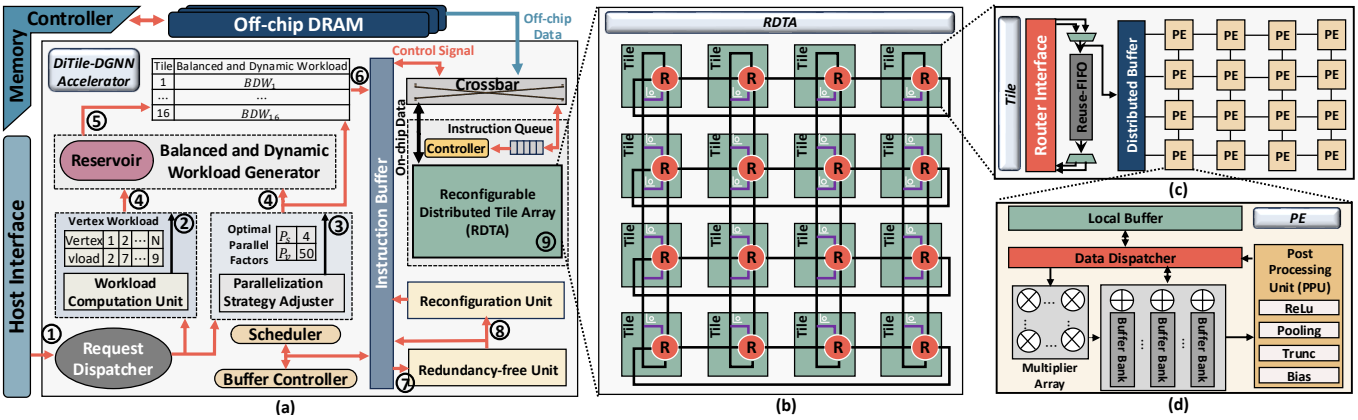


Figure 5: (a) The overview of DiTile-DGNN accelerator, (b) the reconfigurable and distributed tile array, (c) the microarchitecture of the proposed tile, (d) the microarchitecture of the proposed PE.

that $N^1(v_A^1) = 3$ and $N^2(v_A^1) = 1$, the total workload of vertex A in the first snapshot is 7. Thus, the total workload of vertex A across all snapshots ($vload(A)$) is 32.

Once the workloads for all vertices are computed, they are sorted in descending order to prioritize vertices with higher computational demands for allocation. For example, as shown in Figure 4, the workloads for all vertices are sorted in descending order as follows: $vload(C) = 34$, $vload(A) = 32$, $vload(E) = 24$, $vload(F) = 20$, $vload(B) = 19$, $vload(D) = 16$, $vload(G) = 6$. The sorting algorithm ensures an even distribution of vertices across tiles. The sorted workloads are then allocated to distributed tiles using a round-robin method (line 10), effectively balancing the workloads across tiles. This process simultaneously divides the vertices into load-balanced groups(BDW), where each group contains P_s snapshots, and each snapshot comprises P_v vertices. For instance, with the parallel strategy where each group consists of two snapshots and each snapshot contains four vertices, the load-balanced groups are divided as follows: $BDW_{\{1,2,3,4\}}$, as shown in Figure 4.

6 Proposed Accelerator Overview

We present an overview of the proposed accelerator in Figure 5 (a), which includes four distinct designs to support the proposed workload partitioning and dataflow. The **Workload Computation Unit** calculates the workload of each vertex during DGNN execution. The **Parallelization Strategy Adjuster** determines the parallelization strategy, balancing the number of snapshots and vertex counts to enhance efficiency. The **Balanced and Dynamic Workload Generator** produces balanced and dynamic workloads based on load information and the selected parallelization strategy, directing these workloads to the reconfigurable, distributed tile array. Finally, the **Reconfigurable and Distributed Tile Array** executes the balanced and dynamic workloads, pipelining both the GNN and RNN kernels to maximize performance.

We illustrate the specific function of each module following the example shown in Figure 5. First, the host (e.g., CPU) sends requests to the request dispatcher (Step ①). The control unit then dispatches DGNN model information and graph metadata to the Workload Computation Unit and the Parallelization Strategy Adjuster. For a dynamic graph DG, the Workload Computation Unit calculates the workload for each vertex during DGNN execution (Step ②). Simultaneously, using the dynamic graph DG and hardware resource information, the Parallelization Strategy Adjuster determines an efficient parallelization strategy to distribute snapshots and their partitions among tiles to minimize communication overhead (Step ③). Next, the load information and efficient parallel factors are sent to the Balanced and Dynamic Workload Generator (Step ④). This process yields a balanced and dynamic logical partition, denoted as $BDW = BDW_1, BDW_2, ..., BDW_N$. The partition is then stored in the Balanced and Dynamic Workload Reservoir (Step ⑤). In each iteration, the reservoir produces balanced and dynamic workloads for each tile (Step ⑥). These workloads are then routed to the Redundant-Free Unit, which eliminates communication and computation redundancy between consecutive snapshots (Step ⑦).

Furthermore, the generated balanced and dynamic workload, the redundant-free strategy, and graph metadata are sent to the Reconfiguration Unit (Step ⑧). This unit configures the Network-on-Chip (NoC) to support various communication patterns required by different types of data exchanges, including temporal, spatial, and reuse communication within the Reconfigurable and Distributed Tile Array (RDTA) (Step ⑨). Once the configuration is complete, the instruction dispatcher begins issuing instructions, as in conventional accelerators. The RDTA then executes the DGNN, pipelining the GNN and RNN kernels throughout each iteration.

6.1 Reconfigurable and Distributed Tile Array

To address the diverse communication and dataflow requirements of DGNNs, we propose a Reconfigurable and Distributed Tile Array with a 4×4 array of tiles. The tiles are interconnected by a reconfigurable interconnect. For example, as shown Figure 5 (b), the interconnect design utilizes a ring topology for horizontal connections and a combination of ring topology and reconfigurable links (Re-Link) for vertical connections. The horizontal rings are configured to handle regular communication to exchange the temporal dependency for RNN kernel and reuse computations between snapshots, while vertical links support irregular communication for spatial dependency within each graph snapshot.

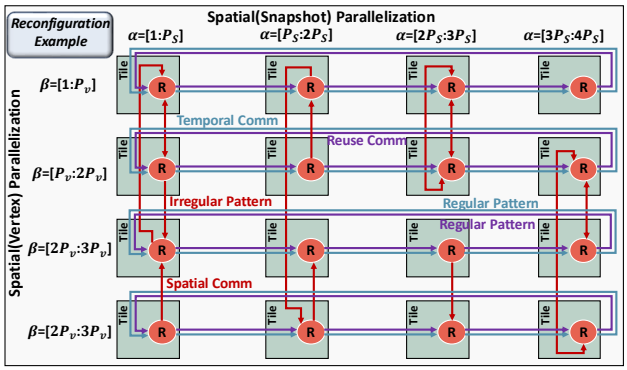


Figure 6: An example for proposed dataflow and topology configuration.

6.1.1 Proposed Dataflow Mapping for DGNNs. Mapping dataflows to the reconfigurable tile array is another critical step to determine the data locality and communication patterns. Conventional dataflow mapping [4, 10, 40] only targets a single model, which has limited applicability for DGNN models with combined models. The DGNN dataflow mapping consists of two parts - mapping partitioned snapshots and vertices for both RNN and GNN kernels. As the intermediate data between RNN and GNN is relatively larger, each tile has both RNN and GNN kernels to accommodate the intermediate data to eliminate inter-tile communication. Specifically, RNN and GNN kernels are duplicated in each row of tiles. As such, as shown in Figure 6, the example illustrates the data mapping of snapshots on a 4×4 tile array, where P_s and P_v are the parallelism factors for snapshots and vertices. α represents the snapshots assigned to distributed tiles, and β represents the vertices assigned to distributed tiles. Since snapshots are spatially parallelized along the horizontal direction, horizontally circulating RNN output and GNN intermediate features could increase data reuse. On the other hand, vertices in each snapshot are parallelized vertically, aggregating vertices from remote tiles located in the same column of the tile array. As such, we restrict the irregular communication patterns within one dimension of the tile array, preventing worst-case data transfers proportional to the network diameter.

Based on the dataflow mapping, the reconfigurable interconnect design is to support both regular and irregular dataflows in DGNN workloads. As shown Figure 5 (b), the interconnect employs a dual-layer topology with horizontal links based on a ring topology and vertical links that combine ring topology with reconfigurable links (Re-Link). Horizontal ring links handle predictable and regular communication patterns, including temporal communication and reuse communication, ensuring low-latency data transfer between adjacent routers. In contrast, vertical communication leverages the flexibility of the Re-Link architecture to address the irregular data exchanges associated with spatial dependencies. Re-Link consists of simple transistors that dynamically enable or disable bypass connections between non-adjacent routers, effectively minimizing signal interference, communication distances, and hop counts.

6.1.2 Proposed Tile and PE Architecture. Each tile integrates the following components: a distributed buffer, a router interface, a 4×4 array of processing elements (PEs), and a reuse First-in-First-Out (FIFO) buffer, as shown in Figure 5 (c). The tile connects to its router through a simplified router interface, significantly reducing complexity and the router's radix. Internally, the PEs in each tile are connected using a mesh topology to facilitate intra-tile communication and computation. The reuse FIFO acts as a double buffer, supporting inter-PE communication and data exchange between distributed buffers within the tile. This design minimizes off-chip memory accesses by enabling inter-tile data reuse, allowing locally-stored data to be shared efficiently across tiles. Each PE within a tile comprises a local buffer, a data dispatcher, a MAC array, and specialized processing units (e.g., ReLU), as depicted in Figure 5 (d). This design ensures efficient local computation while supporting flexible data exchange between PEs. The local buffer stores intermediate data, enabling seamless integration with the reuse FIFO and minimizing reliance on external memory.

Table 1: 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

7 Evaluation

7.1 Evaluation Setup

Accelerator Simulator : We built a cycle-accurate simulator to measure the performance of the DiTile-DGNN accelerator. As such, our simulator can accurately capture the varying graph connectivity and snapshots as well as their impact on DGNN performance. We faithfully implemented their respective characteristics and fine-tuned the models' performance characteristics to match the performance metrics. 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 DiTile-DGNN accurately captures the redundancy-free and dynamic parallelization strategy, balance-aware workload optimization, and distinct system configurations. As CommonGraph and MEGA [5, 12] have identified the high computational overhead associated with graph deletion operations, we also transform expensive deletion operations into addition operations by leveraging the mutually inclusive graph structure across snapshots. 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[36]. The overall execution time is determined by overlapping the off-chip communication time with the on-chip execution time, while accounting for system configuration overheads and control signal delays. The on-chip execution time is further refined by overlapping the on-chip communication latency with the computation latency.

The simulator counts the required amount of on/off-chip communications and computations, which is used to estimate the related energy consumption according to the analytical model proposed in [19]. Additionally, to accurately estimate the area consumption, we

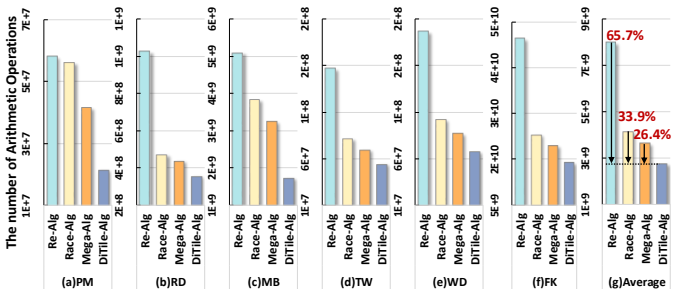


Figure 7: Comparison of the number of arithmetic operations for each dataset with baseline and proposed algorithms.

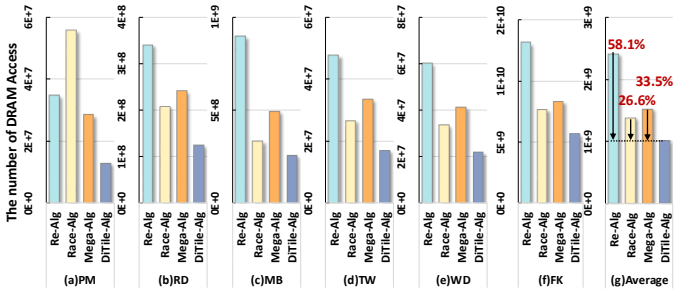


Figure 8: Comparison of the number of the DRAM access for each dataset with baseline and proposed algorithms.

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 set the clock frequency at 1 GHz. We use Cacti 6.0 [33] 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 DiTile-DGNN including 16 × 16 tiles interconnected by the proposed reconfigurable interconnect. Each tile integrates the following components: a distributed buffer, a router interface, a 4 × 4 array of PEs, and a reuse FIFO buffer. Each PE consists of a local buffer, a data dispatcher, a router interface, a post-processing unit(PPU), 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 DiTile-DGNN is 700MHz. The distributed buffer capacity of the DiTile-DGNN is 4MB. The reuse FIFO buffer capacity is 512KB. The local buffer capacity of each PE is 256KB.

Baselines : We compare the DiTile-DGNN with four GNN accelerators (ReaDy [20], DGNN-Booster [8], RACE [51], and MEGA [12]). The baseline accelerators are scaled to be equipped with the same number of multipliers and off-chip/on-chip bandwidth as the DiTile-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. ReaDy [20] and DGNN-Booster [8] employ a recomputation algorithm (Re-Alg) that fully recomputes all graph data whenever edges or vertices change over time. In contrast, RACE [51] adopts a redundancy-aware incremental algorithm (Race-Alg), which eliminates overlapping graph components, such as vertices and edges, between snapshots to reduce subsequent computational costs. MEGA [12] addresses the high computational overhead of graph deletion operations by using an algorithm (Mega-Alg) that transforms costly deletion operations into addition operations, leveraging the mutually inclusive graph structure across snapshots.

Datasets and Benchmarks: Table 1 illustrates six dynamic graphs used for evaluation in this paper with the number of vertices and edges [1–3, 15, 35, 37]. We consider one typical DGCN model [35], including GCN [26] and Long-Short-Term-Memory (LSTM) models [14]. The 32-bit floating-point representation is used in the evaluation, which proves to be sufficient for maintaining inference accuracy [24, 39, 45].

7.2 Arithmetic Operation Analysis

Figure7 provides a comparative analysis of arithmetic operations across various algorithms. The proposed DiTile algorithm achieves a substantial reduction in redundant computations during DGNN execution with the classic DGCN model[35]. On average, it reduces arithmetic operations by 65.7%, 33.9%, and 26.4% across multiple datasets compared to baseline methods, demonstrating its effectiveness in optimizing computational workloads. The algorithm's efficiency stems from several key innovations. Similar to Race-Alg [51], the DiTile algorithm leverages graph dissimilarities between snapshots, processing only the evolved graph structures and avoiding unnecessary recomputation. Additionally, akin to MEGA-Alg [12], it transforms expensive deletion operations into addition operations by exploiting the mutually inclusive graph structure across snapshots, significantly reducing the computational overhead of dynamic graph updates. Moreover, the RNN kernel within the DiTile accelerator selectively processes a limited set of output features from the GNN kernel, further reducing unnecessary computations. These optimizations collectively contribute to the algorithm's enhanced performance and its ability to efficiently reduce computational overhead in DGNN execution.

7.3 Off-chip DRAM Access Analysis

Figure 8 presents a comparison of DRAM access volumes for various algorithms across multiple datasets, where lower values indicate better performance. The DRAM access volume includes memory access for weights, adjacency matrix, input features, intermediate features, and output features. The proposed algorithm consistently outperforms the baseline approaches, achieving average reductions in DRAM access of 58.1%, 26.6%, and 33.5%. These results highlight the proposed algorithm's effectiveness in significantly reducing off-chip DRAM access.

The reduction in DRAM access is achieved through several key strategies. First, the proposed algorithm incorporates an efficient tiling optimization that partitions the entire graph into appropriately sized subgraphs, minimizing overall DRAM access. Second, the

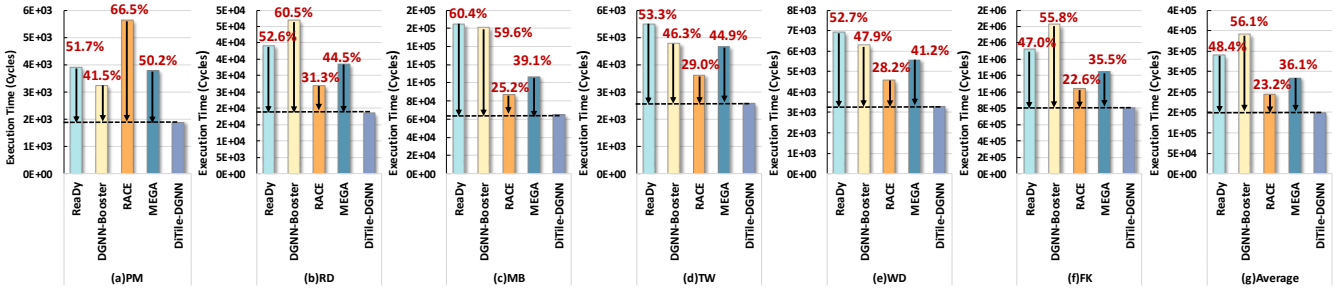


Figure 9: Execution time comparison of ReaDy, DGNN-Booster, RACE, MEGA, and DiTile-DGNN for different datasets.

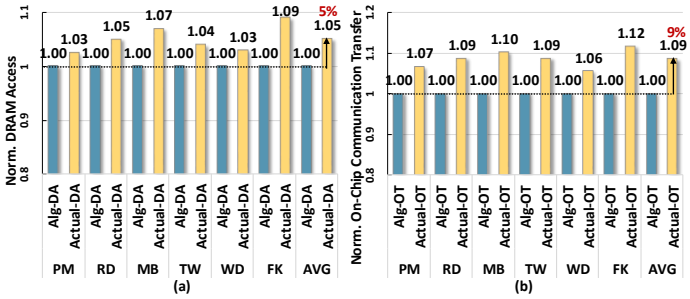


Figure 10: (a)Comparison of the normalized estimated off-chip DRAM access and actual DRAM access(Normalized to the estimated off-chip DRAM access) (b)Comparison of the normalized estimated on-chip data transfer and actual on-chip data transfer(Normalized to the estimated on-chip data transfer).

proposed redundancy-free dynamic parallelization strategy eliminates unnecessary redundant DRAM access by focusing exclusively on necessary computations, thereby further reducing memory traffic. Third, the proposed workload balance optimization ensures even data distribution across distributed tiles, enhancing on-chip memory utilization and maximizing data reuse. Together, these optimizations not only reduce off-chip memory access but also improve overall computational efficiency.

In contrast, baseline methods such as Re-Alg, Race-Alg, and Mega-Alg show notable limitations. Re-Alg performs repetitive computations for all snapshots without optimizing the redundancy elimination process, resulting in substantial redundant DRAM access. Race-Alg improves upon Re-Alg by eliminating redundant computations for vertices with identical output and intermediate features of the GNN kernel across snapshots. Mega-Alg reduces some redundant computations by addressing vertices with identical output features across snapshots but does not address redundancies related to intermediate features. Moreover, none of the baselines optimize the graph tiling process, and uneven data distribution across tiles continues to limit on-chip memory utilization, leading to increased DRAM access. Overall, the proposed algorithm's combination of efficient tiling, redundancy elimination, and workload balancing offers a robust solution for minimizing DRAM access.

7.4 Performance Analysis

Figure 9 illustrates the execution time of the DiTile-DGNN compared to previous approaches, measured by the total number of execution cycles. Baseline architectures, ReaDy and DGNN-Booster employ the Re-Alg, RACE utilizes the Race-Alg, and MEGA adopts the Mega-Alg. The overall execution time includes overlapping off-chip communication with on-chip execution while considering system configuration overheads and control signal delays. The DiTile-DGNN achieves significant performance improvements, with an average execution time reduction of 48.4%, 56.1%, 23.2%, and 36.1% across multiple GNN datasets compared to the baselines. These enhancements are attributed to three main factors: reduced computation, optimized off-chip communication, and efficient on-chip communication.

The reduction in computation is achieved through the proposed redundancy-free dynamic parallelization strategy, which eliminates redundant operations and reduces overall computational complexity. By focusing on necessary computations, the algorithm minimizes unnecessary processing overhead, contributing directly to reduced execution cycles. The off-chip communication optimization is realized by eliminating all memory transactions unrelated to graph updates between consecutive snapshots. Moreover, the graph tiling technique divides the entire graph into appropriately sized subgraphs, reducing the overall DRAM access and ensuring efficient memory utilization. The on-chip communication efficiency is enhanced through three complementary strategies. First, the proposed parallelism optimization identifies efficient parallelization strategies and parallel factors to reduce costly inter-tile communication. To assess how closely DiTile-DGNN approaches the estimated data transfers, we performed a detailed comparison of estimated and actual data movement with WD dataset. The estimated data movement refers to values calculated from our analytical model. We compare the normalized estimated off-chip DRAM access (Alg-DA) with the actual off-chip DRAM access (Actual-DA). As shown in Figure 10 (a), the actual off-chip access exceeds the estimated minimum by only 5% on average. This is because, in theory, we assume that subgraphs within the same snapshot share identical sparsity characteristics, whereas in practice, sparsity variations exist across subgraphs. We also compare the normalized estimated on-chip data transfer (Alg-OT) and actual on-chip data transfer (Actual-OT). The results shown in Figure 10 (b) indicate that the actual on-chip data transfer exceeds the estimated lower bound by 9% on average. This is due to the theoretical assumption that subgraphs across snapshots contain identical numbers of vertices and edges, whereas, in practice, even highly similar snapshots exhibit slight variations in graph structure. Second, the proposed workload balance optimization ensures balanced workload distribution

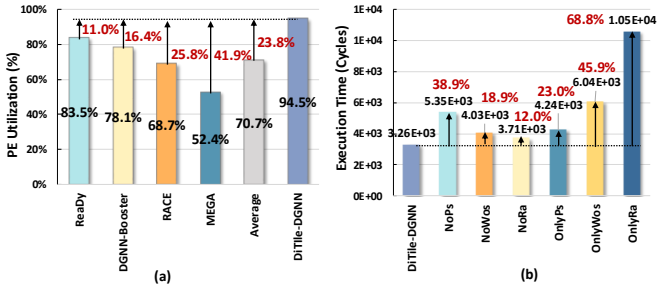


Figure 11: (a)PE utilization and (b)Execution time comparison of DiTile-DGNN vs. baseline variants—NoPs (without parallelism strategy), NoWos (without workload optimization), NoRa (without reconfigurable architecture), OnlyPs (only parallelism strategy), OnlyWos (only workload optimization), and OnlyRa (only reconfigurable architecture).

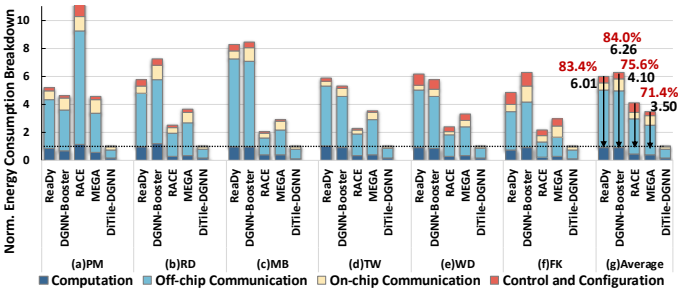


Figure 12: Comparison of the Normalized Energy Consumption Breakdown for different datasets (Normalized to the energy consumption of the DiTile-DGNN).

across tiles and minimizes synchronization overhead. Third, the reconfigurable interconnect design provides scalable and flexible communication tailored for DGNN workloads. This approach minimizes hop counts, signal interference, and communication latency, ensuring adaptability to varying workload demands.

The proposed design DiTile-DGNN performs 1.9–2.5× better than ReaDy, 1.7–2.7× better than DGNN-Booster, 1.3-3.0× better than RACE, and 1.6-2.1× better than MEGA. 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.

To evaluate the computational efficiency of DiTile-DGNN, we analyze its PE utilization and compare it with baseline accelerators. Our results shown in Figure 11 (a) indicate that DiTile-DGNN achieves a 23.8% average improvement in PE utilization over baseline accelerators on WD dataset. This improvement stems from two key factors: homogeneous architecture and proposed workload optimization strategy. The uniform design of our compute tiles allows for a more balanced workload distribution, reducing idle cycles and enhancing overall PE usage. By dynamically adjusting workload mapping, our accelerator effectively mitigates load imbalance caused by varying graph sparsity, ensuring consistently high utilization. These findings highlight the effectiveness of our architectural choices and workload management strategies in maximizing hardware efficiency.

7.5 Ablation Study

To evaluate the effectiveness of the proposed contributions in enhancing performance, we conduct an ablation study by incrementally removing or isolating each component of our design. The three key contributions include: proposed parallelism strategy, proposed workload optimization strategy, and proposed reconfigurable distributed-tiled accelerator architecture. We compare the execution time of our DiTile-DGNN accelerator against six baseline variants: DiTile-DGNN without the parallelism strategy (NoPs), DiTile-DGNN without the workload optimization strategy (NoWos), DiTile-DGNN without the reconfigurable architecture (NoRa), an accelerator implementing only the parallelism strategy (OnlyPs), an accelerator implementing only the workload optimization strategy (OnlyWos), and an accelerator implementing only the reconfigurable architecture (OnlyRa). Figure 11 (b) presents the execution time comparison, demonstrating the impact of each component on overall performance. Compared to DiTile-DGNN, removing the parallelism strategy (NoPs) results in a 38.9% increase in execution time, indicating that efficient data reuse and minimized off-chip/on-chip communication play a crucial role in accelerating DGNN computations. The absence of workload optimization (NoWos) increases execution time by 18.9%, highlighting the importance of evenly distributing workloads to avoid underutilization of computational resources. Eliminating the reconfigurable architecture (NoRa) results in a 12.0% execution time increase, suggesting that flexible interconnects significantly reduce communication bottlenecks. Among the individual components, OnlyPs, OnlyWos, and OnlyRa increase execution time by 23.0%, 45.9%, and 68.8%, respectively. These results indicate that while each contribution independently enhances performance, their combined effect in DiTile-DGNN achieves the best efficiency. Notably, the standalone reconfigurable architecture (OnlyRa) yields the highest execution time increase, underscoring the necessity of jointly optimizing both computation and communication. This ablation study confirms that the proposed parallelism strategy, workload optimization, and reconfigurable architecture collectively contribute to significant performance improvements. Their integration in DiTile-DGNN ensures efficient parallel execution, balanced workloads, and scalable communication, making it well-suited for dynamic graph workloads.

7.6 Energy Efficiency Analysis

In the energy analysis, we assess the total energy consumption of the entire execution process, encompassing computation, on-chip communication, off-chip communication, and control and configuration overheads. Figure 12 provides a breakdown of the normalized energy consumption for the DiTile-DGNN compared to baseline approaches. The DiTile-DGNN demonstrates significant energy savings, achieving average reductions of 83.4%, 84.0%, 75.6%, and 71.4% across multiple datasets compared to the baselines. These values, normalized to the energy consumption of the DiTile-DGNN, underscore its improved energy efficiency. The primary contributors to

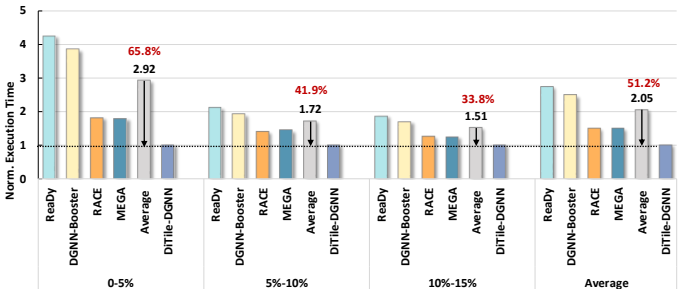


Figure 13: The sensitivity study of graph dissimilarity proportion between consecutive snapshots for the proposed and baseline accelerators (Normalized to the execution time of the DiTile-DGNN with same graph dissimilarity proportion).

these energy savings include reduced DRAM access, decreased computation, minimized inter-tile communication, and reduced on-chip communication latency. These improvements are driven by several key strategies, including the proposed redundancy-free dynamic parallelization strategy, the proposed workload balance optimization, and the reconfigurable NoC design, as detailed previously. Furthermore, the energy consumption for control and configuration accounts for less than 7% of the total energy consumption, reflecting the effectiveness of the proposed design in minimizing system overheads. In summary, the combination of reduced memory access, optimized computation, and efficient communication strategies establishes the DiTile-DGNN as a highly energy-efficient solution for DGNN workloads. This holistic approach ensures substantial energy savings without compromising performance.

7.7 Sensitivity Analysis

Since the proportion of dissimilarity between consecutive snapshots varies from 4.1% to 13.3%[51], we adjusted this proportion to demonstrate that DiTile-DGNN consistently outperforms baseline accelerators across different levels of dissimilarity. Figure13 illustrates the normalized execution time as the proportion of dissimilarity between consecutive snapshots increases from 0% to 15% using the Wikipedia dataset. The execution time of the baseline accelerators is normalized to the execution time of the DiTile-DGNN at the same graph dissimilarity ratio. The DiTile-DGNN achieves execution time reductions of 65.8%, 41.9%, and 33.8% compared to the baselines as the dissimilarity proportion changes from 0%-5%, 5%-10%, and 10%-15%, respectively. Although the performance speedup of DiTile-DGNN decreases as dissimilarity decreases, the proposed algorithm continues to eliminate a substantial number of redundant computations and communications while significantly reducing on-chip and off-chip communication consumption compared to prior approaches. From this study, we conclude that the performance gains of our proposed algorithm diminish as the dissimilarity increases, but DiTile-DGNN remains highly effective across varying levels of graph dissimilarity.

7.8 Area Consumption Analysis

Figure14 (a) shows the breakdown of the overall area of the DiTile-DGNN. This includes the tiles, on-chip buffer, reconfigurable interconnects, and logic components for control and configuration.

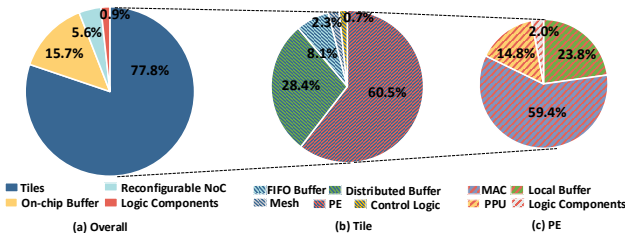


Figure 14: (a) Area breakdown of the DiTile-DGNN, (b) the proposed tile, and (c) the proposed PE

The tile array accounts for 77.8% of the total chip area. The on-chip buffer makes up 15.7% of the chip area. The reconfigurable interconnect uses 5.6% of the chip area. The controller's area consumption is negligible at 0.9% of the total chip area. Figure14 (b) provides a breakdown of area consumption for the proposed Tile. The PE array accounts for 60.5% of the total PE area, while the distributed buffer and reuse FIFO buffer account for 28.4% and 8.1% respectively. The mesh topology that connected the PE array account for 2.3% and router reconfigurable Muxes and local control logic have a negligible area consumption of 0.7% of the total tile area. Figure14 (c) provides a breakdown of area consumption for the proposed PE. The MACs array accounts for 59.4% of the total PE area, while the local buffer account for 23.8%. The local control logic have a negligible area consumption of 2.0% of the total PE area.

8 Conclusion

In this paper, we propose DiTile-DGNN, an efficient accelerator for large-scale DGNN execution. The proposed DiTile-GNN consists of a redundancy-free parallelism strategy, workload balance optimization, and a reconfigurale accelerator architecture. Specifically, we propose a redundancy-free framework that can efficiently find an efficient parallelism strategy that can fully eliminate the data redundancy between graph snapshots while minimizing the communication complexity. Additionally, we propose a workload balance optimization for combined GNN and DGNN models to enhance resource utilization and eliminate synchronization overhead between snapshots. Lastly, we propose a reconfigurable accelerator architecture, with a flexible interconnect, that can be dynamically configured in support of various DGNN dataflows. Our simulations demonstrate that DiTile-DGNN achieves 48.4%, 56.1%, 23.2%, and 36.1% reductions in execution time and 83.4%, 84.0%, 75.6%, and 71.4% improvements in energy efficiency compared to state-of-the-art accelerators, including ReaDy [20], DGNN-Booster [8], RACE [51], and MEGA [12], on average across multiple DGNN datasets. These results highlight the strength of our algorithm-architecture co-optimization approach, establishing DiTile-DGNN as a robust and scalable solution for efficient DGNN execution.

Acknowledgments

This research was partially supported by NSF grants CCF-1901165, CCF-195398, CCF-2131946, CCF-1901165, CCF-2441973, and CNS-2321224. We sincerely thank the anonymous reviewers for their excellent and constructive feedback.

References

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

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

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

[4] 2017. *NVDLA Deep Learning Accelerator*. http://nvdla.org

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

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

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

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

[9] Jinyin Chen, Xueke Wang, and Xuanheng Xu. 2022. GC-LSTM: Graph convolution embedded LSTM for dynamic network link prediction. *Applied Intelligence* (2022), 1–16.

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

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

[12] Chao Gao, Mahbod Afarin, Shafiur Rahman, Nael Abu-Ghazaleh, and Rajiv Gupta. 2023. MEGA Evolving Graph Accelerator. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 310–323.

[13] Tong Geng, Chunshu Wu, Yongan Zhang, Cheng Tan, Chenhao Xie, Haoran You, Martin Herbordt, Yingyan Lin, and Ang Li. 2021. I-GCN: A graph convolutional network accelerator with runtime locality enhancement through islandization. In *In proceedings of IEEE/ACM international symposium on microarchitecture*. 1051–1063.

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

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

[16] Mingyu Guan, Anand Padmanabha Iyer, and Taesoo Kim. 2022. DynaGraph: dynamic graph neural networks at scale. In *Proceedings of the ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*. 1–10.

[17] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. *Advances in Neural Information Processing Systems* 30 (2017).

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

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

[20] Yu Huang, Long Zheng, Pengcheng Yao, Qinggang Wang, Haifeng Liu, Xiaofei Liao, Hai Jin, and Jingling Xue. 2022. 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 (2022), 3567–3578.

[21] Ajay Kumar Jaiswal, Shiwei Liu, Tianlong Chen, Ying Ding, and Zhangyang Wang. 2023. Graph ladling: Shockingly simple parallel gnn training without intermediate communication. In *International Conference on Machine Learning*. PMLR, 14679–14690.

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

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

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

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

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

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

[28] Kai Lei, Meng Qin, Bo Bai, Gong Zhang, and Min Yang. 2019. GCN-GAN: A non-linear temporal link prediction model for weighted dynamic networks. In *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*. 388–396.

[29] Hongxi Li, Zuxuan Zhang, Dengzhe Liang, and Yuncheng Jiang. 2024. K-Truss Based Temporal Graph Convolutional Network for Dynamic Graphs. In *Asian Conference on Machine Learning*. PMLR, 739–754.

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

[31] Shengwen Liang, Ying Wang, Cheng Liu, Lei He, LI Huawei, Dawen Xu, and Xiaowei Li. 2020. EnGN: A high-throughput and energy-efficient accelerator for large graph neural networks. *IEEE Trans. Comput.* 70, 9 (2020), 1511–1525.

[32] Osman Asif Malik, Shashanka Ubaru, Lior Horesh, Misha E Kilmer, and Haim Avron. 2021. Dynamic graph convolutional networks using the tensor M-product. In *Proceedings of the SIAM International Conference on Data Mining (SDM)*. 729–737.

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

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

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

[36] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. 2011. DRAMSim2: A cycle accurate memory system simulator. *IEEE Computer Architecture Letters* 10, 1 (2011), 16–19.

[37] Ryan Rossi and Nesreen Ahmed. 2015. The network data repository with interactive graph analytics and visualization. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 29.

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

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

[40] Yakun Sophia Shao, Jason Clemons, Rangharajan Venkatesan, Brian Zimmer, Matthew Fojtik, Nan Jiang, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, et al. 2019. Simba: Scaling deep-learning inference with multi-chip-module-based architecture. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 14–27.

[41] Lubos Takac and Michal Zabovsky. 2012. Data analysis in public social networks. In *Proceeding of the International Scientific Conference and International Workshop Present Day Trends of Innovations*, Vol. 1.

[42] Stylianos I Venieris, Christos-Savvas Bouganis, and Nicholas D Lane. 2022. Multi-DNN accelerators for next-generation AI systems. *arXiv preprint arXiv:2205.09376* (2022).

[43] Cheng Wan, Youjie Li, Ang Li, Nam Sung Kim, and Yingyan Lin. 2022. Bns-gcn: Efficient full-graph training of graph convolutional networks with partition-parallelism and random boundary node sampling. *arXiv preprint arXiv:2203.10983* (2022).

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

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

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

[47] Jiaqi Yang, Hao Zheng, and Ahmed Louri. 2024. Aurora: A Versatile and Flexible Accelerator for Graph Neural Networks. In *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 890–902. https://doi.org/10.1109/IPDPS57955.2024.00084

[48] Jiaqi Yang, Hao Zheng, and Ahmed Louri. 2025. I-DGNN: A Graph Dissimilarity-based Framework for Designing Scalable and Efficient DGNN Accelerators. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 1038–1051. https://doi.org/10.1109/HPCA61900.2025.00081

[49] Fangzhou Ye, Lingxiang Yin, Amir Ghazizadeh Ahsaei, and Hao Zheng. 2024. EGMA: Enhancing Data Reuse and Workload Balancing in Message Passing GNN Acceleration via Gram Matrix Optimization. In *Proceedings of the 61st ACM/IEEE Design Automation Conference* (San Francisco, CA, USA) *(DAC '24)*.

Association for Computing Machinery, New York, NY, USA, Article 304, 6 pages. https://doi.org/10.1145/3649329.3655962

[50] Lingxiang Yin, Sanjay Gandham, Mingjie Lin, and Hao Zheng. 2024. SCALE: A Structure-Centric Accelerator for Message Passing Graph Neural Networks. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 580–593. https://doi.org/10.1109/MICRO61859.2024.00050

[51] Hui Yu, Yu Zhang, Jin Zhao, Yujian Liao, Zhiying Huang, Donghao He, Lin Gu, Hai Jin, Xiaofei Liao, Haikun Liu, et al. 2023. RACE: An Efficient Redundancy-aware Accelerator for Dynamic Graph Neural Network. *ACM Transactions on Architecture and Code Optimization* 20, 4 (2023), 1–26.

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

[53] Yingnan Zhao, Ke Wang, and Ahmed Louri. 2024. OPT-GCN: A Unified and Scalable Chiplet-Based Accelerator for High-Performance and Energy-Efficient GCN Computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 43, 12 (2024), 4827–4840. https://doi.org/10.1109/TCAD.2024.3401543

[54] Yingnan Zhao, Ke Wang, Jiaqi Yang, and Ahmed Louri. 2024. An Efficient Hardware Accelerator Design for Dynamic Graph Convolutional Network (DGCN) Inference. In *Proceedings of the 61st ACM/IEEE Design Automation Conference* (San Francisco, CA, USA) *(DAC '24)*. Association for Computing Machinery, New York, NY, USA, Article 324, 6 pages. https://doi.org/10.1145/3649329.3658254

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

[56] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. 2019. Aligraph: A comprehensive graph neural network platform. *arXiv preprint arXiv:1902.08730* (2019).