RTGA: A Redundancy-free Accelerator for High-Performance Temporal Graph Neural Network Inference

Hui Yu, Yu Zhang, Andong Tan, Chenze Lu, Jin Zhao, Xiaofei Liao, Hai Jin, 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
{huiy, zhyu, adtan, czl, zjin, xfliao, hjin, hkliu}@hust.edu.cn

Abstract

Temporal Graph Neural Network (TGNN) has attracted much research attention because it can capture the dynamic nature of complex networks. However, existing solutions suffer from *redundant computation overhead* and *excessive off-chip communications* for TGNN inference because they often rely on redundant graph sampling and repeatedly fetching the features and vertex memory. This paper proposes a redundancy-free accelerator, RTGA, for high-performance TGNN inference. Specifically, RTGA proposes a redundancy-aware execution approach with *temporal tree* into a novel accelerator design to effectively eliminate unnecessary data processing for fewer redundant computations and off-chip communications and also designs a temporal-aware data caching method to improve data locality for TGNN. We have implemented and evaluated RTGA on a Xilinx Alveo U280 FPGA card. Compared with cutting-edge software solutions (i.e., TGN and TGL) and hardware solutions (i.e., BlockGNN and FlowGNN), RTGA improves the performance of TGNN inference by an average of 473.2x, 87.4x, 8.2x, and 6.9x and saves energy by 542.8x, 102.2x, 9.4x, and 8.3x, respectively.

ACM Reference Format:
Hui Yu, Yu Zhang, Andong Tan, Chenze Lu, Jin Zhao, Xiaofei Liao, Hai Jin, Haikun Liu. 2024. RTGA: A Redundancy-free Accelerator for High-Performance Temporal Graph Neural Network Inference. In *61st ACM/IEEE Design Automation Conference (DAC '24), June 23–27, 2024, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3649329.3656241

Yu Zhang (zhyu@hust.edu.cn) is the corresponding author of this paper.
This paper is supported by National Key Research and Development Program of China (No. 2022YFB2404202), Key Research and Development Program of Hubei Province (No. 2023BAB078), Knowledge Innovation Program of Wuhan-Basi Research (No. 2022013301015177), and Huawei Technologies Co., Ltd (No. YBN2021035018A6).

DAC '24, June 23–27, 2024, San Francisco, CA, USA

ACM ISBN 979-8-4007-0601-1/24/06…$15.00
https://doi.org/10.1145/3649329.3656241

1 Introduction

Many real-world graphs are constantly evolving over time (e.g., edge/vertex deletion/addition and the mutation of vertex feature vector), which arise naturally in many real-world settings. To effectively model temporal dependencies and learn spatio-temporal representations, many *Temporal Graph Neural Networks* (TGNNs) have been proposed [6, 8, 11]. TGNNs show significant performance gains over static GNNs in a variety of applications, e.g., traffic forecasting [14] and epidemiological modeling [5].

To efficiently support TGNN models, some solutions, e.g., TGL [14] and FlowGNN [7], have been recently proposed to significantly reduce the data transfer time and keep the workload balanced. However, these solutions usually suffer from significant redundant computations, because they treat each target vertex independently without considering the potential overlap in their temporal neighbors when the graph updates. As a result, the same subgraph may be sampled and computed repeatedly, leading to massive unnecessary computations and memory accesses. Besides, the vertex and edge features, as well as the vertex memory, are too large to be stored in off-chip memory, requiring frequent irregular data transfers between the processing units and the memory. This not only increases the memory access latency but also consumes a significant amount of memory bandwidth.

Challenges come with opportunities. Through analyzing the data access patterns and computational characteristics of TGNN inference, we propose constructing a *temporal tree* (detailed in Section 2.3) for each target vertex when the graph updates in a batch, which efficiently encodes the temporal and structural dependencies of the TGNN model. Based on the temporal tree, we further develop a redundancy-aware execution approach to effectively eliminate unnecessary computations and memory accesses, leading to substantially improved processing efficiency. Furthermore, our in-depth analysis reveals that vertices with high connectivity, particularly those involved in interactions at earlier timestamps, exhibit a higher probability of being sampled during the inference process. This insight suggests that the

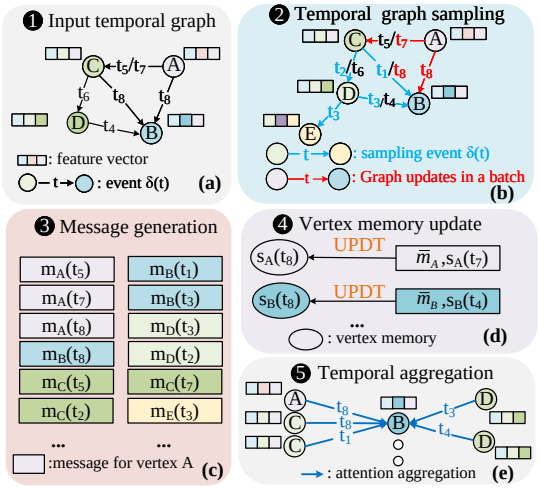


Figure 1. Illustration of the execution of TGNN inference

vertex features and memory of these frequently accessed vertices should be strategically cached in on-chip memory to minimize costly off-chip communications. Motivated by these findings, we introduce RTGA, a redundancy-free hardware accelerator specifically designed for efficient TGNN inference. RTGA leverages the temporal tree structure to guide a redundancy-aware execution approach, ensuring optimal computation and enhanced data locality, ultimately delivering unparalleled performance in TGNN inference.

We have implemented and evaluated RTGA on a Xilinx Alveo U280 FPGA card. The experimental results show that RTGA outperforms the cutting-edge software TGNN solutions (i.e., TGN [6] and TGL [14]) running on Intel Xeon CPU and NVIDIA A100 GPU by, on average, 473.2x and 87.4x with 542.8x and 102.2x energy savings, respectively. Compared with the cutting-edge static GNN accelerators, i.e., BlockGNN [16] and FlowGNN [7], RTGA achieves a speedup of 7.4x-11.9x and 5.2x-9.3x with 8.4x-17.3x and 7.8x-14.2x energy savings, respectively.

2 Background and Motivation

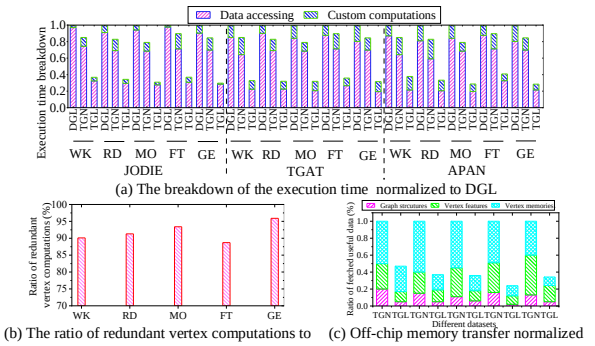
2.1 Background of TGNN

Temporal Graph Neural Network (TGNN). Many TGNN models adhere to the principle of temporal message passing [6] to effectively harness the temporal dynamics present within the graph structure. TGNN inference process is typically segmented into five distinct stages. As shown in Figure 1, ❶ **Input temporal graph**, where the graph data along with its temporal aspects are fed into the model. ❷ **Temporal graph sampling**, which involves accessing the temporal graph and sampling the most recent neighbors (i.e., the candidate edges from all past neighbors of the target vertex). ❸ **Message generation**, for each vertex $v \in V$, TGNN maintains a vertex memory vector s_v , which is initialized to be a zero vector. When an edge e_{uv} connects vertex u and vertex v appear at timestamp t , two *messages* are generated at vertex u and vertex v:

$$\mathbf{m}_u = [\mathbf{s}_u \,||\, \mathbf{s}_v \,||\, \Phi(t - t_u^-) \,||\, \mathbf{e}_{uv}] \tag{1}$$

Table 1. Dataset information. V and E denote the number of vertices and edges, respectively. $|d_v|$ and $|d_e|$ show the dimensions of vertex and edge states, respectively. The max(t) column represents the maximum edge timestamp. Note that the minimum edge timestamp is 0 in all datasets.

Datasets	#V	#E	$\lvert d_v\rvert$	$\lvert d_e\rvert$	max(t)
Wikipedia (WK) [14]	9,227	157,474	1,433	172	2.7e6
Reddit (RD) [14]	10,984	672,447	3,703	172	2.7e6
MOOC (MO) [5]	7,144	411,749	500	285	2.6e7
Flights (FT) [15]	13,169	1,927,145	61,278	100	1.0e7
GDELT (GE) [15]	16,682	191,290,882	602	413	1.6e8



(a) The breakdown of the execution time normalized to DGL

(b) The ratio of redundant vertex computations to all vertex computations for APAN model on TGL

(c) Off-chip memory transfer normalized TGN for APAN model on different datasets

Figure 2. Performance of existing solutions

$$\mathbf{m}_v = [\mathbf{s}_v \,||\, \mathbf{s}_u \,||\, \Phi(t - t_v^-) \,||\, \mathbf{e}_{uv}] \tag{2}$$

where $\Phi(\cdot)$ is time coding [5], t_u^- is the timestamp when s_u is last updated, and e_{uv} is the edge feature. ❹ **Vertex memory update**, most of TGNN models use an update function UPDT to update the vertex memory of vertex u and vertex v,

$$s_u = UPDT(s_u, \bar{m}_u), s_v = UPDT(s_v, \bar{m}_v)$$

The update function can be implemented using any sequence model. In attention-based TGNN models [5], UPDT$(\cdot)$ is implemented as GRU cells. The $\bar{m}_u$ is the sample neighbors' aggregated message of vertex u. Since the UPDT function is only called when a related graph event occurs, the lengths of the hidden status of different vertices in the graph are different. ❺ **Temporal aggregation**, after updating the vertex memory, TGNN models use a one-layer temporal attention layer [11] or other aggregation function [6] to gather and aggregate information from the vertex memory of the most recent neighbors $S_w, w \in N_v$ to compute the dynamic vertex embedding h_v for vertex v.

2.2 Limitation of State-of-the-art Solutions

Many software solutions [6, 14] have been proposed to handle the TGNN model on general-purpose processors. However, TGNN inference exhibits *massive redundant computations* and *excessive off-chip communications* because they process the temporal neighbors of each target vertex independently, without considering the temporal dependencies and potential overlap among the neighbors. As a result, the same messages may be computed multiple times for the same set of neighbors.

Figure 1 illustrates the operation of TGNN during a batch of graph updates (i.e., timestamps t_7 and t_8 in Figure 1 (b)). For the target vertex B, TGNN models sample its most recent

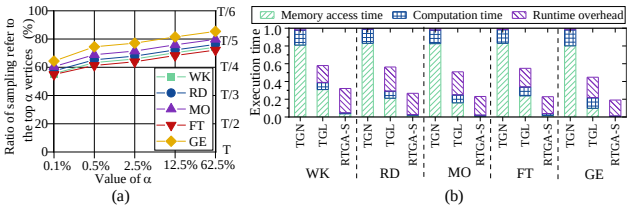


Figure 3. Performance of our approach: (a) the ratio of sampled neighbors from the top α vertices to those from all vertices; (b) the execution time of our software-only approach (i.e., RTGA-S) normalized to that of TGL over one GPU

neighbors, namely the vertex C at timestamp t_1 and the vertices D at timestamps t_3 and t_4. It becomes evident that the edge $C \longrightarrow B$ at timestamp $t-1$ and $D \longrightarrow B$ at timestamps t_3 and t_4 are highly likely to be sampled twice because the vertex B engaging in two new interactions in timestamp t_8.

To prove it, we evaluate three cutting-edge software TGNN systems (i.e., TGL [14], TGN [6], and DGL [9]) running on NVIDIA A100 GPU. The datasets are listed in Table 1 and the details of the platform and benchmarks are presented in Section 4. Although TGL outperforms other systems in all cases as shown in Figure 2 (a), for the APAN model, only 8.9%-11.2% of the total computation of TGN is useful as shown in Figure 2 (b). It indicates that many off-chip communications are unnecessary, and take more than 88.9% of overall execution time, as shown in Figure 2 (a). The accesses to vertex features and vertex memory occupy 81.2%-91.4% of all data accesses, as shown in Figure 2 (c).

Compared with the solutions based on general-purpose processors, specialized hardware accelerators [7, 16] gain significant energy savings and performance speedup for GNN. However, they are also inefficient for TGNN because they also suffer from significant redundant computations and off-chip communications due to the above described unnecessary updates and computations. In other words, the conventional architectures are not the best platforms for running TGNN models, which requires designing a new hardware accelerator to address the challenges of redundant computations for TGNN.

2.3 Motivation

2.3.1 Our Redundancy-aware Execution Approach with Temporal Tree.

To improve TGNN inference's efficiency, we propose a redundancy-aware execution approach with temporal tree. We first give the definition and then present our approach.

Temporal Tree. Given a temporal graph $G = \{\delta(t_1), \delta(t_2), \ldots, \delta(t)\}$, where $\delta(t) = \{v_i, v_j, e_{ij}, t\}$. For a given destination vertex v and a set of time-order graph updates in a batch $U = \{(e_1, t_1), (e_2, t_2), \ldots, (e_n, t_n)\}$, where $e_i \in E$ and $t_i \in T$, a temporal tree TT_v is defined as follows:

Root Vertex. The root of TT_v is the destination vertex v.

Child Vertices and Edge Labels. Each child vertex u of v in TT_v corresponds to a vertex v that has interacted with v through an edge e_{vu} at timestamp $t \in T$. These child vertices are denoted as u. Additionally, each edge in TT_v is labeled

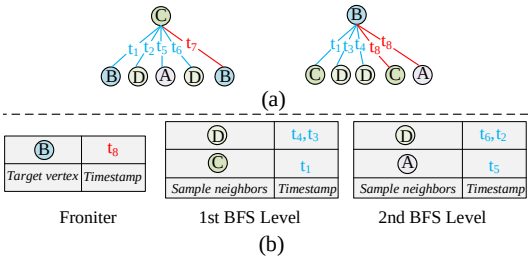


Figure 4. An example to illustrate our approach: (a) constructing the temporal tree for vertex C at timestamp t_7 and vertex B at timestamp t_8, respectively; (b) redundancy-free sampling based on TT_C and TT_B

with the timestamp t of the interaction, representing the temporal aspect of the graph.

Temporal Tree Structure. The tree is constructed such that for each time-ordered update (e_i, t_i) relevant to v, a corresponding child vertex u_{t_i} is linked to v in TT_v. Mathematically, TT_v can be represented as a set of triplets $\{(u, v, t_1), (u, v, t_2), \ldots, (u, v, t_n)\}$, where each triplet corresponds to an edge in the tree with its respective timestamp.

Redundancy-free Processing. Leveraging the temporal tree TT_v in TGNN inference involves a streamlined process where the tree structure guides both the sampling and updating phases. The key idea is to process the target vertices in descending order of their maximum timestamp, ensuring that each relevant interaction edge is sampled only once. Specifically, for each target vertex, through traversing its temporal neighbors and performing sampling based on the temporal tree. The sampled neighbors are then marked as visited to avoid redundant processing. If the next target vertex appears in the list of already sampled neighbor vertices, it is treated as a new root vertex to traverse its temporal neighbors and sample only the unvisited temporal neighbors. This process is repeated until all target vertices have been processed.

Figure 4 illustrates our approach with a toy example. In Figure 4 (a), it constructs the temporal trees TT_C and TT_B for target vertices C and B at timestamps t_7 and t_6, respectively. Based on TT_B, it can efficiently identify the neighbors of vertex B at timestamp t_8, i.e., vertices C and D connected by edges (D, B, t_4), (D, B, t_3), and (C, B, t_1). Similarly, based on TT_C, the neighbors of vertex C at timestamp t_7 are vertices D and A connected by edges (D, C, t_6), (D, C, t_2), (A, C, t_5), and (B, C, t_1). Next, as shown in Figure 4 (b), it starts with the target vertex B at timestamp t_7 as the root, traverses TT_C on the fly to sample the temporal neighbors of B, and marks the sampled edges (C, B, t_1), (D, B, t_3), and (D, B, t_4) as visited at the first BFS level. At the second BFS level, it takes vertex C as the root to traverse TT_C for sampling its temporal neighbors. However, since edge (B, C, t_1) has already been sampled at timestamp t_8, it only needs to sample the unvisited temporal edges (D, C, t_6), (D, C, t_2), and (A, C, t_5).

Besides, because of the power-law property [3], a noteworthy observation pertains to the sampling probability of

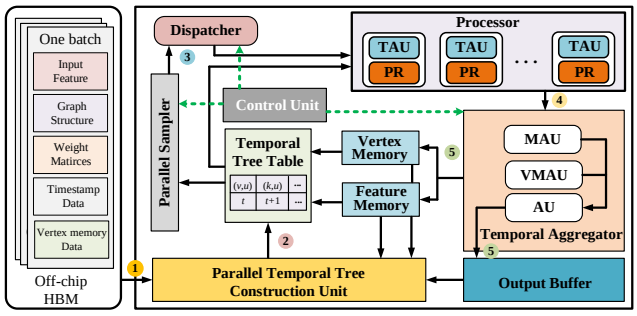


Figure 5. Architecture of RTGA

vertices which are influenced by their timestamp and degree. It has been discerned that vertices with smaller timestamps (t) and higher degrees (d) exhibit a higher likelihood of being sampled. As shown in Figure 3 (a), when the timestamp t is less than $5/T$ (T is the maximum timestamp), and the degree d of a vertex falls within the top 0.5% percentile, the probability of that vertex being sampled escalates to 62.4%. Thus, our approach also caches the features and vertex memory of the vertices on these frequently-used in the on-chip memory for fewer off-chip communications.

2.3.2 Challenges. Although our approach ensures much fewer redundant computations and data accesses, the software-only implementation remains challenging. As shown in Figure 3 (b), it suffers from high runtime overhead (which occupies 85.8%-94.2% of total execution time for our tested cases). This substantial overhead primarily stems from the generation and maintenance of temporal trees. In fact, when the dynamic graph is updated, our approach needs to dynamically and irregularly traverse all the sampled temporal subgraphs of the affected vertices to construct the temporal trees. It not only incurs many irregular data access operations and generates many extra instructions, but also these instructions have data-dependent branches that limit the parallelism. It motivates us to design this approach into our hardware accelerator for efficient TGNN inference.

3 RTGA Architecture

3.1 RTGA Overview

Figure 5 shows the architecture of RTGA, which is designed to efficiently support the proposed redundancy-aware execution approach with temporal tree to reduce unnecessary computations and off-chip communications for TGNN inference. Specifically, RTGA contains several key hardware units, i.e., *Parallel Temporal Tree Construction Unit* (TTCU), *Parallel Sampler*, *Dispatcher*, *Processor with Temporal Arithmetic Units (TAUs)*, *Temporal Aggregator*, control unit, and some on-chip buffers.

TTCU. TTCU is dedicated to constructing Temporal Trees for each destination vertex in real-time. Specifically, TTCU contains a *Graph update Batch Scheduler* (GUBS), which is responsible for receiving the incoming graph update batches and efficiently distributing them among the available *Construction Units* (CUs). The CUs are capable of independently constructing a temporal tree for a specific target vertex. The number of CUs can be flexibly configured based on the available hardware resources and the desired level of parallelism.

Parallel Sampler. This unit can efficiently perform parallel temporal neighbor sampling for multiple target vertices, which leverages the temporal tree constructed by TTCU to identify the most relevant neighbors for each target vertex.

Dispatcher and Processor. The *Dispatcher*'s primary responsibility is to efficiently distribute the sampled edge data among the TAUs to ensure optimal load balancing and maximum resource utilization. The *Processor* contains multiple *Temporal Arithmetic Units* (TAUs) to generate the messages between the sample edges of target vertices.

Temporal Aggregator. It contains three computation units, i.e., *Message Aggregation Unit* (MAU), *Vertex Memory Aggregation Unit* (VMAU), and *Activation Unit* (AU). MAU aggregates messages from different parts of the graph. VAMU is responsible for updating the vertex memory by assimilating the most recent and relevant interaction data. AU applies necessary nonlinear transformations or activation functions to the processed data.

On-chip Buffers and Control Unit. The on-chip memory is composed of several buffers, e.g., *Vertex Memory* Buffer, *Feature Memory* Buffer, and *Output Buffer*, which are employed to cache various data (e.g., *vertex features, vertex memory, and partial output results*) to improve data reuse and reduce unnecessary off-chip communications. Note that RTGA adopts the ping-pong buffering technology [7] to decouple the different operations for all buffers to hide the access latency.

Workflow of RTGA. RTGA loads the temporal graph data from off-chip HBM to on-chip memory ❶. Then TTCU quickly constructs the temporal tree for each destination vertex by traversing the temporal graph structure on the fly, and the temporal tree can be stored in the *temporal tree table*, which uses a triplet to record the temporal tree information ❷. Based on the data in *temporal tree table*, *Parallel Sampler* selects the most relevant subgraphs for processing, guided by the *temporal trees table*. Then *Dispatcher* is responsible for assigning sampled edges to the *Processor* and generating corresponding vertex messages ❸. The *Processor*, equipped with multiple TAUs, then generates a message for each sampled edge and records the edge's features between the source vertex and target vertex in *Private Register* (PR) to improve the data locality. As messages are generated, they are passed on to the *Temporal Aggregator* ❹. MAU is responsible for aggregating messages from various vertices to obtain the aggregation results of temporal neighbors. VMAU updates the memory states of the vertices with the latest information, while AU applies activation functions to the aggregated data. Finally, the processed data is moved to the output buffer ❺, where it is stored and made ready for retrieval.

3.2 Redundancy-aware Parallel Sampling

To reduce the latency of temporal tree construction and graph sampling when graph updates, four stages, i.e., *graph*

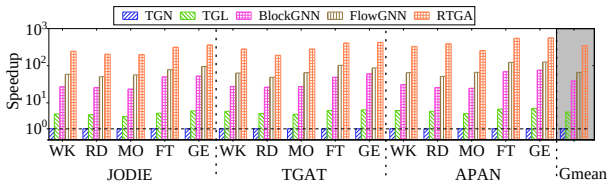


Figure 6. Performance normalized to that of TGN

updates, *incremental tree construction*, *temporal tree update*, and *redundancy-aware sampling*, are performed, which are implemented as a pipeline between TTCU and parallel sampler. In the *graph updates* stage, TTCU utilizes high-bandwidth memory interfaces and efficient data parsing modules to quickly ingest the incoming graph updates, and record them in *UTable*. In the *incremental tree construction* stage, TTCU fetches them from the *UTable* and employs multiple CUs to incrementally construct and update the temporal trees in parallel. Each CU operates independently, focusing on different segments of the graph, to ensure that updates are processed efficiently and without unnecessary delays. Once the temporal trees are constructed, the *temporal tree update* stage utilizes high-speed interconnects and efficient synchronization primitives to transfer the updated tree structures and relevant data to the *Parallel Sampler*. In the *redundancy-aware sampling* stage, it leverages multiple sampling units to perform efficient and redundancy-free neighborhood sampling. Each sampling unit is equipped with a tree traversal engine, which is implemented using the *Finite State Machines* (FSMs) and address generation units (which are responsible for generating the memory addresses for reading and writing tree vertices and edges) to identify the relevant neighbors for each target vertex. Note that it also employs comparators and data filtering modules to eliminate redundant neighbors.

3.3 Temporal-aware Data Caching

In the sophisticated design of RTGA, special emphasis is placed on the strategic caching of vertices, particularly those with smaller timestamps and higher degrees. *Temporal-aware Data Caching* (TADC) strategy is implemented in both *Vertex Memory* and *Feature Memory* buffers through an intricate hardware implementation that efficiently identifies and prioritizes these key vertices for enhanced caching. Specifically, when the features and vertex memory of a vertex v are requested, TADC first checks if the *Vertex Memory* and *Feature Memory* buffers have available space. If the buffer is not full, v's feature and vertex memory are directly cached in the *Vertex Memory* and *Feature Memory*, respectively. However, if the buffer is already full, TADC calculates v's priority based on its temporal and structural significance in the graph. The priority of vertex v is denoted as $Pri(v)$, i.e., $Pri(v) = D(v)/T(v)$, where $T(v)$ represents the timestamp index of vertex v from the off-chip *Timestamp data*, and $D(v)$ is the degree of vertex v. Note that the vertex degree information can be easily obtained via the vertex array. Also, the identification operation is as simple as a compactor, which can be easily implemented with a few ALUs.

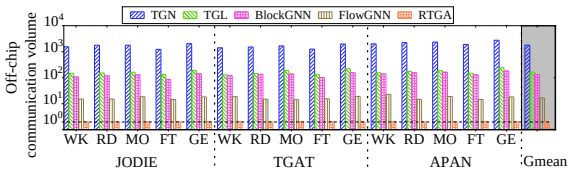


Figure 7. Off-chip communications normalized to that of RTGA

Once the priority score is computed, TADC compares v's priority with the priorities of the vertices currently residing in the *Vertex Memory* and *Feature Memory*. If v has a higher priority than the lowest-priority vertex in the buffer, TADC replaces the lower-priority vertex with v. This ensures that the most temporally and structurally significant vertices are retained in the cache.

4 EVALUATION

4.1 Experiment Setup

RTGA Settings. We have implemented RTGA 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, and two 4 GB HBM2 stacks. RTGA contains 8 TAUs and the on-chip memory of RTGA is implemented using the BRAM resources. Similar to BlockGNN [16], for the PE Group of RTGA, we adopt programmable ALUs, which implement different operations depending on the properties of the TGNN model. *Activate PEs* support activation operations, while the PEs in the TAUs are constructed as a systolic array. Each PE is a MAC unit with some glue logic. We get the clock rate of RTGA using Xilinx Vivado 2019.1 and conservatively use 330 MHz in our experiments.

Datasets and Benchmarks. We use five datasets (listed in Table 1) and three popular TGNN models, i.e., JODIE [5], TGAT [11], and APAN [10], which are commonly used in TGNN acceleration studies [14].

Baseline. RTGA is compared with four solutions, i.e., TGN [6], TGL [14], BlockGNN [16], and FlowGNN [7]. TGN runs on the CPU platform (which has an Intel Xeon 6151 processor with 65 cores at 3.0 GHz and 696 GB DRAM). Intel Product Specifications are used to estimate the CPU energy consumption and TGL runs on the NVIDIA Tesla A100 with 6,912 cores and 80 GB HBM. The GPU power is measured by *Nvidia-smi*. RTGA is also compared with the cutting-edge static GNN accelerators, i.e., BlockGNN and FlowGNN, which run at 330 MHz on the Xilinx Alveo U280 FPGA.

4.2 Experimental Results

Overall Performance. Figure 6 shows the execution time of different solutions normalized to that of TGN. Compared with TGL, BlockGNN, and FlowGNN, RTGA improves the performance by 413.2-627.9x (473.2x on average), 62.5x-103.9x (87.4x on average), 7.4x-11.9x (8.2x on average), and 5.2x-9.3x (6.9x on average), respectively. The better performance achieved by RTGA comes from fewer redundant computations and off-chip communications. Note that our CADC strategy contributes 6.1%-9.4% of RTGA's performance improvement. Figure 7 shows that the number of communications of TGN, TGL, BlockGNN, and FlowGNN are 29.5x,

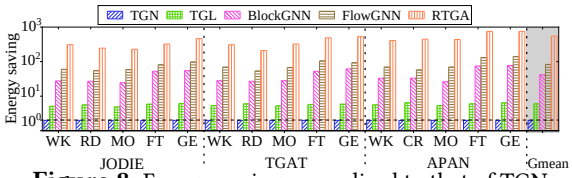


Figure 8. Energy saving normalized to that of TGN

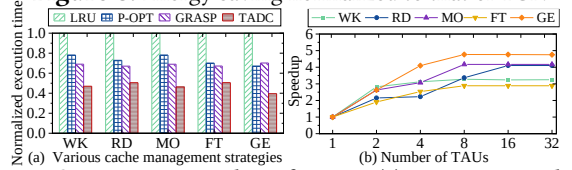


Figure 9. Sensitivity studies of RTGA: (a) sensitivity to the replacement strategies; (b) sensitivity to the number of TAU

17.4x, 9.2x, and 7.2x more than that of RTGA on average, respectively. There are two reasons. First, in RTGA, most redundant calculations are eliminated by our approach. Second, RTGA further eliminates irregular off-chip communication through the parallel sampling acceleration module. Figure 8 shows that the energy savings of RTGA are 414.2x-862.5x (542.8x on average) and 92.5x-114.9x (102.2x on average) are higher than TGN and TGL, respectively. Compared with BlockGNN and FlowGNN, the energy savings of RTGA are improved by an average of 9.4x and 8.3x.

Sensitivity Studies. Figure 9 (a) shows the performance of RTGA with different replacement strategies (i.e., LRU [4], P-OPT [1], and GRASP [2]) for the management of the *Feature memory* on the APAN model. It shows that our TADC scheme outperforms the others because the features of the vertices that are frequently accessed can be effectively cached in the on-chip memory. Figure 9 (b) describes the sensitivity to the number of TAUs. It shows that RTGA obtains better performance as the number of TAUs increases until reaches eight because the memory bandwidth is saturated.

5 Related Work

Software TGNN Systems and GNN Accelerators. TGL [14] exploits a stochastic chunk scheduling methodology to enhance training efficiency. To further improve the training efficiency, TGN [6] proposes a general framework to implement the ubiquitous support for many TGNN models. Besides, to accelerate the process of GNN inference, FlowGNN [7] uses an approach based on multi-queue streaming, while BlockGNN [16] proposes a pipelined architecture to support block-circulant matrices computation. However, these solutions suffer from massive redundant computations.

Temporal Graph Processing Solutions. EGraph [12] proposes the Loading-Processing-Switching execution model. SaGraph [13] introduces a domain-specific hardware accelerator to exploit the data access similarity of temporal graph processing. However, these solutions can not support the neural network computation of TGNN inference [14].

6 Conclusion

This paper proposes a redundancy-free hardware accelerator, *RTGA*, for efficient TGNN inference. RTGA can construct the temporal tree to reveal the temporal dependency flow and efficiently cache the frequently-sampled vertex features and memory associated with high-degree vertices and low timestamps. Compared with the cutting-edge software solutions, i.e., TGN and TGL, RTGA gains an average of 473.2x, 87.4x speedup, and 542.8x, 102.2x energy savings, respectively.

References

[1] Vignesh Balaji, Neal Crago, Aamer Jaleel, and Brandon Lucia. 2021. P-OPT: Practical Optimal Cache Replacement for Graph Analytics. In *Proceedings of HPCA*. 668–681.

[2] Priyank Faldu, Jeff Diamond, and Boris Grot. 2020. Domain-Specialized Cache Management for Graph Analytics. In *Proceedings of HPCA*. 234–248.

[3] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *Proceedings of OSDI*. 17–30.

[4] Daniel A. Jiménez. 2013. Insertion and promotion for tree-based PseudoLRU last-level caches. In *Proceedings of MICRO*. 284–296.

[5] Srijan Kumar, Xikun Zhang, and Jure Leskovec. 2019. Predicting Dynamic Embedding Trajectory in Temporal Interaction Networks. In *Proceedings of KDD*. 1269–1278.

[6] Emanuele Rossi, Ben Chamberlain, Fabrizio Frasca, Davide Eynard, Federico Monti, and Michael M. Bronstein. 2020. Temporal Graph Networks for Deep Learning on Dynamic Graphs. In *Proceedings of ICLR*. 1–15.

[7] Rishov Sarkar, Stefan Abi-Karam, Yuqi He, Lakshmi Sathidevi, and Cong Hao. 2022. FlowGNN: A Dataflow Architecture for Universal Graph Neural Network Inference via Multi-Queue Streaming. In *Proceedings of HPCA*. 1099–1112.

[8] Rakshit Trivedi, Mehrdad Farajtabar, Prasenjeet Biswal, and Hongyuan Zha. 2019. DyRep: Learning Representations over Dynamic Graphs. In *Proceedings of ICLR*. 1–15.

[9] 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. In *Proceedings of ICLR*. 1–18.

[10] Xuhong Wang, Ding Lyu, Mengjian Li, Yang Xia, Qi Yang, Xinwen Wang, Xinguang Wang, Ping Cui, Yupu Yang, Bowen Sun, and Zhenyu Guo. 2021. APAN: Asynchronous Propagation Attention Network for Real-time Temporal Graph Embedding. In *Proceedings of ICMD*. 2628–2638.

[11] Da Xu, Chuanwei Ruan, Evren Körpeoglu, Sushant Kumar, and Kannan Achan. 2020. Inductive representation learning on temporal graphs. In *Proceedings of ICLR*. 1–12.

[12] Yu Zhang, Yuxuan Liang, Jin Zhao, Fubing Mao, Lin Gu, Xiaofei Liao, Hai Jin, Haikun Liu, Song Guo, Yangqing Zeng, Hang Hu, Chen Li, Ji Zhang, and Biao Wang. 2023. EGraph: Efficient Concurrent GPU-Based Dynamic Graph Processing. *IEEE Transactions on Knowledge and Data Engineering* 35, 6 (2023), 5823–5836.

[13] Jin Zhao, Yu Zhang, Jian Cheng, Yiyang Wu, Chuyue Ye, Hui Yu, Zhiying Huang, Hai Jin, Xiaofei Liao, Lin Gu, and Haikun Liu. 2023. SaGraph: A Similarity-aware Hardware Accelerator for Temporal Graph Processing. In *Proceedings of DAC*. 1–6.

[14] Hongkuan Zhou, Da Zheng, Israt Nisa, Vassilis N. Ioannidis, Xiang Song, and George Karypis. 2022. TGL: A General Framework for Temporal GNN Training on Billion-Scale Graphs. *Proceedings of the VLDB Endowment* 15, 8 (2022), 1572–1580.

[15] Hongkuan Zhou, Da Zheng, Xiang Song, George Karypis, and Viktor K. Prasanna. 2023. DistTGL: Distributed Memory-Based Temporal Graph Neural Network Training. In *Proceedings of SC*. 39:1–39:12.

[16] 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 DAC*. 1009–1014.