Model-Architecture Co-Design for High Performance Temporal GNN Inference on FPGA

Hongkuan Zhou[*][§], Bingyi Zhang[*][§], Rajgopal Kannan[†], Viktor Prasanna[*], Carl Busart[†]

[*]University of Southern California [†]US Army Research Lab

[*]{hongkuaz, bingyizh, prasanna}@usc.edu [†]{rajgopal.kannan.civ, carl.e.busart.civ}@army.mil

Abstract—**Temporal Graph Neural Networks (TGNNs) are powerful models to capture temporal, structural, and contextual information on temporal graphs. The generated temporal node embeddings outperform other methods in many downstream tasks. Real-world applications require high performance inference on real-time streaming dynamic graphs. However, these models usually rely on complex attention mechanisms to capture relationships between temporal neighbors. In addition, maintaining vertex memory suffers from intrinsic temporal data dependency that hinders task-level parallelism, making it inefficient on general-purpose processors. In this work, we present a novel model-architecture co-design for inference in memory-based TGNNs on FPGAs. The key modeling optimizations we propose include a light-weight method to compute attention scores and a related temporal neighbor pruning strategy to further reduce computation and memory accesses. These are holistically coupled with key hardware optimizations that leverage FPGA hardware. We replace the temporal sampler with an on-chip FIFO based hardware sampler and the time encoder with a look-up-table. We train our simplified models using knowledge distillation to ensure similar accuracy vis-á-vis the original model. Taking advantage of the model optimizations, we propose a principled hardware architecture using batching, pipelining, and prefetching techniques to further improve the performance. We also propose a hardware mechanism to ensure the chronological vertex updating without sacrificing the computation parallelism. We evaluate the performance of the proposed hardware accelerator on three real-world datasets. The proposed model reduces the computation complexity by 84% and memory accesses by 67% with less than 0.33% accuracy loss. Compared with CPU/GPU, our FPGA accelerator achieves 16.4/2.3× speedup in latency and 0.27% improvement in accuracy compared with the state-of-the-art inference algorithm. To the best of our knowledge, this is the first work that performs model-architecture co-design on memory-based Temporal Graph Neural Networks.**

Index Terms—**Temporal Graph Neural Network, Hardware Architecture, FPGA**

I. INTRODUCTION

The acceleration of static GNNs operating on *structural* and *contextual* information to produce node/link *embeddings* on static graphs is a burgeoning area of research. Proposed techniques range from algorithmic model optimizations (pruning and compression [1]–[3]) to dedicated hardware accelerators [4]–[6]. These efforts achieve massive task parallelism, allowing static GNNs to be deployed accurately and efficiently on large-scale graphs for a variety of applications (recommender systems [7], [8], knowledge graph reasoning [9], [10], fraud detection [11] etc.).

However, most real world graphs also contain *temporal* information that is sensitive both to duration (longevity) and chronology. For example, interactions in social networks are often timestamped, components of knowledge graphs have duration-limited validity and chronological ordering of graph signals is critical for real-world tasks like fraud detection and event prediction. Thus Temporal GNNs (TGNNs) [12]–[16] that generate node embeddings also capturing (evolving) temporal information have become popular.

In production environments, TGNNs are usually used to compute dynamic node embeddings on the upcoming stream of graph signals for downstream tasks. However, several unique characteristics of TGNNs make them inefficient for deployment on General Purpose Processors (GPPs). First, TGNNs are significantly more compute-intensive. In order to accurately capture the evolving nature of temporal neighborhoods, most TGNNs [12], [13], [17], [18] rely on a temporal attention mechanism (adopted from Transformer [19]) to aggregate features from temporal neighbors along with additional sequence models like RNNs and GRUs. An artifact of this mechanism is that it requires computing additional "keys" and "queries" for each temporal neighbor (more than 2× the number of operations than a mean or max pooling aggregator). Second, graph signals can appear asynchronously, at varying rates. Temporal neighbor sampling and vertex information updates associated with these signals lead to intrinsic sequential dependencies which require the system to process small batches. This raises the issue of implementation platform. Current TGNN implementations are mostly GPU focused, where the coarse-grained parallelism usually leads to significantly worse performance on small versus large batches. Further, both latency *and* throughput of processing these signals is important. State-of-the-art works like [17] hide latency by offlining part of the message passing process. This requires exponential amount of extra memory to cache intermediate results making it hard to scale to large dynamic graphs while also not decreasing computational complexity.

We believe that while algorithmic optimizations per se are useful in partially solving the above challenges, a more holistic approach that also leverages the fine-grained parallelism, low-latency on-chip memory (for customized memory access patterns), and high density resources (programmable DSPs for customized data paths) of hardware platforms such as FPGAs can provide a superior overall solution.

In this paper, we present a model-architecture co-design

for high-performance TGNN inference on FPGA. To the best of our knowledge, this is the first such co-design framework that jointly optimizes both throughput and latency, consisting of a suite of algorithmic model optimizations to solve computational and memory bottlenecks imposed by model constraints and carefully mapped to maximize FPGA architectural support for performance acceleration. We begin with an analysis of the computation processes of general memory-based TGNNs along with a case study evaluating computation-communication characteristics. Based on the identified bottlenecks, we perform both algorithmic and hardware-specific optimizations to make TGNN inference computationally tractable with negligible accuracy loss. We design and implement our hardware accelerator on two different FPGAs and demonstrate high throughput and low latency TGNN inference with negligible accuracy loss on real-world datasets. We summarize our main contributions below:

- We propose a light-weight temporal attention mechanism and a related neighbor pruning strategy which greatly reduce the computation and memory accesses at inference. We design a knowledge distillation setup to train our simplified models to ensure comparable accuracy.
- To better leverage the FPGA hardware, we propose FPGA-specific optimizations that replace the temporal sampler with a FIFO-based hardware sampler and replace the time encoder with a Look-up-Table (LUT) based time encoder. We use a hardware-based mechanism to rapidly maintain the vertex information up-to-date. We design and implement a hardware accelerator using techniques including batching, pipelining, and prefetching to achieve massive computation parallelism.
- We propose a predictive performance model for our hardware accelerator to estimate the performance based on algorithm parameters, design configurations, and memory characteristics.
- We implement the proposed design on state-of-the-art FPGA platforms Xilinx Alveo U200 and Xilinx ZCU104. Compared with the baseline, our hardware accelerators on ZCU104/U200 achieve $2.00/5.04\times$ improvement in latency and $2.46/8.81\times$ improvement in throughput compared with the CPU/GPU baselines.

II. TEMPORAL GRAPH NEURAL NETWORKS

Given a dynamic graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ where nodes and edges are associated with timestamps representing their appearance/disappearance, the TGNN inference problem aims at encoding contextual, structural, and temporal information of nodes at specific times into dynamic node embeddings and maintaining up-to-date node memory. Note that TGNN inference is distinct from the inference (link prediction etc.) carried out by downstream tasks. In this work, we choose to use the memory-based TGNNs which achieve state-of-the-art accuracy as the baseline models to optimize. We briefly describe the memory-based TGNN inference process below.

For each node v, memory-based TGNNs maintain a node memory vector s_v that summarizes its status. A message

MSG(m) is generated whenever a graph signal m related to v occurs. Multiple related messages from v's receptive field (of neighbors) $\mathcal{N}(v)$, are combined by an aggregator function AGGR. These are then used to update s_v via an update function UPDT. s_v is then fed as an input feature into an attention-based GNN to generate the output node embedding at the current timestamp h_v.

$$s_v = \text{UPDT}(s_v, \text{AGGR}(\text{MSG}(\{m, m \in \mathcal{N}(v)\}))) \quad (1)$$

$$h_v = \text{GNN}(\mathcal{G}_v, \{s_u, u \in \mathcal{G}_v\}), \quad (2)$$

where $\mathcal{G}_v$ denotes v's supporting temporal neighbors (sampled from the past neighbors of v).

The process in Equation (1)-(2) is computed both at training and inference. Since dynamic node labels are difficult to acquire and hardly provide enough information on evolving graphs, TGNNs are usually trained by self-supervision from the temporal edges. Under this setup, the temporal edges which need to be predicted by an external downstream edge classifier are itself fed to the TGNN models as input during training, creating an "information leak" problem [13]. To solve this, the TGNN *caches* the messages of the current graph signals rather than directly use them as input. When a new graph signal arrives, the UPDT function takes input from the cached messages instead of using the message of the current signal to update the node memory. Therefore, at inference, to be consistent with the training phase, we need to first update the node memory by the cached messages (Equation (1)) before aggregating from them (Equation (2)).

A. Inference Performance Metrics

When deployed for real-world applications, TGNN-based systems usually operate on upcoming graph signals in batches, formed either by fixed number of graph signals or by the graph signals in fixed time windows. To quantify the performance of TGNN inference, we formally define our evaluation metrics of throughput and latency. Since most existing datasets only have new edges as graph signals, we define the throughput as the number of new edges that can be processed per second. Defining the execution time as the total time to generate the dynamic node embeddings and maintain the node memory up-to-date, we define throughput as

$$\text{TGNN throughput (E/s)} = \frac{\text{\# of new edges}}{\text{execution time (s)}}, \quad (3)$$

We define TGNN inference latency as the time to output the required dynamic node embeddings once we receive a batch of graph signals. We follow the general setup in [13], [16], [17] for processing incoming graph signals, where temporal dependencies are ignored for nodes in the same batch while the node memory and the cached messages are updated in order. For the graph signals in one batch, we perform one forward propagation and generate the corresponding dynamic node embeddings for *all* the involved nodes, which is the common inference scenario for many applications. For example, a fraud detection application would like to frequently examine *all*

TABLE I

NUMBER (#) AND PERCENTAGES (%) OF THOUSANDS OF MEMs (kMEM) AND THOUSANDS OF MACs (kMAC) AND THE AVERAGE EXECUTION TIME ON CPU AND GPU PER DYNAMIC NODE EMBEDDING.

	Wikipedia							Reddit						
	kMEM		kMAC		Exec. Time (ns)			kMEM		kMAC		Exec. Time (ns)		
	#	%	#	%	1 Thread	32 Threads	GPU	#	%	#	%	1 Thread	32 Threads	GPU
sample	0.0	0.3%	0	0%	9	9	8	0.1	1.1%	0	0%	11	9	8
memory	5.2	91.4%	48.4	6.0%	273	40	8	5.2	90.7%	48.4	6.0%	198	47	9
GNN	0	0%	703.5	93.6%	296	33	4	0	0%	703.5	93.6%	297	31	3
update	0.5	8.3%	0	0%	23	21	19	0.5	8.2%	0	0%	27	25	15
total	5.7	100%	751.9	100%	601	103	39	5.8	100%	751.9	100%	533	112	35

users involved in newly appearing transactions. Our goal is to increase the throughput and decrease the latency for TGNN inference while maintaining similar accuracy.

B. Case Study: Memory-Based TGNN Inference

In this subsection, we perform a detailed case study to analyze the computation complexity, memory accesses, and runtime profiling of the memory-based TGNN inference. TGN [13] provides a general framework for node memory-based TGNNs and benchmarks the performance with different UPDT and GNN functions. Among these memory-based TGNN variants, 1-layer attention-based GNN with GRU memory updater (TGN-attn) has the highest accuracy to complexity ratio. Hence, we focus on optimizing the performance of TGN-attn in this work. Nevertheless, our proposed optimizations also apply to other TGNNs.

In TGN-attn, when a graph signal of new interaction between nodes i and j appears, two messages are generated

$$\boldsymbol{m}_i = \boldsymbol{s}_i || \boldsymbol{s}_j || \boldsymbol{f}_e || \Phi(\Delta t) \tag{4}$$

$$\boldsymbol{m}_j = \boldsymbol{s}_j || \boldsymbol{s}_i || \boldsymbol{f}_e || \Phi(\Delta t) \tag{5}$$

where $||$ denotes concatenation, $\boldsymbol{f}_e$ is the edge feature vector, and Δt the time difference between the most recent node memory $\boldsymbol{s}$ and the timestamp of the graph signal. $\Phi(\cdot)$ is a time encoder similar to the positional encoder in Transformer with two learnable vectors $\boldsymbol{\omega}$ and $\boldsymbol{\phi}$

$$\Phi(\Delta t) = \cos(\boldsymbol{\omega}\Delta t + \boldsymbol{\phi}) \tag{6}$$

All Messages to a node in a batch are aggregated with the "Most-Recent" aggregator which simply keeps the most recent message of each temporal node. The aggregated message $\overline{\boldsymbol{m}_i}$ is then sent to a GRU cell which updates the node memory $\boldsymbol{s}_i$ using the previous node memory as hidden state and aggregated messages as input features, as shown below.

$$\boldsymbol{r}_i = \sigma(\boldsymbol{W}_{ir}\overline{\boldsymbol{m}_i} + \boldsymbol{b}_{ir} + \boldsymbol{W}_{hr}\boldsymbol{s}_i + \boldsymbol{b}_{hr}) \tag{7}$$

$$\boldsymbol{z}_i = \sigma(\boldsymbol{W}_{iz}\overline{\boldsymbol{m}_i} + \boldsymbol{b}_{iz} + \boldsymbol{W}_{hz}\boldsymbol{s}_i + \boldsymbol{b}_{hz}) \tag{8}$$

$$\boldsymbol{n}_i = \tanh(\boldsymbol{W}_{in}\overline{\boldsymbol{m}_i} + \boldsymbol{b}_{in} + \boldsymbol{r}_i(\boldsymbol{W}_{hn}\boldsymbol{s}_i + \boldsymbol{b}_{hn})) \tag{9}$$

$$\boldsymbol{s}_i = (1 - \boldsymbol{z}_i)\boldsymbol{n}_i + \boldsymbol{z}_i\boldsymbol{s}_i \tag{10}$$

where $\boldsymbol{W}$ and $\boldsymbol{b}$ denote learnable weights and biases. The updated node memory is now combined with the static node

features $\boldsymbol{f}_i$ and fed into the attention aggregator. Since the node memory already contains the status of the nodes in the past, the attention aggregator can focus on recent graph signals. A fixed amount of most recent temporal neighbors $j \in \mathcal{N}_i$ are sampled and fed to the attention aggregator

$$\boldsymbol{f}_i' = \boldsymbol{s}_i + \boldsymbol{W}_s\boldsymbol{f}_i + \boldsymbol{b}_s \tag{11}$$

$$\boldsymbol{q} = \boldsymbol{W}_q\left[\boldsymbol{f}_i'||\Phi(0)\right] + \boldsymbol{b}_q \tag{12}$$

$$\boldsymbol{K} = \boldsymbol{W}_k\left[\boldsymbol{f}_j'||\boldsymbol{e}_{ij}||\Phi(\Delta t)\right] + \boldsymbol{b}_k \tag{13}$$

$$\boldsymbol{V} = \boldsymbol{W}_v\left[\boldsymbol{f}_j'||\boldsymbol{e}_{ij}||\Phi(\Delta t)\right] + \boldsymbol{b}_v \tag{14}$$

$$\boldsymbol{h}_i = \text{softmax}\left(\frac{\boldsymbol{q}\boldsymbol{K}^{\mathrm{T}}}{\sqrt{|\mathcal{N}_v|}}\right)\boldsymbol{V} \tag{15}$$

where the attention aggregator performs vector-vector multiplication between the queries $\boldsymbol{q}$ and keys $\boldsymbol{K}$ to determine attention scores $\boldsymbol{h}_i$. After computing the required dynamic node embeddings in each batch, we obtain the updated node memory $\{\boldsymbol{s}_i\}$ and the messages $\{\boldsymbol{m}_i\}$ of the involved nodes $\{i\}$. We update the node memory and cache the messages in a global copy which is usually stored in the external memory.

For analysis, we divide the whole process into four parts: sample, memory, GNN, and update. The sample part accesses the dynamic graph and samples most recent temporal neighbors. The memory part aggregates the messages and computes the updated node memory. The GNN part applies attention aggregator to generate the dynamic node embeddings. The update part writes the updated node memory back and updates the cached messages. We calculate the number of MEMory accesses (MEMs) (assuming the learnable parameters are stored on-chip) and Multiplication and ACcumulations (MACs) in each part on two popular dynamic graphs Wikipedia and Reddit [12]. For each node, we follow the setup in TGN [13] and sample 10 most recent neighbors from all temporal neighbors as the supporting nodes. We run an optimized version of the open-sourced code [1], which replaces the inefficient python loops with batched operations, on 1) a single thread in the Intel Xeon Gold 5120 CPU 2) 32 threads on the same CPU 3) an Nvidia Titan Xp GPU and measure the execution time of the four parts. Table I shows the complexity and execution time in the four parts. The memory accesses are primarily in the memory part to access the messages and

---

[1] https://github.com/twitter-research/tgn

edge features. The computation is dominated by the GNN part to aggregate from selected temporal neighbors. For a serial processor (1CPU), the bottleneck is the computation in the GNN part. For highly parallel machines (32 CPU threads and GPU), the bottleneck lies in the memory part that accesses the memory and aggregate the messages. Although the number of memory updates is not enormous in the update part, due to the need to maintain the sequential order, it still becomes the bottleneck when executed on a highly-paralleled machine with complex cache hierarchy.

III. MODEL-ARCHITECTURE CO-DESIGN

We use our case study on memory-based TGNN inference in Section II-B to identify *three key points* in designing an optimized inference system:

1) GNN computation accounts for more than 80% of the total time and is the bottleneck on a single CPU core with more than 50% of the time spent on computing the attention scores (Equation (12) and (13)). It is linear in the number of supporting temporal neighbors.
2) The time encoding maps a scalar time interval to a vector which is further multiplied with weight matrices W_{ir}, W_{iz}, W_{in}, W_q, W_k, and W_v. These vector-matrix multiplications can be removed if we can reverse the computation order.
3) On a massively parallel architecture, the key bottleneck is fetching and updating the node memory and messages of the supporting nodes from and to external memory.

Based on these key points, we propose a model-architecture co-design by exploiting FPGA-specific features. FPGAs consist of massive on-chip memory – Block RAMs (BRAMs) and Ultra RAMs (URAMs) that allow fast random memory access in high bandwidth. The built-in DSPs can perform large number of arithmetic operations in each cycle. Based on these features, we propose a simplified temporal attention mechanism (Section III-A) and a temporal neighbor pruning strategy (Section III-B) which reduce the execution time in DSPs and enable data pre-fetching. In addition, the DSPs of FPGAs can be programmed into computation arrays that enable fine-grained parallelism for batched processing. We replace the time encoding and the subsequent vector-matrix multiplications with look-up tables (LUTs) (Section III-C) which are stored in the programmable on-chip memory of FPGAs for fast accesses. Besides, the on-chip memory of FPGAs can be programmed into customized cache structures (Section IV-B) which facilitate fast updates to vertex data.

A. Simplified Temporal Attention

As seen in Equation (15), the traditional temporal attention mechanism requires vector-vector multiplication among neighbors which consumes a lot of DSPs on FPGAs. We note that temporal neighbors in dynamic graphs can be naturally ordered by timestamp. We leverage this unique characteristic to design a simplified attention aggregator that operates on fixed-length lists of n timestamp-sorted (not necessarily unique) temporal neighbors. Given a node u at timestamp t^u

with n sorted temporal neighbors at respective timestamps $t^{v_0} \leq t^{v_1} \leq \cdots \leq t^{v_{n-1}}$, we compute its attention score as

$$\boldsymbol{\alpha}'(u) = \text{Softmax}(\boldsymbol{a} + \boldsymbol{W}_t \boldsymbol{\Delta t}^u) \qquad (16)$$

where $\boldsymbol{a}$ is a learnable constant attention vector shared among all nodes and $\boldsymbol{W}_t$ is a learnable weight matrix that maps node-specific time difference $\boldsymbol{\Delta t}^u = [t^u - t^{v_0}, \cdots, t^u - t^{v_{n-1}}]$ to respective offsets of the attention logits. The intuition behind Equation (16) is that on a dynamic graph, attention scores should be sensitive to chronology of neighbors. Since each node u has a specific neighbor interaction frequency, we use this node-specific offset to produce its attention score. Our simplified attention mechanism eliminates the vector-vector multiplication operations (Equation (15)) which dramatically saves the DSP usage on FPGAs.

To learn $\boldsymbol{a}$ and $\boldsymbol{W}_t$, we apply a simplified knowledge distillation [20] setup under which we train *student models* (with our simplified temporal attention aggregators) under both self-supervision from temporal edges and supervision from a teacher model with the vanilla temporal attention aggregator. We add an additional soft cross-entropy loss l_a between the simplified attention logits $\overline{\boldsymbol{\alpha}}' = \boldsymbol{a} + \boldsymbol{W}_t \boldsymbol{\Delta t}$ and the vanilla attention logits $\overline{\boldsymbol{\alpha}}$ to encourage the student models to mimic the behaviour of the teacher model

$$l_a = -\sum_v \text{Softmax}(\overline{\boldsymbol{\alpha}}'(v)/T) \cdot \text{Softmax}(\overline{\boldsymbol{\alpha}}(v)/T) \qquad (17)$$

where T is the temperature that controls how much the student model learns from the teacher model.

B. Temporal Neighbor Pruning

The Transformer attention mechanism shown in Equation (11)-(14) computes the attention scores after the computation of the keys $\boldsymbol{K}$ and queries $\boldsymbol{Q}$. In contrast, our simplified temporal attention mechanism computes the attention scores only using the time difference $\boldsymbol{\Delta t}$ of the temporal neighbors as the input. This allows models to quickly determine which temporal neighbor is more important before fetching the hidden features from them all. Although the amount of computation and number of memory accesses are the same for each temporal neighbor, the neighbors with higher attention scores contribute more to the output hidden features, which naturally leads to our attention score-based temporal neighbor pruning strategy. Under the simplified attention mechanism where only the values $\boldsymbol{V}$ needs to be computed, performing temporal neighbor pruning directly leads to a linear reduction in computation and memory accesses. For a given pruning budget (number of temporal neighbors to aggregate from), after computing the logits of the simplified attention scores, we apply softmax function only on the temporal neighbors with top logit values and compute their $\boldsymbol{V}$.

C. Time Encoding Look-Up-Table

The time encoder in Equation (6) maps scalar time frames to vectors. These vectors are later multiplied with the weight
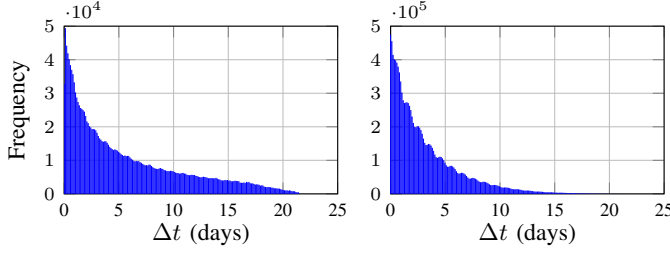
Fig. 1. Frequency of input Δt of the time encoder on the Wikipedia (left) and Reddit (right) datasets.

matrices in the GRU memory updater and the GNN aggregator. This whole process accounts for around 30% of the total computation in the TGNN model with our simplified attention mechanism. This can be completely avoided if the computation order is reversed and the vector-matrix multiplication is pre-computed. However, the time encoding process involves a nonlinear trigonometric function which does not permit pre-computation. To solve this problem, we replace the time encoding process with LUT operations which transforms the nonlinear operations to piece-wise linear ones. We analyze the input Δt of the time encoder and observe that it follows the power law where most inputs are close to 0. Based on the intuition that the output vectors should distinguish different length of time frames, we divide the range of the input Δt to 128 intervals with equal number of Δt occurrences in each interval. The output time encoding vectors in each interval are stored in one entry in the LUT, which is learned in the training process. At inference, we pre-compute the product of each entry in the LUTs with the weight matrices and store them in the fast on-chip memory. Our LUT-based time encoder can directly output the hidden features after weight application for any given time frame within 1 clock cycle.

IV. HARDWARE MAPPING AND OPTIMIZATIONS

Figure 2 illustrates the overview of the proposed architecture that executes the inference process of TGNN (Algorithm 1). The proposed architecture consists of the following parts:

- **Input**: The data packets of the new edges are sent to FPGA accelerators through the Direct Memory Access (DMA) unit.
- **Graph Storage** The FPGA board consists of an external DDR memory and an FPGA chip. The external DDR memory stores various vertex information, including the Vertex Mailbox $\{\boldsymbol{m}_v : v \in \mathcal{V}\}$, the Vertex Memory Table $\{\boldsymbol{s}_v : v \in \mathcal{V}\}$, the vertex feature vectors $\{\boldsymbol{f}_v : v \in \mathcal{V}\}$, and the Vertex Neighbor Table $\{\mathcal{N}_{mr}(v) : v \in \mathcal{V}\}$.
- **Accelerator**: The proposed hardware accelerator consists of four parts – the Edge Parser, the Data Loader, the Updater, and the Computation Units. The memory controller handles the data transmissions between the external memory and the accelerator. The Edge Parser receives the new edges from the host processor through DMA and parses the raw information of the new edges. The Data loader loads the required inputs from external memory. The Updater ensures the chronological order

---

Algorithm 1 Inference process of Memory-based TGNN on the proposed accelerator

Input: Graph $\mathcal{G}(\mathcal{E}, \mathcal{V})$; A edges stream $\mathcal{E}_{\text{new}}$ that incoming edges follow the chronological order; Old vertex memory $\{\boldsymbol{s}_v : v \in \mathcal{V}\}$; Old cached messages $\{\boldsymbol{m}_v : v \in \mathcal{V}\}$; Vertex feature vectors $\{\boldsymbol{f}_v : v \in \mathcal{V}\}$; Edge feature vectors $\{\boldsymbol{f}_e : e \in \mathcal{E}\}$;

1: $\{$ % Process batches of new edges in chronological order % $\}$
2: **for** each batch $\{e(u, v, \boldsymbol{f}_e, t_e)\} \in \mathcal{E}_{\text{new}}$ **do**
3:     $\{$ % update vertex memory % $\}$
4:     $\{\boldsymbol{s}_u\} = \{\text{UPDT}(\boldsymbol{m}_u, \boldsymbol{s}_u, t_e)\}$
5:     $\{\boldsymbol{s}_v\} = \{\text{UPDT}(\boldsymbol{m}_v, \boldsymbol{s}_v, t_e)\}$
6:     $\{$ % update cached messages % $\}$
7:     $\{\boldsymbol{m}_u\} = \{\boldsymbol{s}_u || \boldsymbol{s}_v || \boldsymbol{f}_e || \Phi(t_e)\}$
8:     $\{\boldsymbol{m}_v\} = \{\boldsymbol{s}_v || \boldsymbol{s}_u || \boldsymbol{f}_e || \Phi(t_e)\}$
9:     $\{$ % compute output embeddings % $\}$
10:    $\{\boldsymbol{h}_u\} = \{\text{GNN}((\boldsymbol{s}_u, \boldsymbol{f}_u, t_e), \{(\boldsymbol{s}_z, \boldsymbol{f}_z, t_z), z \in \mathcal{N}(u)\})\}$
11:    $\{\boldsymbol{h}_v\} = \{\text{GNN}((\boldsymbol{s}_v, \boldsymbol{f}_v, t_v), \{(\boldsymbol{s}_z, \boldsymbol{f}_z, t_z), z \in \mathcal{N}(v)\})\}$
12:    $\{$ % update vertex neighbors % $\}$
13:    $\{\mathcal{N}(v) = \text{UpdateNeighbor}(\mathcal{N}(u), v)\}$
14:    $\{\mathcal{N}(u) = \text{UpdateNeighbor}(\mathcal{N}(v), u)\}$
15: **end for**

---

of the updated vertex information and sends the updated vertex information back to external memory. The Computation Units (CUs) perform the key computation stages of TGNN inference where each CU has its own Memory Update Unit (MUU) to generate the updated vertex memory and an Embedding Unit (EU) to generate the updated vertex embedding.

Runtime: At runtime, the accelerator first receives a new batch of edges (line 2 of Algorithm 1). Then, the MUU calculates the vertex memory of involving vertices and the Updater updates the vertex memory and vertex cache messages (line 3-8 of Algorithm 1) in the external memory of FPGA board. After that, the EU calculates the vertex embeddings for the downstream tasks (line 9-11 of Algorithm 1) and the vertex neighbor table is updated based on the new edges (line 12-14 of Algorithm 1). To improve the overall throughput, we adapt batching, fine-grained task pipelining and prefetching to fully exploit the parallelism (Section IV-C).

A. Data Structure

We represent each edge in a dynamic graph as $e(src, dst, \boldsymbol{f}_e, t_e)$ where src, dst, $\boldsymbol{f}_e, t_e$ denote the source index, destination index, feature vector, and timestamp, respectively. Vertex information consists of the Vertex Mailbox (cached messages), the Vertex Memory Table, and the Vertex Neighbor Table. Each row of the Vertex Mailbox contains the most recent message $\boldsymbol{m}_v$ of a vertex. Each row of the Vertex Memory Table contains the most recent vertex memory $\boldsymbol{s}_v$ of a vertex. Each row of the Vertex Neighbor Table contains the indices of the most recent mr neighbors $\mathcal{N}_{mr}(v)$ of a vertex.

B. Hardware Modules

In the proposed accelerator, the UPDT$(\cdot)$ function is implemented as a GRU (Equation (7), (8), (9), (10)) and the GNN$(\cdot)$ function is implemented as a 1-layer GNN with attention mechanism (Equation (16)). The GRU is mapped to the MUU and the 1-layer GNN is mapped to the EU. The CU performs
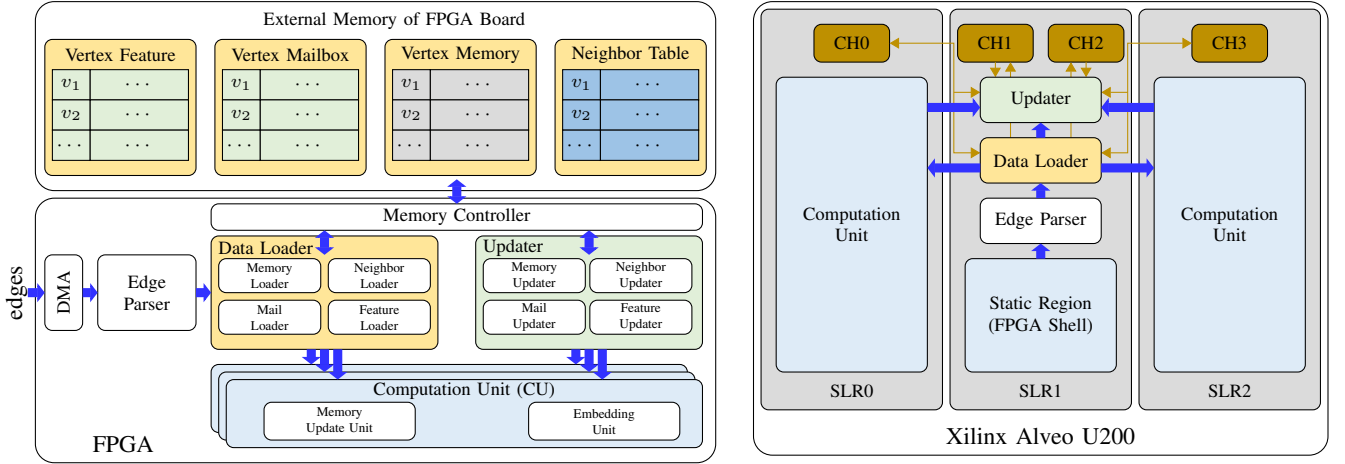
Fig. 2. The principled hardware architecture for TGNN on FPGA platform (Left). Mapping of the architecture on Xilinx Alveo U200 platform (Right).

batched execution for vertex information updating. When MUU is ready to receive new inputs, the vertex messages $\{\boldsymbol{m}_v : v \in \mathcal{V}_b\}$ and vertex memory $\{\boldsymbol{s}_v : v \in \mathcal{V}_b\}$ of N_b vertices $\mathcal{V}_b$ ($|\mathcal{V}_b| = N_b$) are sent to MUU for memory updating. The updated vertex memory are sent to EU to generate vertex embedding.

Memory Update Unit: In GRU, there are an Update Gate (Equation (7)), a Reset Gate (Equation (8)), a Memory Gate (Equation (9)) ,and a Merging Gate (Equation (10)). The Update Gate, Reset Gate, and Memory Gate involve matrix multiplication between the vertex messages (Equation (4), (5)) of N_b vertices and the weight matrices. For each one of the three gates, a Multiply-Accumulate Array of size $S_g \times S_g$ is implemented for efficient matrix multiplication. The four gates are connected through the on-chip FIFO and their execution is pipelined to achieve task-level parallelism.

Embedding Unit: The EU performs 1-layer message passing with an Attention Module (AM) to calculate the attention scores of the neighbors $\{\alpha(u) : u \in \mathcal{N}_{mr}(v)\}$, a Feature Aggregation Module (FAM) to perform feature aggregation: $\boldsymbol{h}_v = aggregate\{\alpha(u) \cdot \boldsymbol{s}_u : u \in \mathcal{N}_{mr}(v) \cup \{v\}\}$ and a Feature Transformation Module (FTM) to perform feature transformation $\boldsymbol{h}_v = transform(\boldsymbol{h}_v, \boldsymbol{s}_v, \boldsymbol{W}_v)$. The FAM uses the multiply-add tree-based design to aggregate information from the neighbors. FTU performs multiplication of the aggregated vertex memory vectors and the weight matrix which is also implemented as a Multiply-Accumulate Array.

Updater: The function of Updater is to ❶ receive the vertex

Fig. 3. Updater using a fully-associative cache with rotating pointers ($N_{cu} = 2$).

information from the computation units, ❷ write the vertex information back to the external memory, ❸ ensure the chronological order of the updated vertex information, and ❹ eliminate the redundant vertex updating. To ensure the chronological order, the new edges are assigned to the CUs in Round-Robin style. Similarly, the Updater receives the updated vertex information from the CUs in the same Round-Robin order. Figure 3 shows the diagram of the Updater. The Updater is organized as a fully-associative cache with rotating pointers. Each cache line stores one vertex information (message/memory/neighbors), the index of the vertex, and a flag bit that indicates whether the current cache line is valid. When the Updater receives the vertex information from multiple CUs concurrently, the chronological order of the multiple vertex information is ensured by the relative position of the write pointers. Each write pointer points to the write position of a CU. The commit pointer scans multiple consecutive cache lines at a time to check if a valid cache link can be sent back to the external memory. When multiple new updated vertex information are received by the Updater, their vertex indices are compared with vertex index (vid) of each cache line. If an uncommitted cache line has the same vertex index with new vertex information, this uncommitted cache line will be invalidated.

Multi-die Implementation: Advanced FPGA platforms are usually integrated with multiple dies with limited number of inter-die connections. The right side of the Figure 2 demonstrates our multi-die implementation on Xilinx Alveo U200 board. Multiple memory channels are connected to the data loader and updater. The hardware modules are distributed into different dies (Super Logic Regions) with on-chip FIFOs as the interconnection.

C. Dataflow Optimizations

We exploit three design principles to optimize the overall performance: batching, task pipelining, and prefetching. Figure 4 depicts the detailed task scheduling.

Batching: In each time interval T_p, the computation unit groups a set of edges to start the execution. This exploits data parallelism within the CUs.
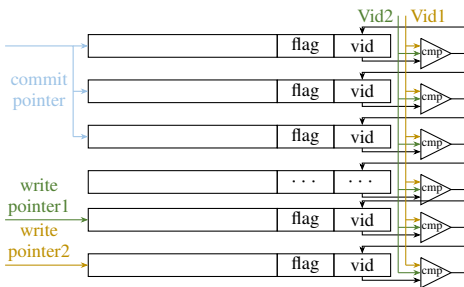
Fig. 4. Task scheduling.

1. Load edges
2. Load neighbors, vertex memory, mail vector
3. Prefetch memories of neighbors
4. Update neighbors, vertex memory, mail vector
5. Update vertex embedding
6. Memory Update Unit: (1) Time encoding,
   (2) Update gate, (3) Reset gate,
   (4) Memory gate, (5)Merging Gate
7. Embedding Unit: (1) Attention, (2) Time encoding,
   (3) Feature aggregation, (4) Feature transformation
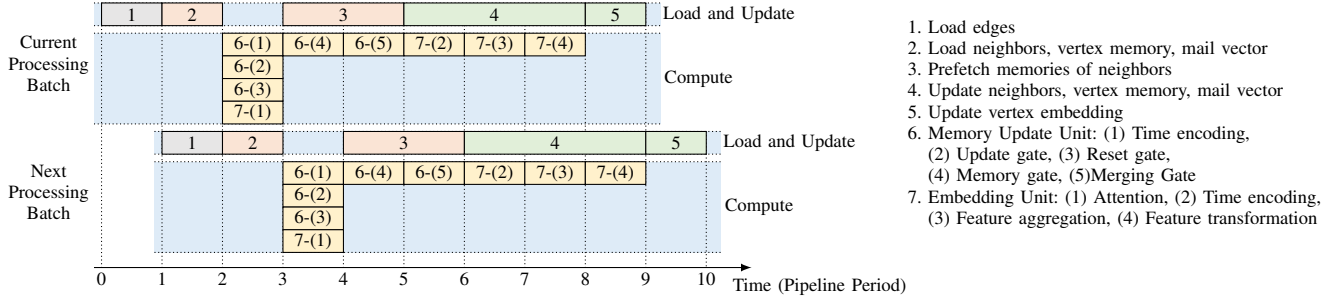
Task pipelining: The execution of consecutive processing batches are fully pipelined to improve the overall throughput. To facilitate the pipelined execution, (1) the computation operations and the memory accesses are overlapped, (2) the MMU and EU are pipelined, and (3) the individual hardware modules within MUU and EU are pipelined.

Prefetching: GNN($\cdot$) requires the vertex memory of the neighbors where loading from external memory leads to large latency. Therefore, in the task scheduling (Feature 4), the calculation of the attention score (7-(2)) in the EU is executed after loading the timestamps of the neighbors. The EU uses the attention score to obtain the indices of the required neighbors and then prefetches the memory of the neighbors before the MUU finishes execution. Note that the prefetching of vertex memory is promoted by the proposed attention mechanism (Equation 16) which does not require the updated vertex memory to obtain the attention scores.

V. PERFORMANCE MODEL

In this section, we introduce our accurate performance model which predicts the performance of the proposed model-architecture co-design on a given FPGA. We define the following notations:

- Length of feature vector, message vector, memory, neighbor list, and embedding of a vertex are f_{feat}, f_{mail}, f_{mem}, mr, and f_{emb}. The size of each data is Z_d bytes.
- Number of CUs: N_{cu}. Computation Parallelism of each Gate in MUU: $S_g \times S_g$. Computation parallelism of FAM: S_{FAM}. Computation parallelism of FTM: S_{FTM}.
- Number of edges processed concurrently in a pipeline stage (size of a processing batch): N_b (see Figure 4).
- Frequency of FPGA design: F_{freq}

Due to our task scheduling (Figure 4), the execution of a processing batch of N_b edges are divided into 9 stages. The time period T_p of a pipeline stage is decided by the longest stage $T_{\text{comp}}^{\max}$ or the time of loading and storing data from/to external memory T_{LS}:

$$T_p = \max(T_{\text{comp}}^{\max}, T_{\text{LS}}) \tag{18}$$

where

$$T_{\text{comp}}^{\max} = \max(\{T_{6-(i)} : i = 1, .., 5\}, \\ \{T_{7-(i)} : i = 1, .., 4\}) \tag{19}$$

For example, $T_{6-(1)}$ is the execution latency of the time encoding stage in MUU as shown in Figure 4. $T_{\text{comp}}^{\max}$ can be approximated by the dominant terms

$$T_{\text{comp}}^{\max} \approx \frac{1}{F_{\text{freq}}} \cdot \max(\frac{3 \cdot N_b \cdot f_{\text{mail}} \cdot f_{\text{mem}}}{S_g \cdot S_g}, \\ \frac{3 \cdot N_b \cdot mr \cdot (f_{\text{mem}} + f_{\text{feat}})}{S_{\text{FAM}}}, \frac{3 \cdot N_b \cdot (f_{\text{mem}} + f_{\text{feat}}) \cdot f_{\text{emb}}}{S_{\text{FTM}}}). \tag{20}$$

To drive an expression for T_{LS}, we assume the external memory bandwidth is $\alpha(l) \cdot BW$, where $\alpha(l), (0 < \alpha(l) \leqslant 1)$ is a factor specifying the effective bandwidth when the length of burst data transaction is l [21] and BW is the peak memory bandwidth between FPGA and external memory. Similarly, T_{LS} can be approximated as:

$$T_{\text{LS}} \approx \frac{6 \cdot N_b \cdot f_{\text{mail}} \cdot Z_d}{\alpha(f_{\text{mail}}) \cdot BW} + \frac{3 \cdot N_b \cdot (2 + mr) \cdot f_{\text{mem}} \cdot Z_d}{\alpha(f_{\text{mem}}) \cdot BW} \\ + \frac{3 \cdot N_b \cdot mr \cdot f_{\text{feat}} \cdot Z_d}{\alpha(f_{\text{feat}}) \cdot BW} + \frac{3 \cdot N_b \cdot f_{\text{emb}} \cdot Z_d}{\alpha(f_{\text{emb}}) \cdot BW} \tag{21}$$

When the batch size N is far larger than the processing batch size N_b, the maximum throughput and latency can be expressed as:

$$\text{Maximum Throughput} \approx \frac{N_b}{T_p}$$
$$\text{Latency} \approx (\beta - 1 + \lceil \frac{N}{N_b} \rceil) \cdot T_p \tag{22}$$

where β is the number of pipeline stages in task scheduling.

VI. IMPLEMENTATION AND EXPERIMENTAL RESULTS

We evaluate the performance of our proposed model-architecture co-design with the widely-studied temporal link prediction task on three datasets – Wikipedia [12], Reddit [12], and GDELT [9]. For the GDELT dataset, we use the pre-trained 200-dimensional node embedding from SeDyT [10] as the input node features. These three datasets cover the dynamic graphs with edge features (Wikipedia and Reddit) and with node features (GDELT). We implement our hardware accelerators on two FPGAs. Table III shows the specifications of the hardware platforms used in this work. We train three different sizes of the simplified models NP(L/M/S) with 6/4/2 temporal neighbors as the neighbor pruning budgets. We set

TABLE II
Accuracy, computation complexity, latency, and throughput (with a single CPU thread) of the original and optimized models. The SAT row denotes our simplified attention mechanism. The LUT row denotes LUT-based time encoder. The NP(L/M/S) rows denotes the neighbor pruning with 6/4/2 neighbors. We show the model with accumulated optimizations row by row.

	Model	$\|v_i\|$	$\|e_{ij}\|$	$\|\mathcal{N}(v)\|$	kMEM #	kMEM %	#(GRU)	#(GNN)	#(Tot.)	%(Tot.)	AP	(difference)	Thpt. (kE/s)	Speedup
Wikipedia	Baseline	0	172	10	5.7	100%	48.4	703.5	751.9	100%	0.9900	(-0.0000)	0.85	1×
	+SAT	0	172	10	5.7	100%	48.4	351.1	399.5	53.1%	0.9821	(-0.0079)	1.10	1.29×
	+LUT	0	172	10	3.8	100%	38.3	240.0	278.3	37.0%	0.9891	(-0.0009)	1.12	1.32×
	+NP(L)	0	172	6	3.8	66.7%	38.3	156.4	194.5	25.9%	0.9891	(-0.0009)	1.71	2.01×
	+NP(M)	0	172	4	2.9	50.9%	38.3	114.6	152.9	20.3%	0.9887	(-0.0013)	2.71	3.19×
	+NP(S)	0	172	2	1.9	33.3%	38.3	72.8	111.1	14.8%	0.9878	(-0.0022)	3.22	3.79×
Reddit	Baseline	0	172	10	5.8	100%	48.4	703.5	751.9	100%	0.9978	(-0.0000)	0.92	1×
	+SAT	0	172	10	5.8	100%	48.4	351.1	399.5	53.1%	0.9967	(-0.0011)	1.22	1.33×
	+LUT	0	172	10	5.8	100%	38.3	240.0	278.3	37.0%	0.9978	(-0.0000)	1.21	1.32×
	+NP(L)	0	172	6	3.9	67.2%	38.3	156.4	194.5	25.9%	0.9971	(-0.0007)	1.51	1.64×
	+NP(M)	0	172	4	3.0	51.7%	38.3	114.6	152.9	20.3%	0.9971	(-0.0007)	1.93	2.10×
	+NP(S)	0	172	2	2.0	34.4%	38.3	72.8	111.1	14.8%	0.9948	(-0.0030)	2.21	2.40×
GDELT	Baseline	200	0	10	5.1	100%	51.2	733.8	785.0	100%	0.9623	(-0.0000)	1.29	1×
	+SAT	200	0	10	5.1	100%	51.2	371.3	422.5	53.8%	0.9612	(-0.0011)	1.83	1.42×
	+LUT	200	0	10	5.1	100%	41.1	260.2	301.3	38.4%	0.9605	(-0.0018)	1.85	1.43×
	+NP(L)	200	0	6	3.4	66.7%	41.1	176.6	217.7	27.7%	0.9598	(-0.0025)	3.01	2.33×
	+NP(M)	200	0	4	2.5	49.1%	41.1	134.8	175.9	22.4%	0.9596	(-0.0027)	3.62	2.81×
	+NP(S)	200	0	2	1.7	31.5%	41.4	93.0	134.1	17.1%	0.9590	(-0.0033)	4.43	3.43×

the temperature T in the knowledge distillation loss to 1 during training. For the rest of the hyper-parameters, we follow the training setup in the baseline code [13].

A. FPGA Implementation Results

We implement our design on two state-of-the-art FPGA platforms – Xilinx Alveo U200 and Xilinx ZCU104[2]. The accelerators are developed using Xilinx High-level Synthesis (HLS). HLS is a pragma-directive programming language that allows user to develop the accelerator design using C/C++. Alveo U200 is a cloud-based FPGA board while ZCU104 is an embedded FPGA board with a built-in ARM processor. We use ZCU104 to demonstrate that the proposed design can be deployed on light-weight embedded platforms, which is useful for applications on edge devices such as Internet of Things (IoTs). We set the commit pointer in the Updater to scan 3 consecutive cache lines each cycle. A BRAM has the size of 36K bits and an URAM has the size of 288K bits on both FPGAs. With the IEEE float32 data format, each multiplier consumes 3 DSPs while each accumulator consumes

TABLE III
Specifications of various hardware platforms

Hardware	# of dies /sockets/boards	Computation Resources Per Die	External Memory Bandwidth
Xilinx Alveo U200	3	394K LUTs, 2280 DSPs 720 BRAMs, 320 URAMs	77 GB/s DDR4
Xilinx ZCU104	1	230K LUTs, 1728 DSPs 312 BRAMs, 96 URAMs	19.2 GB/s DDR4
Dual Intel Xeon Gold 5120 CPU (CPU baseline)	2	14 Cores, 28 Threads 2.20 GHz	89 GB/s DDR4
Nvidia Titan X (GPU baseline)	1	3840 CUDA cores 1532 MHz	547 GB/s HBM

2 DSPs. The design configurations and resource utilization of the accelerators on the two FPGAs are shown in Table IV. We use the Xilinx Vitis 2020.2 for hardware synthesis and place & route. The reported resource utilization and frequency are obtained after place & route.

B. Evaluation Results

Effect of Model Optimization: To compare the performance of our proposed model-architecture co-design, we first evaluate the effect of our simplified models on a single CPU core. Table II shows the reduction in the number of operations and the loss in Average Precision (AP) of the simplified models. Our neighbor pruning strategy achieves near-linear reduction in the memory accesses and computation with respect to the number of neighbors left. The simplified attention mechanism greatly reduces half of the total computation while the LUT-based time encoder reduces another 15%. On a single CPU core, our simplified model achieves an average of $3.21\times$ speedup in throughput with less than 0.0033 drop in AP. Note that our LUT-based time encoder does not show evident improvement due to the hardware limitation that CPU cannot store the LUT on-chip for fast access.

Cross-Platform Comparison: We compare the performance of our FPGA accelerators with the same optimized code as in Section II-B on CPU (32 threads) and GPU. On the FPGA platforms, we run three models with different sizes NP(L/M/S). Due to the limited size of external memory,

TABLE IV
Design configurations and Resource utilization

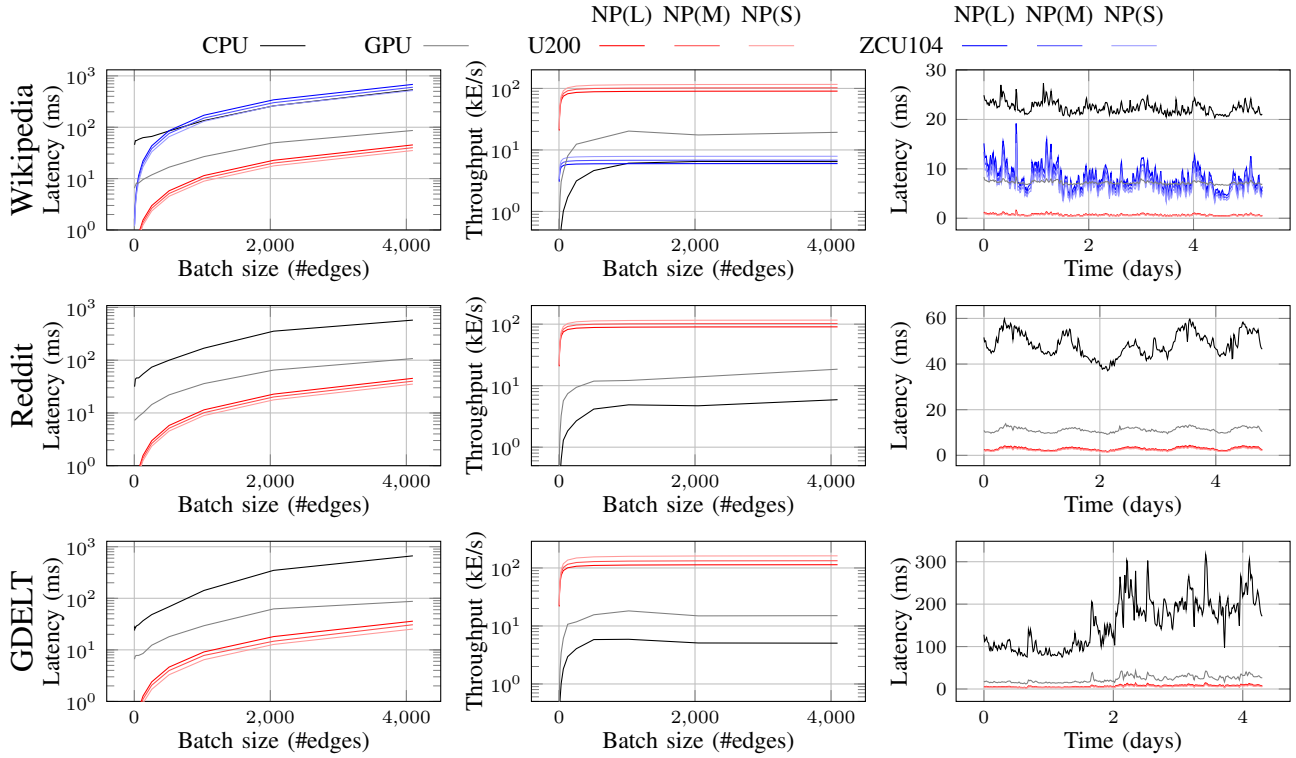|	Design configurations N_{cu}	S_g^2	S_{FAM}	S_{FTM}	Resource utilization LUT	DSP	BRAM	URAM	Freq. (MHz)
U200	2	8^2	16	8×8	563k	2512	1415	448	250
ZCU104	1	4^2	8	4×4	125k	744	240	0	125

Fig. 5. Performance of our hardware accelerator on two FPGAs compared with the baseline on CPU and GPU. The left two plots show the latency and throughput (in log scale) under various batch sizes. The right plots show the real-time inference latency on the test set.

we only test the performance of the Wikipedia dataset on Xilinx ZCU104. Figure 5 shows the latency and throughput of the baselines and our hardware accelerators. We show the latency and throughput under various batch sizes (the first two columns). Our accelerator on Xilinx Alveo U200 achieves more than $13.9/15.8/17.9\times$ speedup compared with CPU and more than $4.6/5.2/6.0\times$ speedup compared with GPU using the NP(L/M/S) models. Note that the accuracy of our simplified models are the same on FPGAs as on CPU (see Table II). The obtained speedup on FPGAs is due to our model-architecture co-design that results in low complexity and takes advantages of the FPGA features (Section III). To simulate the performance when deployed in production environments, we also evaluate the real-time latency of inferencing on the upcoming graph signals every 15 minutes on the three datasets. We perform inference on batches of new edges in every 15 minutes and measure the latency of each batch. On Xilinx ZCU104, our hardware accelerators achieve similar latency as GPU but with larger fluctuation due to the limited computing resources. On Xilinx Alveo U200, we achieve remarkable speedup (more than $3.66\times$) in latency compared with GPU. Our smallest model NP(S) has less than 10ms latency on all three datasets which meets the requirements of most real-time applications.

Evaluation of Performance Model: We evaluate our performance model by comparing the predicted latency and throughput with the experimental results. On average, the
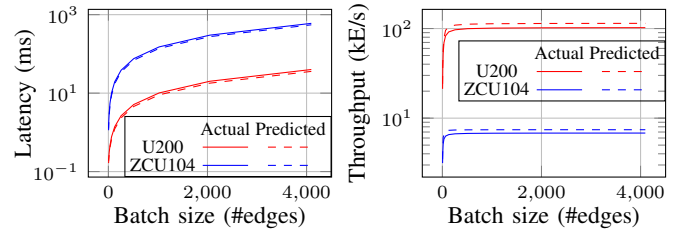

Fig. 6. Predicted and actual performance on two FPGAs with the NP(M) model on the Wikipedia dataset.

prediction error ranges from 9.9% to 12.8%. Figure 6 shows the prediction error using our performance model on the Wikipedia dataset. We attribute the prediction error to two reasons: (1) the fine-grained hardware pipelines generated by the Xilinx Vitis have some flushing & draining cycles that are not included in the performance model. This is hard to estimate since the extra cycles are usually decided by the platforms and the version of the compiler. (2) The refreshing behavior of the DDR memory is hard to predict which leads to periodic extra memory latency.

Comparison with State-of-the-Art Method: Figure 7 shows the accuracy-latency curve of our model-architecture co-design, our baseline method TGN [13], and latency-targeted TGNN APAN [17]. We show the accuracy and latency of TGN and APAN on CPU and GPU since there is no dedicated hardware accelerator designed for these methods. Our co-
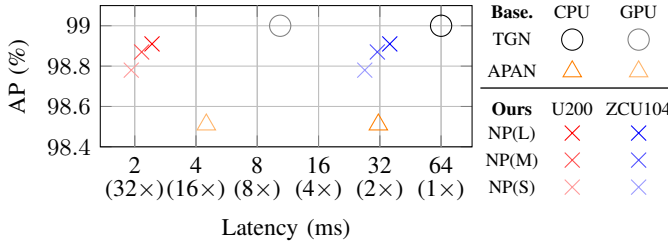
Fig. 7. Accuracy and latency on the Wikipedia dataset (batch size 200).

design achieves significantly higher accuracy than APAN while maintaining similar latency using ZCU104 with CPU and around $2\times$ improvement in latency using U200 with GPU.

VII. RELATED WORKS

The acceleration of GNN on static graphs has been extensively studied before. For example, researchers have proposed algorithm optimizations such as pruning [1], [2] and compression [3] techniques to reduce the computation complexity. However, the only work on accelerating TGNNs that we are aware of is APAN [17] which reduces the latency by asynchronous processing. On the other hand, there are many hardware accelerators proposed for GNNs on static graphs. HyGCN proposed [5] hybrid accelerators for GNN with a self-adaptive sliding window technique to reduce the memory traffic. GraphACT [4] proposed an FPGA accelerator for subgraph sampling-based GNN training. AWB-GCN [6] proposed a dynamic re-balancing technique to resolve the workload imbalance in GNN inference. Nevertheless, the previous works exploit massive data parallelism and task parallelism on static graphs which are not designed to process the vertex memory updating or deal with temporal dependency on dynamic graphs. To the best of our knowledge, this is the first work to optimize the TGNN inference through comprehensive model-architecture co-design.

VIII. CONCLUSION

In this work, we proposed a model-architecture co-design for temporal GNN on FPGA platforms. We defined performance metrics and conducted a case study to identify the bottlenecks in the existing TGNN inference methods. We designed a light-weight and hardware-friendly TGNN inference algorithm which has low computation complexity and external memory accesses which takes advantage of the programmable on-chip memory and massive data parallelism of FPGAs. We mapped the optimized models to principled hardware architectures and implemented the corresponding hardware accelerators on two FPGAs. Our co-design on FPGA platforms achieved significantly better performance than state-of-the-art methods on CPU and GPU on real-world datasets.

ACKNOWLEDGMENT

REFERENCES

[1] H. Zhou, A. Srivastava, H. Zeng, R. Kannan, and V. Prasanna, "Accelerating large scale real-time gnn inference using channel pruning," *Proc. VLDB Endow.*, 2021.

[2] X. Xu, W. Feng, Y. Jiang, X. Xie, Z. Sun, and Z.-H. Deng, "Dynamically pruned message passing networks for large-scale knowledge graph reasoning," in *ICLR*, 2020.

[3] J. Wang, Y. Wang, Z. Yang, L. Yang, and Y. Guo, "Bi-gcn: Binary graph convolutional network," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2021.

[4] H. Zeng and V. Prasanna, "Graphact: Accelerating gcn training on cpu-fpga heterogeneous platforms," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2020.

[5] M. Yan, L. Deng, X. Hu, L. Liang, Y. Feng, X. Ye, Z. Zhang, D. Fan, and Y. Xie, "Hygcn: A gcn accelerator with hybrid architecture," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020.

[6] T. Geng, A. Li, R. Shi, C. Wu, T. Wang, Y. Li, P. Haghi, A. Tumeo, S. Che, S. Reinhardt *et al.*, "Awb-gcn: A graph convolutional network accelerator with runtime workload rebalancing," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020.

[7] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, "Graph convolutional neural networks for web-scale recommender systems," in *ACM SIGKDD International Conference on Knowledge Discovery Data Mining (KDD)*, 2018.

[8] J. Zhang, X. Shi, S. Zhao, and I. King, "Star-gcn: Stacked and reconstructed graph convolutional networks for recommender systems," in *IJCAI*, 2019.

[9] W. Jin, M. Qu, X. Jin, and X. Ren, "Recurrent event network: Autoregressive structure inference over temporal knowledge graphs," in *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2020.

[10] H. Zhou, J. Orme-Rogers, R. Kannan, and V. Prasanna, "Sedyt: A general framework for multi-step event forecasting via sequence modeling on dynamic entity embeddings," in *ACM International Conference on Information and Knowledge Management (CIKM)*, 2021.

[11] A. Li, Z. Qin, R. Liu, Y. Yang, and D. Li, "Spam review detection with graph convolutional networks," *ACM International Conference on Information and Knowledge Management (CIKM)*, 2019.

[12] da Xu, chuanwei ruan, evren korpeoglu, sushant kumar, and kannan achan, "Inductive representation learning on temporal graphs," in *International Conference on Learning Representations*, 2020.

[13] E. Rossi, B. Chamberlain, F. Frasca, D. Eynard, F. Monti, and M. Bronstein, "Temporal graph networks for deep learning on dynamic graphs," in *ICML 2020 Workshop on Graph Representation Learning*, 2020.

[14] A. Sankar, Y. Wu, L. Gou, W. Zhang, and H. Yang, "Dysat: Deep neural representation learning on dynamic graphs via self-attention networks," in *International Conference on Web Search and Data Mining*, 2020.

[15] A. Pareja, G. Domeniconi, J. Chen, T. Ma, T. Suzumura, H. Kanezashi, T. Kaler, T. B. Schardl, and C. E. Leiserson, "EvolveGCN: Evolving graph convolutional networks for dynamic graphs," in *AAAI Conference on Artificial Intelligence*, 2020.

[16] R. Trivedi, M. Farajtabar, P. Biswal, and H. Zha, "Dyrep: Learning representations over dynamic graphs," in *International Conference on Learning Representations*, 2019.

[17] X. Wang, D. Lyu, M. Li, Y. Xia, Q. Yang, X. Wang, X. Wang, P. Cui, Y. Yang, B. Sun *et al.*, "Apan: Asynchronous propagation attention network for real-time temporal graph embedding," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 2628–2638.

[18] Y. Wang, Y.-Y. Chang, Y. Liu, J. Leskovec, and P. Li, "Inductive representation learning in temporal networks via causal anonymous walks," in *International Conference on Learning Representations*, 2021.

[19] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *International Conference on Neural Information Processing Systems*, 2017.

[20] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," *stat*, p. 9, 2015.

[21] A. Lu, Z. Fang, W. Liu, and L. Shannon, "Demystifying the memory system of modern datacenter fpgas for software programmers through microbenchmarking," in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2021.