TaGNN: An Efficient Topology-aware Accelerator for High-performance Dynamic Graph Neural Network

Hui Yu[*]
National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology
Huazhong University of Science and Technology
Wuhan, China
huiy@hust.edu.cn

Yu Zhang
National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology
Huazhong University of Science and Technology
Wuhan, China
zhyu@hust.edu.cn

Ligang He
University of Warwick
Coventry, United Kingdom
ligang.he@warwick.ac.uk

Bing Peng
National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology
Huazhong University of Science and Technology
Wuhan, China
bpeng@hust.edu.cn

Jin Zhao
National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology
Huazhong University of Science and Technology
Wuhan, China
zjin@hust.edu.cn

Zixiao Wang
National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology
Huazhong University of Science and Technology
Wuhan, China
zwang62@hust.edu.cn

Hao Qi
National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology
Huazhong University of Science and Technology
Wuhan, China
theqihao@hust.edu.cn

Hai Jin
National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology
Huazhong University of Science and Technology
Wuhan, China
hjin@hust.edu.cn

[*]Department of Electronic and Computer Engineering, The Hong Kong University of Science and Technology, Hongkong, China.

ACM ISBN 979-8-4007-1466-5/25/11
https://doi.org/10.1145/3712285.3759818

Abstract

Dynamic Graph Neural Networks (DGNNs) have become powerful tools for analyzing continuously evolving graph data, combining *Graph Neural Network* (GNN) models to extract structural information and *Recurrent Neural Network* (RNN) models to capture temporal semantics across snapshots. However, despite extensive research, existing DGNN solutions still face significant limitations, particularly *low data parallelism* caused by their snapshot-by-snapshot execution. This sequential paradigm exacerbates memory contention due to irregular, repeated vertex feature accesses and enforces strict temporal dependencies. In this paper, we propose *TaGNN*, an efficient topology-aware DGNN accelerator that addresses these performance bottlenecks. Specifically, we present a *topology-aware*

concurrent execution approach into the accelerator design that calculates the final features of affected vertices while ensuring that unaffected vertices are loaded and computed only once per layer across multiple snapshots, maximizing data parallelism while minimizing memory usage. *TaGNN* employs a cache-friendly storage format that compactly organizes affected vertices across multiple snapshots by their timestamps and topological characteristics, reducing indexing overhead and enhancing data locality. In addition, *TaGNN* further proposes a similarity-aware cell skipping strategy to alleviate the stringent temporal data dependencies. It selectively reuses the RNN results from the previous snapshot to bypass RNN operations in the current snapshot when the output features of the GNN module across two consecutive snapshots are similar, achieving significant efficiency gains with minimal accuracy loss. We have implemented and assessed *TaGNN* on a Xilinx Alveo U280 FPGA card. Experimental results show that *TaGNN* achieves average speedups of 535.2x and 84.3x, and energy savings of 742.6x and 104.9x over state-of-the-art software DGNNs on Intel Xeon CPUs and NVIDIA A100 GPUs, respectively. Compared to leading DGNN accelerators (i.e., DGNN-Booster, E-DGCN, and Cambricon-DG), *TaGNN* delivers average speedups of 13.5x, 10.2x, and 6.5x, and energy savings of 15.9x, 11.7x, and 7.8x, respectively.

CCS Concepts

• **Computer systems organization** → *Parallel architectures*; • **Computing methodologies** → Machine learning approaches.

Keywords

DGNN inference, dynamic graphs, hardware architecture

ACM Reference Format:
Hui Yu, Yu Zhang, Ligang He, Bing Peng, Jin Zhao, Zixiao Wang, Hao Qi, and Hai Jin. 2025. TaGNN: An Efficient Topology-aware Accelerator for High-performance Dynamic Graph Neural Network. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '25), November 16–21, 2025, St Louis, MO, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3712285.3759818

1 Introduction

Dynamic Graph Neural Networks (DGNNs) [29] have garnered significant attention as a powerful tool for analyzing dynamic graph-structured data. DGNNs possess the unique ability to process a series of graph snapshots, learning time-varying vertex representations that capture the dynamic patterns and evolutionary processes within the graph structure. This inherent capacity to model the temporal aspects of graphs renders DGNNs particularly well-suited for a wide range of the forefront of various applications, such as dynamic node classification [20, 21], dynamic link prediction [8, 25], and anomaly detection in dynamic graphs [5, 31, 34].

To improve the performance of DGNN models, many software and hardware solutions have been recently proposed, and they strive to make full use of high sequential memory bandwidth [7, 39], reduce the data transfer volume and time [5, 31], enhance data locality [19, 30], etc. Despite extensive efforts to optimize DGNN models, as summarized in Table 1, prevailing solutions continue to face challenge related to low data parallelism, primarily caused by excessive irregular memory access and stringent temporal data dependencies, both of which stem from the following reasons.

Table 1: A comparative analysis of state-of-the-art solutions for supporting DGNN inference

Solutions	Dynamic graph	Alleviate data dependencies	Better data locality	High data parallelism
DGL [32]	✗	✗	✗	✗
DGNN-Booster [7]	✓	✗	✗	✗
E-DGCN [46]	✓	✗	✗	✗
Cambricon-DG [40]	✓	✗	✓	✗
TaGNN	✓	✓	✓	✓

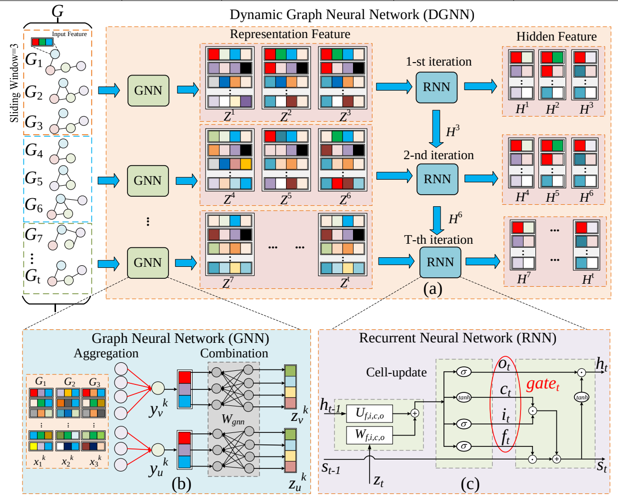


Figure 1: An illustrative example for DGNN inference with the sliding window parameter set to 3, including (a) the comprehensive execution flow, (b) a representative GNN module, and (c) a characteristic RNN module

First, existing solutions predominantly operate on isolated snapshot graphs, performing intact propagation computations per timestamp (i.e., snapshot-by-snapshot execution pattern). This leads to *excessive irregular memory access* overhead, as the graph structure is not only sparse but also dynamically changing. Specifically, these solutions need to frequently and inefficiently process the features of vertices during DGNN inference, however, the size of these features is typically too large to fit within on-chip memory (e.g., 512 or 1,024 dimensions [37, 39]), resulting in irregular off-chip communication. Worse still, an overwhelming majority of this off-chip communication involves redundant access to unaffected vertices (i.e., vertices that maintain identical neighbors, features, and neighbors' features across multiple snapshots), which further exacerbates the inefficiency by consuming additional bandwidth and computational resources without improving inference efficiency.

Second, DGNN models incorporate substantial temporal components (e.g., LSTM [41]) for capturing the temporal semantics. It means the computation must adhere to the time sequence for capturing the temporal information, which results in *temporal data dependency* bottleneck. Specifically, the inter-snapshot and intra-snapshot temporal data dependency within the DGNN models collectively limit the computation parallelism from different dimensions, leading to low hardware utilization and suboptimal performance on parallel computing platforms. Furthermore, the traditional snapshot-by-snapshot execution pattern may exacerbate this situation, as it fails to exploit the potential parallelism across multiple snapshots, and instead processes each snapshot sequentially, causing severe underutilization of hardware resources.

To enhance the data parallelism of DGNNs, the most straightforward approach is to enable simultaneous inference of multiple snapshots (i.e., multi-snapshot execution pattern), as the GNN operations across different snapshots during DGNN inference are inherently independent of data dependencies [20, 21, 24]. However, directly extending existing solutions to support multi-snapshot inference presents two significant challenges. First, concurrently processing multiple snapshots requires a substantially larger memory footprint. For example, supporting four snapshots of the *Flickr* dataset in Table 2 would demand over 11.2 GB of memory, which exceeds the capacity of existing architectures. Second, although multi-snapshot execution pattern could improve the data parallelism of DGNN inference, the presence of irregular memory access and temporal data dependencies within DGNNs fundamentally constrains the achievable parallelism.

To address performance bottlenecks in DGNNs, our investigation uncovers two critical insights based on the unique characteristics of these models. First, we observe a significant overlap of vertices across multiple snapshots, with only a small fraction of vertices and their associated features being updated between consecutive snapshots [3, 5, 31]. This finding supports the implementation of a multi-snapshot execution pattern, which allows for concurrent processing of multiple graph inputs, thereby enhancing throughput by leveraging overlapping features across snapshots. Second, our analysis indicates that a substantial portion of computations involved in updating RNN cells may be excessive due to the inherent stability of DGNN models. We find a strong correlation between the similarity of final embeddings and output features of the GNN module across consecutive snapshots. This allows us to judiciously skip cell update computations for vertices with similar input features, reusing previous output results and effectively eliminating unnecessary computations and memory accesses with minimal impact on model accuracy. Based on these insights, we propose a *topology-aware concurrent execution approach* to achieve high data parallelism while reducing redundant memory usage and alleviating temporal data dependencies for DGNN inference.

To fully leverage our approach, we introduce *TaGNN*, a topology-aware DGNN accelerator designed for enhanced performance and energy efficiency in DGNN inference. *TaGNN* enables multi-snapshot execution by identifying affected vertices, which serve as root vertices to prefetch their neighbors impacted by graph updates, thus constructing the *affected subgraph*. This subgraph encapsulates all vertices and their neighbors affected across snapshots (i.e., affected vertices), allowing *TaGNN* to efficiently compute and load unaffected vertices only once, significantly reducing redundant memory footprint and massive off-chip communication. To further optimize parallel processing and achieve better data locality of the affected subgraph, *TaGNN* employs a cache-friendly *Overlap-Compressed Sparse Row* (O-CSR) storage format to effectively represents vertex and temporal information across multiple snapshots, capitalizing on the sparsity and temporal locality of graph data for efficient storage and rapid access. Additionally, *TaGNN* integrates a similarity-aware cell skip strategy to minimize unnecessary computations and address intra-snapshot and inter-snapshot temporal data dependencies. By evaluating the similarity of output features from the GNN module across two consecutive snapshots, *TaGNN* dynamically determines whether to perform cell updates for each vertex at current snapshot. When input features exhibit high similarity, *TaGNN* can skip the cell update computation and reuse the previous results, thereby significantly reducing massive unnecessary computations with negligible model accuracy loss.

TaGNN features an overlap-aware data loading mechanism designed to minimize data loading overhead through specialized hardware pipelines that operate in a dataflow style of parallelism. Additionally, TaGNN incorporates an adaptive data similarity computation scheme that utilizes dedicated hardware to efficiently calculate the similarity scores of affected vertices between consecutive snapshots. This information then guides subsequent cell updates, optimizing computation efficiency and mitigating data dependencies.

We have implemented and evaluated *TaGNN* on a Xilinx Alveo U280 FPGA card. Experimental results demonstrate that for DGNN inference, *TaGNN* achieves average speedups of 535.2x and 84.3x, and energy savings of 742.6x and 104.9x, compared to the cutting-edge software DGNN solutions running on an Intel Xeon CPU and an NVIDIA A100 GPU, respectively. Furthermore, *TaGNN* consistently outperforms leading DGNN inference accelerators, i.e., DGNN-Booster, E-DGCN, and Cambricon-DG, achieving average speedups of 6.5x, 10.2x, and 13.5x, and improvements in energy consumption of 7.8x, 11.7x, and 15.9x on average, respectively.

2 Background and Motivation

We review the background for DGNNs and undertake a comprehensive analysis to identify the gap between DGNNs and the analogous solutions in the domain, and describe our motivation.

2.1 Background of DGNN

Dynamic Graphs [5, 29, 38]. In the real-world scenarios [29, 44], graphs frequently evolve over time through the addition or deletion of vertices and edges, as well as feature mutations. Formally, a dynamic graph is defined as $G = \{G_1, G_2, \cdots, G_T\}$, where $G_t = (V_t, E_t, X_t)$ denotes the snapshot t. Here V_t is the set of vertices that be added/removed over time, E_t signifies the changing set of edges at snapshot t, and X_t represents the evolving vertex feature vectors across snapshots. Note that we exploit the CSR format [42, 43] to store each graph snapshot of the dynamic graph.

Dynamic Graph Neural Network (DGNN) [8, 20, 21, 24, 29, 31]. DGNN inference operates under a sliding window processing paradigm and may consist of multiple layers. As illustrated in Figure 1, DGNN inference has two key components: the *Graph Neural Network* (GNN) [9, 16] module and the *Recurrent Neural Network* (RNN) [22, 41] module, each tasked with processing individual graph snapshots. In Figure 1 (b), the GNN module extracts representative vector data for all vertices within a snapshot, generating the output feature Z^k for snapshot G_k. This process involves essential operations of aggregation and combination, after which the output feature is passed to the RNN module. As shown in Figure 1 (c), the RNN module synthesizes hidden feature representations that capture both the structural dynamics of the graph and its temporal characteristics, ultimately producing the final feature H^k.

We observe that the above DGNN methodology provides the flexibility to construct a DGNN model utilizing various implementations of GNN and RNN modules [8, 20, 21, 24]. While this paper focuses on GCN-based models due to their widespread use in DGNNs [29], *TaGNN* is highly versatile and adaptable to a broad range of DGNN models, including those that do not rely on RNNs.

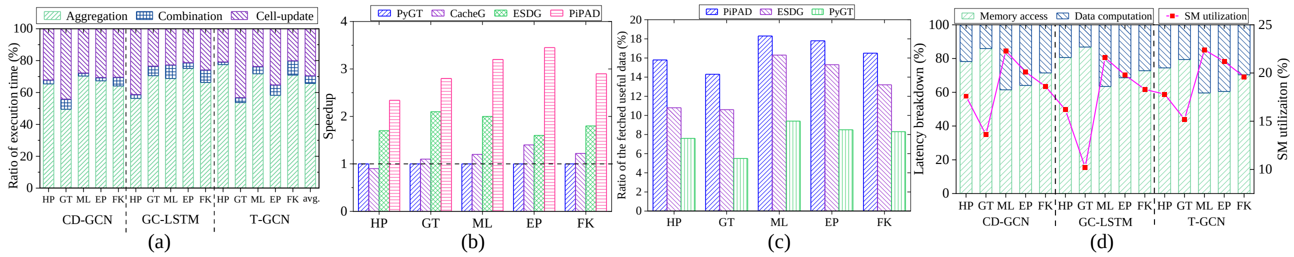


Figure 2: Studies of the performance of DGNN inference: (a) the execution time breakdown of PiPAD; (b) the execution time normalized to that of PyGT for T-GCN model; (c) the ratio of fetched useful data to all data accesses across four snapshots for the T-GCN model on different systems; (d) latency breakdown and SM utilization of PiPAD on NVIDIA Tesla A100

Table 2: The real-life dynamic graph datasets

Datasets	#*Vertices*	#*Edges*	#*Dimension*	#*Snapshots*	# *Granularity*
HepPh(HP)	28,090	1,543,901	172	243	1 day
Gdelt (GT)	7,398	238,765	248	288	1 month
MovieLens (ML)	9,992	1,000,209	500	100	4 days
Epinions (EP)	876,252	13,668,320	220	51	10 day
Flicker (FK)	2,302,925	33,140,017	162	134	1.5 days

2.2 Pitfalls of Exiting DGNN Solutions

Current solutions for DGNN inference suffer from significantly low data parallelism, primarily due to massive irregular memory access and temporal data dependencies. They typically employ a snapshot-by-snapshot execution paradigm, requiring the repeated and inefficient loading of vertex features via frequent off-chip transfers, which exacerbates memory contention and leads to substantial irregular access patterns. Moreover, the temporal data dependencies further constrain the degree of parallelism achievable in these solutions. These limitations collectively hinder the scalability and throughput of existing DGNN solutions.

To elucidate the aforementioned challenges, we conducted an evaluation of four cutting-edge DGNN software frameworks: PyGT [27], CacheG [19], ESDG [5], and PiPAD [31]. These frameworks adhere to the snapshot-by-snapshot execution paradigm and are assessed by executing various DGNN models across different datasets, as depicted in Table 2. An in-depth discussion of the DGNN models and system configurations is provided in Section 5.

Figure 2 (a) presents the execution time breakdown across five datasets using three DGNN models: CD-GCN [21], GC-LSTM [8], and T-GCN [45]. While execution times vary based on the dataset and model combination, the aggregation and update (comprising both the GNN module's combination and the RNN module's cell-update operations) operations are consistently time-consuming. For instance, the execution time for the aggregation can reach 77.48% for T-GCN on the HP dataset, while it may drop to 49.51% for CD-GCN on the GT dataset. Therefore, optimizing both the aggregation and update operations is essential for enhancing DGNN execution efficiency. As depicted in Figure 2 (b), although PiPAD outperforms other DGNN systems in all evaluated scenarios, it remains suboptimal due to two primary issues:

Irregular Memory Access. Current solutions for DGNN inference predominantly utilize the snapshot-by-snapshot execution paradigm (i.e., sliding window = 1) [5, 13, 19, 27, 31, 39]. Under this execution pattern, each graph snapshot is processed sequentially, leading to significant low data parallelism. Although some methods [4, 5, 19, 31, 39] attempt to leverage various caching strategies to reduce off-chip communication, they remain constrained by excessive redundant accesses and unnecessary data transfers, resulting in suboptimal performance. Figure 4 (a) illustrates this issue, showing that vertex v_1 shares identical features, 1-hop neighbors (i.e., v_0 and v_2), and neighbors features across three consecutive snapshots. Consequently, these neighbors, features, and weight matrices (e.g., W_{RNN}) are accessed repeatedly, and computations (e.g., aggregation operations) are unnecessarily repeated three times. Therefore, existing solutions lead to substantial redundant data accesses in DGNN processing. As shown in Figure 2 (c), although PiPAD outperforms other methods in all tested scenarios, over 81.7% of its data accesses remain redundant.

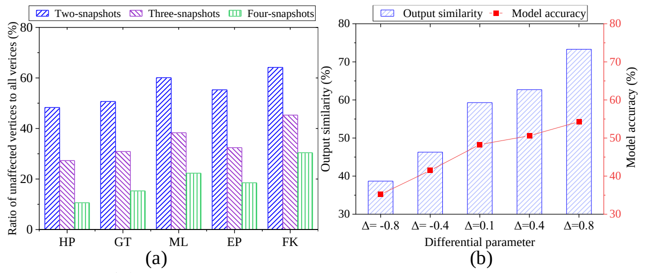


Figure 3: (a) the ratio of unaffected vertices to all vertices across different snapshots; (b) impact of output feature difference on final feature similarity and model accuracy

Temporal Data Dependency. DGNN inference encounters both inter-snapshot and intra-snapshot temporal data dependencies that significantly limit parallelism and hardware resource utilization. First, the inter-snapshot dependency within the RNN module requires that the computation of the current snapshot relies on the hidden state from the previous snapshot, thereby preventing parallelization across snapshots. Second, the intra-snapshot dependency between the GNN and RNN modules dictates that RNN computations can only begin after the GNN has completed feature extraction, hindering effective pipelining between these modules. Collectively, these dependencies impose severe constraints on the parallelism of DGNNs, resulting in low hardware utilization and suboptimal performance. As shown in Figure 2 (d), the SM utilization of PiPAD remains below 22.3% across various datasets and DGNN models. Furthermore, significant memory access overhead (accounting for on average 70.4% of the total execution time) exacerbates these inefficiencies. Thus, the interplay of temporal data dependencies and redundant accesses presents a formidable challenge to the efficient execution of DGNN inference on modern hardware architectures.

2.3 Our Insights

Figure 3 presents the outcomes of statistical analyses conducted on DGNN models. From these studies, we derive two critical insights concerning DGNN inference, which offer avenues to overcome the constraints posed by current architectures and to improve inference performance significantly.

Insight One. *In real-world dynamic graphs, the substantial overlaps in features and topology across multiple snapshots present a significant opportunity to enhance the throughput and data parallelism of DGNN inference through concurrent processing of these snapshots.* Figure 3 (a) demonstrates that unaffected vertices across three and four snapshots account for an average of 27.3%-45.3% and 10.6%-24.4% of all vertices, respectively, across various real-world dynamic graphs. This indicates that only a small subset of vertices is affected by graph updates, underscoring the potential for concurrent processing to improve DGNN inference performance. This highlights the advantages of leveraging overlaps across snapshots to reduce redundant memory footprint. Additionally, our findings reveal that processing three snapshots in a single batch is, on average, approximately 1.6x faster than sequential processing.

Insight Two. *In the RNN module of DGNN inference, the similarity of the final cell-update features across two consecutive snapshots correlates positively with the similarity of output features from the GNN module.* Our correlation analysis indicates a strong stability in DGNN inference, as shown in Figure 3 (b), which illustrates the relationship among output feature differences, final feature similarities, and model accuracy. The output feature difference Δ is computed by measuring the cosine similarity [18] between the output features of the same vertex across consecutive snapshots. As depicted in Figure 3 (b), increasing Δ from -0.6 to 0.6 increases output feature similarity from 35.2% to 73.3%. This positive correlation highlights the stability of the RNN module, where similar output features correspond to similar final features across snapshots. However, traditional approximation methods neglect the influence of graph structure on vertex features, directly applying these methods to DGNNs may result in the loss of critical structural information, adversely affecting the model accuracy. For instance, Figure 3 (b) demonstrates that the T-GCN model achieves an accuracy below 54.3% on the FK dataset (compared to the baseline accuracy-58.4%), even when Δ is set to 0.8.

3 Overview of Our Approach

Leveraging these two insights, we propose a topology-aware concurrent execution approach to support multi-snapshot DGNN inference, enhancing data parallelism while significantly reducing redundant memory usage and alleviating temporal data dependencies. This section presents our main ideas and discusses the limitations of software-only implementations.

3.1 Topology-aware Concurrent Execution

In this subsection, we present the core concept of our topology-aware concurrent execution approach, which efficiently identifies the *affected subgraph* and *unaffected vertices* across multiple snapshots (i.e., sliding windows ≥ 2). It facilitates the concurrent computation of multiple snapshots, thereby enhancing the throughput of DGNN inference. In our approach, the vertices are classified into three categories: *unaffected vertices*, *stable vertices* (i.e., a vertex whose feature remains unchanged across multiple consecutive snapshots while its neighbors change), and *affected vertices*. In other words, the set of unaffected vertices is a subset of the stable vertices.

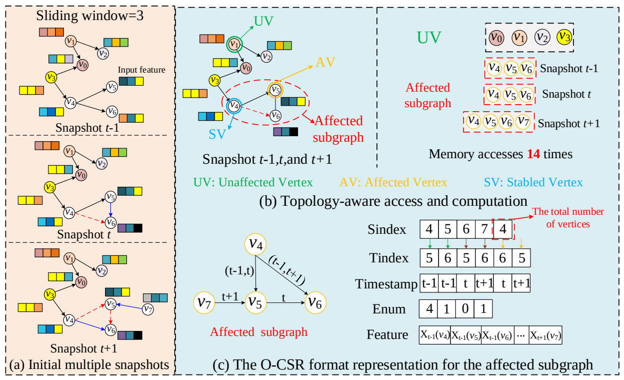


Figure 4: An example to explain our approach

Topology-aware Concurrent Processing. To effectively capture overlapping characteristics across multiple snapshots, our approach first tracks the dependencies between stable vertices[1] and their neighbors based on the topology of each snapshot. These stable vertices are designated as roots to concurrently prefetch affected vertices on the fly along their respective graph topologies using a *Depth-First Search* (DFS) strategy [23] for better data locality, leveraging previously tracked information. This method efficiently delineates the boundaries of the affected subgraph while concurrently identifying unaffected vertices.

Specifically, our approach begins by categorizing vertices across multiple snapshots based on the mutability of their features and topological information. Vertices with unchanged features are classified as stable vertices, which are further designated as affected or unaffected based on the variability of their topological information across snapshots. Subsequently, the stable vertices serve as roots for a concurrent DFS traversal to retrieve other vertices and capture the affected subgraph. For each neighbor of the root vertex, our approach determines its categorization by meticulously examining whether its features or topological information have undergone changes across the snapshots. If a neighbor is identified as an affected vertex, it is seamlessly incorporated into the affected subgraph, and the graph traversal proceeds recursively from that neighbor. This process persists until all reachable vertices from the stable vertices have been thoroughly explored, ensuring a comprehensive identification of the affected subgraph.

We illustrate the above process using the example in Figure 4 (b). Initially, vertices v_0, v_1, v_2, and v_3 are identified as unaffected vertices due to their unchanged features, neighbors and neighbors’features across the three snapshots. Vertex v_4 is classified as a stable vertex, while vertices v_5, v_6, and v_7 are recognized as affected vertices. Subsequently, vertex v_4 serves as the root for a concurrent DFS based on the graph topology of snapshots $t-1$, t, and $t+1$, thereby obtaining the affected subgraph, which includes vertices v_4, v_5, v_6, and v_7 along with their topology and feature information.

Dynamic Graph Representation for Affected Subgraph. To facilitate the concurrent computation and enhance data locality of affected subgraphs across multiple snapshots, we introduce a cache-friendly dynamic graph representation, termed *Overlap-aware Compressed Sparse Row* (O-CSR). As depicted in Figure 4 (c),

[1]Stable vertices are analogous to cut vertices in a graph, allowing for the segmentation of affected and unaffected subgraphs across multiple snapshots.

the O-CSR consists of four arrays: *Sindex*, which denotes the source vertex IDs of each outgoing edge within the affected subgraph and includes an additional entry for the total number of vertices; *Tindex*, representing the target vertex IDs corresponding to these source vertices in the respective snapshots; *Timestamp*, indicating the snapshot ID of each target vertex; *Enum*, specifies the number of edges for each source vertex across multiple snapshots; and *Feature*, which stores the features of these vertices across all snapshots. Note that O-CSR stores the feature of stable vertices only once.

The total space complexity of O-CSR is $o(2|E_s| + (K * D + 2)|V_s|)$, where $|E_s|$, $|V_s|$, K, and D represent the number of edges, vertices, snapshots, and the size of feature dimension in the affected subgraph, respectively. The O-CSR data structure efficiently accommodates dynamic changes, such as inserting, updating, and deleting edges and vertices, by adjusting the appropriate entries in the indices and timestamp arrays. We illustrate how O-CSR stores affected vertices using the example from Figure 4 (c). For the stable vertex v_4, it has neighbors v_5 and v_6 at $t-1$, v_5 at t, and v_6 at $t+1$. In O-CSR, this is represented as follows: *Sindex*[0] = 4, *Tindex*[0:3] = [5, 6, 5, 6], *Timestamp*[0:3] = [t-1, t-1, t, t+1], *Enum*[0] = 4, and *Feature*[0:4] = $[X_{t-1}(v_4), X_{t-1}(v_5), X_{t-1}(v_6), X_t(v_5), X_{t+1}(v_6)]$.

By leveraging the O-CSR format, our approach efficiently retrieves the neighbor IDs and features for each source vertex across all snapshots in a continuous manner, facilitating fast and cache-friendly access to the necessary data. The sequential storage layout of target vertex IDs and features within the O-CSR structure ensures that the data required for computing the affected subgraph is stored contiguously, thereby minimizing cache misses and reducing off-chip memory access latency. Furthermore, the O-CSR representation enables concurrent computation of the affected subgraph across multiple snapshots by organizing data in a snapshot-wise manner and utilizing the *Timestamp* array to identify the snapshot ID of each target vertex, enhancing DGNN inference efficiency.

Similarity-aware Cell Skipping. To alleviate temporal data dependencies and minimize unnecessary computations in the cell-update phase of DGNN inference, we propose an efficient similarity-aware cell skipping strategy. It introduces a similarity score θ to quantify the consistency between two feature vectors (i.e., the output features of the same vertex after the GNN operation across two consecutive snapshots). The similarity score θ is defined as follow:

$$\theta(Z^t(v), Z^{t+1}(v)) = (\frac{Z^t(v) \odot Z^{t+1}(v)}{\|Z^t(v)\| \times \left\|Z^{t+1}(v)\right\|}) \times \frac{|\mathcal{N}_{sv}(v)|}{|\mathcal{N}^t(v) \cap \mathcal{N}^{(t+1)}(v)|}$$

where $\mathcal{Z}^t(v)$ and $\mathcal{Z}^{t+1}(v)$ denote the output features of vertex v at snapshots t and $t+1$ after GNN operation, respectively, $\mathcal{N}^t(v)$ and $\mathcal{N}^{(t+1)}(v)$ represent the neighbors of v at snapshots t and $t+1$, respectively. $\mathcal{N}sv(v)$ denotes the set of stable vertices among the common neighbors of v across snapshots t and $t+1$.

The similarity score θ integrates three key factors: the cosine similarity between vertex features, which quantifies the dissimilarity between the features of a vertex across two consecutive snapshots; the overlap ratio of neighboring vertices, reflecting topological consistency by measuring the proportion of shared neighbors between snapshots; and the proportion of stable vertices among common neighbors, indicating the stability of the local graph structure by assessing how many common neighbors remain stable over time.

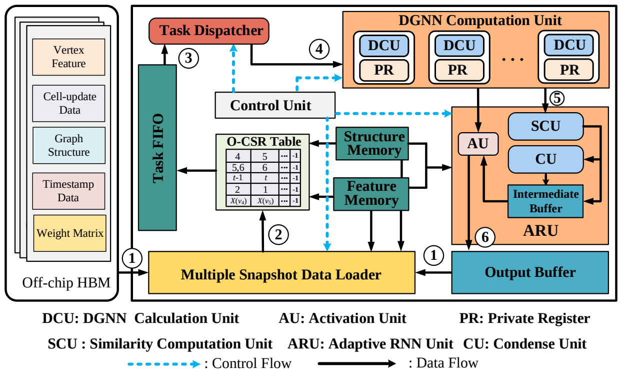


Figure 5: *TaGNN* hardware architecture

By combining these factors, the similarity score provides a comprehensive measure of vertex consistency across two consecutive snapshots, ranging from -1 to 1. A higher similarity score signifies greater consistency, whereas a lower score indicates substantial changes in the vertex's features and local topology.

Based on the similarity score θ, the cell skipping strategy works as follows. For each stable and affected vertex v, it computes $\theta\left(Z^t(v), Z^{t+1}(v)\right)$ across snapshots t and $t+1$. It establishes two thresholds, θ_s and θ_e to guide the cell-update operation. If the similarity score exceeds θ_e, it classifies the vertex as highly consistent and skip the cell-update operation entirely, reusing the final feature from the previous snapshot. If the similarity score falls between θ_s and θ_e, it performs a partial cell-update by computing the vector difference Δ [11] between the output features and adding it to the final feature from the previous snapshot. If the similarity score is below θ_s, it executes the normal cell-update operation of the RNN module to produce the final features.

3.2 Benefits of Customization

Our topology-aware concurrent execution approach significantly enhances performance on general-purpose processors, as illustrated in Figure 8 (a). However, several factors may constrain the full potential of this approach. First, the need to dynamically capture the *affected subgraph* through simultaneous and irregular traversal of multiple graphs poses a challenge, as the graph structure of DGNN is not only sparse but also varies dynamically. Second, the extensive set operations involved in similarity score computation introduce additional computational instructions and suffer from low levels of instruction-level parallelism due to data-dependent branches. Furthermore, the adaptive selection of the cell-update computation model based on similarity scores is not well-suited for general-purpose processors; frequent switching between different computation models can lead to pipeline stalls and degraded cache performance. Given these challenges, it is prudent to consider the development of a bespoke accelerator specifically designed for the efficient execution of DGNN inference. By integrating specialized hardware components and optimized memory hierarchies that align with our topology-aware concurrent execution approach, such an accelerator could substantially enhance the performance and efficiency of DGNN inference, thereby unlocking its full potential.

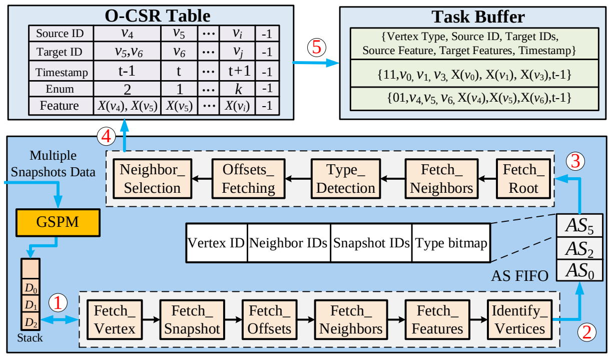

O-CSR Table
Source ID | v_4 | v_5 | ⋯ | v_i | -1
Target ID | v_5,v_6 | v_6 | ⋯ | v_j | -1
Timestamp | t-1 | t | ⋯ | t+1 | -1
Enum | 2 | 1 | ⋯ | k | -1
Feature | $X(v_4)$, $X(v_5)$ | $X(v_5)$ | ⋯ | $X(v_i)$ | -1
Task Buffer
{Vertex Type, Source ID, Target IDs, Source Feature, Target Features, Timestamp}
{11,v_0, v_1, v_3, X(v_0), X(v_1), X(v_3),t-1}
{01,v_4,v_5, v_6, X(v_4),X(v_5),X(v_6),t-1}
5
4
Multiple Snapshots Data
Neighbor_ Selection
Offsets_ Fetching
Type_ Detection
Fetch_ Neighbors
Fetch_ Root
3
GSPM
Vertex ID | Neighbor IDs | Snapshot IDs | Type bitmap
AS_5
AS_2
AS_0
AS FIFO
D_0
D_1
D_2
Stack
1
Fetch_ Vertex
Fetch_ Snapshot
Fetch_ Offsets
Fetch_ Neighbors
Fetch_ Features
Identify_ Vertices
2

Figure 6: Microarchitecture of MSDL unit

4 *TaGNN* Overview

In this section, we introduce the topology-aware accelerator, *TaGNN*, designed to maximize the performance of our approach. *TaGNN* comprises several hardware units, including the *Multiple Snapshots Data Loader*, *Task Dispatcher*, *DGNN Computation Unit*, *Adaptive RNN Unit*, and various on-chip buffers, as illustrated in Figure 5. These components collaborate to efficiently implement topology-aware concurrent execution for DGNN inference.

Multiple Snapshots Data Loader (MSDL) divides all snapshots into multiple batches, each containing a predefined number of snapshots. DGNN inference begins with the *Graph Snapshot Partition Module* (GSPM), which retrieves a partition from the current batch. Note that GSPM can support various partitioning strategies [5, 45]. For each retrieved partition, the required data (i.e., D_0) are stored in a stack (①). MSDL utilizes multiple data loading hardware pipelines to identify and prioritize the loading of unaffected vertices while capturing the *affected subgraph* across multiple snapshots (②). Subsequently, the affected subgraph data are stored in the O-CSR table, enabling rapid traversal and retrieval of vertex, feature, and timestamp information by organizing the affected subgraph in a compact format. Leveraging the O-CSR table, *TaGNN* assembles these data and orchestrates them into discrete computing tasks, which are sequentially streamed into the *Task FIFO*. Each entry comprises six elements: *Vertex Type*, the *Source ID*, *Target IDs*, *Source Feature*, *Target Features*, and *Timestamp* (③).

Task Dispatcher is activated when the *Task FIFO* is not empty and retrieves information from the FIFO to generate fine-grained computation tasks for each vertex across multiple snapshots. The *Task Dispatcher* assigns tasks to idle computing units through evenly divides the vertices within each batch based on the number of neighbors associated with them (④). This approach maximizes the utilization of computation units and minimizes idle time by maintaining a steady supply of tasks.

DGNN Computation Unit is a specialized processing element engineered to efficiently execute the hybrid computations required for the aggregation and combination phases of DGNN inference. Each *DGNN Calculation Unit* (DCU) comprises two types of computational components: the *Combination Processing Element* (CPE) and the *Aggregation Processing Element* (APE). The CPE employs row-wise matrix multiplication to effectively facilitate the combination operation, while the APE is tasked with performing the aggregation operation. The partial sum (*Psum*) generated by the CPE can be stored in the *Private Register* (PR) of the DCU to enhance the efficiency of regular matrix computations. Subsequently, the aggregation results from the APE are transferred to the *Adaptive RNN Unit* to derive the final features of each vertex (⑤). Note that the computational units are designed to be flexible and configurable, enabling them to adapt to various DGNN models.

Adaptive RNN Unit comprises three integral components: the *Similarity Core Unit*, the *Condense Unit*, and the *Activation Unit*. These modules collaboratively manage and efficiently support cell-update and activation operations based on the similarity scores of vertices across multiple snapshots. The *Similarity Core Unit* (SCU) computes the similarity scores for each stable and affected vertex by utilizing their output features from the GNN module across two consecutive snapshots. Subsequently, the appropriate cell-update computation mode is selected by comparing the similarity score against two threshold values (i.e., θ_s and θ_e). For vertices with similarity scores that fall between θ_s and θ_e, the *Delta Generation* module calculates the delta values representing the differences in these vertices' features between the current and previous snapshots. *Condense Unit* then condenses these delta values into a dense representation to facilitate efficient computation in the subsequent *DGNN Computation Unit*. The *Activation Unit* supports the necessary nonlinear transformations or activation functions to the processed data. Upon completion of these computations, the resulting output is conveyed to the output buffer for either immediate retrieval or eventual write-back to off-chip memory (⑥).

On-chip Buffers consist of several components, including the *Structure Memory* buffer, *Feature Memory* buffer, *Weight Matrix* buffer, and *Output Buffer*. These buffers are employed to cache various types of data (e.g., vertex structure, vertex features, weight matrices, and output results) to enhance data reuse and minimize unnecessary off-chip communications. Notably, *TaGNN* employs ping-pong buffering technology [6] to decouple different operations across all buffers, thereby mitigating access latency.

4.1 Overlap-aware Data Loading

To efficiently classify vertices and capture affected subgraphs across multiple snapshots, as illustrated in Figure 6, the MSDL employs a meticulously designed 6-stage pipeline comprising the following stages: *Fetch_Vertex*, *Fetch_Snapshot*, *Fetch_Offsets*, *Fetch_Neighbors*, *Fetch_Features*, and *Identify_Vertices* (①). Upon the arrival of a new graph snapshot partition, the *Fetch_Vertex* stage initiates the pipeline by randomly selecting a vertex (e.g., v_i at snapshot $t-1$) from the *Structure Buffer*. Once a vertex is selected, it is flagged as visited to eliminate redundancy in subsequent processing stages. For each selected vertex, the *Fetch_Snapshot* stage checks for the presence of the vertex ID in the snapshots to be processed (i.e., snapshots t, $t+1$, and $t+2$). If the vertex ID is absent in any of these snapshots, it is immediately classified as an affected vertex, as its absence signifies a structural change in the graph across the snapshots. Subsequently, the *Fetch_Offsets* stage retrieves the starting and ending offsets of v_i within the relevant snapshots by leveraging the *Vertex_Offset* array in the CSR format. The *Fetch_Neighbors* stage retrieves the neighboring vertices of v_i from the respective snapshots. Following this, the *Fetch_Features* stage acquires the input features associated with the selected vertex v_i and its neighbors across the relevant snapshots. Finally, the *Identify_Vertices* stage

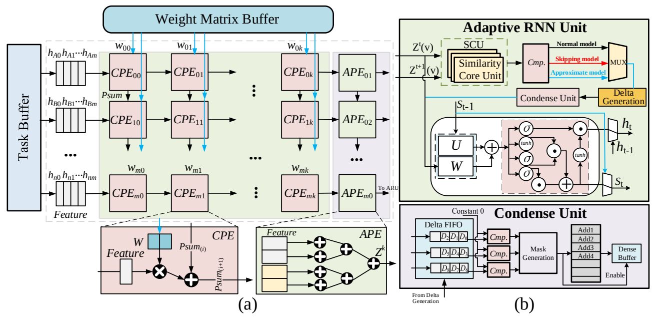


Figure 7: Microarchitecture of *TaGNN*; (a) the detail of *DGNN Computation Unit*; (b) the detail of *Condense Unit*

performs the critical task of classifying vertices based on their structural and feature-level changes across the snapshots. This stage considers factors such as the consistency of vertex IDs, variations in features, and modifications in the local graph structure. By comparing these attributes across the snapshots, the *Identify_Vertices* stage accurately categorizes each vertex as either a stable vertex, an unaffected vertex, or an affected vertex. Note that *TaGNN* replicates the *Fetch_Neighbors* and *Fetch_Features* units and parallelizes their data accesses to balance the pipeline design.

The collected data of stable vertices, now structured in the format *Vertex ID*, *Neighbor IDs*, *Snapshot IDs*, *Vertex Bitmap*, is efficiently streamed into the *Affected Subgraph FIFO* (AS FIFO) to expedite the identification of the *affected subgraph*, thereby optimizing the subsequent computational phases of the DGNN computing pipeline (2). Note that the identified unaffected vertices are directly used by *TaGNN* to generate computational tasks. Upon the insertion of an entry into the AS FIFO buffer by the *Identify_Vertices* stage, MSDL commences its efficient detection of the *affected subgraph* by employing a meticulously designed five-stage pipeline (3). Each stage of this pipeline is orchestrated by a *Traversal Finite State Machine* (TFSM) and encompasses the following steps: *Fetch_Root*, *Fetch_Neighbors*, *Type_Detection*, *Offsets_Fetching*, and *Neighbors_Selection*.

The *Fetch_Root* stage initiates the pipeline by extracting an element (e.g., AS_0) from the AS FIFO, which serves as the basis for subsequent stages. In the *Fetch_Neighbors* stage, the neighbor IDs of the root vertex are retrieved from the FIFO, facilitating efficient exploration of the local graph structure. During the *Type_Detection* stage, the pipeline classifies each neighbor vertex by examining the corresponding type bitmap, enabling accurate categorization based on structural and temporal characteristics. The *Offsets_Fetching* stage then retrieves the offsets of the identified affected neighbors from graph structure memory, serving as critical pointers for locating relevant information associated with these vertices. In the *Neighbors_Selection* stage, the affected neighbors, along with their offsets and type information, undergo a refined selection process, ensuring that only pertinent stable and affected vertices are considered for further processing. Upon completion of neighbor selection, MSDL updates the O-CSR table with details of the selected affected vertices, including source ID, target ID, timestamp, and features (4). To reduce redundant memory footprint, *TaGNN* stores the features of stable and unaffected vertices only once across snapshots. Using the O-CSR table, *TaGNN* generates computation tasks

Table 3: Resource utilization of *TaGNN* on U280 FPGA

Resource	CD-GCN	GC-LSTM	T-GCN
DSP	77.2%	80.2%	73.6%
LUT	42.6%	49.5%	40.1%
FF	34.9%	35.2%	30.4%
BRAM	62.4%	69.7%	59.3%
UltraRAM	82.4%	89.7%	80.3%

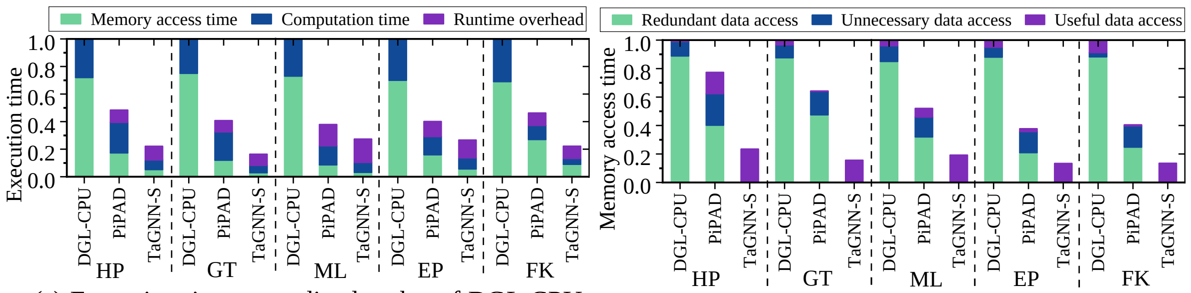
and populates the *Task FIFO* (5) efficiently by leveraging information from the O-CSR table and MSDL. Notably, unaffected vertices are directly processed as computation tasks by the *Task Dispatcher*.

4.2 Adaptive Data Similarity Computation

When the *Task FIFO* buffer is not empty, the *Task Dispatcher* assigns computation tasks to the *DGNN Computation Unit*. As depicted in Figure 7 (a), each DCU features two specialized types of processing elements to facilitate the aggregation and combination operations required for DGNN inference: the CPEs and APEs. Specifically, CPEs consist of an array of *multiply-and-accumulate* (MAC) units, responsible for executing row-wise matrix multiplication to update features, while APEs are designed for efficient vertex aggregation, employing a parallel adder tree structure to expedite the GNN module's aggregation process. The output features of each vertex (e.g., $Z^t(v)$ and $Z^{t+1}(v)$) across two consecutive snapshots are then transferred to the *Adaptive RNN Unit* to complete the RNN module computation and produce the final features.

To alleviate temporal data dependencies between snapshots during DGNN inference, *TaGNN* integrates *Adaptive RNN Unit* to eliminate unnecessary cell-update computations. As illustrated in Figure 7 (b), the SCU calculates similarity scores for affected and stable vertices. The SCU consists of multiple *Similarity Core Units* (SCUs), each designed to efficiently perform the similarity analysis computations. These units operate in parallel, with each one calculating the cosine similarity between the feature vectors of the target vertex and its corresponding neighbors or associated vertices. To achieve this, the SCU employs a multi-stage pipeline architecture, where each stage processes the input data through a series of specialized hardware components. The first stage consists of vector multiplication units that compute the dot product between the feature vectors of the vertices. In the second stage, these dot products are then passed through normalization units to compute the cosine similarity by dividing the dot product by the product of the norms of the individual vectors. In parallel, topological overlap stages calculate the intersection of neighboring vertices in the graph, providing a measure of how well the target vertex overlaps with its neighbors in terms of graph structure. Finally, the stability weighting stages adjust the similarity scores based on the stability of the vertices across consecutive snapshots, ensuring that the impact of stable vertices is appropriately factored into the computation.

For stable vertices, the SCU bypasses the vector difference computation to further reducing computational latency, as their feature vectors remain unchanged across consecutive snapshots. Subsequently, *Adaptive RNN Unit* compares these scores against predefined thresholds θ_s and θ_e to determine the appropriate cell-update computation mode. In normal mode, standard RNN computation is performed using the complete vertex features from the current



(a) Execution time normalized to that of DGL-CPU with sliding window is set to 4 on T-GCN model (b) Computation time normalized to that of DGL-CPU with sliding window is set to 4 on T-GCN model

Figure 8: Performance of *TaGNN*-S against different software systems over different datasets

Table 4: System configurations of compared accelerator

	DGNN-Booster [7]	E-DGCN [46]	Cambricon-DG [40]	TaGNN
Compute	280 MHz @ 4,096 MACs	1 GHz @ 4,096 MACs, where it has 8×8 PEs, each PE has 4×4 adders	1 GHz @ 4,096 MACs, where it has 1 DU, 32 TUs, and 32 SUs	280 MHz @ 4,096 MACs, where it has 16 DCUs and 8 SCUs, each DCU has 256 CPEs and 128 APEs
On-chip Memory	5 MB	12 MB	4 MB	2 MB (Feature Memory), 256 KB (Task FIFO), 128 KB (Intermediate Buffer) 1 MB (O-CSR Table), 512 KB (Structure Memory), 128 KB (Output Buffer)
Off-chip Memory	256 GB/s HBM 2.0	256 GB/s HBM 2.0	256 GB/s HBM 2.0	256 GB/s HBM 2.0

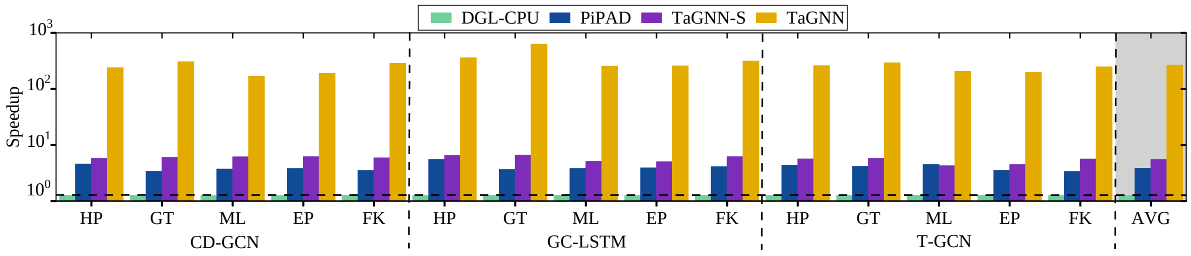


Figure 9: Comparative performance of normalized to that of DGL-CPU

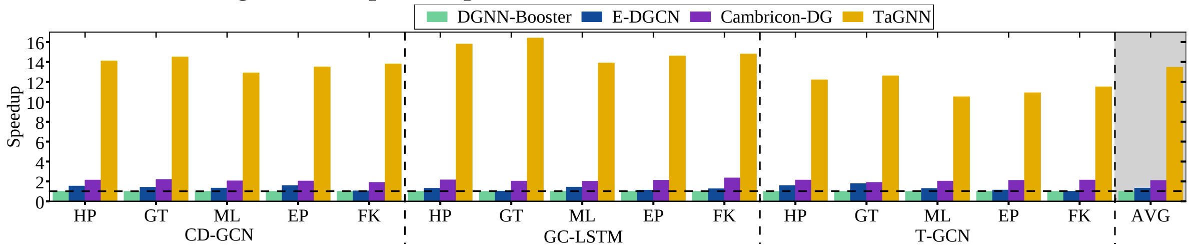


Figure 10: Performance of different schemes normalized to that of DGNN-Booster

snapshot, with the *DGNN Computation Unit* executing conventional RNN cell-update operations to accurately capture temporal dependencies. For vertices with similarity scores exceeding θ_e, a skipping mode is activated, bypassing RNN computation and directly reusing the final features from the previous snapshot. This approach significantly reduces computational overhead for vertices exhibiting high consistency across snapshots. In similarity computation mode, triggered when scores fall between θ_s and θ_e, the *Delta Generation* module calculates the differences (Δ) between the output features of consecutive snapshots. However, because both feature and topological similarities are considered in the similarity score calculation, the resulting delta values often contain numerous zero elements, reflecting unchanged components.

To efficiently eliminate these zero values, the *Condense Unit* employs a multi-level parallel zero-value filtering mechanism. The *Mask Generation Unit* creates a mask for each vertex's delta values, identifying non-zero elements. These masks, along with the corresponding vertex IDs, are used to generate addresses stored in the *Address Register*, ensuring proper alignment and retrieval of final results. The non-zero delta values are then stored in the *Dense Buffer*, serving as a compact representation of relevant information. The *DGNN Computation Unit* performs cell-update calculations for these non-zero delta values. Finally, the computed results from the *DGNN Computation Unit* are combined with the corresponding vertex's cell-update values from the previous snapshot (stored in the *Intermediate* buffer) to produce the final output features. Importantly, the state values S_{t-1} from the previous snapshot are also updated through this process, ensuring consistency and accuracy across multiple snapshots during DGNN inference. To preserve the fidelity of inference results and mitigate error accumulation from prolonged skipping, *TaGNN* recalculates similarity scores for each vertex in the new batch, rather than reusing scores and skipping decisions from the previous batch. This approach ensures robust and precise computation over time.

5 Experimental Evaluation

5.1 Experimental Setup

TaGNN Setting. We implemente *TaGNN* on a Xilinx Alveo U280 FPGA accelerator card, which features a XCU280 FPGA chip equipped with 1.08 million LUTs, 4.5 MB of on-chip BRAM, 30 MB of on-chip UltraRAM, 9,024 DSP slices, and two 4 GB HBM2 stacks, providing a total memory bandwidth of 460 GB/s. To determine the clock rate for *TaGNN*, we employ Xilinx Vivado 2019.1 and conservatively set the operating frequency to 225 MHz for our experiments.

Benchmarks and Dynamic Datasets. Table 2 summarizes five real-world datasets for DGNN research [26, 40] used in the evaluation, i.e., *HepPh* (HP), *Gdelt* (GT), *MovieLens* (ML), *Epinions* (EP), and *Flicker* (FK). We evaluate *TaGNN* using three widely recognized DGNN models: CD-GCN [21], GC-LSTM [8], and T-GCN [45]. These models are configured with four, three, and two layers, respectively [8, 15, 20, 21, 45]. The resource utilization of *TaGNN* across all evaluated models is presented in Table 3.

Baselines and Evaluation Metrics. The performance of *TaGNN* is compared with five solutions, i.e., DGL-CPU (v2.4.0) [32], PiPAD [31], DGNN-Booster [7], E-DGCN [46], and Cambricon-DG [40]. DGL-CPU is the best-performing solution on the CPU platform, and PiPAD is the state-of-the-art framework for DGNN on GPU. In our experiments, DGL-CPU is running on the Intel Xeon 6151 processor with 65 cores at 3.0 GHz and 696 GB DRAM. PiPAD runs on the NVIDIA Tesla A100 GPU with 6,912 cores and 80 GB HBM. DGNN-Booster, E-DGCN, and Cambricon-DG, are the cutting-edge hardware DGNN accelerators. The configurations of these baseline accelerators and our *TaGNN* are listed in Table 4. Intel Product Specifications [2] are used to estimate the CPU energy consumption. The GPU energy is obtained by NVIDIA System Management Interface (nvidia-smi) [1]. Note that, to evaluate our software approach, we have also modified DGL to use our *topology-aware concurrent execution approach* to support DGNN inference, and this software implementation is called *TaGNN-S*. Note that *TaGNN-S* runs on the above A100 GPU in the following experiments and the default number of snapshots of a batch is set to 4 in our approach.

5.2 Experimental Results

Comparison with Software Systems. Figure 8 (a) presents the normalized execution time of various solutions, with the time decomposed into memory access time, computation time, and runtime overhead. The results demonstrate that the overall performance of *TaGNN*-S surpasses PiPAD, primarily because PiPAD incurs significantly higher data access time and lower data parallelism than *TaGNN*-S. Specifically, the memory access time of PiPAD is 2.7x to 4.1x greater than that of *TaGNN*-S across the tested instances. This disparity arises from PiPAD's requirement to transmit the features of all vertices, even when the features of most vertices remain identical across the four snapshots.

Figure 8 (b) provides a detailed breakdown of the memory access time from Figure 8 (a). The results indicate that *TaGNN*-S reduces redundant access time by 21.2%-47.5% and unnecessary computation time by 14.2%-22.2% for the T-GCN model. This improvement is attributed to *TaGNN*-S's capability to minimize the access frequency of unaffected and stable vertices across the four snapshots to a single access, thereby significantly reducing unnecessary off-chip communications and improving data locality. Additionally, *TaGNN*-S alleviates temporal data dependencies and minimizes redundant computations in the RNN module. However, *TaGNN*-S only slightly outperforms PiPAD overall due to its high runtime overhead. As shown in Figure 8 (a), the runtime overhead accounts for 40.1%-62.3% of *TaGNN*-S's total execution time. In contrast, *TaGNN* further enhances data locality through a customized memory subsystem and significantly reduces the runtime overhead inherent to *TaGNN*-S, resulting in a marked improvement in overall performance. As illustrated in Figure 9, *TaGNN* outperforms DGL-CPU and PiPAD by 415.2x-612.6x (535.2x on average) and 62.8x-146.4x (84.3x on average), respectively.

Table 5: Accuracy comparison of *TaGNN* against *TaGNN* with other cutting-edge RNN approximation methods, including DeltaRNN [11] (i.e., *TaGNN*-DR), ALSTM [14] (i.e., *TaGNN*-AM), and ATLAS [17] (i.e., *TaGNN*-AS)

Models	Methods	Accuracy (%)				
		HepPh	Gdelt	MovieLens	Epinions	Flicker
CD-GCN	Baseline	75.3±0.8	78.2±0.4	80.4±1.1	70.2±0.8	61.4±1.2
	TaGNN-DR	61.2±0.4	62.4±0.8	60.5±0.7	55.4±2.5	43.4±2.8
	TaGNN-AM	63.4±0.5	61.4±0.2	61.2±2.1	54.2±1.3	40.3±1.8
	TaGNN-AS	70.0±0.7	72.3±1.0	65.4±1.2	60.1±1.8	53.4±0.5
	TaGNN (Ours)	**74.9±0.6**	**78.1±0.4**	**80.1±0.9**	**69.6±0.5**	**60.8±0.8**
	TaGNN loss accuracy	0.1% ~ 0.8%				
GC-LSTM	Baseline	89.5±0.4	80.5±0.9	91.2±0.4	87.3±0.4	72.4±1.2
	TaGNN-DR	73.4±0.5	72.8±1.1	72.5±0.8	61.6±1.5	59.8±1.8
	TaGNN-AM	74.8±1.2	74.4±0.9	77.3±0.5	67.1±2.8	64.3±1.2
	TaGNN-AS	80.0±0.6	71.5±1.1	80.2±1.4	77.2±2.1	63.5±1.9
	TaGNN (Ours)	**88.7±0.5**	**79.9±0.9**	**90.4±0.5**	**86.5±0.4**	**71.9±1.1**
	TaGNN loss accuracy	0.5% ~ 0.8%				
T-GCN	Baseline	75.3±0.5	81.4±0.2	75.6±0.7	85.2±0.5	58.4±1.2
	TaGNN-DR	59.7±0.8	59.4±0.8	44.4±0.9	62.7±2.1	49.3±2.7
	TaGNN-AM	68.4±0.2	72.5±0.2	61.5±0.4	63.8±1.4	50.4±1.5
	TaGNN-AS	64.2±0.8	70.3±1.2	64.2±0.3	65.4±0.8	49.8±2.9
	TaGNN (Ours)	**74.6±0.8**	**80.9±0.2**	**74.8±0.5**	**84.9±0.2**	**57.8±0.6**
	TaGNN loss accuracy	0.2% ~ 0.9%				

Comparison with DGNN Accelerators. Figure 10 demonstrates that *TaGNN* outperforms DGNN-Booster, E-DGCN, and Cambricon-DG with average speedups of 13.5x, 10.2x, and 6.5x, respectively. This performance advantage stems from *TaGNN*'s ability to reduce redundant and unnecessary memory accesses by 78.3%-84.6%, 69.2%-72.5%, and 52.1%-63.4% compared to DGNN-Booster, E-DGCN, and Cambricon-DG, respectively. In comparison to these accelerators, *TaGNN* effectively avoids redundant accesses to unaffected and stable vertices across multiple snapshots, reduces unnecessary RNN computations, and improves data parallelism.

Analysis of Energy Consumption. Figure 11 illustrates the energy consumption of various solutions, showing that *TaGNN* achieves energy reductions of 621.3x to 901.5x (averaging 742.6x) and 88.9x to 135.2x (averaging 104.9x) compared to DGL-CPU and PiPAD, respectively. Against existing hardware accelerators, including DGNN-Booster, E-DGCN, and Cambricon-DG, *TaGNN* achieves average energy savings of 15.9x, 11.7x, and 7.8x, respectively. These substantial energy efficiency improvements are primarily attributed to *TaGNN*'s multi-stage hardware pipeline and customized memory subsystem, which effectively minimize redundant combinational logic along the critical path and reduce unnecessary on-chip resource utilization.

5.3 DGNN Inference Accuracy Analysis

Table 5 provides a comparative analysis of the accuracy achieved by *TaGNN* and its variants that incorporate state-of-the-art RNN approximation methods [11, 14, 17], namely TaGNN-DR, TaGNN-AM,

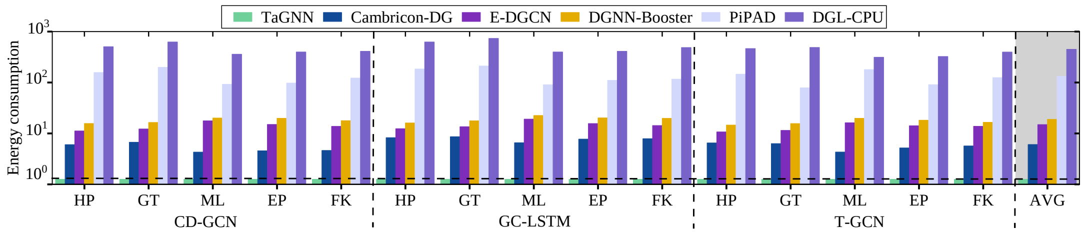


Figure 11: Energy consumption of different solutions normalized to that of *TaGNN*

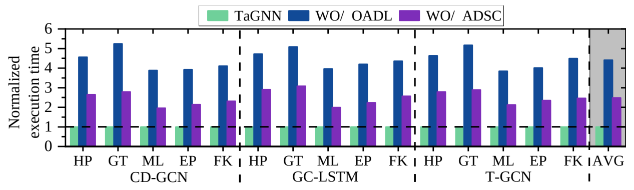


Figure 12: Performance breakdown of *TaGNN*

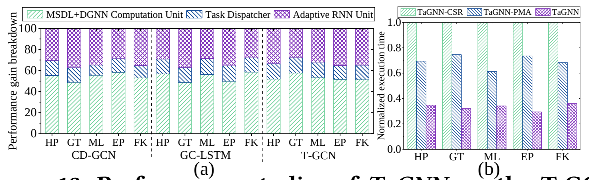


Figure 13: Performance studies of *TaGNN* on the T-GCN model: (a) breakdown of architecture performance improvement; (b) the execution time normalized to *TaGNN*-CSR

and TaGNN-AS, across various datasets and DGNN models. The results reveal that *TaGNN* consistently delivers high accuracy with minimal degradation compared to the baseline models. Specifically, the accuracy drop for *TaGNN* ranges from 0.1% to 0.8% for CD-GCN, 0.5% to 0.8% for GC-LSTM, and 0.2% to 0.9% for T-GCN, underscoring its ability to preserve model performance while significantly improving inference efficiency. Importantly, this accuracy loss remains within an acceptable range [8, 14, 20, 21, 24], validating the effectiveness of our approach. Additionally, the similarity-aware cell skipping strategy employed by *TaGNN* plays a pivotal role in maintaining accuracy loss below 1%. By exploiting topological and feature overlaps between consecutive snapshots, this strategy reduces unnecessary computations and memory accesses, further enhancing efficiency without compromising accuracy.

5.4 Effectiveness of *TaGNN*'s Designs

Figure 12 delineates the advantages conferred by the *TaGNN*, which encompasses our *Overlap-aware Data Loading* (OADL) mechanism and the *Adaptive Data Similarity Computation* (ADSC) scheme.

OADL. Benefiting from the OADL mechanism, which can further eliminate redundant data partition loading and the features of stable vertices can also be reduced to once. Consequently, we discern that OADL furnishes an average performance enhancement of 4.41x compared to the version of *TaGNN* without OADL strategy (i.e., WO/ OADL), constituting 71.38% of the total performance gains achieved by *TaGNN*.

ADSC. By applying the ADSC scheme, the similarity scores of all vertices in two consecutive snapshots can be efficiently calculated, and the results can be quickly reused based on this score. This enhancement translates to an average improvement in total execution time by a factor of 2.48x over the version of *TaGNN* without

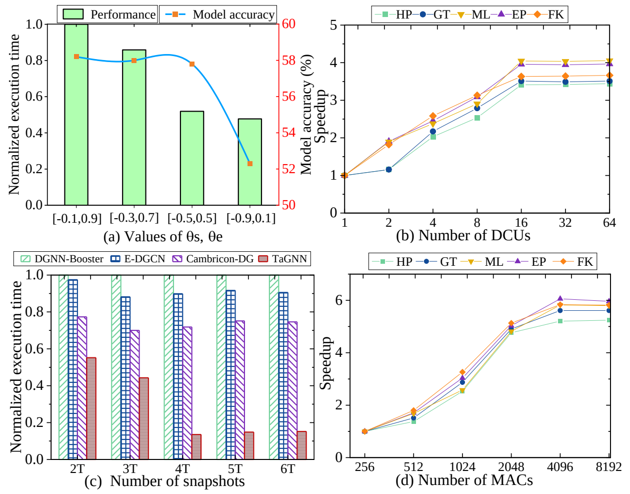


Figure 14: Sensitivity studies of *TaGNN*: (a) sensitivity to the values $[\theta_s, \theta_e]$ over *Fk* dataset; (b) sensitivity to the number of DCUs; (c) sensitivity to the number of snapshots over *FK* dataset; (d) sensitivity to the number of MAC units

ADSC (i.e., WO/ ADSC), accounting for 28.62% of the aggregate performance improvement.

Figure 13 (a) presents the performance gain breakdown of the *TaGNN* architecture, driven by three key components: the MSDL+DGNN Computation Unit, which minimizes irregular traversal and data loading overhead, contributing 53.6% on average of the total performance improvement; the Task Dispatcher, which balances DGNN workloads, contributing 13.8% on average of the total performance improvement; and the Adaptive RNN Unit, which implements the similarity-aware cell skipping strategy, contributing 32.6% on average of the total performance improvement. Figure 13 (b) highlights that the O-CSR of *TaGNN* outperforms its counterparts employing other state-of-the-art dynamic graph formats, i.e., *TaGNN*-CSR [37, 39, 42] and *TaGNN*-PMA [28, 33], by factors of 2.3-3.4 and 1.8-2.5, respectively. This performance advantage is largely attributed to the O-CSR format, which provides superior data locality and parallelism while reducing redundant storage overhead by 73.5%-82.4% and 53.2%-61.8% for four snapshots compared to CSR and PMA, respectively.

5.5 Sensitivity Studies

Figure 14 (a) shows the performance of *TaGNN* on the T-GCN model with varying values of θ_s and θ_e (as detailed in Section 3.1), ranging from -0.9 to 0.9. Experimental results indicate that the interval between θ_s and θ_e from -0.5 to 0.5 is optimal for achieving satisfactory performance while maintaining better accuracy, averaging 57.8% on the FK dataset. Figure 14 (b) evaluates the sensitivity of *TaGNN* to the number of DCUs on the T-GCN model. Performance improves as the number of DCUs increases, peaking at 16, beyond which memory bandwidth saturation limits further gains. Figure 14 (c) compares the execution time of *TaGNN* with leading DGNN accelerators on the T-GCN model across varying snapshot counts. *TaGNN* demonstrates strong performance at different snapshot counts, with optimal results at four snapshots. However, performance slightly declines as the snapshot count increases due to the rising runtime overhead of identifying unaffected vertices and computing similarity scores, despite the increased inference throughput. Figure 14 (d) explores the impact of the number of MAC units on *TaGNN*'s performance on the T-GCN model. Performance improves with more MAC units but levels off due to resource and memory bandwidth limitations. For fairness and consistency with state-of-the-art works (e.g., E-DGCN [46]), 4096 MAC units are selected.

6 Related Work

Software Frameworks for GNN and DGNN. Several software frameworks for GNNs have emerged, including DGL [32] and PyG [10], which adopt a message-passing programming model and utilize the torch-scatter library to enhance edge-wise and node-wise parallelism, respectively. However, these frameworks are not well-suited for DGNNs due to their reliance on SpMM-based kernels. To address this limitation, ESDG [5] employs a graph difference-based strategy for DGNN training, while PiPAD [31] leverages dynamic pipeline training strategy to improve DGNN training. However, they predominantly adopt snapshot-by-snapshot training approach, leading to low data parallelism.

Hardware GNN and DGNN Accelerators. Numerous GNN accelerators have been proposed [6, 12, 36, 39, 47]. HyGCN [35] employs a window-based sliding and shrinking method to improve the locality of GNN inference. However, these approaches primarily focus on static graphs and fail to address the temporal dynamics of DGNNs. For dynamic graphs, DGNN-Booster [7] designs a multi-level parallelism accelerator for DGNN inference and E-DGCN [46] employs the reconfigurable processing elements that efficiently support diverse types of data computations required by DGCN layers. Cambricon-DG [40] proposes a nonlinear isolation mechanism to reduce redundant aggregation operations. While these solutions can also adopt a multi-snapshot execution pattern, they still struggle with excessive memory usage and stringent temporal data dependencies, leading to limited data parallelism.

RNN Approximate Computation. To improve RNN inference, various solutions utilize approximate computation techniques to reduce unnecessary calculations. ALSTM [14] applies approximation to LSTM operations in speech recognition, reducing energy consumption without compromising accuracy, while ATLAS [17] proposes a low-power LSTM accelerator using approximate multipliers. However, these methods rely on graph snapshots that fundamentally differ from traditional text and speech data. As a result, they overlook critical graph topology properties essential for effective approximation in DGNNs.

7 Conclusion

This paper introduces *TaGNN*, a topology-aware accelerator designed for high-performance DGNN inference. *TaGNN* employs a topology-aware concurrent execution approach that leverages overlap characteristics across multiple snapshots to reduce irregular memory accesses and mitigate temporal data dependencies while improving data parallelism. The experiment results show that *TaGNN* achieves speedups of 84.3x to 535.2x compared to cutting-edge software frameworks.

Acknowledgments

The authors would like to thank the anonymous reviewers for their insightful comments. This paper is supported by National Key Research and Development Program of China (No. 2024YFB4504200), National Natural Science Foundation of China (No. 62472183 and No. 62402457). Yu Zhang (zhyu@hust.edu.cn) is the corresponding author of this paper.

References

[1] 2021. Nvidia, Nvidia System Management Interface. https://developer.nvidia.com/nvidia-system-management-interface,2021..

[2] 2023. Intel Product Specifications. https://wcm-stg.intel.com/content/www/us/en/ark.html..

[3] Mahbod Afarin, Chao Gao, Shafiur Rahman, Nael B. Abu-Ghazaleh, and Rajiv Gupta. 2023. CommonGraph: Graph Analytics on Evolving Data. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 133–145.

[4] Vignesh Balaji, Neal Crago, Aamer Jaleel, and Brandon Lucia. 2021. P-OPT: Practical Optimal Cache Replacement for Graph Analytics. In *Proceedings of the 27th IEEE International Symposium on High-Performance Computer Architecture*. 668–681.

[5] 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 2021 International Conference for High Performance Computing, Networking, Storage and Analysis*. 53–77.

[6] Cen Chen, Kenli Li, Yangfan Li, and Xiaofeng Zou. 2022. ReGNN: A Redundancy-Eliminated Graph Neural Networks Accelerator. In *Proceedings of the 28th IEEE International Symposium on High-Performance Computer Architecture*. 1–14.

[7] Hanqiu Chen and Cong Hao. 2023. DGNN-Booster: A Generic FPGA Accelerator Framework For Dynamic Graph Neural Network Inference. In *Proceedings of the 31st IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*. 195–201.

[8] Jinyin Chen, Xueke Wang, and Xuanheng Xu. 2021. GC-LSTM: Graph Convolution Embedded LSTM for Dynamic Network Link Prediction. *Applied Intelligence* 12, 1 (2021), 1–16.

[9] Michael Defferrard, Xavier Bresson, and Pierre Vandergheynst. 2016. Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering. In *Proceedings of the 2016 Annual Conference on Neural Information Processing Systems*. 3837–3845.

[10] Matthias Fey and Jan Eric Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. *CoRR* abs/1903.02428 (2019), 1–14.

[11] Chang Gao, Daniel Neil, Enea Ceolini, Shih-Chii Liu, and Tobi Delbrück. 2018. DeltaRNN: A Power-efficient Recurrent Neural Network Accelerator. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 21–30.

[12] Tong Geng, Chunshu Wu, Yongan Zhang, Cheng Tan, Chenhao Xie, Haoran You, Martin C. Herbordt, Yingyan Lin, and Ang Li. 2021. I-GCN: A Graph Convolutional Network Accelerator with Runtime Locality Enhancement through Islandization. In *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 1051–1063.

[13] Mingyu Guan, Anand Padmanabha Iyer, and Taesoo Kim. 2022. DynaGraph: dynamic graph neural networks at scale. In *Proceedings of the 5th ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics*. 6:1–6:10.

[14] Junseo Jo, Jaeha Kung, and Youngjoo Lee. 2020. Approximate LSTM computing for energy-efficient speech recognition. *Electronics* 9, 12 (2020), 1983–2004.

[15] 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, 5 (2020), 70:1–70:73.

[16] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *Proceedings of the 5th International Conference on Learning Representations*. 1–14.

[17] Fabian Kreß, Alexey Serdyuk, Micha Hiegle, Disnebio Waldmann, Tim Hotfilter, Julian Höfer, Tim Hamann, Jens Barth, Peter Kämpf, Tanja Harbaum, and Jürgen Becker. 2023. ATLAS: An Approximate Time-Series LSTM Accelerator for Low-Power IoT Applications. In *Proceedings of the 26th Euromicro Conference on Digital*

System Design. 569–576.

[18] Alfirna Rizqi Lahitani, Adhistya Erna Permanasari, and Noor Akhmad Setiawan. 2016. Cosine similarity to determine similarity measure: Study case in online essay assessment. In *Proceedings of the 4th International Conference on Cyber and IT Service Management*. 1–6.

[19] Haoyang Li and Lei Chen. 2021. Cache-based GNN System for Dynamic Graphs. In *Proceedings of the 30th ACM International Conference on Information and Knowledge Management*. 937–946.

[20] 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 2021 SIAM International Conference on Data Mining*. 729–737.

[21] Franco Manessi, Alessandro Rozza, and Mario Manzo. 2020. Dynamic graph convolutional networks. *Pattern Recognition* 97, 1 (2020), 1–18.

[22] Tomas Mikolov, Martin Karafiat, Lukas Burget, Jan Cernocky, and Sanjeev Khudanpur. 2010. Recurrent Neural Network based Language Model. In *Proceedings of the 11th Annual Conference of The International Speech Communication Association*. 1045–1048.

[23] Anurag Mukkara, Nathan Beckmann, Maleen Abeydeera, Xiaosong Ma, and Daniel Sánchez. 2018. Exploiting Locality in Graph Analytics through Hardware-Accelerated Traversal Scheduling. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*. 1–14.

[24] Aldo Pareja, Giacomo Domeniconi, Jie Chen, Tengfei Ma, Toyotaro Suzumura, Hiroki Kanezashi, Tim Kaler, Tao B. Schardl, and Charles E. Leiserson. 2020. EvolveGCN: Evolving Graph Convolutional Networks for Dynamic Graphs. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence*. 5363–5370.

[25] Xuehai Qian. 2021. Graph Processing and Machine Learning Architectures with Emerging Memory Technologies: A Survey. *Science China Information Sciences* 64, 6 (2021), 1–25.

[26] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *Proceedings of the 29th AAAI Conference on Artificial Intelligence*. 4292–4293.

[27] Benedek Rozemberczki, Paul Scherer, Yixuan He, George Panagopoulos, Alexander Riedel, Maria Sinziana Astefanoaei, Oliver Kiss, Ferenc Béres, Guzmán López, Nicolas Collignon, and Rik Sarkar. 2021. PyTorch Geometric Temporal: Spatiotemporal Signal Processing with Neural Machine Learning Models. In *Proceedings of the 30th ACM International Conference on Information and Knowledge Management*. 4564–4573.

[28] Mo Sha, Yuchen Li, Bingsheng He, and Kian-Lee Tan. 2017. Accelerating Dynamic Graph Analytics on GPUs. In *Proceedings of the VLDB Endowment*. 107–120.

[29] Joakim Skarding, Bogdan Gabrys, and Katarzyna Musial. 2021. Foundations and Modeling of Dynamic Networks Using Dynamic Graph Neural Networks: A Survey. *IEEE Access* 9 (2021), 79143–79168.

[30] Xinkai Song, Tian Zhi, Zhe Fan, Zhenxing Zhang, Xi Zeng, Wei Li, Xing Hu, Zidong Du, Qi Guo, and Yunji Chen. 2022. Cambricon-G: A Polyvalent Energy-Efficient Accelerator for Dynamic Graph Neural Networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 1 (2022), 116–128.

[31] Chunyang Wang, Desen Sun, and Yuebin Bai. 2023. PiPAD: Pipelined and Parallel Dynamic GNN Training on GPUs. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 405–418.

[32] Minjie Wang, Lingfan Yu, Da Zheng, Quan Gan, Yu Gai, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Chao Ma, Ziyue Huang, Qipeng Guo, Hao Zhang, Haibin Lin, Junbo Zhao, Jinyang Li, Alexander J. Smola, and Zheng Zhang. 2019. Deep Graph Library: Towards Efficient and Scalable Deep Learning on Graphs. *CoRR* abs/1909.01315 (2019).

[33] Qinggang Wang, Long Zheng, Yu Huang, Pengcheng Yao, Chuangyi Gui, Xiaofei Liao, Hai Jin, Wenbin Jiang, and Fubing Mao. 2021. GraSU: A Fast Graph Update Library for FPGA-based Dynamic Graph Processing. In *Proceedings of the 2021 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. 149–159.

[34] Jin Xu and Zheng Bao. 2002. Neural Networks and Graph Theory. *Science China Information Sciences* 45, 1 (2002), 1–24.

[35] 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 *Proceedings of the 2020 IEEE International Symposium on High Performance Computer Architecture*. 15–29.

[36] Hui Yu, Yu Zhang, Ligang He, Donghao He, Qikun Li, Jin Zhao, Xiaofei Liao, Hai Jin, Lin Gu, and Haikun Liu. 2024. CDA-GNN: A Chain-driven Accelerator for Efficient Asynchronous Graph Neural Network. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*. 109:1–109:6.

[37] Hui Yu, Yu Zhang, Ligang He, Yingqi Zhao, Xintao Li, Ruida Xin, Jin Zhao, Xiaofei Liao, Haikun Liu, Bingsheng He, and Hai Jin. 2024. RAHP: A Redundancy-aware Accelerator for High-performance Hypergraph Neural Network. In *Proceedings of the 2024 International Symposium on Microarchitecture*. 1264–1277.

[38] Hui Yu, Yu Zhang, Andong Tan, Chenze Lu, Jin Zhao, Xiaofei Liao, Hai Jin, and Haikun Liu. 2024. RTGA: A Redundancy-free Accelerator for High-Performance Temporal Graph Neural Network Inference. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*. 111:1–111:6.

[39] Hui Yu, Yu Zhang, Jin Zhao, Yujian Liao, Zhiying Huang, Donghao He, Lin Gu, Hai Jin, Xiaofei Liao, Haikun Liu, Bingsheng He, and Jianhui Yue. 2023. RACE: An Efficient Redundancy-aware Accelerator for Dynamic Graph Neural Network. *ACM Transactions on Architecture and Code Optimization* 20, 4 (2023), 53:1–53:26.

[40] Zhifei Yue, Xinkai Song, Tianbo Liu, Xing Hu, Rui Zhang, Zidong Du, Wei Li Li, Qi Guo, and Tianshi Chen. 2025. Cambricon-DG: An Accelerator for Redundant-Free Dynamic Graph Neural Networks Based on Nonlinear Isolation. In *Proceedings of the 2025 IEEE International Symposium on High Performance Computer Architecture*. 934–948.

[41] Xingyao Zhang, Haojun Xia, Donglin Zhuang, Hao Sun, Xin Fu, Michael B. Taylor, and Shuaiwen Leon Song. 2021. η-LSTM: Co-Designing Highly-Efficient Large LSTM Training via Exploiting Memory-Saving and Architectural Design Opportunities. In *Proceeding of the 48th ACM/IEEE Annual International Symposium on Computer Architecture*. 567–580.

[42] Yu Zhang, Xiaofei Liao, Hai Jin, Bingsheng He, Haikun Liu, and Lin Gu. 2019. DiGraph: An Efficient Path-based Iterative Directed Graph Processing System on Multiple GPUs. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*. 601–614.

[43] Yu Zhang, Jin Zhao, Xiaofei Liao, Hai Jin, Lin Gu, Haikun Liu, Bingsheng He, and Ligang He. 2019. CGraph: A Distributed Storage and Processing System for Concurrent Iterative Graph Analysis Jobs. *ACM Transactions on Storage* 15, 2 (2019), 10:1–10:26.

[44] Jin Zhao, Yun Yang, Yu Zhang, Xiaofei Liao, Lin Gu, Ligang He, Bingsheng He, Hai Jin, Haikun Liu, Xinyu Jiang, and Hui Yu. 2022. TDGraph: a topology-driven accelerator for high-performance streaming graph processing. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 116–129.

[45] Ling Zhao, Yujiao Song, Chao Zhang, Yu Liu, Pu Wang, Tao Lin, Min Deng, and Haifeng Li. 2020. T-GCN: A Temporal Graph Convolutional Network for Traffic Prediction. *IEEE Transactions on Intelligent Transportation Systems* 21, 9 (2020), 3848–3858.

[46] 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*. 324:1–324:6.

[47] Zhe Zhou, Bizhao Shi, Zhe Zhang, Yijin Guan, Guangyu Sun, and Guojie Luo. 2021. BlockGNN: Towards Efficient GNN Acceleration Using Block-Circulant Weight Matrices. In *Proceedings of the 58th ACM/IEEE Design Automation Conference*. 1009–1014.