12-14-2023

RACE: An Efficient Redundancy-aware Accelerator for Dynamic Graph Neural Network

Hui Yu
Huazhong University of Science and Technology

Yu Zhang
Huazhong University of Science and Technology

Jin Zhao
Huazhong University of Science and Technology

Yujian Liao
Huazhong University of Science and Technology

Zhiying Huang
Huazhong University of Science and Technology

See next page for additional authors

Recommended Citation

Yu, H., Zhang, Y., Zhao, J., Liao, Y., Huang, Z., He, D., Gu, L., Jin, H., Liao, X., Liu, H., He, B., & Yue, J. (2023). RACE: An Efficient Redundancy-aware Accelerator for Dynamic Graph Neural Network. *ACM Transactions on Architecture and Code Optimization, 20*(4), 1-26. http://doi.org/10.1145/3617685
Retrieved from: https://digitalcommons.mtu.edu/michigantech-p2/460

Follow this and additional works at: https://digitalcommons.mtu.edu/michigantech-p2

 Part of the Computer Sciences Commons

RACE: An Efficient Redundancy-aware Accelerator for Dynamic Graph Neural Network

HUI YU, YU ZHANG, JIN ZHAO, YUJIAN LIAO, ZHIYING HUANG, DONGHAO HE, LIN GU, HAI JIN, XIAOFEI LIAO, and HAIKUN LIU, National Engineering Research Center for Big Data Technology and System, Service Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, China
BINGSHENG HE, National University of Singapore, Singapore
JIANHUI YUE, Michigan Technological University, America

Dynamic Graph Neural Network (DGNN) has recently attracted a significant amount of research attention from various domains, because most real-world graphs are inherently dynamic. Despite many research efforts, for DGNN, existing hardware/software solutions still suffer significantly from redundant computation and memory access overhead, because they need to irregularly access and recompute all graph data of each graph snapshot. To address these issues, we propose an efficient redundancy-aware accelerator, *RACE*, which enables energy-efficient execution of DGNN models. Specifically, we propose a *redundancy-aware incremental execution approach* into the accelerator design for DGNN to instantly achieve the output features of the latest graph snapshot by correctly and incrementally refining the output features of the previous graph snapshot and also enable regular accesses of vertices' input features. Through traversing the graph on the fly, RACE identifies the vertices that are not affected by graph updates between successive snapshots to reuse these vertices' states (i.e., their output features) of the previous snapshot for the processing of the latest snapshot. The vertices affected by graph updates are also tracked to incrementally recompute their new states using their neighbors' input features of the latest snapshot for correctness. In this way, the processing and accessing of many graph data that are not affected by graph updates can be correctly eliminated, enabling smaller redundant computation and memory access overhead. Besides, the input features, which are accessed more frequently, are dynamically identified according to graph topology and are preferentially resident in the on-chip memory for less off-chip communications. Experimental results show that RACE achieves on average 1139× and 84.7× speedups for DGNN inference, with average 2242× and 234.2× energy savings, in comparison with the state-of-the-art software DGNN running on Intel Xeon CPU and

This is a new article, not an extension of a conference paper.
This article is supported by National Key Research and Development Program of China under Grant No. 2022YFB2404202, NSFC (No. 62072193), Major Scientific Research Project of Zhejiang Lab No. 2022PI0AC03, CCF-AFSG Research Fund No. RF20220211, the Young Top-notch Talent Cultivation Program of Hubei Province, Key Research and Development Program of Hubei Province No. 2023BAB078, and Knowledge Innovation Program of Wuhan-Basi Research No. 2022013301015177.
Authors' addresses: H. Yu, Y. Zhang (Corresponding author), J. Zhao, Y. Liao, Z. Huang, D. He, L. Gu, H. Jin, X. Liao, and H. Liu, National Engineering Research Center for Big Data Technology and System, Service Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, China; e-mails: huiy@hust.edu.cn, zhyu@hust.edu.cn, zjin@hust.edu.cn, yjlong@hust.edu.cn, hzying@hust.edu.cn, hdh@hust.edu.cn, lingu@hust.edu.cn, hjin@hust.edu.cn, xfliao@hust.edu.cn, hkliu@hust.edu.cn; B. He, National University of Singapore, Singapore; e-mail: hebs@comp.nus.edu.sg; J. Yue, Michigan Technological University, America; e-mail: jyue@mtu.edu.cn.

1544-3566/2023/12-ART53
https://doi.org/10.1145/3617685

NVIDIA A100 GPU, respectively. Moreover, for DGNN inference, RACE obtains on average 13.1×, 11.7×, 10.4×, and 7.9× speedup and 14.8×, 12.9×, 11.5×, and 8.9× energy savings over the state-of-the-art Graph Neural Network accelerators, i.e., AWB-GCN, GCNAX, ReGNN, and I-GCN, respectively.

CCS Concepts: • **Computer systems organization → Special purpose systems**; **Parallel architectures;**

Additional Key Words and Phrases: Redundancy-aware, dynamic graph neural network, hardware accelerator, efficiency

ACM Reference format:
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 Trans. Arch. Code Optim.* 20, 4, Article 53 (December 2023), 26 pages.
https://doi.org/10.1145/3617685

1 INTRODUCTION

Dynamic graphs, e.g., traffic networks [23] and biology antibiotic graphs [42], constantly evolve over time [11, 47]. The continuously arriving graph updates (e.g., edge/vertex deletion/addition and the mutation of vertex state) are usually deployed in batches to the graph, and a sequence of graph snapshots at different time intervals are produced accordingly [8, 31, 39]. To extract latent information from these dynamic graphs, many **Dynamic Graph Neural Network (DGNN)** models [10, 24], which are neural networks encoding dynamic graph, are recently designed and used for dynamic node categorization [28, 29, 38], dynamic link prediction [24], real-time graph clustering [62], real-world E-recommendation [37], and so on.

To efficiently support DGNN models, software DGNN frameworks, e.g., DGNNS [8], have been recently proposed to significantly diminish the transfer time and reduce the memory usage. However, these software frameworks usually adopt coarse-grained (i.e., snapshot by snapshot) execution, resulting in a substantially sub-optimal performance. Although some hardware solutions [9, 14, 15, 25] have been designed for high-performance **Graph Neural Network (GNN)** models, they are mainly designed to accelerate the processing of static graphs. As a result, as shown in Table 1, both existing software and hardware solutions still suffer from significant redundant computation and high data access cost due to the following two reasons, which motivates us to design an efficient accelerator for DGNN workloads.

First, the vertices with the same input features, neighbors, and neighbors' input features are repeatedly accessed and processed for different snapshots, because the existing solutions need to recompute all graph data of each graph snapshot, which results in *redundant computation overhead*. Specifically, when handling multiple snapshots, the final states of the vertices with the same input features, neighbors, and neighbors' input features are identical across snapshots, because their weight matrices are the same among multiple snapshots. This eventually causes serious redundant computation, which wastes the memory and computation resources.

Second, the processing of the DGNN models suffers from serious *irregular memory access*, because the input features of the vertices are too large to be stored in the on-chip memory [14, 25, 51]. Specifically, usually only a small number of vertices need to be processed for each graph snapshot. It incurs significant random accesses to these vertices' input features and their neighbors' input features, because these input features are very sparsely dispersed in the off-chip memory. Besides, the input features of some vertices are repeatedly accessed for the processing of different snapshots via irregular off-chip communications although these vertices' final states are the same for different snapshots.

To address the above two issues, we analyze the characteristics of DGNN models and have two observations. First, the majority of vertices in two successive graph snapshots have the same input

Table 1. A Comparison of State-of-the-art Solutions in Terms of Support for Dynamic Graph Neural Network

Solutions	Dynamic graph	Incremental execution	Inter-snapshot redundancy aware
DGNNS [8]	✓	✗	✗
AWB-GCN [14]	✗	✗	✗
GCNAX [25]	✗	✗	✗
ReGNN [9]	✗	✗	✗
I-GCN [15]	✗	✗	✗
RACE	✓	✓	✓

features, neighbors, and neighbors' input features, because only a small portion of the vertices' information changes between these snapshots [8, 39]. Hence, these vertices can reuse their states of the previous snapshot for the latest snapshot. It implies that DGNN model shows significant *temporal similarity*. Second, due to the power-law property [16], some vertices are frequently accessed for the processing of consecutive snapshots in the DGNN models, because most vertices have to be updated through them to obtain final states for different snapshots. So, the DGNN models show significant *spatial locality*, which allows us to cache these vertices' input features in the on-chip memory for better locality and smaller off-chip communications. Based on these observations, we propose an effective *redundancy-aware incremental execution approach* to reduce redundant computations and data access cost in the processing of DGNN models. However, the software-based solution does not improve overall performance, because its overheads may outweigh its benefits.

We further propose the redundancy-aware accelerator, i.e., *RACE*, which enables efficient execution of the DGNN model, to reverse this situation. Specifically, RACE dynamically identifies the vertices with the same input features, neighbors, and neighbors' input features across successive graph snapshots through traversing the graph on the fly and then reuses these vertices' states of the previous snapshot for the latest snapshot. The states of the vertices affected by graph updates are recomputed based on the tracked information of their neighbors' input features. By such means, RACE can correctly and incrementally refine the results of the previous graph snapshot to quickly obtain the results for the latest graph snapshot for DGNN by fully exploiting the temporal similarity of DGNN models. Besides, RACE also dynamically identifies the most frequently accessed input features of vertices according to graph topology and then caches these data in the on-chip memory to fully exploit the spatial locality of DGNN models for fewer off-chip communications.

We have implemented RACE in an RTL targeting TSMC 12-nm library and conducted extensive experiments to verify its effectiveness. For DGNN inference, the results show that RACE outperforms the state-of-the-art software DGNN framework [8] running on Intel Xeon CPU and NVIDIA A100 GPU by on average 1139× and 84.7×, with on average 2242× and 234.2× energy savings, respectively. Compared with the state-of-the-art GNN accelerators, i.e., AWB-GCN [14], GCNAX [25], ReGNN [9], and I-GCN [15], RACE obtains on average 13.1×, 11.7×, 10.4×, and 7.9× speedup and 14.8×, 12.9×, 11.5×, and 8.9× energy savings, respectively, for DGNN inference.

2 BACKGROUND AND MOTIVATION

2.1 Background of DGNN

Discrete Time Dynamic Graphs (DTDG) [20, 29, 38]. A dynamic graph snapshot $G = \{G_1, \ldots, G_t, \cdots G_T\}$ and the input feature H make up a DTDG, where $G_t = (V_t, E_t)$, $t \in [1, T]$. V_t and E_t are the sets of vertices and edges of snapshot t, respectively. The E_t may be different for different snapshots due to edge insertion/deletion. A vertex addition/deletion is modeled by adding/deleting

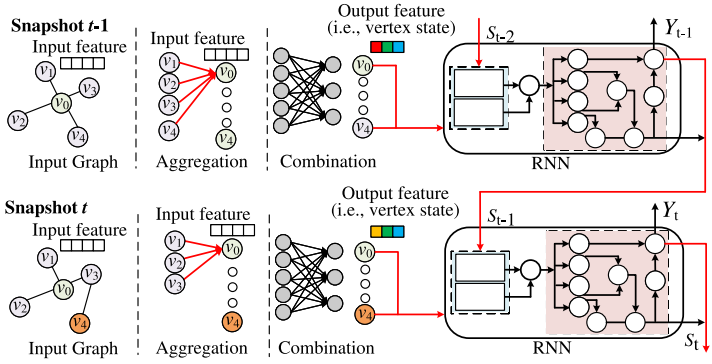


Fig. 1. Illustration of the processing of DGNN inference over a snapshot t.

this vertex's edges [8, 10, 20, 28, 29, 38]. G_t can be represented as a adjacency matrix A_t. The input feature vector of snapshot t is H_t. Note that A_t is extremely sparse and H_t is dense [25].

Dynamic graph neural network for DTDG (DGNN) [20, 28, 29, 38]. The inference of DGNN model [8] operates both GNN module and **Recurrent Neural Network (RNN)** module [34] on each snapshot of DTDG, and the snapshots of DGNN are processed one by one [8]. Figure 1 shows the processing of DGNN model over the snapshot t. In detail, let Y_t, A_t, and H_t represent the final output states, the input adjacency matrix, and the input features of all vertices over snapshot t, respectively. For each vertex v, the GNN module produces v's intermediate state of snapshot t through operating *Aggregation* and *Combination* phases according to A_t and $H_t[v]$, where $H_t[v]$ represents the vertex v's input feature of snapshot t. After that, RNN module produces the hidden state $S_t[v]$ and the final state $Y_t[v]$ using v's intermediate state of snapshot t and the hidden state $S_{t-1}[v]$, where $Y_t[v]$ is v's final output state of snapshot t and $S_{t-1}[v]$ is v's hidden state of snapshot t-1.

In detail, the computing pattern of GNN on layer k can be represented in Equation (1). After K layers of propagation, we will get the final features for each vertex over the snapshot t,

$$H_t{}^k[v] = Combination^k(Aggregation^k(H_t^{k-1}[u])|\forall u \in N(v) \cup \{v\}). \tag{1}$$

Different GNNs differ in the specific functions used for the two key phases. In **Graph Convolution Network (GCN)** [22], the aggregation processes the gathered features with *mean* function to obtain the aggregation result for each vertex (e.g., v) through aggregating the input states of the vertex v's neighbors (i.e., $N(v)$), while the FC-layer-based update computes $H_t{}^k[v]$ using the weights W^k.

For the RNN model of DGNN, the output states at snapshot t are calculated based on the input data at snapshot t and the hidden state at snapshot t-1. This computation involves matrix multiplication, element-wise multiplication and addition, and activation functions. We note that DGNN can be composed of different GNN models (e.g., Graph Attention Network [46]) and RNN models (e.g., **Long Short-Term Memory (LSTM)** [56]), which means that DGNN involves diverse and complex computational patterns.

2.2 Problems of Existing Solutions

When using existing solutions [8, 14, 15, 25, 51, 53] for handling DGNN inference, they suffer from redundant data accesses and computations across two consecutive snapshots. Specifically, since two consecutive snapshots evolve slightly [13] in terms of both graph structures and input features, the computation of GNN for the current snapshot involves the accessing of identical edge

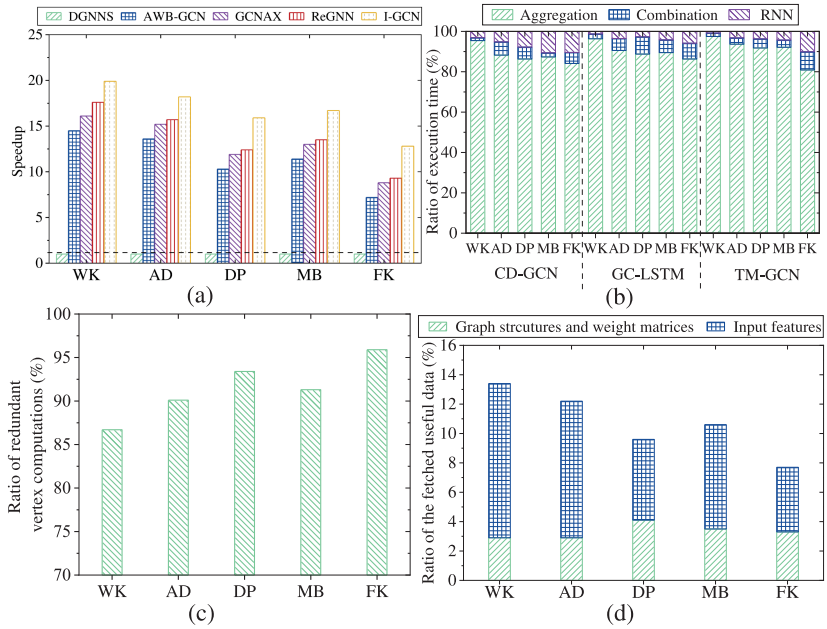


Fig. 2. Studies of the performance of DGNN inference: (a) the execution time normalized to that of DGNNS for TM-GCN; (b) the execution time breakdown of I-GCN; (c) the ratio of redundant vertex computations across two consecutive snapshots to all vertex computations for TM-GCN on I-GCN; (d) the ratio of the fetched useful data (i.e., after removing the fetched data associated with the redundant computations) to all fetched data for TM-GCN on I-GCN.

lists and the same input features for most vertices with respect with the previous snapshot. These redundant data accesses include the following: (1) the accesses to edge lists for *unaffected vertices* (which are the vertices with the same input features, one-hop neighbors, and one-hop neighbors' input features between consecutive snapshots) across consecutive snapshots and (2) the accesses to input features of the unaffected vertices (such a set of unaffected vertices are called a *unaffected vertices cluster*). Because the processing of each vertex needs to access its input feature and its one-hop neighbors' input features, the state aggregation and combination for each vertex in the *unaffected vertices cluster* is identical for two successive snapshots. Due to the above reasons, massive redundant data accesses and computations exist in DGNN when using existing solutions.

To demonstrate these issues, we evaluate five state-of-the-art solutions (i.e., DGNNS [8], AWB-GCN [14], GCNAX [25], ReGNN [9], and I-GCN [15], which recompute all graph data for each snapshot in DGNNs) by running various DGNN models on different datasets shown in Table 2. Section 4 depicts the details of the platform and benchmarks. Figure 2(a) shows that I-GCN outperforms the other solutions in all cases. However, in Figure 2(b), the aggregation phase still consumes the majority of the total execution time (e.g., more than 80.9% on the TM-GCN model over the dataset *FK*) for I-GCN. This is because of the following two main problems caused by recomputing all the graph data for each snapshot.

Redundant Computation Overhead. We explain it using the example of Figure 3. In this example, through aggregating the input features of the vertices v_0, v_1, and v_3 associated with the snapshot t using GCN [22], we can get the final state of v_0 for snapshot t. Similarly, the final states of v_1, v_2, v_3, v_4, v_5, and v_6 can be obtained. In the same way, we can also get the final states of all vertices of the snapshot t+1. However, because the weight matrices of all snapshots are the same

Table 2. Characteristic Statistics of the Real-world Datasets

Datasets	#*Vertices*	#*Edges*	*D*	*S*	*T*	$\Delta V \uparrow$	$\Delta V \downarrow$	$\Delta E \uparrow$	$\Delta E \downarrow$
Wikidata (WK) [4]	11,134	150,779	1,572	243	3 months	1.49%–2.1%	1.47%–2.92%	1.25%–2.12%	0.24%–1.1%
Academic (AD) [1]	51,060	794,552	2,849	568	1 month	1.14%–1.93%	1.22%–2.31%	0.62%–1.32%	0.61%–1.42%
DBLP (DP) [2]	315,159	1,615,400	25,468	200	1 month	0.76%–0.98%	0.91%–1.28%	0.96%–1.55%	0.93%–1.9%
Mobile (MB) [2]	340,751	2,200,203	13,452	397	1 day	0.98%–1.4%	0.67%–1.24%	1.13%–1.7%	1.10%–2.1%
Flicker (FK) [3]	1,715,256	22,613,981	32,105	134	1 second	0.48%–0.72%	0.31%–0.62%	0.23%–0.52%	0.22%–0.44%

D denotes the number of dimensions of feature vector; *S* denotes the number of snapshots; *T* denotes the time granularity; $\Delta V \uparrow$ and $\Delta V \downarrow$ represent the proportion of vertex increase and vertex deletion between two successive snapshots, respectively; $\Delta E \uparrow$ and $\Delta E \downarrow$ represent the proportion of edge increase and edge deletion between two successive snapshots, respectively.

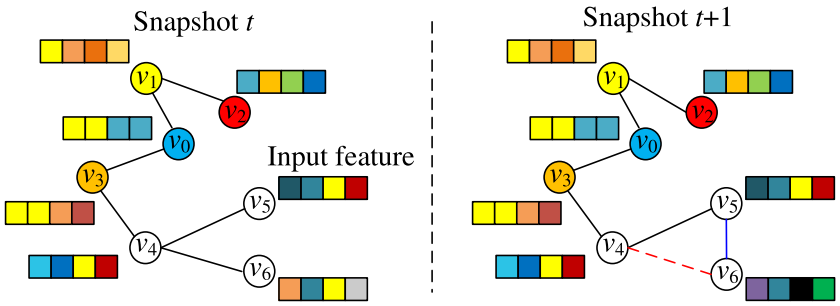


Fig. 3. An example graph, where the snapshot t+1 is the snapshot t with the deletion of the edge $v_4 \rightarrow v_6$, the addition of the edge $v_5 \rightarrow v_6$, and the mutation of v_6's input feature.

and v_0 has the same input feature, neighbors, and neighbors' input features between the snapshots t and t+1, the final states of v_0 of these two snapshots are completely identical for DGNN models. In other words, v_0 is an unaffected vertex across these two snapshots. Therefore, we do not need to recompute the final state of v_0 of the snapshot t+1 through accessing and aggregating the input features of both v_0 and v_0's neighbors of the snapshot t+1. Thus, massive redundant computations and data accesses can be eliminated. As shown in Figure 2(c), although I-GCN outperforms other solutions for all tested instances, more than 86.7% of vertex computations (i.e., the aggregation and combination of the vertex states) across two consecutive snapshots are redundant.

Irregular Memory Access. Using existing solutions [8, 14, 15, 25, 51, 53] for the DGNN inference, the input features of most vertices need frequent off-chip communications for the processing of each graph snapshot, because the size of these data is typically larger than that of the on-chip memory [14, 25, 51]. Specifically, for each batch of graph updates, only the input features of a small portion of vertices need to be loaded for processing in the DGNN models [8], and these graph data are sparsely dispersed in the off-chip memory. It incurs many irregular memory accesses to these graph data. Besides, the redundant computations incur many irregular memory accesses to the same input features for different snapshots, which exacerbates the above problem. As shown in Figure 2(d), most graph data (more than 86.6%) accessed by I-GCN via off-chip communication are redundant, because they are the same for two successive snapshots, resulting in the underutilization of memory bandwidth and on-chip memory. For the tested cases, the access time of graph data occupies 93.4%–97.6% of the execution time for I-GCN.

2.3 Similarities in DGNN Inference

Figure 4 shows the statistical studies on the characteristic of DGNNS. We have two observations regarding DGNN inference in this study, inspiring us to design our solution toward efficient DGNN inference.

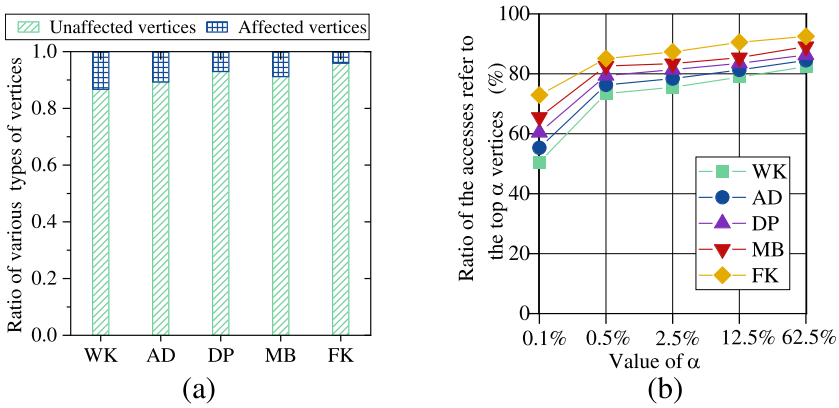


Fig. 4. Performance of TM-GCN model: (a) the ratio of different types of vertices to all vertices; (b) the ratio of the accesses refer to the input features of the top α vertices to those of all vertices.

Observation one: Most real-world dynamic graphs show strong temporal similarity*, which implies that there is substantial overlaps in the input features and neighbors between consecutive snapshots.* As shown in Figure 4(a), the unaffected vertices between two consecutive snapshots occupy 86.7%–95.9% of all vertices on average for different real-world dynamic graphs, because each batch of graph updates only affect a small portion of the vertices. It allows us to reuse most vertices' final states of the previous snapshot to incrementally obtain the final vertices' states of the latest snapshot quickly.

Observation two: The processing of DGNN models shows strong spatial locality*, i.e., most accesses of input features refer to those of a small set of vertices.* We evaluate the proportion of the accesses referring to the input features of the top α vertices to those of all vertices for the processing of each snapshot, where the vertices are sorted in the descending order according to the number of times for which their input features are accessed. Figure 4(b) shows that more than 73.4% of the accesses refer to the top 0.5% vertices' input features. Besides, for two successive snapshots, more than 80.3% of the accesses referring to the input features of the top 0.5% vertices of these two snapshots are the same on average. That is, most input features of the top 0.5% vertices in the previous snapshot are still frequently accessed when processing the latest snapshot. It motivates us to cache these vertices' input features in the on-chip memory for smaller off-chip communications across the processing of multiple snapshots.

3 OVERVIEW OF OUR SOLUTION

Based on the above observations, we propose an efficient accelerator *RACE*, which supports an effective *redundancy-aware incremental execution approach* for DGNN inference. This section first introduces our main idea and then discusses the implementation details of RACE.

3.1 Redundancy-aware Incremental Execution Approach

In this subsection, we present our redundancy-aware incremental execution approach, which can correctly and incrementally refine the results of the previous graph snapshot to quickly obtain the results for the latest graph snapshot and also enable regular accesses of vertices' input features. The details of our proposed approach are introduced below.

3.1.1 Redundancy-aware Incremental Processing. In fact, if all n-hop neighbors of a vertex (e.g., v_{root}) are unaffected vertices across two successive snapshots (e.g., the snapshots t and t+1 in

ALGORITHM 1: Redundancy-aware Incremental Processing

```
▷ Identify the unaffected vertices
for each vertex v ∈ G_{t+1} do
    if v's neighbors and their input features are the same between the snapshots t and t + 1 then
        v is set as an unaffected vertex
    end if
end for
▷ Calculate the n-hop aggregation dependency
for each vertex v ∈ Frontier do /*the nth BFS level*/
    Insert v's neighbors into the nth BFS level
end for
for each vertex u ∈ the nth BFS level do
    if all direct neighbors of u ∈ Frontier then
        u has n-hop aggregation dependency
    end if
end for
```

Figure 3), then we can consider that a dependency (called as *n-hop aggregation dependency*) exists between these two snapshots for this vertex (i.e., v_{root}). Under such circumstances, as proved in Section 3.1.3, this vertex's state on the nth layer of the snapshot t+1 keeps the same with that on the nth layer of the snapshot t. Thus, we can correctly reuse the state of this vertex on the nth layer of the snapshot t as its state on the nth layer of the snapshot t+1. This indicates that all state computations before the (n+1)th layer of the snapshot t+1 can be skipped for this vertex (i.e., v_{root}), significantly reducing the redundant computations and data accesses.

To efficiently identify the vertices' states that can be reused across multiple snapshots, as shown in Algorithm 1, it takes all unaffected vertices as the roots to cooperatively traverse the graph together in a breadth-first fashion on the fly to calculate *n-hop aggregation dependency*[1] for the vertices. Specifically, as shown in Figure 5, it obtains the unaffected vertices (e.g., v_0, v_1, v_2, and v_3 in Figure 3) and takes these vertices as the roots to traverse the graph. In Figure 5(a), it first calculates the *one-hop aggregation dependency* between two successive snapshots (e.g., the snapshots t and t+1 in Figure 3) for the vertices at the first BFS level. We can consider that v_0, v_1, and v_2 have *one-hop aggregation dependency*, because all direct neighbors of these vertices are unaffected vertices. At the second BFS level, as depicted in Figure 5(b), it can calculate *two-hop aggregation dependency* for v_1 and v_2, because all direct neighbors of these vertices have *one-hop aggregation dependency*. Then, at the third BFS level, as shown in Figure 5(c), the *three-hop aggregation dependency* will be calculated for v_2 because all direct neighbors of v_2 have *two-hop aggregation dependency*. With these *n-hop aggregation dependencies*, many redundant computations and data accesses can be eliminated. For example, because v_2 has a *three-hop aggregation dependency*, v_2's state on the third layer of the snapshot t can be reused as its state on the third layer of the snapshot t+1, sparing the computations of v_2's state on the first, second, and third layers of the snapshot t+1.

3.1.2 Topology-aware Data Caching. To further reduce the data access cost for DGNN, we alleviate the problem of irregular memory accesses via exploiting the spatial locality of DGNN.

[1]Note that if all direct neighbors of a vertex have (n-1)-*hop aggregation dependency* across two successive snapshots, then this vertex has a *n-hop aggregation dependency* between these two snapshots.

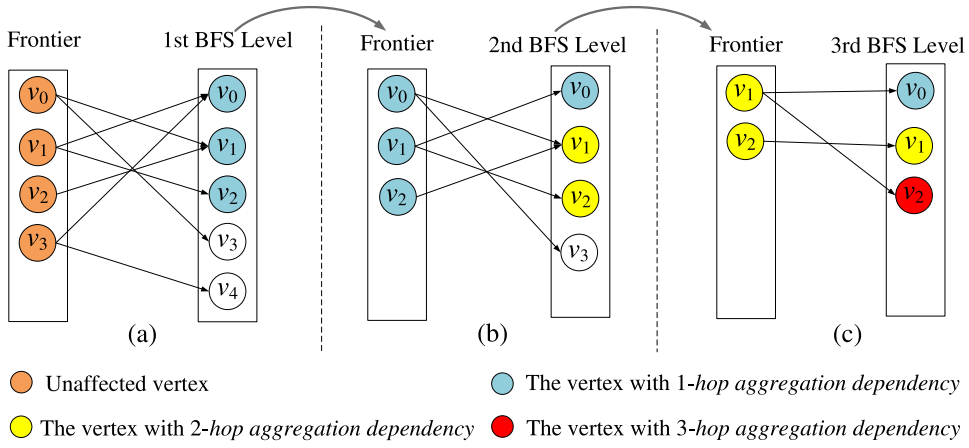


Fig. 5. Calculation of *n-hop aggregation dependency*.

Specifically, as shown in Figure 4(b), in the DGNN processing, most accesses of input features usually refer to those of a small set of vertices for the real-world graphs [16]. What is more, the input feature of a vertex would be accessed more times when this vertex has more neighbors that are not the unaffected vertices across two successive snapshots, because these neighbors need to access this vertex's input feature to recompute their states. Thus, before the processing of a snapshot, we can profile the number of each vertex's neighbors that are not the unaffected vertices across this snapshot and the previous snapshot and use this number to approximately predict the access frequency of this vertex's input feature in the processing of this snapshot.

Then, the most frequently accessed input features during the processing of this snapshot can be preferentially cached in the on-chip memory to achieve better data locality for the accesses to these input features. Besides, most cached input features can be reused in the processing of multiple snapshots, because the frequently accessed input features in the current snapshot are still frequently accessed in the processing of the latter snapshots. In other words, our approach enables one off-chip communication to serve multiple vertex state computations (which need to access the off-chip memory more times in existing solutions).

3.1.3 Correctness of Our Approach. This subsection proves the correctness of our *redundancy-aware incremental execution approach*. First, for each vertex with one-hop aggregation dependency between two successive snapshots, this vertex's state on the first layer of the current snapshot is the same as that of the previous snapshot, because the input features, neighbors, and neighbors' input features are the same for this vertex across the first layer of these two successive snapshots and the weight matrix W is shared with all snapshots. Next, we prove the following theorem to ensure the correctness of our approach under any circumstance.

Theorem 1. *If a vertex v has n-hop ($n > 1$) aggregation dependency between two successive snapshots, then v's state on the nth layer of the current snapshot is the same as that on the nth layer of the previous snapshot.*

Proof. We use inductive hypothesis to prove it. When $n = 1$, it is correct as the above described. Then, assume that it is correct for n-1, and we need to prove it is correct for n. It can be obtained that $H_t^n(v) = GNN(A_t, H_t^{n-1}(N(v))), W)$ according to the definition of GNN, where $N(v)$ and $H_t^n(v)$ denote the neighbors and input feature of v on the nth layer of snapshot t, respectively. In

this case, we have

$$\begin{aligned} H_t^{n-1}(v) &= GNN\left(A_t, H_t^{n-2}\left(N\left(v\right)\right), W\right) \\ &= H_{t+1}^{n-1}(v) \\ &= GNN\left(A_{t+1}, H_{t+1}^{n-2}\left(N\left(v\right)\right), W\right). \end{aligned} \tag{2}$$

After substituting Equation (2) into Equation (3), we have

$$\begin{aligned} H_t^{n}(v) &= GNN\left(A_t, H_t^{n-1}\left(N\left(v\right)\right), W\right) \\ &= GNN\left(A_t, GNN\left(A_t, H_t^{n-2}\left(N\left(N\left(v\right)\right)\right), W\right), W\right) \\ &= GNN\left(A_t, GNN\left(A_{t+1}, H_{t+1}^{n-2}\left(N\left(N\left(v\right)\right)\right), W\right), W\right) \\ &= GNN\left(A_t, H_{t+1}^{n-1}\left(N\left(v\right)\right), W\right). \end{aligned} \tag{3}$$

Because the vertex v has n-hop ($n \geq 1$) aggregation dependency between the snapshots t and $t+1$, according to the definition, we have

$$\begin{aligned} H_t^{n}(v) &= GNN\left(A_t, H_{t+1}^{n-1}\left(N\left(v\right)\right), W\right) \\ &= GNN\left(A_{t+1}, H_{t+1}^{n-1}\left(N\left(v\right)\right), W\right) \\ &= H_{t+1}^{n}(v). \end{aligned} \tag{4}$$

Equation (4) shows that the state of v on the nth layer of the current snapshot $t+1$ is the same as that on the nth layer of the previous snapshot t and also proves the correctness of our approach. Now, we can conclude that the state of v is the same for two successive snapshots if this vertex has n-hop ($n \geq 1$) aggregation dependency between these two snapshots. □

3.1.4 Benefits of Customization. Although our *redundancy-aware incremental execution approach* ensures much fewer redundant computations and data accesses, the software-only implementation of our approach suffers from high runtime cost on existing architectures (see Figure 11(a)) due to the following two reasons. First, our approach needs to track the unaffected vertices through irregularly traversing the graph on the fly. Such irregularity makes our approach ill suited to the **processing elements (PEs)** and the memory hierarchy of the general-purpose processors (e.g., CPU and GPU). Second, our approach incurs additional instructions and also low instruction level parallelism in the general-purpose processors due to the data-dependent branches (which depend on irregular graph structures) of these instructions. Such high runtime overhead outweighs the performance improvement brought by our approach in general-purpose processors. To tackle the inefficiency and non-adaptability of existing architectures, there is an urgent need for a custom accelerator for efficient execution of DGNN inference by efficiently supporting our approach. It is because that the custom accelerator can use customized PEs with multiple specific hardware pipelines to accelerate the operations and run them with the dataflow style of parallelism. Besides, it can use specialized memory hierarchy to obtain the higher memory-level parallelism, better data locality, and the shorter data access latency. The experiments show that our accelerator significantly outperforms existing solutions.

3.2 RACE Overview

To achieve lower redundant computation and data access overhead, we design *RACE*, a customized accelerator for efficient DGNN inference. RACE contains two key hardware units (i.e., **redundancy identification unit (RIU)** and **redundancy-aware processing unit (RPU)**) and on-chip buffers, as depicted in Figure 6. The RIU computes the *n-hop aggregation dependency* for each vertex and stores the computed dependency in a queue, called *Dependency_Queue*, where each entry

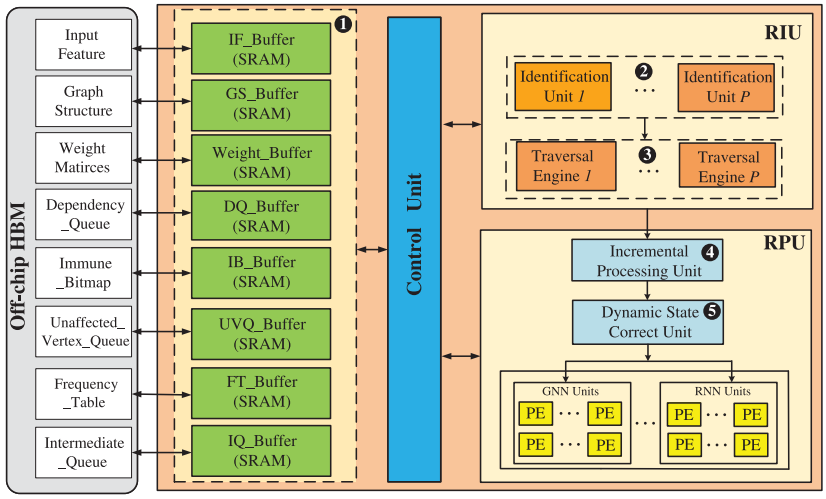


Fig. 6. Architecture of RACE.

stores the hop n for each vertex. The RPU follows RIU and conducts incremental computation of DGNN inference (i.e., GNN module and RNN module) to eliminate GNN operations for vertices involving redundant computation by consulting the *Dependency_Queue* and forwards the remaining vertices to the GNN units. The outputs of the GNN units are fed into the RNN units to produce the final results of current snapshot. Both the GNN unit and RNN unit have multiple PEs, and each PE has multiple **multiply and accumulate units (MACs)**. Since the elimination of redundant GNN operations increases the sparsity of adjacency matrices, the GNN unit adopts columnwise sparse matrix matrix multiplication as AWB-GCN [14]. The on-chip buffers are used to cache different types of data to reduce off-chip communications. The main functionalities of RIU, RPU, and on-chip buffers are as follows.

RIU. The RIU has two types of components: the **identification unit (IU)** and the **traversal engine (TE)**, as shown in Figure 6. The IU reads and compares the input features in two consecutive snapshots to mark the vertices with immune input features across these two snapshots. The comparison result of each vertex is stored in a bitmap, i.e., *Immune_Bitmap*, which is used to indicate whether a vertex's input feature is immune across two successive snapshots. For each vertex, the IU further detects whether this vertex's one-hop neighbors and their input features are immune across two successive snapshots and updates the queue, i.e., *Unaffected_Vertex_Queue*, which stores the IDs of all vertices in the *unaffected vertices cluster*. The TE determines whether a vertex has *n-hop aggregation dependency* with the help of the *Unaffected_Vertex_Queue*. If a vertex has *n-hop aggregation dependency*, then the value stored in the *Dependency_Queue*'s entry associated with this vertex is updated to n, where $n \in [1, N]$ and N is the number of GNN layers. The vertex v's *Dependency_Queue* entry with the value of n indicates that the calculation of the hidden features of v for the first n GNN layers can be eliminated.

RPU. The RPU contains an **Incremental Processing Unit (IPU)**, a **Dynamic State Correction Unit (DSCU)**, and multiple PEs. The IPU decides whether the GNN computation of the layer n for a vertex v can be skipped, to avoid the redundant computation. If the value (e.g., k) of the *Dependency_Queue* entry for the vertx v is larger than or equal to n (i.e., $k \geq n$), then the computations of v's hidden features for the first n GNN layers can be removed. Otherwise, the DSCU needs to assign and perform the computing of the hidden feature of v for the $(k+1)$th GNN layer of the latest snapshot on the PEs. When all GNN layers have been processed for the current snapshot, the

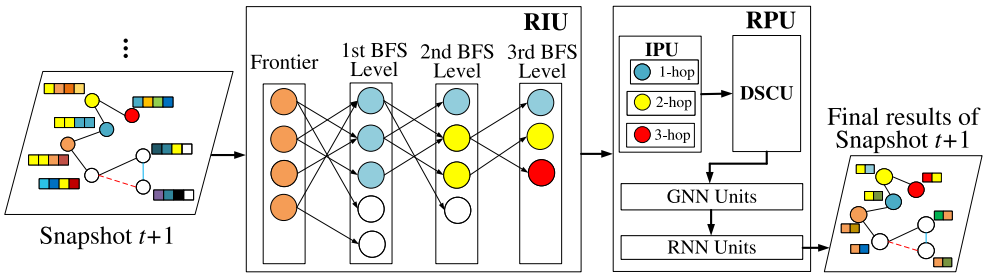


Fig. 7. Workflow of RACE.

outputs of the last GNN layer are used by the RNN to produce the latest snapshot's final results. The weight matrices of both GNN and RNN are cached in the *Weight_Buffer*.

On-chip Buffers. RACE has multiple on-chip buffers, e.g., *IF_Buffer*, *GS_Buffer*, and *Weight_Buffer*, to isolate the accesses of different types of data (e.g., input features, graph structure data, and weight matrices). This can avoid frequent data thrashing among different types of data. To fully and efficiently exploit the spatial locality of the DGNN, we also use our topology-aware data caching scheme with hardware implementations to avoid the data thrashing of the frequently accessed input features, and the access frequency of each vertex is recorded in a table, i.e., *Frequency_Table*.

The Workflow of RACE. The processing of each snapshot for DGNN includes both GNN and RNN operations. For the processing of the first snapshot, the RACE computes the hidden features for each vertex in all GNN layers, and the hidden features of the last GNN layer are forwarded to the RNN unit. The computed hidden features of the GNN layers are stored in the off-chip memory and are prepared for their potential reuse in the next snapshot.

After that, as shown in Figure 7, upon an arrival of a snapshot t+1, RACE loads the data from the off-chip **High Bandwidth Memory (HBM)** to the on-chip SRAM (step ❶). The RIU compares the input features and graph topology of the current snapshot t+1 and the previous snapshot t (step ❷), determines whether a vertex has *n-hop aggregation dependency*, and updates *Dependency_Queue* (step ❸). The operations of RIU are done after processing all vertices. Then, if the value of a vertex (e.g., v) in the *Dependency_Queue* is larger than or equal to n, then the RPU can avoid the computation of the hidden features of v for the first n GNN layers by reusing the states of the first n GNN layers computed in the previous snapshot, respectively (step ❹). Otherwise, the RPU instructs the GNN units to conduct user-defined GNN operations (e.g., GCN [22] of CD-GCN [29]) to compute the states of the remaining GNN layers for v. When the processing of GNN is done, the RNN unit is then initiated to conduct user-defined RNN operations (e.g., LSTM [34] of CD-GCN [29] and M-transform [28] of TM-GCN [28]) to compute the final results of the current snapshot t+1 (step ❺).

3.3 Hardware Design

This subsection describes the details of the key components of RACE. Note that each graph snapshot is stored in the **Compressed Sparse Row (CSR)** [8, 25, 27, 33] format by default, because CSR is the most popular format. Specifically, three arrays, i.e., *Offset_Array*, *Neighbor_Array*, and *Vertex_Feature_Array*, are used. The *Offset_Array* records the beginning and end offsets of each vertex's neighbors in the *Neighbor_Array*, while the *Neighbor_Array* stores the outgoing neighbors for each vertex. *Vertex_Feature_Array* maintains the algorithm-specific feature for each vertex. Note that, like HyGCN [51], each snapshot is divided into multiple partitions, which are maintained in the off-chip HBM. When the required data are not maintained in the on-chip SRAM, the relevant partition is retrieved from the off-chip HBM.

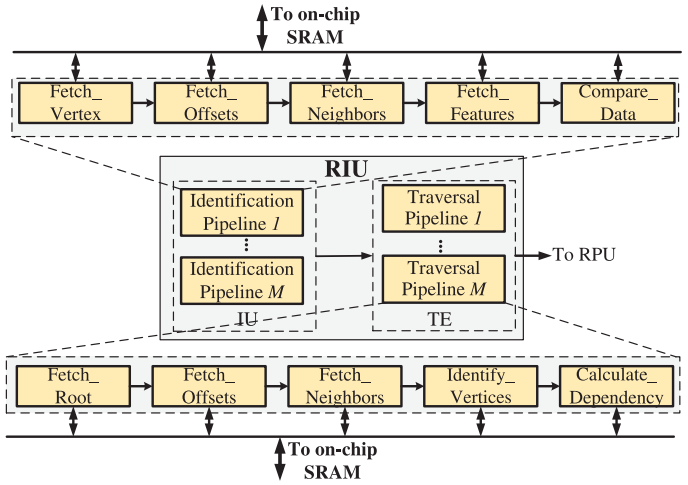


Fig. 8. Microarchitecture of RIU.

3.3.1 Microarchitecture of RIU. Upon receiving a new snapshot, the RIU begins to compute the *n-hop aggregation dependency* between the current and the previous snapshots for each vertex. Figure 8 describes the microarchitecture of RIU. The RIU first updates *Immune_Bitmap* and calculates the *unaffected vertices cluster* and then obtains the *n-hop aggregation dependency* through traversing the graph accordingly.

Identification of Unaffected Vertices. For each vertex, the IU updates *Immune_Bitmap* and calculates the *unaffected vertices cluster* through fetching and comparing its input feature, its one-hop neighbors, and these neighbors' input features from the previous snapshot t and the current snapshot t+1. To efficiently identify the unaffected vertices across these two snapshots, five stages, i.e., *Fetch_Vertex*, *Fetch_Offsets*, *Fetch_Neighbors*, *Fetch_Features*, and *Compare_Data*, are performed, which are implemented as a pipeline, i.e, the *Identification Pipeline*, as shown in Figure 8.

In detail, the *Fetch_Vertex* stage sequentially scans the vertices of the snapshots t and t+1 and outputs the ID of the same vertices (e.g., v_i) of these two snapshots. Then, the *Fetch_Offsets* stage fetches the beginning and end offsets of v_i's neighbors from the *Vertices_array* of the snapshots t and t+1, respectively. Next, the IDs of v_i's neighbors are fetched by the *Fetch_Neighbors* stage from the *Edges_Array* for the snapshots t and t+1, respectively. In the *Fetch_Features* stage, the input features of v_i and v_i's neighbors are fetched from the *Vertex_Feature_Array* for the snapshots t and t+1, respectively. Finally, the *Compare_Data* stage checks whether v_i's input features of the snapshots t and t+1 are different. If so, then nothing will be done. Otherwise, the corresponding entry of v_i in the *Immune_Bitmap* is set as 1, which indicates that v_i has the same input feature across the snapshots t and t+1. Then, it further identifies whether v_i's neighbors and their input features are immune across the snapshots t and t+1. If so, then v_i's ID is inserted into the *Unaffected_Vertex_Queue*. Otherwise, and v_i is set as an affected vertex and is sent to the RPU. Note that if the corresponding entry of a vertex (e.g., v_j) in the *Immune_Bitmap* is 1 (which indicates that the input features $H_t[v_j]$ and $H_{t+1}[v_j]$ are the same), then the accesses and comparison of $H_t[v_j]$ and $H_{t+1}[v_j]$ can be skipped. To balance the pipeline design, RACE replicates the *Fetch_Neighbors* and *Fetch_Features* units and parallelizes their accesses.

Calculation of *n*-hop aggregation dependency. When the *unaffected vertices cluster* has been obtained by the IU, the TE of RIU calculates the *n-hop aggregation dependency* between the snapshots t and t+1 for each vertex by taking the unaffected vertices as the roots to cooperatively traverse the snapshot t+1 in a breadth-first fashion on the fly.

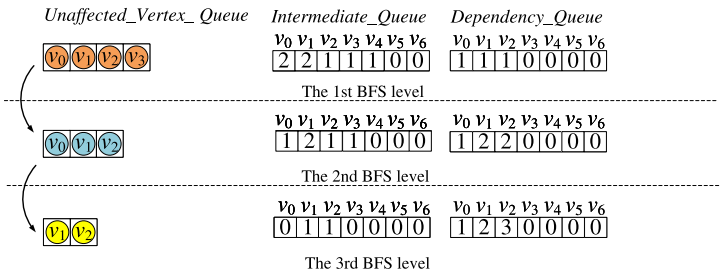


Fig. 9. The values in the data structures during the calculation of *n-hop aggregation dependency*.

To efficiently calculate the *n-hop aggregation dependency* in parallel, TE evenly divides the *Unaffected_Vertex_Queue* into several continuous ranges and multiple *Traversal Pipelines* are used. Each *Traversal Pipeline* is responsible for the graph traversals originated from the vertices in a continuous range of the *Unaffected_Vertex_Queue*. Each pipeline has five stages, i.e., *Fetch_Root*, *Fetch_Offsets*, *Fetch_Neighbors*, *Identify_Vertices*, and *Calculate_Dependency*, which are managed by a Traversal Finite State Machine. During the graph traversal, a register is employed to record the value of the current BFS level (i.e., L), and a queue (i.e., *Intermediate_Queue*) is used for each vertex to store the number of its direct neighbors with (L-1)-*hop aggregation dependency*. Note that the initial value of L is 1, and each vertex's initial value in *Intermediate_Queue* is 0.

During the graph traversal, for each pipeline, as shown in Figure 8, in the *Fetch_Root* stage, it sequentially fetches a vertex (e.g., v_2 in Figure 9) from its corresponding range of the *Unaffected_Vertex_Queue*. Then, the *Fetch_Offsets* stage fetches the beginning and end offsets of v_2 from the *Offset_Array* of the snapshot t+1. In the *Fetch_Neighbors* stage, v_2's neighbors are fetched from the *Neighbor_Array* of the snapshot t+1, and the values associated with these fetched neighbors in *Intermediate_Queue* are increased by one, respectively. The above four stages repeat until all vertices in the *Unaffected_Vertex_Queue* have been traversed.

The *Calculate_Dependency* stage identifies whether the value associated with each vertex in *Intermediate_Queue* equals the number of this vertex's neighbors, which is obtained by subtracting this vertex's beginning offset from its end offset (the beginning and the end offsets can be obtained from the *Offset_Array*). If so, then the ID of this vertex (e.g., v_2 in the third BFS level of Figure 9) is inserted into the *Unaffected_Vertex_Queue* and v_2's value stored in the *Dependency_Queue*'s entry is updated to be L (e.g., L=3 in the third BFS level). It means that v_2 has three-*three-hop aggregation dependency* between the snapshots t and t+1, since all direct neighbors of v_2 have *two-hop aggregation dependency* across these snapshots. To approximately evaluate the access frequency of each vertex's input feature, the *Calculate_Dependency* stage also calculates the number of the Lth hop neighbors that are not the unaffected vertices for this vertex.

When all vertices in the *Intermediate_Queue* have been processed in the *Calculate_Dependency* stage, the value of L is set to L+1 and the value of each entry in *Intermediate_Queue* is initialized with 0. The above five stages repeat until there is no vertex in the *Unaffected_Vertex_Queue*. Note that, RACE replicates the *Fetch_Neighbors* unit and parallelizes their accesses to balance the pipeline.

3.3.2 Microarchitecture of RPU. According to the *n-hop aggregation dependency* obtained by the RIU, the RPU is employed to reuse the results of the previous graph snapshot to quickly obtain the results for the current graph snapshot. Specifically, the RPU processes each snapshot t+1 layer by layer for GNN. When processing a vertex (e.g., v) in a GNN layer (e.g., the nth GNN layer) for the snapshot t+1, the IPU of RPU first scans the *Dependency_Queue* to identify whether the value (e.g., k) of the *Dependency_Queue* entry associated with this vertex v is larger than or equal to n. If so,

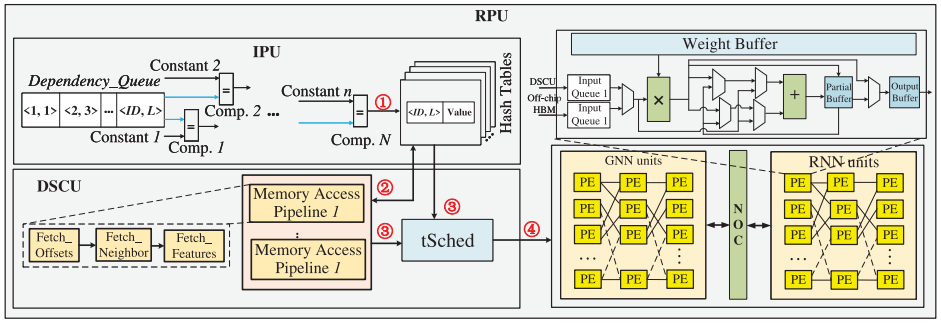


Fig. 10. Microarchitecture of RPU.

then IPU directly reuses v's state of the nth GNN layer of the previous snapshot t as that of v for the nth GNN layer of the current snapshot t+1, thereby skipping many redundant computations. Otherwise, the DSCU of RPU fetches the graph data (i.e., its input feature, its neighbors, and these neighbors' input features) associated with this vertex v of the current snapshot t+1 to recompute its state for the nth GNN layer of the snapshot t+1 on the GNN units.

In detail, as shown in Figure 10, IPU judges whether the value of a vertex in the *Dependency_Queue* is greater than or equal to n by using the comparator. The comparator (e.g., Comp.1) compares the L value of the vertex ID with the constant value n and outputs the final result. The output result is then saved in a hash table in the form of a triple <*ID, L, Value*> (step ❶), where *ID* is the vertex (e.g., v) with aggregation dependency in the current snapshot t+1, L denotes the vertex v's state of the Lth GNN layer of the previous snapshot t is identical with the current snapshot t + 1, and $Value$ is the output state of v in the Lth layer of GNN of the previous snapshot. To further accelerate this process, RPU employs multiple comparators and works in a pipeline manner. Specifically, multiple comparators are used to compare the L value of the vertex ID with a sequence of constant values: $1, 2, \ldots, n$. Note that IPU keeps the constant value n in the register. The comparators are connected in a pipeline manner, with the output of one comparator serving as the input to the next one.

When a comparator (e.g., Comp.1) compares the L value of the vertex ID with the constant value 1 and outputs a match signal, the data is stored in the hash table. If the data does not match, then it is passed to the next comparator (i.e., Comp.2) in the pipeline, which compares the L value of the vertex ID with the constant value 2. This process continues until the data matches one of the comparators in the pipeline. Using this pipeline of comparators, greatly improves the efficiency of the value-matching process. To prevent pipeline stalls and efficiently fetch the $Value$ of vertex v of previous snapshot t, because they are stored in the off-chip memory (step ❸), three stages are implemented by the **Memory Access (MA)** as a pipeline (step ❷). First, in the *Fetch_Offsets* stage, MA fetches the beginning and end offsets of v's neighbors from the *Vertices_array* of the current snapshot. Second, in the *Fetch_Neighbors* stage, the IDs of v's neighbors are repeatedly fetched from the *Edges_Array*. Finally, in the *Fetch_Features* stage, MA fetches the input features or the states of v and v's neighbors of the Lth layer of the GNN of the previous snapshot from the off-chip HBM or on-chip SRAM accordingly.

To efficiently fetch and process the graph data of v for its recomputation of a GNN layer of the snapshot t+1, DSUC uses multiple MA pipelines to accelerate the memory access of the vertex v (step ❸). The GNN and RNN task scheduler (*tSched*) guarantees the pipeline execution and assigns the workloads to the PE group, which works in a task disperse aggregation mode similar to HyGCN [51] for workload balance and task-level parallelism (step ❹). To minimize the data movement, RPU adopts a few-to-all PE crossbar between different compute units, which has been

widely used in neural network accelerators and can effectively reduce the number and area of the crossbar. To efficiently support DGNN computations, RPU proposed a unique PE that supports both GNN computation and RNN computation.

Specifically, each PE unit contains a multiplier and an adder. There are four multiplexers (MUX) inserted between the multiplier and adder, and one MUX inserted before the multiplier. As shown in Figure 10, the first MUX on the left is used to select the current computation pattern, that is, whether to perform GNN computation or RNN computation. If the GNN computation is to be performed, then MUX will select the graph data of the affected vertices that need to be computed in the current snapshot. Otherwise, the MUX will choose to obtain the time information of the RNN in the previous snapshot and the intermediate states of GNN in the current snapshot. Then, the two MUXs on the left are flipped in each cycle to ensure that data is evenly transmitted to the two input ports of the adder, while the two MUXs on the right select the adder's input as the input of the PE or the output of the multiplier. The output of the adder is then sent to the partial output queue for accumulation. Finally, before the output queue, a MUX is used to select the appropriate output from the multiplier or the adder.

However, when the GNN computation of the affected vertices is not completed, the RNN unit will be idle, because the outputs of the last GNN layer are fed into the RNN units to produce the final results of the current snapshot t+1, resulting in lower hardware resource utilization. To address this issue, *tSched* of DSCU assigns GNN computations to idle computing units within the RNN unit. In this way, the GNN unit and RNN unit can be further assembled together to form a large computation unit, which enables a larger set of elements to perform matrix–matrix multiplication and element-wise multiplication/addition operations simultaneously.

3.3.3 Topology-aware Memory Hierarchy. The memory hierarchy is shown in Figure 6. Eight on-chip SRAM buffers, i.e., *IF_Buffer*, *GS_Buffer*, *Weight_Buffer*, *DQ_Buffer*, *IB_Buffer*, *UVQ_Buffer*, *FT_Buffer*, and *IQ_Buffer*, are used to cache the input features, graph structure data, weight matrices, *Dependency_Queue*, *Immune_Bitmap*, *Unaffected_Vertex_Queue*, *Frequency_Table*, *Intermediate_Queue*, respectively, aiming to exploit data locality to reduce data access latency. With these buffers, the different types of data are isolated effectively, avoiding the potential access conflicts and data thrashing between them. When the memory requests (e.g., requested by different stages of the pipelines in RIU) are generated, these requests are assigned to a FIFO *request buffer* so as to access the data in the corresponding on-chip SRAM buffer in parallel.

To fully leverage the spatial locality of DGNN for smaller data access cost, RACE preferentially caches the frequently accessed input features in the *IF_Buffer*, while the other input features are directly fetched from the off-chip memory on demand. Specifically, RACE employs our proposed topology-aware data caching scheme to manage the *IF_Buffer*. It uses a threshold *TD* to record the lowest value of the access frequency of all input features cached in the *IF_Buffer*, where the initial value of *TD* is zero and is updated by RACE at runtime. The access frequency of each input feature is profiled by RIU. For each input feature (e.g., $H_t[v]$) fetched from the off-chip memory, RACE determines whether the access frequency of $H_t[v]$ is less than the value of *TD*. If so, then $H_t[v]$ will not be cached in the *IF_Buffer*. Otherwise, an input feature with the access frequency equals to *TD* is dynamically evicted from the *IF_Buffer*, and $H_t[v]$ is then cached in the *IF_Buffer* at runtime. The value of *TD* is also updated as the minimum value of the access frequencies of all input features cached in the *IF_Buffer* at this moment. Note that the fetched input feature is directly cached in the *IF_Buffer* if it is not full.

By such means, the input feature with the highest access frequency can be cached and directly obtained from the *IF_Buffer*, thereby achieving lower access latency and high utilization of the on-chip SRAM. Since the frequently accessed input features in the current snapshot are still frequently

Table 3. System Configurations of Compared Accelerator

	AWB-GCN	GCNAX	ReGNN	I-GCN	RACE
Compute	1 GHz @ 4096 MACs	1 GHz @ 1 × 16 MAC array	1 GHz @ 512 A-PEs, 2 × 16 P-PEs with radix-64 float tree adder and 2 systolic modules (each with 32 × 128 arrays)	1 GHz @ 4096 MACs	1 GHz @ 4096 MACs, 8 IUs and 8 TEs, 1 DSCU, and 1 IPU
On-chip Memory	12 MB	4 MB	20 MB	12 MB	4 MB
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	256 GB/s HBM 2.0

Table 4. Parameter Details of RACE on Xilinx Alveo U280 FPGA

Processing units	280 MHz @ 4096 MACs, 8 IUs, 8TEs, 1 DSCU, and 1 IPU
On-chip memory	*IF_Buffer* (2 MB), *GS_Buffer* (1 MB), *GS_Buffer* (1 MB), *DQ_Buffer* (32 KB), *IB_Buffer* (16 KB), *UVQ_Buffer* (24 KB), *IQ_Buffer* (48 KB), and *FT_Buffer* (32 KB)
Off-chip memory	256 GB/s HBM 2.0

accessed by the processing of the next snapshot, the cached input features can typically serve the processing of multiple snapshots, ensuring better data locality in DGNN.

4 EVALUATION

4.1 Experimental Setup

Hardware Simulator. To evaluate the performance of our accelerator RACE, we have built a cycle-accurate simulator. This simulator models each module of RACE faithfully and the modules' timing behaviors are co-verified with the synthesized RTL design. The simulator is also integrated with Ramulator [21], which supports HBM to estimate HBM timings and produce a command trace.

ASIC Synthesis. We implement and synthesize each module to measure the area, power, and critical path delay (in cycles). The Synopsys Design Compiler with the TSMC 12-nm library is used for the synthesis, and Synopsys PrimeTime PX is used to estimate the power. The area, power, and access latency of on-chip buffers are estimated using Cacti [36].

Baselines. The performance of RACE is compared with seven solutions, i.e., DGL (v0.9.0) [48], DGNNS-CPU, DGNNS [8], AWB-GCN [14], GCNAX [25], ReGNN [9], and I-GCN [15]. DGL is the most popular GNN framework. DGNNS is the state-of-the-art framework for DGNN. Both DGL and DGNNS run on the NVIDIA Tesla A100 with 6,912 cores and 80 GB HBM. DGNNS-CPU is the version of DGNNS running on the CPU platform (which has an Intel Xeon 6151 processor with 65 cores at 3.0 GHz and 696 GB DRAM). AWB-GCN, GCNAX, ReGNN, and I-GCN are the state-of-the-art GCN accelerators. Similarly to ReGNN [9], these accelerators run at 1 GHz, and their hardware configurations are listed in Table 3. We also conducted FPGA-based implementations to validate the RACE simulation infrastructures. Specifically, we have implemented RACE on a Xilinx Alveo U280 FPGA card, which is equipped with a XCU280 FPGA chip. The FPGA provides 9-MB BRAM resources, 1.3-M LUTs, 2.6-M Registers, 9,024 DSP slices, and two 4-GB HBM2 stacks. In addition, we use the BRAM resources to implemente the on-chip memory of RACE. More parameter details of RACE are listed in Table 4. We employ Xilinx Vivado 2019.1 to obtain the clock rate of RACE and conservatively use 280 MHz in our experiments. The resource utilization of all models are reported in Table 5. Although the existing state-of-the-art static GNN accelerators cannot directly support DGNN inference, we divide the workload of DGNN into two parts, enabling it to be translated into static GNN and RNN workloads. (1) The first part is pre-processing each snapshot of the input graph to conform to the input format of the existing accelerators. For instance, in the case of ReGNN [9], an additional edge feature array must be added besides the current graph

Table 5. Resource Utilization on Xilinx Alveo U280 FPGA

Resource	CDA-GNN	GC-LSTM	TM-GCN
DSP	24.3%	17.5%	21.4%
LUT	34.7%	26.3%	24.2%
FF	31.4%	29.4%	39.1%
BRAM	42.1%	46.2%	44.5%

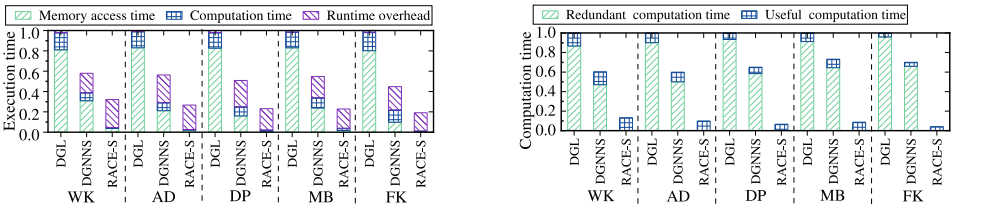
snapshot in the CSR format. Note that the time taken for this part is not considered in the DGNN inference time. (2) The second part is handling GNN and RNN computations of DGNN separately in the existing GNN accelerators. After GNN computations are completed, these intermediately states of GNN are stored in off-chip memory, and then the final result is output after completing the RNN computations. This is because the existing GNN accelerators cannot simultaneously support GNN and RNN computations. The communication time between GNN and RNN computation units is included in the DGNN inference time. Note that, to evaluate our software approach, we have also modified DGL to use our *redundancy-aware incremental execution approach* to support DGNN inference, and this software implementation is called *RACE-S*. Note that RACE-S runs on the above NVIDIA Tesla A100 in the following experiments, and it outperforms RACE-S running on the above CPU platform by 11.3× on average. To further verify the effectiveness and irreplaceability of each hardware module in RACE, we implemented the version of RACE using I-GCN to perform GCN computation, and this hardware implementation is called *RACE-I*. The reported results are measured for end-to-end system.

DGNN Models. To evaluate the performance of RACE, three typical DGNN models, i.e., CD-GCN [29], GC-LSTM [10], and TM-GCN [28], are used. For RNN operation, both CD-GCN and GC-LSTM use the typical LSTM [34], and the TM-GCN model uses M-transform [28], which is a parameter-less temporal aggregation mechanism. The number of layers are five, three, and three for CD-GCN, GC-LSTM, and TM-GCN, respectively. CD-GCN contains three graph convolution layers, one LSTM layer, and one fully connected layer. GC-LSTM has two graph convolution layers and one LSTM layer. TM-GCN has two graph convolution layers and one M-transform layer. Note that EvolveGCN [38] updates the weights along the timeline, AWB-GCN [14], GCNAX [25], ReGNN [9], and I-GCN [15]) cannot support EvolveGCN [38]. Therefore, we did not use it as the benchmark.

Dynamic Graph Datasets. Table 2 lists the five real-world dynamic graph datasets used in the evaluation, namely Wikidata (WK), Academic (AD), DBLP (DB), Mobile (MB), and Flicker (FK), where the graph edges are undirected. These datasets are characterized by various properties such as the number of vertices, edges, dimensions of feature vectors, number of snapshots, time granularity, and update statistics.

4.2 Experimental Results

4.2.1 Comparison with Software Approaches. Figure 11(a) shows the normalized execution time of different solutions over the TM-GCN model, where the execution time is broken down into memory access time, computation time, and the runtime overhead. This figure shows that the overall performance of RACE-S is superior to DGNNS, because DGNNS needs much more computation time and data access time than RACE-S. As shown in Figure 11(a), the memory access time of DGNNS is 8.6×–19.8× more than that of RACE-S for our tested instances. It is because DGNNS needs to recompute all graph data for each graph snapshot and suffers from significant redundant memory access overhead. Figure 11(b) shows the computation time breakdown of Figure 11(a). We can find that RACE-S reduces 86.7%–95.9% redundant computation time of DGNNS for the



(a) Execution time of various systems normalized to that of DGL (b) Computation time of various systems normalized to that of DGL

Fig. 11. Performance of different systems normalized to that of DGL over the NVIDIA Tesla A100.

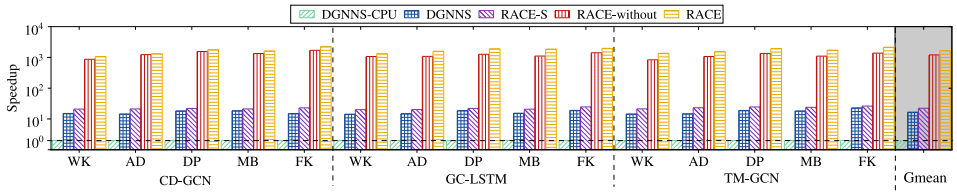


Fig. 12. Performance of different schemes normalized to that of DGNNS-CPU.

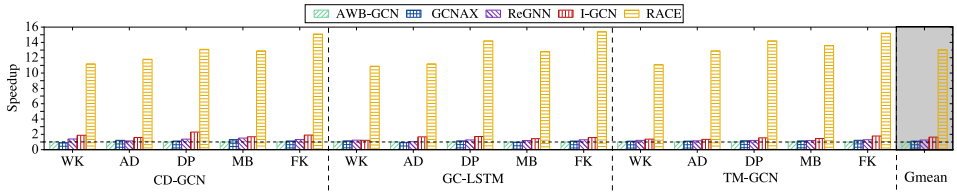


Fig. 13. Performance of different schemes normalized to that of AWB-GCN.

TM-GCN model, because RACE-S only needs to refine a very small proportion of the results of the previous snapshot to get the final results for the latest snapshot.

However, because RACE-S suffers from high runtime cost, RACE-S only outperforms DGNNS slightly. Figure 11(a) shows that the runtime overhead occupies 85.6%–96.3% of RACE-S's total execution time. It is because that RACE-S needs much time to traverse the graph on the fly and refine the results of previous snapshot. Compared with RACE-S, RACE not only ensures fewer redundant computations and memory accesses but also reduces the runtime cost of RACE-S, thereby enabling much better performance than existing solutions. Figure 12 shows that RACE outperforms DGNNS-CPU and DGNNS by 1038.4×–1240.8× and 111.8×–157.6×, respectively.

Note that the execution time to identify redundancy takes 14.9%–23.5% of RACE's total execution time, and the off-chip memory transfers that are used to identify redundancy take 15.3%–23.8% of RACE's total off-chip memory transfers.

Figure 12 also presents the performance contribution from different hardware units of RACE. It shows that RACE outperforms RACE-without (i.e., only the units associated with the redundancy-aware incremental processing are enabled while the units associated with the topology-aware data caching scheme are disabled) by 1.8×–2.6×, because the topology-aware data caching scheme can fully exploit the spatial locality in the DGNN models to ensure better data locality.

4.2.2 Comparison with Hardware Accelerators. Figure 13 shows that RACE outperforms AWB-GCN, GCNAX, ReGNN, and I-GCN by on average 13.1×, 11.7×, 10.4×, and 7.9×, respectively, because RACE can significantly reduce the redundant computation and data access overhead. Figure 14 depicts the volume of off-chip memory transfers of the accelerators normalized to

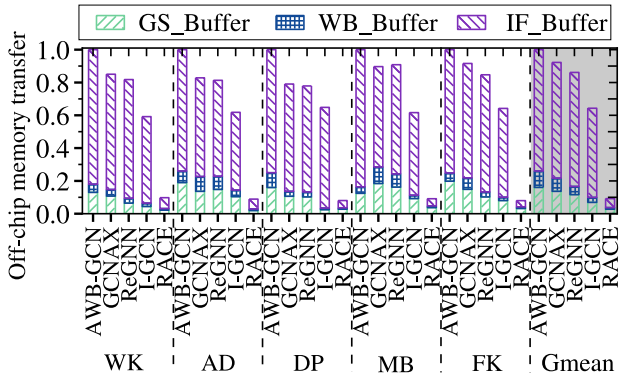


Fig. 14. Off-chip memory traffic of different types of data for different accelerators.

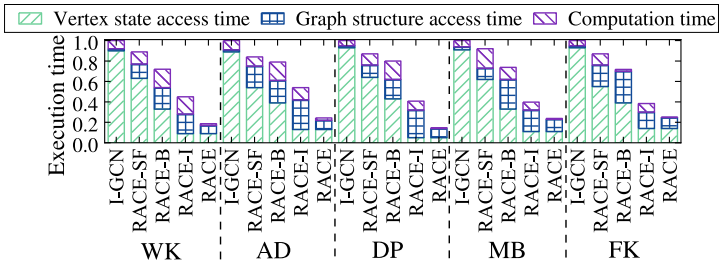


Fig. 15. Execution time of various schemes normalized to that of I-GCN.

that of AWB-GCN on TM-GCN. We can observe that I-GCN requires smaller volume of off-chip memory transfers than both AWB-GCN and GCNAX for the DGNN inference because of its better data locality and less computations. However, in Figure 14, the volume of off-chip memory transfers of I-GCN is still 4.2×–8.1× more than that of RACE. As shown in Figure 15, RACE-I performs worse than RACE, because I-GCN cannot use the information of n-hop aggregation dependency to perform incremental computations and needs to recalculate all vertices on each snapshot. Thus, RACE can effectively perform incremental computation when processing DGNN inference. Figure 16 shows the volume of off-chip memory transfers associated with different types of vertices normalized to that of AWB-GCN on TM-GCN. Most off-chip memory transfers (account for 83.1% of all off-chip memory transfers on average) of I-GCN refer to the unaffected vertices of multiple snapshots. Compared with I-GCN, RACE can avoid many redundant computations across multiple snapshots for DGNN, significantly reducing the off-chip memory transfers.

4.2.3 Area and Power Overhead. The total area of RACE is only 4.73 mm^2 under TSMC 12-nm technology. Figure 17 shows the area breakdown of the major components of RACE. The results show that most area is contributed by on-chip SRAM buffers and MAC arrays, which occupy 78.4% of the total area, while the total area of IU, TE, IPU, and DSCU only occupies 16.9%. It indicates that the IU, TE, IPU, and DSCU are designed to introduce only a small amount of area overhead, but it can significantly eliminate the redundancy in DGNNs and reduce the computation and off-chip communication. Figure 18 depicts the energy savings of RACE. The energy savings of RACE are 2002.4×–2482.2× and 198.9×–269.5× higher than that of DGNNS-CPU and DGNNS, respectively. Compared with existing hardware accelerators, i.e., AWB-GCN, GCNAX, ReGNN, and I-GCN, the energy savings of RACE are improved by on average 14.8×, 12.9×, 11.5×, and 8.9×, respectively.

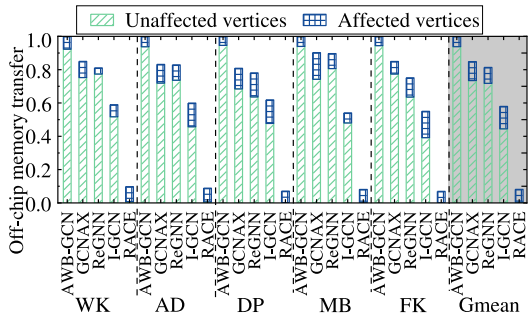


Fig. 16. Off-chip memory traffic associated with different types of vertices for different accelerators.

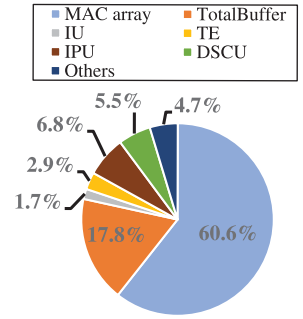


Fig. 17. Area breakdown of RACE.

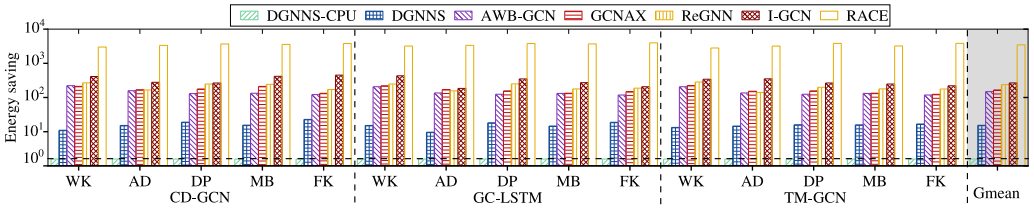


Fig. 18. Energy saving of different solutions normalized to that of DGNNS-CPU.

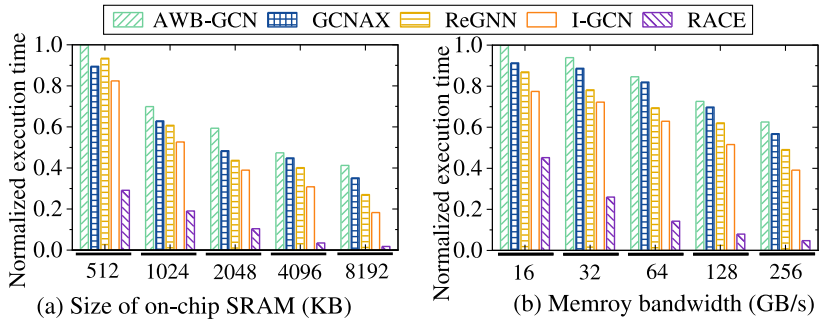


Fig. 19. Sensitivity studies of RACE over *FK*: (a) sensitivity to the total size of on-chip SRAM; (b) sensitivity to the memory bandwidth.

RACE achieves more energy savings than existing solutions due to much fewer redundant computations and off-chip memory transfers.

4.2.4 Sensitivity Studies. Figure 19(a) evaluates the sensitivity to the total size of the on-chip SRAM for TM-GCN. When the buffer size increases, RACE ensures higher performance than the other solutions, because of the higher utilization of on-chip SRAM. Figure 19(b) depicts the performance of various solutions with different values of memory bandwidth for TM-GCN. The results show that RACE outperforms the other solutions in all cases, because it fully exploits the memory bandwidth. Figure 20(a) shows the performance of RACE with different replacement strategies for the management of the *IF_Buffer*, i.e., LRU [19], DRRIP [18], P-OPT [6], and GRASP [12], on TM-GCN. It shows that our *topology-aware data caching* scheme outperforms the other schemes due to the fact that it can avoid the data thrashing of the input features that are most frequently

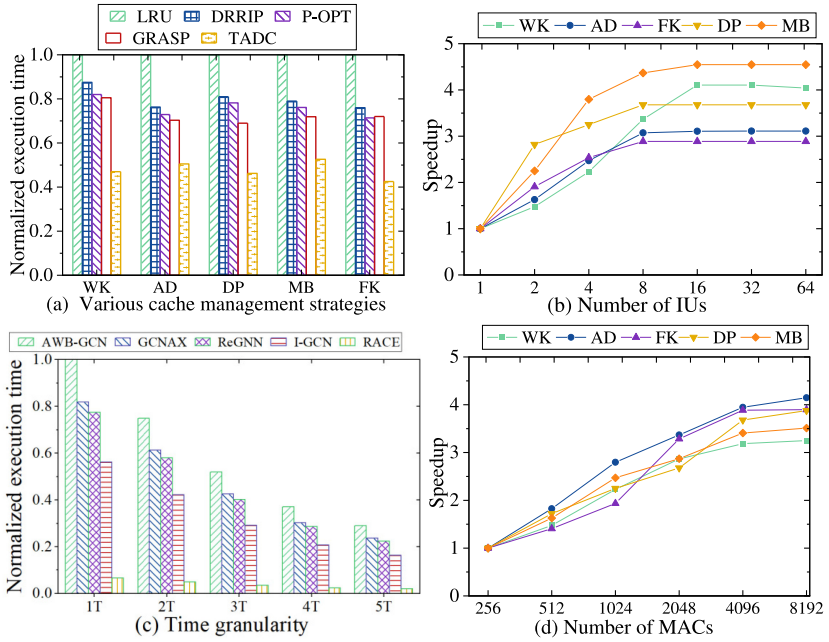


Fig. 20. Sensitivity studies of RACE: (a) sensitivity to the replacement strategies; (b) sensitivity to the number of IUs; (c) sensitivity to time granularity; (d) sensitivity to the number of MAC units.

accessed in the DGNN processing. Figure 20(b) describes the sensitivity to the number of IUs on TM-GCN. It shows that RACE can obtain better performance as the number of IUs increases until the the number of IUs reaches 8, because the memory bandwidth is saturated. Figure 20(c) evaluates the sensitivity to time granularity and shows that RACE still outperforms the other solutions in all cases. Figure 20(d) shows the performance of RACE with varying number of MAC units. As the number of MAC units increases, RACE exhibits an increasing performance gain. However, for sake of fairness and limited resource and memory bandwidth, we selected 4,096 MAC units as it is similar to the choices made by other state-of-the-art works (e.g., AWB-GCN [14] and I-GCN [15]).

5 RELATED WORK

Incremental computation and Software DGNN Solutions. Incremental computation [40, 45] is widely used. Kineograph [11] and KickStarter [47] use incremental computation for efficient dynamic graph processing, however, may yield inaccurate results under some circumstances. GraphBlot [31] and DZiG [30] are further proposed to trace the dependencies between intermediate results for correctness. Although all these software systems can support incremental computation, they do not support neural network computation in DGNN. Meanwhile, some solutions, e.g., DGNNS [8], are also proposed for efficient execution of DGNN through reducing the transfer time of vertices' adjacency matrix, and so on. Unfortunately, these solutions need to recompute all graph data of each graph snapshot and thus suffer from serious redundant computation and irregular memory access overhead. Note that many software temporal graph processing solutions [41, 43, 60] have been proposed to efficiently support temporal graph processing. However, these solutions also suffer from inter-snapshot redundant memory accesses for DGNN.

Hardware Dynamic Graph Processing Accelerators. Although many static graph processing accelerators [17, 35] have been developed, they suffer from significant redundant computation

and access overhead when processing the dynamic graph. To track these issues, some dynamic graph processing accelerators have been recently proposed. GraSU [49] employs a differential data management for graph updating, and Basak *et al.* [7] propose an input-aware software and hardware co-design to accelerate graph updating. DREDGE [32] provides a hardware accelerator to speedup the repartitioning of dynamic graphs. JetStream [39] is designed to use an event-driven computation model to efficiently support the delta-accumulative incremental computation [57] of dynamic graphs. TDGraph [59] further optimizes the incremental computation of dynamic graphs via synchronizing the vertex state propagations. Mint [44] presents a novel accelerator architecture and a programming model for mining temporal motifs efficiently. However, the GNN operations of DGNN are non-linear function [22] and cannot be expressed as the delta-accumulative computation [57] and the computation pattern of temporal motifs [44]. Thus, these accelerators cannot support the execution of the DGNN, although they can efficiently support the processing of the dynamic graphs.

Hardware GNN Accelerators. Many GNN accelerators [5, 26, 50, 52, 54–56, 58, 61] have been recently developed. HyGCN [51] proposes a window-based sliding and shrinking method to reduce the redundant accesses and improve the locality of non-zero elements in the adjacency matrix. To further reduce data movement, GCNAX [25] employs an adaptable and efficient dataflow to reconfigure the loop order and loop fusion strategy of matrix multiplication to adapt to different GCN accelerators configurations. To alleviate the data access irregularity of GCN, GCoD [53] proposes a split and conquer GCN training strategy. For load balancing of GCN inference, AWB-GCN [14] leverages three auto-tuning techniques to dynamically balance the workload for all processing elements to boost the efficiency. To reduce redundant computations and memory accesses of GNN, I-GCN [15] proposes an online graph restructuring algorithm, and ReGNN [9] proposes a dynamic redundancy elimination algorithm and a dynamic clipping algorithm.

However, these hardware accelerators are inefficient in resolving the redundant computations and irregular memory accesses for DGNN inference, because they also need to recompute all data of each snapshot and suffer from significant inter-snapshot redundancy (i.e., the redundant computation and accesses between successive snapshots) for DGNN.

6 CONCLUSION

This article proposes a redundancy-aware hardware accelerator *RACE* toward efficient DGNN inference. Through correctly and incrementally processing multiple snapshots for DGNN, RACE enables much smaller redundant computation and data access overhead. Experimental results show that RACE obtains on average 1139× and 84.7× speedups with average 2242× and 234.2× energy savings for DGNN inference over the start-of-the-art DGNN software systems running on the CPU and GPU, respectively. Moreover, RACE obtains on average 13.1×, 11.7×, 10.4×, 7.9× speedup and 14.8×, 12.9×, 11.5×, 8.9× energy savings over the state-of-the-art GNN accelerators, i.e., AWB-GCN, GCNAX, ReGNN, and I-GCN, respectively.

REFERENCES

[1] 2022. Academic. Retrieved from https://west.uni-koblenz.de/konect/networks
[2] 2022. DBLP and Mobile. Retrieved from https://dblp.uni-trier.de/xml/
[3] 2022. Flicker. Retrieved from https://socialnetworks.mpi-sws.org/data-imc2007.html
[4] 2022. Wikidata. Retrieved from https://github.com/mniepert/mmkb/tree/master/TemporalKGs/wikidata
[5] Adam Auten, Matthew Tomei, and Rakesh Kumar. 2020. Hardware acceleration of graph neural networks. In *Proceedings of the 57th ACM/IEEE Design Automation Conference*. 1–6.
[6] 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.

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

[8] 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*. 77:1–77:15.

[9] 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.

[10] Jinyin Chen, Xueke Wang, and Xuanheng Xu. 2021. GC-LSTM: Graph convolution embedded LSTM for dynamic network link prediction. *Appl. Intell.* 12, 1 (2021), 1–16.

[11] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. 2012. Kineograph: Taking the pulse of a fast-changing and connected world. In *Proceedings of the 7th European Conference on Computer Systems*. 85–98.

[12] Priyank Faldu, Jeff Diamond, and Boris Grot. 2020. Domain-specialized cache management for graph analytics. In *Proceedings of the 26th IEEE International Symposium on High Performance Computer Architecture*. 234–248.

[13] Wenfei Fan, Chunming Hu, and Chao Tian. 2017. Incremental graph computations: Doable and undoable. In *Proceedings of the ACM International Conference on Management of Data*. 155–169.

[14] Tong Geng, Ang Li, Runbin Shi, Chunshu Wu, Tianqi Wang, Yanfei Li, Pouya Haghi, Antonino Tumeo, Shuai Che, Steven K. Reinhardt, and Martin C. Herbordt. 2020. AWB-GCN: A graph convolutional network accelerator with runtime workload rebalancing. In *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture*. 922–936.

[15] 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.

[16] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*. 17–30.

[17] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. 2016. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*. 56:1–56:13.

[18] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely Jr., and Joel S. Emer. 2010. High performance cache replacement using re-reference interval prediction. In *Proceedings of the 37th International Symposium on Computer Architecture*. 60–71.

[19] Daniel A. Jiménez. 2013. Insertion and promotion for tree-based PseudoLRU last-level caches. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. 284–296.

[20] Seyed Mehran Kazemi, Rishab Goel, Kshitij Jain, Ivan Kobyzev, Akshay Sethi, Peter Forsyth, and Pascal Poupart. 2020. Representation learning for dynamic graphs: A survey. *J. Mach. Learn. Res.* 21, 5 (2020), 70:1–70:73.

[21] Yoongu Kim, Weikun Yang, and Onur Mutlu. 2016. Ramulator: A fast and extensible DRAM simulator. *IEEE Comput. Arch. Lette.* 15, 1 (2016), 45–49.

[22] 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.

[23] Ekkehard Köhler, Rolf H. Möhring, and Martin Skutella. 2009. Traffic networks and flows over time. In *Algorithmics of Large and Complex Networks*. Springer, 166–196.

[24] 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 2019 IEEE Conference on Computer Communications*. 388–396.

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

[26] Shengwen Liang, Cheng Liu, Ying Wang, Huawei Li, and Xiaowei Li. 2020. DeepBurning-GL: An automated framework for generating graph neural network accelerators. In *Proceedings of the IEEE/ACM International Conference On Computer Aided Design*. 72:1–72:9.

[27] Xianzhu Liu, Zhijian Ji, and Ting Hou. 2019. Graph partitions and the controllability of directed signed networks. *Sci. Chin. Inf. Sci.* 62, 4 (2019), 42202:1–42202:11.

[28] 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*. 729–737.

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

[30] Mugilan Mariappan, Joanna Che, and Keval Vora. 2021. DZiG: Sparsity-aware incremental processing of streaming graphs. In *Proceedings of the 16th European Conference on Computer Systems*. 83–98.

[31] Mugilan Mariappan and Keval Vora. 2019. GraphBolt: Dependency-driven synchronous processing of streaming graphs. In *Proceedings of the 14th European Conference on Computer Systems*. 25:1–25:16.

[32] Andrew McCrabb, Eric Winsor, and Valeria Bertacco. 2019. DREDGE: Dynamic repartitioning during dynamic graph execution. In *Proceedings of the 56th Annual Design Automation Conference*. 15–28.

[33] Shaohui Mei, Yunhao Geng, Junhui Hou, and Qian Du. 2022. Learning hyperspectral images from RGB images via a coarse-to-fine CNN. *Sci. Chin. Inf. Sci.* 65, 5 (2022), 1–14.

[34] 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.

[35] 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 51th Annual IEEE/ACM International Symposium on Microarchitecture*. 1–14.

[36] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. 2016. CACTI 6.0: A tool to model large caches. *HP Lab.* 27, 1 (2016), 1–28.

[37] Huy Trung Nguyen, Quoc Dung Ngo, and Van Hoang Le. 2018. IoT botnet detection approach Based on PSI graph and DGCNN classifier. In *Proceedings of the IEEE International Conference on Information Communication and Signal Processing*. 118–122.

[38] 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.

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

[40] Dipanjan Sengupta, Narayanan Sundaram, Xia Zhu, Theodore L. Willke, Jeffrey S. Young, Matthew Wolf, and Karsten Schwan. 2016. GraphIn: An online high performance incremental graph processing framework. In *Proceedings of the 22nd International Conference on Parallel and Distributed Computing*. 319–333.

[41] Matthias Steinbauer and Gabriele Anderst-Kotsis. 2016. DynamoGraph: Extending the Pregel paradigm for large-scale temporal graph processing. *Int. J. Grid Util. Comput.* 7, 2 (2016), 141–151.

[42] Jonathan M. Stokes, Kevin Yang, Kyle Swanson, Wengong Jin, Andres Cubillos-Ruiz, Nina M. Donghia, Craig R. MacNair, Shawn French, Lindsey A. Carfrae, Zohar Bloom-Ackermann, et al. 2020. A deep learning approach to antibiotic discovery. *Cell* 180, 4 (2020), 688–702.

[43] Nishil Talati, Di Jin, Haojie Ye, Ajay Brahmakshatriya, Ganesh Dasika, Saman Amarasinghe, Trevor Mudge, Danai Koutra, and Ronald Dreslinski. 2021. A deep dive into understanding the random walk-based temporal graph learning. In *Proceedings of the IEEE International Symposium on Workload Characterization*. 87–100.

[44] Nishil Talati, Haojie Ye, Sanketh Vedula, Kuan-Yu Chen, Yuhan Chen, Daniel Liu, Yichao Yuan, David Blaauw, Alex Bronstein, Trevor Mudge, et al. 2022. Mint: An accelerator for mining temporal motifs. In *Proceedings of the 55th IEEE/ACM International Symposium on Microarchitecture*. 1270–1287.

[45] Pourya Vaziri and Keval Vora. 2021. Controlling memory footprint of stateful streaming graph processing. In *Proceedings of theUSENIX Annual Technical Conference*. 269–283.

[46] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, Yoshua Bengio, et al. 2017. Graph attention networks. In *Proceedings of the International Conference on Learning Representations*. 1–12.

[47] Keval Vora, Rajiv Gupta, and Guoqing Xu. 2017. KickStarter: Fast and accurate computations on streaming graphs via trimmed approximations. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*. 237–251.

[48] 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. arXiv:1909.01315. Retreived from https://arxiv.org/abs/1909.01315

[49] 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 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. 149–159.

[50] Zhao Wang, Yijin Guan, Guangyu Sun, Dimin Niu, Yuhao Wang, Hongzhong Zheng, and Yinhe Han. 2020. GNN-PIM: A processing-in-memory architecture for graph neural networks. In *Proceedings of the 13th Advanced Computer Architecture*, Vol. 1256. 73–86.

[51] 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 IEEE International Symposium on High Performance Computer Architecture*. 15–29.

[52] Mingi Yoo, Jaeyong Song, Jounghoo Lee, Namhyung Kim, Youngsok Kim, and Jinho Lee. 2021. Making a better use of caches for GCN accelerators with feature slicing and automatic tile morphing. *IEEE Comput. Arch. Lett.* 20, 2 (2021), 102–105.

[53] Haoran You, Tong Geng, Yongan Zhang, Ang Li, and Yingyan Lin. 2022. GCoD: Graph convolutional network acceleration via dedicated algorithm and accelerator co-design. In *Proceedings of the 28th IEEE International Symposium on High-Performance Computer Architecture*. 15–27.

[54] Hanqing Zeng and Viktor K. Prasanna. 2020. GraphACT: Accelerating GCN training on CPU-FPGA heterogeneous platforms. In *Proceedings ot the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 255–265.

[55] Bingyi Zhang, Rajgopal Kannan, and Viktor K. Prasanna. 2021. BoostGCN: A framework for optimizing GCN inference on FPGA. In *Proceedings of the 29th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*. 29–39.

[56] 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.

[57] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. 2014. Maiter: An asynchronous graph processing framework for delta-based accumulative iterative computation. *IEEE Trans. Parallel Distrib. Syst.* 25, 8 (2014), 2091–2100.

[58] Yongan Zhang, Haoran You, Yonggan Fu, Tong Geng, Ang Li, and Yingyan Lin. 2021. G-CoS: GNN-accelerator co-search towards both better accuracy and efficiency. In *Proceedings to the IEEE/ACM International Conference On Computer Aided Design*. 1–9.

[59] 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.

[60] Hongkuan Zhou, Da Zheng, Israt Nisa, Vasileios Ioannidis, Xiang Song, and George Karypis. 2022. TGL: A general framework for temporal GNN training on billion-scale graphs. *Proc. VLDB Endow.* 15, 8 (2022), 1572–1580.

[61] 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.

[62] 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. *Proc. VLDB Endow.* 12, 12 (2019), 2094–2105.

Received 21 October 2022; revised 27 June 2023; accepted 4 August 2023