A Deep Dive Into Understanding The Random Walk-Based Temporal Graph Learning

Nishil Talati[†§] Di Jin[†§] Haojie Ye[†] Ajay Brahmakshatriya[‡]
Ganesh Dasika[*] Saman Amarasinghe[‡] Trevor Mudge[†] Danai Koutra[†] Ronald Dreslinski[†]
[†]University of Michigan [‡]Massachusetts Institute of Technology [*]Advanced Micro Devices, Inc.
Email: talatin@umich.edu

***Abstract*—Machine learning on graph data has gained significant interest because of its applicability to various domains ranging from product recommendations to drug discovery. While there is a rapid growth in the algorithmic community, the computer architecture community has so far focused on a subset of graph learning algorithms including Graph Convolution Network (GCN), and a few others. In this paper, we study another, more scalable, graph learning algorithm based on *random walks*, which operates on dynamic input graphs and has attracted less attention in the architecture community compared to GCN. We propose high-performance CPU and GPU implementations of two important graph learning tasks, that cover a broad class of applications, using random walks on continuous-time dynamic graphs: link prediction and node classification. We show that the resulting workload exhibits distinct characteristics, measured in terms of irregularity, core and memory utilization, and cache hit rates, compared to graph traversals, deep learning, and GCN. We further conduct an in-depth performance analysis focused on both algorithm and hardware to guide future software optimization and architecture exploration. The algorithm-focused study presents a rich trade-off space between algorithmic performance and runtime complexity to identify optimization opportunities. We find an optimal hyperparameter setting that strikes balance in this trade-off space. Using this setting, we also perform a detailed microarchitectural characterization to analyze hardware behavior of these applications and uncover execution bottlenecks, which include high cache misses and dependency-related stalls. The outcome of our study includes recommendations for further performance optimization, and open-source implementations for future investigation.**

***Index Terms*—Characterization, CPU, dynamic graph, graph learning, GPU, random walk, temporal graph, and word2vec.**

I. Introduction

A graph[1] is a ubiquitous data structure that models entities and their interactions through the collections of nodes and edges. It is widely employed in many domains ranging from social media [1] to bioinformatics [2], [3]. More recently, the process of learning representation of graph structured data, *i.e., graph representation learning*, has gained significant popularity in the algorithmic community [4]–[8]. This is due to its superiority on multiple machine learning tasks in domains ranging from social science [9], [10], computer vision [11], physics, chemistry, and biology [12]–[15]. Following this algorithmic evolution, several works in the architecture community have analyzed its workload characteristics [16]–[18], and built domain-specific hardware [19]–[21] for acceleration.

[§]Equal contribution.

[1]In this work, we use the term "graph" and "network" interchangeably.

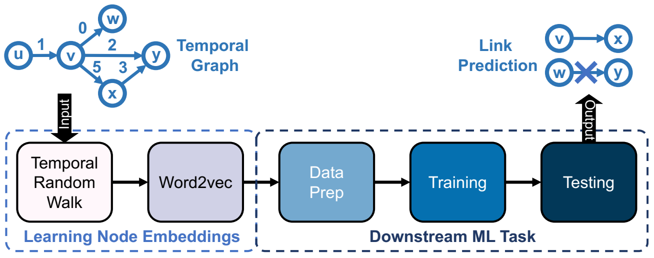

Figure 1: A high-level overview of our modeled pipeline that takes a temporal graph as an input and learns the network dynamics to encode each node into a low-dimension embedding space by using *temporal random walk* and *word2vec*. These embeddings are then fed into a downstream machine learning task such as link prediction or node classification.

The scope of these works, however, has so far been limited to (a) *static* input graphs [22], and (b) a *subset* of graph learning algorithms including Graph Convolution Network (GCN) [7], and a few others [5], [23]. Nonetheless, most real-word graphs are dynamic in nature, *i.e.,* naturally evolving over time by adding, deleting, or changing their nodes and edges. Modeling these dynamic graphs as static would inevitably incur information loss and performance deterioration of downstream predictive tasks. Moreover, while GCN has shown state-of-the-art algorithmic performance on various prediction tasks [22], it mostly works on static graphs and cannot model the graph dynamics such as the sequential interactions between nodes and temporal dependency between graph snapshots. Besides, high computation and memory complexity of GCN makes it difficult to scale to large-scale graphs [22].

In this paper, we investigate the behavior of a fundamentally new class of graph learning algorithms for temporal graphs based on random walks, namely, temporal random walk [6]. Temporal graphs are a category of dynamically evolving networks with timestamp information associated with each network interaction (*i.e.,* temporal edge). Informally, a temporal walk is defined as a sequence of temporally-valid edges $\{(u, v_1, t_1), (v_1, v_2, t_2), \cdots, (v_{i-1}, v_i, t_i)\}$, where $t_{i-1} \leq t_i$. As an example, for the temporal graph shown in Fig. 1, the walk $\{u, v, x\}$ is temporally-valid as it naturally indicates how the node u interacts with its neighbors with respect to time, while $\{u, v, w\}$ is invalid. Temporal random walk is an important algorithm that underlies a wide range of applications on graphs such as information cascading [24], user behavior modeling [25]. It is also the foundation of many follow-up research in the field of machine learning and representation learning [26]–[28]. However, temporal random walk has gained relatively less popularity in the architecture community so far. Additionally, this conceptually straightforward algorithm could effectively model the temporally-valid node interactions while being more scalable [4] to handle large-scale graphs. Furthermore, we show that a workload resulting from temporal random walks exhibits distinct characteristics compared to traditional graph processing and GCN algorithms (see §IV-D).

Fig. 1 shows an overview of a canonical pipeline based on a prior algorithmic work [6]. We model high-performance implementations of two variants of this pipeline for both the CPU and GPU-based computing. The front-end of the pipeline employs temporally-valid random walks and word2vec, a technique from Natural Language Processing (NLP), to map nodes into a low-dimension embedding space. This process translates the similarity between nodes in the original network into closeness in the embedding space. Then, these node embeddings are fed into downstream machine learning tasks. Specifically, this part models the two most widely-known graph learning tasks, used in several applications, as follows.

- **Link prediction.** This task predicts the presence/absence of an edge between a given pair of nodes. A concrete application of this task is product recommendation from the online sales websites such as Amazon.
- **Node classification.** This task assigns labels to nodes. Its concrete application is identifying the professional role of a user in social networks such as LinkedIn.

Based on this pipeline, we perform detailed two-step performance characterization: (a) algorithm-focused, and (b) hardware-focused. This reveals a rich design space and performance acceleration opportunities as listed below.

(a) Accuracy-complexity trade-off. While high prediction accuracy is desirable, it *does not* always come with high cost. We use three hyperparameters to show this: (a) number of random walks per node, (b) random walk length, and (c) embedding space dimensionality. While increasing these values monotonically increases workload memory consumption and execution time, their benefit in accuracy are limited. While prior works [4], [6], [29] often over-provision these values, we find optimal parameters balancing accuracy and complexity.

(b.1) Instruction diversity. By analyzing dynamic instruction types of individual kernels, we find the dominance of both memory and compute instructions, indicating the necessity to optimize both types of operations. This is particularly interesting for temporal random walk that executes more compute operations than traditional graph processing.

(b.2) Thread scalability. Despite irregularity, individual workload kernels can scale well using work stealing.

(b.3) Time Breakdown and CPU versus GPU. Classifier training dominates the execution time of end-to-end workload; accelerating training will yield high workload speedup. A cross-platform workload comparison reveals that the GPU outperforms CPU at large graph sizes.

(b.4) Execution Bottlenecks. GPU workload characterization reveals that individual kernels exhibit diversity of bottlenecks including cache misses, and compute and memory dependency.

Using these insights, we discuss strategies to optimize this workload for future exploration using: algorithm, ML framework, GEMM library, compiler, and hardware.

This is the first work introducing the random walk-based learning pipeline on dynamic graphs for computer architecture research. In summary, we make the following contributions:

- High-performance CPU and GPU implementations of random walk-based temporal graph learning tasks.
- A detailed algorithic workload characterization presenting a rich accuracy-complexity trade-off space.
- An in-depth hardware-focused performance characterization uncovering future optimization opportunities.
- Open-source benchmark implementations and datasets for the benefit of the broader research community at https://github.com/talnish/iiswc21_rwalk.

II. Related Work

A. Graph Representation Learning

Recently, graph representation learning or node embedding has attracted massive research attention from both academia and industry due to its success in downstream tasks like link prediction and node classification. Inspired by the notion of word proximity from NLP, early research in graph learning focused mainly on leveraging the node proximity in a graph, such as DeepWalk [4] and node2vec [29]. These works either leverage first or second-order node proximity [30], or higher-order (> 2) [31] to construct the global node representations. Additionally, there are works based on graph structural properties. For example, struc2vec [14] defines similarity in terms of degree sequences in node-centric subgraphs, and role2vec [32] inductively learns structural similarity by introducing attributed random walk atop relational operators. Furthermore, other works attempt to incorporate external node features with the graph structures [5], [7], [8]. For instance, Graph Neural Network (GNN) [33], [34] and its variants propose to aggregate node features in its dependent contexts with arbitrary depth via propagation/diffusion. Representative works include GCN [7], GraphSAGE [5], and GAT [8].

B. Temporal Network Modeling

Temporal network modeling has been widely studied in dynamic network analysis [35], [36]. Most existing works in the field of machine learning and representation learning empirically process the temporal graph as a sequence of snapshots [28], [37], [38]. While the sequential order of the snapshots models the evolution of temporal dynamics, each individual snapshot is static and analyzed without the temporal information. Streaming graph models can be seen as an extreme case of the snapshot model, where the most recent snapshot is a dynamically changing graph in real time [39], [40]. Another direction that is orthogonal to snapshot-based methods is based on sequential interactions between node pairs in the graph. In this paper, we follow an earlier algorithmic

Symbol	Definition
$G(\mathcal{V}, \mathcal{E})$	a directed temporal network with $\|\mathcal{V}\|$ nodes and $\|\mathcal{E}\|$ edges
$G_t(\mathcal{V}_t, \mathcal{E}_t)$	a snapshot of the temporal network G at time t with $\|\mathcal{V}_t\|$ nodes and $\|\mathcal{E}_t\|$ temporal edges
$\mathbf{A}$, $\mathbf{A}_t$	adjacency matrix for graph G and G_t, respectively
$w_{u,v}$	a temporal walk reaching out from u to v
f	arbitrary base embedding method
d	dimensionality of the embedding
$\mathbf{Z}$	$\|\mathcal{V}\| \times d$ embedding matrix

Table I: Summary of notation.

work CTDNE [6], which proposes the notion of temporal walks and leverages it to learn embeddings directly from the stream of timestamped edges at the finest temporal granularity. Other works [41], [42] propose to model the sequential interaction as the point-process to predict the occurrence of link over time.

C. Software Frameworks

Several software frameworks have been proposed to understand performance implications of different graph learning algorithms [43]–[47]. However, these frameworks mostly model GCN algorithm and a few others [7]. This paper, on the other hand, models random walk-based graph learning. Additionally, there has been tremendous efforts for developing high-performance implementations for traditional bulk-synchronous graph applications on shared memory systems [48]–[60]. These frameworks implement abstractions for programming graph applications as a library of high-level primitives or a new programming language and compilers [61]–[63]. They also combine optimizations with different iteration orders, data structures, direction-optimization [63] *etc.* to improve performance across different graph inputs and applications.

D. Hardware Proposals

Several prior works accelerate similar algorithms using novel hardware designs. In the context of our paper, similar algorithms include graph traversals, traditional deep learning, and graph neural networks. A subset of prior works focus on optimizing graph algorithms on the CPU using techniques such as hardware prefetching [64]–[66]. Other works optimize graph algorithms on GPUs [67]–[69]. Additionally, several accelerators have also been proposed to accelerate graph traversals [70]–[73]. Both traditional deep learning and graph neural networks have been extensively optimized using hardware accelerators [19]–[21], [74]–[77]. However, random walk based graph learning is not well studied in the context of hardware accelerators. In §IV-D, we show that random walk-based graph learning exhibits significantly different nature in terms of its characteristics compared to aforementioned well-studies application domains, motivating the need for our study.

III. Preliminaries

This section provides the definitions of notions used in this paper. The related symbols are listed in Table I.

Definition III.1 (Temporal Graph)**.** A temporal graph G consists of a set of nodes $\mathcal{V}$ and a set of temporal edges $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V} \times \mathbb{R}^+$, where $t \in \mathbb{R}^+$ represents the timestamp of an edge $(u, v, t) \in \mathcal{E}$.

At a high level, a collection of temporal edges $\{(u, v, t)\}$ forms a time-evolving network structure. For example, the time-evolving email exchange network is constituted by individual contacts from user u to v at time t. Comparing with static networks, the edge timestamps endorse in-depth analysis of the network dynamics over time. A fundamental data structure defined in temporal networks is a set of temporal walks, *i.e.,* a sequence of walks with respect to time [6], [25].

Definition III.2 (Temporal Walk)**.** A temporal walk w from u to v in the network $G(\mathcal{V}, \mathcal{E})$ is defined as a sequence of connected edges $w_{u,v} = \{(u, u_1, t_1), (u_1, u_2, t_2), \cdots (u_k, v, t_k)\}$ where $t_i < t_{i+1}$ for $i = 1, 2, \cdots, k$.

A temporal walk indicates the reachability from the source to destination node in a time-increasing order, which encapsulates detailed information about network dynamics as well as node characteristics. In the email exchange network example, temporal walks denote the paths of a user reaching out to another. These walks reflect how people get to know each other and further expand their social networks over time. In this process, detailed user activities such as reply, forward, *etc.* are critical to user profiling and behavioral analysis.

In order to mathematically characterize such node properties in the graph, the notion of graph representation learning has been proposed and widely applied in practice. The high-level idea is to map the nodes from the graph space to a low-dimensional distance space (*e.g.*, 128-d Euclidean space) such that the computational complexity is reduced while the similarity between nodes is preserved. As a result, the low-dimensional representation can be applied to various machine learning tasks such as link prediction, clustering, and node classification. The formal definition of graph representation learning is given as follows.

Definition III.3 (Graph Representation Learning)**.** Given a graph $G(\mathcal{V}, \mathcal{E})$, graph representation learning aims to learn a function $f : G(\mathcal{V}, \mathcal{E}) \to \mathbb{R}^d$ that maps nodes from the graph to a low-dimensional space such that $d \ll |\mathcal{V}|$ and $d \ll |\mathcal{E}|$ while preserving the notion of similarity between nodes.

Depending on specific approaches, the notion of similarity can be defined as the proximity between nodes. Intuitively, a node is more similar to its 1-hop neighbors than its 2-hop neighbors and other distant nodes. Thus, nodes that share common neighbors are embedded closely. On the other hand, node similarity can be measured through the functionality or structural role of a node in terms of its connection to its neighbors. For example, the centers of two star-like subgraphs are structurally similar to each other because they both are at the center and thus behave like "hubs" that bridge other nodes. In this work, we address the first type of node similarity in graphs through temporal proximity.

IV. Background and Motivation

In this section, we discuss the workload that performs link prediction and node classification based on temporal random walk on the graph. Specifically, the workload first generates

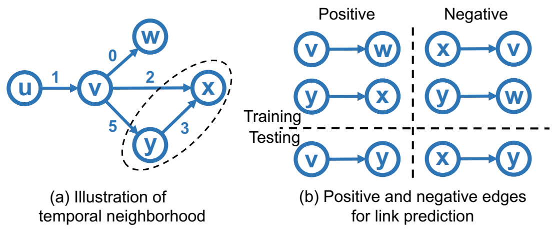
u
v
w
x
y
0
1
2
3
5
(a) Illustration of temporal neighborhood
Positive
Negative
Training
Testing
(b) Positive and negative edges for link prediction

Figure 2: Illustration of temporal the neighborhood and positive/negative edges. At timestamp 1, the random walker reaches node v, then the set of nodes $\{x, y\}$ forms the temporal neighbors of node v.

the temporally-valid walks to characterize the structure of the subgraph centering around each node, and then leverages word2vec to encode it into the low-dimensional Euclidean space as the node embeddings (§IV-A). Then, depending on specific downstream tasks, the workload feeds the derived node embeddings into the neural network architectures, and trains the model to minimize the training loss (§IV-B). Finally, training is performed. After discussing this background, we briefly show how this workload is different from other standard benchmarks (specifically GCN) to motivate our study.

A. Background: Temporal Random Walk based Representation

1) Temporal Random Walk

We follow an earlier algorithmic work, CTDNE [6], to deploy the workload. Specifically, for a node v in graph G, our workload leverages a set of temporally-valid walks originating from v as the characteristic features to derive the embeddings. As mentioned in Definition III.2, the temporally-valid walks reflect the reachability of nodes following the graph structure over time, which further reflects how a node v dynamically interacts with its neighbors in the graph.

We leverage temporal random walk to collect the neighborhood information for each node $v \in G$. In typical temporal random walks, the nodes along the walks are chosen randomly without a specific destination as long as the associated timestamps are increasing. The transitional probability $p(v|u)$ is denoted as $p(v|u) = \frac{1}{|\mathcal{N}_u|}$, where $\mathcal{N}_u$ denotes the set of nodes that are reachable from u following the connected edges. Thus, as long as $\mathcal{N}_u$ for $u \in \mathcal{V}$ is computed efficiently, the temporal walks can be collected efficiently. We detail the implementation of $\mathcal{N}_u$ in §V-A. As an example shown in Fig. 2, the random walker currently reaches node v following the edge with timestamp 1. The next node it reaches would be either node x or y with equal probability 0.5.

While a typical transitional probability marks an efficient way to gather temporal walks, it fails to incorporate the temporal continuity. Again, in the example shown in Fig. 2, an edge from node v to x appears immediately after the edge from the source node u to v. Compared to node y that appears later in time, node x is more correlated with v.

In order to capture this notion of temporal continuity in the graph dynamics, we follow Jin *et al.* [25] to model the transition probability using the softmax function:

$$Pr[v|u] = \frac{\exp\left(-\tau(u,v)/r\right)}{\sum_{i \in \mathcal{N}_u} \exp\left(-\tau(u,i)/r\right)}, \tag{1}$$

where $\tau(u, v)$ denotes the timestamp associated with the edge u, v in the graph, and r is the normalization term that denotes the total range of timespan.

With the transitional probability, our workload performs $|\mathcal{W}|$ walks with lengths L per node, and collect them as the features to describe each individual node in the graph $\mathcal{W}_u = \{v_1, v_2, \cdots, v_L\}$. Next, we describe the derivation of node embeddings based on these walks.

2) Node Embedding

Given the set of temporal walks as features per node, our workload then leverages the skip-gram model [6], [29] to learn the node embeddings, where the objective function is

$$\max_f \log Pr(\mathcal{W}_u | f(u)), \tag{2}$$

where $f(u)$ denotes the embedding for node u to optimize. To solve Equation (2), we assume conditional independence between nodes in $\mathcal{W}_u$, generating a relaxed objective

$$Pr(\mathcal{W}_u | f(u)) = \prod_{v \in \mathcal{W}_u} Pr(v | f(u)), \tag{3}$$

where Pr denotes the softmax function (Eq. (1)). As the output, our workload generates the embedding function $f = G \rightarrow \mathbb{R}^d$ for each node $u \in G$. For our implementation we leverage the word2vec [78] framework.

B. Background: Downstream Tasks

Given the d-dimensional embedding vector per node, our workload leverages the feed forward neural network architecture (FNN) to perform two representative downstream tasks: link prediction and (multi-class) node classification. The parameters of FNN are updated in the training set S^{tr} and tested on the testing set S^{te}. The optimizers used for both tasks are Stochastic Gradient Descent (SGD).

Depending on the tasks, the specific network architecture and loss function adopted in our workload is given as follows.

Link Prediction. The goal of link prediction is to correctly predict the existence of edges that occur later in time based on the initial graph temporal connectivity. Our workload casts link prediction as a classification task, so that the trained FNN can distinguish edges in temporal graph G (positive edges) from the non-existing ones (negative edges). An example is shown in Fig. 2(b), where the goal is to predict the recent edge $e_{(v,y)}$ in the toy graph. Our workload randomly samples two early edges as the positive samples with the same number of negative edges to train the neural network. In the testing stage, the same amount of negative samples are generated as well. The embedding for edge $e_{(u,v)}$ is derived by concatenating the embedding of the source and destination nodes, *i.e.*, $f(e_{(u,v)}) = [f(u), f(v)]$ following [31].

In this task, we deploy the 2-layer FNN, where the output layer generates the probability of classification. We use a binary cross-entropy loss function in the training stage

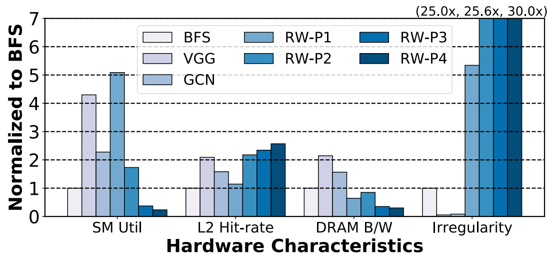

Figure 3: Hardware metric comparison of purely graph traversal (BFS), deep learning inference (VGG), graph convolution network inference (GCN), and modelled pipeline: RW-P1 (random walk), RW-P2 (word2vec), RW-P3 (training), and RW-P4 (testing) The figure showcases unique behavior of modeled application compared to other well-studied benchmarks.

$L = -\sum_{k=1}^{2} p_k \log q_k$, where p_k is the binary target ($\{0, 1\}$) and q_k is the output probability of the neural net, *i.e.*, $q = \text{FNN}_{\text{LP}}(f(e_+, f(e_-)))$.

Node Classification. Multi-class classification is another widely studied task, where the goal is to classify the multi-class labels of nodes in the graph. In our workload, we cast the multi-class classification task by feeding the node-wise embeddings as well as their labels to a 3-layer neural network. The output layer has $|\mathcal{C}|$ neurons, each of which indicates the probability of the input node belonging to the class $c \in \mathcal{C}$. The loss function used is negative log likelihood loss $L = -\log(q_c)$, where q_c is the output probability of a node belonging to the ground-truth class c, *i.e.*, $q = \text{FNN}_{\text{NC}}(f(u), l(u)))$, where $l(u)$ denotes the label for node u.

C. Motivation: GCN versus Random Walk-based Graph Learning

In comparison with GCN that performs spectral convolutional operation over a node's neighbors up to a pre-defined number of hops, temporally-valid random walk captures the sequential interactions with respect to time. As a basic way to explore the spatial property on temporal graphs, the presented algorithm exploits global graph property that is beyond the local node-centric subgraphs. Therefore, it is more powerful in predictive tasks such as link prediction. Furthermore, the presented algorithm works on feature-less graphs and uses a single-integer vertex-identifier as a feature, whereas GCN requires vertex-wise long feature vectors. Interestingly, there is connection between GCN and random walk, for example, [79] shows that random walk can be used to supplement GCN to improve performance on static graphs. However, the difference in these patterns result in different workload characterization and performance optimization strategies on temporal graphs.

D. Motivation: Why Study this Workload?

Fig. 3 compares the hardware characteristics of a traditional graph traversal (BFS), deep learning inference (VGG), graph convolution network inference (GCN), and different workload phases of random walk based graph learning application (RW-P[1:4]) on a GPU. The figure shows GPU core utilization (SM Util), L2 cache hit rate, DRAM bandwidth utilization, load imbalance, and a measure of irregularity (ratio of number of replayed to issued instructions) [80] normalized to BFS. The datasets used for these tasks are the following: BFS—a synthetic graph using `graphgen` utility from Rodinia [81] with 16M nodes and 117M edges, VGG—ImageNet [82], GCN—Reddit [5], and this work—a synthetic Erdős-Renyi graph with 10M nodes and 200M edges.

The figure clearly shows that random walk based graph learning pipeline yields unique characteristics compared to other applications, which warrants its further investigation. Specifically, the amount of irregularity (measured using a ratio of the number of replayed to issued GPU instructions) is high, which can be because of long-latency load instructions and/or load/branch divergence. These characteristics further results in low SM and DRAM bandwidth utilization.

V. Benchmark Implementation

This section presents implementation details of modeled graph learning applications for both CPU and GPU. At a high-level, this follows the flow presented in Fig. 1. We first present the temporal random walk algorithm and a modified version of word2vec that outperforms its open-source counterparts. Then, we briefly discuss the data preparation and classifier steps.

A. Temporal Random Walk

This is the first step of modeled pipeline that takes a temporal graph G as an input, and outputs temporally-valid random walks starting from each node in the graph. We build this kernel by extending a high-performance graph processing framework — the GAP benchmark suite (GAPBS) [83]. We use the weighted graph structure `WGraph` for storing a temporal network, which stores graph edges as an array of structures (*i.e.*, destination and weight). The weight field is repurposed to store timestamps with appropriate changes in the data type. Furthermore, we add support to preserve multiple edges between the same source and destination vertices. This is important to preserve multiple temporally-distant interactions between the same set of nodes.

This algorithm is shown in Algorithm 1. Its time complexity is $\mathcal{O}(KN|\mathcal{V}|M)$, where K is the number of random walks per node, N is the length of each random walk, $|\mathcal{V}|$ is the total number of vertices in the graph, and M is the max degree of all the vertices in the graph. The factor of M comes from the call to the `G.sampleLatent` function (line 12) that iterates through all the neighbors of the vertex and compares each edge against the timestamp. With any value of `currVertex`, this would have to process edges equal to the maximum degree in the graph. There are three nested loops: 1) the outer loop to iterate over the walk number per node when performing multiple random walks per node (line 4); 2) the middle loop to iterate over all the vertices in the graph (line 5); and 3) the inner loop to iterate over an individual step of a walk (line 8). In our implementation, we parallelize the middle loop that iterates over all vertices, based on an empirical finding that it offers optimal performance compared to alternative settings.

B. Word2vec

This algorithm takes a series of temporally-valid random walks as an input and outputs node embeddings. For the CPU,

Algorithm 1 Pseudocode for temporal random walk

```
1: Input: Graph G in CSR format, temporal walk length N, Number of walks per vertex K
2: Output: Temporal walk output matrix of dimensions $|G.\mathcal{V}| \times$ K $\times$ N, W
3: $W \leftarrow new\ matrix[|G.\mathcal{V}|][K][N]$
4: for $w : 0 \rightarrow K$ do
5:     par_for $v : 0 \rightarrow |G.\mathcal{V}|$ do
6:         $currVertex \leftarrow v$
7:         $currTime \leftarrow 0$
8:         for $i : 0 \rightarrow N$ do
9:             if $G.neighbors(currVertex) == 0$ then
10:                break
11:            end if
12:            $currVertex, currTime \leftarrow G.sampleLatent(currVertex, currTime)$
13:            $W[v][w][i] \leftarrow currVertex$
14:        end for
15:    end par_for
16: end for
```

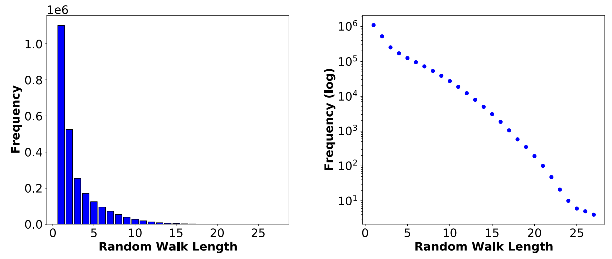

Figure 4: The power-law distribution of temporal random walk lengths on wiki-talk dataset (in linear and log scales). Most walks are of short lengths, and the frequency of longer walk length decreases exponentially. Other datasets also show similar patterns.

we adopt an open-source implementation [84]. However, we find that the available GPU implementations [85], [86] have sub-optimal performance when applied to the graph learning problem. This is because of their parallelism model. These implementations parallelize word embedding updates *within* each sentence, and processes different sentences sequentially. While this might be optimal in NLP with long sentences, it leads to poor parallelism in the graph learning context. This is because, as shown in Fig. 4, the random walk lengths (*i.e.,* the number of walks that complete for a given length given the timestamp constraints) are centered around 1 to 5. As the walk length is analogous to sentence length, the word2vec input constitutes a large number of *short* sentences. This causes the GPU resources to be under-utilized and launches a large number of GPU kernels, one launch for each sentence.

To improve this implementation, we propose the following optimizations. First, we batch multiple sentences together, and process sentences within a batch in parallel. This adds a new possibility to read from a stale word embedding model, potentially reducing accuracy, as we process multiple word embedding updates concurrently. However, because the model update is a sparse operation [4], concurrently updating word embedding model does not result in an accuracy loss. On the flip side, this technique greatly improves the GPU core utilization. Empirically, Fig. 5 shows that the batch size of 16k achieves a *124.2× speedup* over no batching *without accuracy loss*. The speedup is attributed to (a) improved GPU core utilization, (b) CPU-GPU data transfer cost amortization over long computation, and (c) reduced kernel launch overhead.

Second, a prior implementation [86] uses cache line padding to address false sharing at the private L1 caches. This heavily under-utilizes cache lines as our embedding space dimension is small (*i.e.,* 8 as shown in §VII-A). To optimize cache line utilization, we remove the cache line padding (No-pad) and add support to bypass the L1 cache. Third, we assign multiple GPU threads to process each embedding dimension in a coalesced manner (Coalesced), and use parallel reduction for accumulation (Par-red). With a small embedding dimension, we also eliminate all the `__syncthreads()`, and rely on the in-warp synchronization. Fig. 6 shows the benefit of each of these optimizations, leading up to an end-to-end speedup of 220.5× on the wiki-talk dataset without accuracy loss.

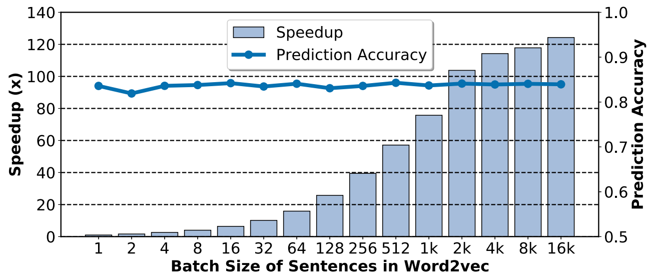

Figure 5: Sensitivity of word2vec phase speedup and end-to-end link prediction accuracy for different batched sentence sizes on a GPU using wiki-talk dataset. Compared to a baseline open-source implementation [85], [86], our batch implementation gains 124.2× speedup without a loss in accuracy at a batch size of 16k sentences.

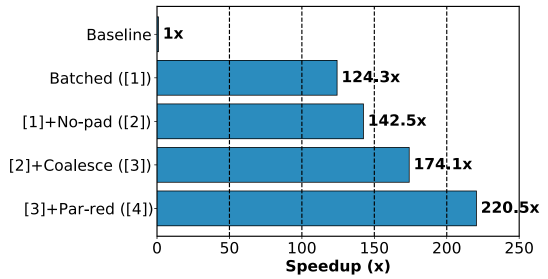

Figure 6: Speedup of the word2vec phase on a GPU for different optimizations. Compared to baseline, batched sentences (Batched), no cache line padding (No-pad), memory operation coalescing (Coalesce), and parallel reduction (Par-red) result in an end-to-end speedup of 220.5× on wiki-talk dataset.

C. Data Preparation

Inputs to this step include the node embeddings from word2vec, and a temporal edge list/a labeled node list for link prediction/node classification. This step outputs datasets for training (S^{tr}), validation (S^{vd}), and testing (S^{te}).

Fig. 7 shows the data preparation algorithm for link prediction. First, the input edges are sorted by their timestamps (❶) and then 20% of the edges are chosen for testing from the end of this list. The intuition behind sorting the edges is

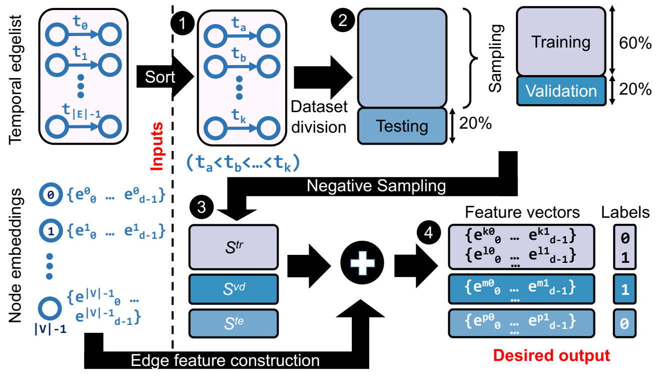
Temporal edgelist
t_0
t_1
$t_{|E|-1}$
Sort
Inputs
1
t_a
t_b
t_k
$(t_a<t_b<\ldots<t_k)$
2
Dataset division
Testing
20%
Sampling
Training
60%
Validation
20%
Negative Sampling
Node embeddings
$\{e^0_0 \ldots e^0_{d-1}\}$
$\{e^1_0 \ldots e^1_{d-1}\}$
$\{e^{|V|-1}_0 \ldots e^{|V|-1}_{d-1}\}$
$|V|-1$
3
S^{tr}
S^{vd}
S^{te}
4
Feature vectors
Labels
$\{e^{k0}_0 \ldots e^{k1}_{d-1}\}$
$\{e^{l0}_0 \ldots e^{l1}_{d-1}\}$
$\{e^{m0}_0 \ldots e^{m1}_{d-1}\}$
$\{e^{p0}_0 \ldots e^{p1}_{d-1}\}$
0
1
1
0
Edge feature construction
Desired output

Figure 7: Data preparation step for link prediction.

to train the classifier on the past edges and test it on the future edges. Excluding the testing edges, 60% and 20% of the total edges are randomly sampled for training and validation (❷), respectively. Because these edges exist in the original input network, they form *positive edge sets* with a label 1. Negative sampling (❸) is used to construct negative edges with a label 0. This is done by altering one/both vertex IDs of positive edges so that the resulting edge is absent in the input graph. After constructing these sets, edge features are computed by concatenating node embeddings as described in §IV-B (❹). A similar mechanism is employed for node classification, where labeled dataset precludes the need for negative sampling.

D. Classifier

Data obtained in the previous stage is fed into the classifier, which goes through training and testing phases. We use an FNN-based classifier as discussed in §IV-B.

VI. Experimental Methodology

This section details our experimental methodology. Specifically, we talk about our hardware platforms, software toolchain, and input graph datasets used for evaluation.

A. Hardware Platforms

We characterize modeled applications on two platforms — CPU and GPU. We use a dual-socket server with two AMD EPYC^{TM} 7742 CPUs with 128 physical cores (256 SMT threads). The aggregate Last Level Cache (LLC) size is 2×256MB. The size of main memory is 512GB. Additionally, we use a discrete NVIDIA GPU with Ampere architecture.

B. Software Toolchain

We model our applications in `C++` and compile them using the `g++ v7.5` compiler with `-O3` optimization level for the CPU. We compile CUDA programs using `nvcc v11.2` with `-O3` and `-arch=sm_80` flags. For hardware profiling, we use manual instrumentation and MICA Pintool [87] for the CPU, and NVIDIA Nsight Compute [88] for the GPU. We use dynamically scheduled OpenMP threads for CPU parallelism. The downstream ML task is implemented using the `PyTorch-C++` API [89].

C. Input Datasets

We use both real-world and synthetic graphs for evaluation. Because the publicly available real-world temporal datasets are limited in size, we use them for algorithmic evaluation. Table II shows the list of these datasets and their properties. For hardware study, we use large-scale synthetic graph datasets generated using Python-based `networkx` library. Specifically, we generate Erdős-Renyi random graphs, with varying sizes and degrees, with synthetic timestamps.

VII. Results and Analysis

Presented analysis is divided into two parts: (a) algorithm-focused study, and (b) hardware-focused study. The former presents the trade-off between prediction accuracy and runtime performance. The latter focuses on understanding the workload characteristics to find performance optimization opportunities.

A. Algorithmic Analysis

We study the effect of three important algorithmic parameters: number of random walks per node, walk length, and embedding space dimension. As shown in §V-A, runtime complexity of the random walk algorithm is proportional to the number of random walks per node and walk length. Additionally, the runtime complexities of word2vec and classifier training/testing are dependent on the embedding space dimension as it decides the feature vector length. Therefore, increasing these parameter values will increase the execution times of different kernels. Fig. 8a empirically confirms this finding by showing the increase in random walk execution time when increasing in the number of walks per node for the stackoverflow dataset. A similar trend is observed for random walk length and embedding space dimension. In general, we find that the performance on link prediction tasks is better than node classification. This is because that temporal random walk exploits global graph property that is beyond the local node-centric subgraphs. As the task of node classification requires detailed information centric to specific nodes, temporal random walk is not the optimal algorithm for it. Thus, as link prediction requires more global information about the graph connectivity, the performance is better. Next, we present the parameter sensitivity on prediction accuracy of downstream tasks.

Number of Random Walks Per Node. As shown in prior works [4], [6], [29], the network is best sampled by performing multiple random walks from the same node. This is because one walk can only sample a vertex neighborhood via one of its neighbors. Performing multiple walks from a node can potentially sample a *wider* vertex neighborhood, enriching the amount of information used for downstream learning tasks. Fig. 8b shows the effect of performing multiple random walks from a node on the prediction accuracy of link prediction and node classification. The figure confirms that more walks from the same node increases the prediction accuracy. Interestingly, this improvement saturates after 8-10 walks. This is because of the power-law nature of real-world graphs, *i.e.,* most nodes have few neighbors. In a majority of sparsely connected nodes, performing 8-10 walks are enough to cover most neighbors. Beyond this, there is limited value by performing more walks.

Random Walk Length. Length of the random walks indicates the distance of sampled neighbor from the source.

Task	Dataset Name	#Nodes	#Temporal Edges	Description
Link prediction	ia-email [90], [91]	87,274	1,148,072	Enron email network from Jan. 1998 until Feb. 2004
Link prediction	wiki-talk [92]–[94]	1,140,149	7,833,140	User editing network of Wikipedia Talk pages
Link prediction	stackoverflow [92], [93]	6,024,271	63,497,050	Stack exchange interaction network on Stack Overflow
Node classification	dblp5 [95]	6,606	42,815	Co-author network from DBLP from 5 research areas
Node classification	dblp3 [95]	4,257	23,540	Co-author network from DBLP from 3 research areas
Node classification	brain [95], [96]	5,000	1,955,488	Connectivity network of tidy cubes of brain tissues

Table II: Real-world temporal networks used for algorithmic evaluation.

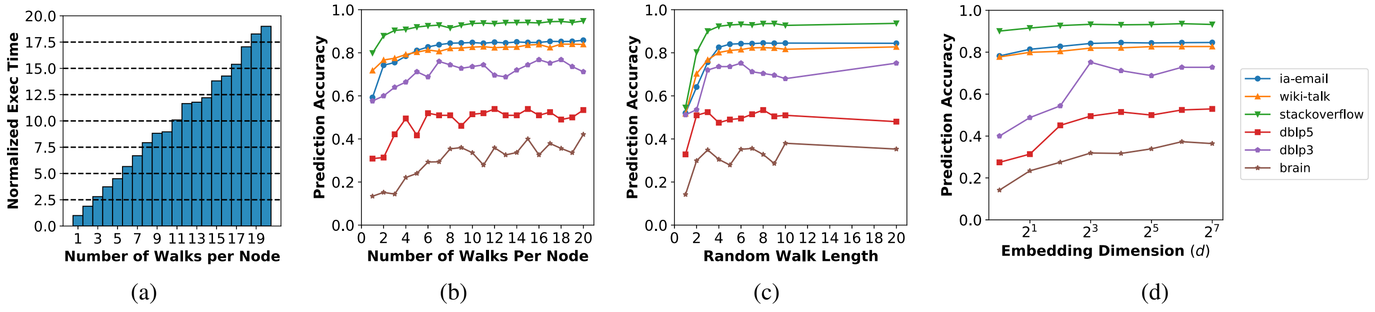

Figure 8: Accuracy-complexity trade-off. (a) Normalized execution time of the random walk kernel for different number of walks per node, and (b-d) Accuracy of link prediction and node classification with respect to different parameter values.

For example, a random walk of length 5 will sample a 5-hop neighbor from a source vertex. While multiple random walks per node sample *wide* neighborhoods, larger random walk length indicates the sampled neighborhood *depth*. Intuitively, larger the length of random walk, deeper the network can be sampled. Fig. 8c shows an increase in prediction accuracy with an increase in the random walk length. This trend, however, saturates after a walk length of 4-6, which can be described using an earlier finding. Fig. 4 shows that the frequency of random walks decreases with increased walk length. This translates into marginal information gain with large walk lengths and saturation in prediction accuracy.

Embedding Space Dimension. At a high level, a graph learning task maps each node to an embedding space, where the dimension of the embedding space defines complexity of interactions that can be modeled. While prior algorithmic works [4], [6], [29] use a fixed dimension size (d) of 128, we analyze how this affects end-to-end accuracy. Fig. 8d shows the effect of changing d on the prediction accuracy. Increasing d from 1 to 8 results in gain in prediction accuracy as higher dimensions can model more complex network interactions. Interestingly, we find that an embedding space of dimension 8 is enough to make meaningful network predictions.

To summarize, there exists a rich trade-off space between algorithmic performance and runtime complexity. *While increasing the value of aforementioned hyperparameters will monotonically increase the execution time of different kernels, their effect on prediction accuracy is limited.* Based on our empirical findings, we find the optimal values of number of random walks per node, random walk length, and embedding space dimension to be 10, 6, and 8, respectively.

B. Hardware Analysis

Next, we perform a detailed hardware analysis based on the optimal parameter values found above. Using real-world and synthetic graph datasets, we study the instruction diversity, scalability, time breakdown, and execution bottlenecks.

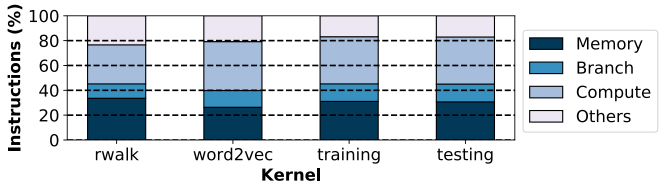

Figure 9: Dynamic instruction breakdown of different kernels involved in link prediction for ia-email dataset. The figure shows that all kernels have a high number of both compute and memory instructions.

Instruction Diversity. Instruction diversity characterization helps understanding the operation types present in a workload, which can be used to make design decisions building specialized hardware. Fig. 9 shows the breakdown of dynamic instruction types of individual kernels on a CPU for the link prediction task on ia-email dataset. This is divided in terms of memory, branch, compute (both arithmetic and floating point), and others. The others category includes instructions for stack usage, bitwise shifts, string operations, SIMD, *etc.*

The figure shows that both compute (36.6% on average) and memory (30.4% on average) operations are dominant in all kernels. Word2vec and classifier training/testing phases use neural network-type computation, hence, this breakdown is not surprising. However, a similar count of compute and memory instructions for random walk is surprising as graph traversals are known to have a low memory-to-compute operation ratio. This distribution is attributed to the compute-intensive operations used in selecting a neighbor to walk as shown in Eq. (1). As a takeaway, system designers should target both compute and memory operations for optimizing all workload kernels.

Scaling Analysis. Fig. 10 shows the thread scaling behavior of temporal random walk and word2vec kernels for stackoverflow. Additionally, it shows GPU performance normalized to a

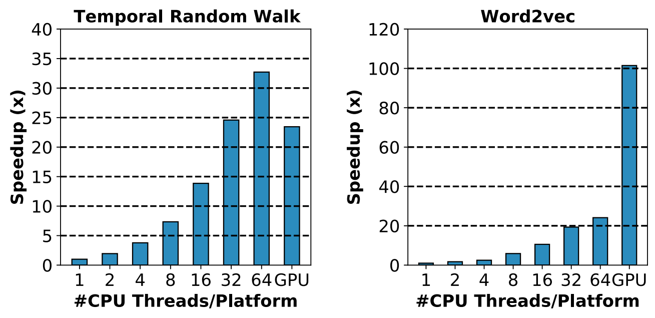

Figure 10: CPU thread scaling analysis and its comparison with GPU implementation for temporal random walk and word2vec kernels on the stackoverflow dataset. The speedups are normalized to a single-thread implementation. The figure shows reasonable scaling trend.

	rwalk		word2vec		training/epoch		testing	
$\vert\mathcal{V}\vert$,$\vert\mathcal{E}\vert$	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU
10k,50k	0.0	1.2	0.3	0.4	0.3	0.7	0.1	0.0
10k,100k	0.0	1.2	0.3	0.3	0.6	0.8	0.4	0.1
10k,200k	0.0	1.2	0.3	0.4	0.8	1.2	0.2	0.2
100k,500k	0.1	1.2	0.7	0.4	2.4	2.3	0.5	0.5
100k,1M	0.1	1.2	0.6	0.4	4.1	3.4	1.2	0.8
100k,2M	0.1	1.1	0.6	0.4	5.1	8.0	2.2	1.6
1M,5M	0.9	1.4	2.7	1.3	13.7	15.5	6.0	4.2
1M,10M	1.2	1.4	3.4	1.4	32.5	28.3	7.8	7.0
1M,20M	1.8	1.4	3.2	1.6	62.2	58.6	20.7	14.7
10M,50M	12.2	4.0	25.4	20.0	147.7	147.1	56.3	44.2
10M,100M	14.2	4.0	27.3	22.1	315.8	303.9	133.0	87.8
10M,200M	18.7	4.2	36.8	27.4	695.2	668.5	233.1	206.9

Table III: Execution times of workload phases in seconds for both CPU and GPU implementations. Cell colored in green indicates a faster implementation between CPU and GPU.

serial CPU implementation. Using more than 64 threads does not improve performance further as the thread creation/logic logic dominates the computation cost. We do not show the scaling of classifier training/testing as its Pytorch-based implementation does not offer an explicit thread-scaling control[2].

The figure shows that both kernels show a reasonable thread scaling trend despite irregularity. For the random walk kernel, the amount of work per thread is dependent on the outgoing degree and timestamp distribution, which leads to heavy load imbalance in a naïve implementation. To alleviate this problem, we employ work stealing using dynamically scheduled OpenMP threads. The GPU performs similar to 32 CPU threads. This is because of the CPU-GPU data transfer time, and workload irregularity leading to branch divergence and non-coalesced memory accesses. On the other hand, the GPU implementation of word2vec performs much better than CPU, despite the data transfer cost and irregularity. This is because of the proposed optimizations discussed in §V-B.

Execution Time Breakdown. Using synthetic Erdős-Renyi graphs of varying sizes and degrees, Table III shows the execution time breakdown of end-to-end workload. The training and testing times are reported for link prediction classifier. A similar trend follows for node classification. Note that Table III shows per-epoch training time; the actual number of training

[2]PyTorch API uses workers for parallel data-loading, which spawns multiple processes replicating the memory space.

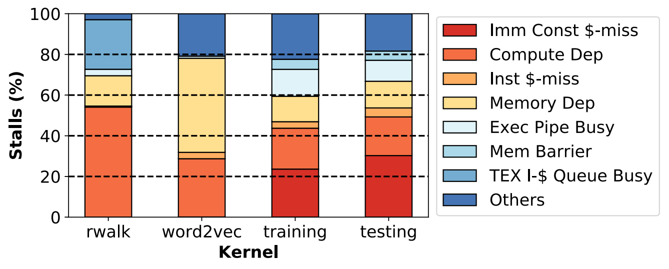

Figure 11: Characterization of stalls in different kernels on a GPU. There is a diversity of stalls observed across kernels; most stalls are caused by immediate constant cache (IMC) misses, and compute and memory dependencies.

epochs is dependent on other hyperparameter values (*e.g.,* batch size, learning rate, and rate decay).

There are two main insights here. *First, the training time dominate an end-to-end execution time of the workload.* The motivation of examining end-to-end workload time breakdown is that in a real-world deployment, the graph evolves over time. With this evolution, an entire pipeline needs to run to account for new nodes/connections. This study shows that optimizing classifier training would yield maximum benefits in reducing the end-to-end workload time. *Second, the execution times of classifier training/testing increase monotonically with the graph size.* To understand this performance further, we compare the *testing time per instruction* for modeled pipeline and VGG. This comparison finds that per-instruction execution time of random walk-based training is *37.4× slower* than VGG. We believe this is because of discrepancy in the matrix sizes. For example, the largest layer size in VGG is *3136×* larger than the largest layer in the studied pipeline limiting its potential for parallelism. Both applications are modeled using PyTorch, which internally calls GEMM kernels. While the performance of GEMM kernels are highly optimized for popular network sizes (*e.g.,* VGG), our study shows that there is a significant room for improvement for other network sizes.

Cross-platform Performance Comparison. Table III also compares the CPU and GPU performance. *GPU implementations outperform its CPU counterpart at large graph sizes.* This is not surprising because CPU-GPU data transfer time dominates computation time with small graphs. With large graph sizes, this time is amortized over longer, more efficient GPU computation, making it faster than the CPU. Additionally, the workload irregularity hurts GPU performance causing divergent thread pools and non-coalesced load operations.

Execution Bottlenecks. Finally, we perform a detailed microarchitectural analysis to characterize stall cycles of different kernels. We perform this analysis on a large synthetic graph with 10M nodes and 200M edges. We use the GPU for this analysis because of its superior performance. Fig. 11 shows the characterization of stalls in terms of (from top to bottom on the legend): 1) immediate constant cache (IMC) misses, 2) compute dependencies (unresolved register dependencies because of long fixed-latency compute instructions), 3) instruction cache misses, 4) scoreboard dependencies on

L1TEX operation, 5) execution pipe and MIO (memory I/O) instruction queue busy, 6) memory/CTA (cooperative thread array) barrier, 7) L1TEX instruction queue busy, and 8) others.

We observe two primary insights. *First, each kernel exhibits unique hardware characteristics and stall cycles.* For example, major causes of stalls in the random walk, word2vec, and classifier training/testing are compute dependencies (*i.e.,* 54.1%), memory dependencies (*i.e.,* 46.2%), and IMC cache misses (*i.e.,* 23.6/30.6%), respectively. As a result, no one optimization strategy can significantly speed up all workload phases, and kernel-wise investigation is necessary.

Second, on average, 65.5% of stall cycles across kernels are caused by IMC cache misses, and memory and compute dependencies. For the *random walk kernel*, the TEX I-cache queuing delay and compute dependencies cause the majority of the stall cycles. TEX I-cache stall is caused by the frequent control flow divergence as a result of the workload imbalance in sampling vertex neighborhoods. This sampling involves several long fixed-latency compute instructions (see Eq. (1)), causing compute dependencies. The memory dependency stall is relatively low because a large portion of the work performed for a single vertex exhibits spatial locality. The *word2vec kernel* is mostly bounded by a significant portion of memory dependencies. This is because this kernel fetches and updates the model weights by sliding through a vertex window. The vertex window being updated is dependent on the random walk result, which contains a random set of vertex IDs, generating irregular memory accesses. The *training and testing phases* show a similar stall distribution, which is attributed to small dimensions of our kernels [97], launching a small number of warps. This is further corroborated by the SM utilization for training/testing classifier being less than 10%. Therefore, loading immediate data has low reuse, causing high stall rates.

VIII. Discussion

This section discusses the employment of this framework to conduct optimization studies and incorporate new tasks.

A. Optimization Opportunities

For algorithm designers. In this work, we leverage the forward neural network for learning (§IV-B) as a basic model for the workload analysis. It can be easily replaced by more advanced neural network architectures such as ResNet [98] or DenseNet [99]. Empirically, we observe at least $\sim$ 2% accuracy improvement for link prediction using ResNet, and we leave the detailed investigation for future work.

For PyTorch framework designers. As briefly discussed in §VII-B, the PyTorch framework uses multi-processing to employ multiple data loading workers. This significantly increases the memory consumption of the workload and hurts scalability. Multi-threading support with optimized memory usage will significantly improve the classifier performance.

For GEMM library designers. As shown in §VII-B, training time per instruction of the modeled pipeline is 37.4× slower than VGG. This is owing to the differences in matrix sizes, and low-level demand-based math library optimization model. Optimizing the GEMM kernel performance for matrix sizes used in our pipeline can improve the performance of classifier training/testing by one-to-two orders of magnitude.

```
1.  #include </* std header files */>
2.  #include <rwalk.h>
3.  #include <word2vec.h>
4.  #include <data_preproc.h>
5.  #include <model.h>
6.  #include <classifier.h>
7.
8.  int main( args ) {
9.    // Call graph reading API
10.   compute_rwalk( ... );
11.   word2vec( ... );
12.   data_preproc( ... ); // Implement data_preproc.h
13.   model_train( ... );  // Modify model.h, classifier.h
14.   model_test( ... );   // Modify model.h, classifier.h
15.   // Memory cleanup
16.   return 0;
17. }
```

Figure 12: Sample source code for incorporating new tasks.

For compiler and hardware designers. Based on the execution stall characterization shown in §VII-B, compiler optimization techniques such as operator fusion, loop interchange, and data structure changes can alleviate kernel launch and data transfer overheads. Additionally, compiler-based blocking, graph partitioning, and tiling [100] can improve memory performance. Furthermore, employing domain-specific hardware acceleration can significantly optimize this workload. The word2vec and classifier phases are similar to traditional deep-learning pipelines, hence, mapping them to an already existing accelerator [74] would be sufficient. However, the random walk kernel exhibits significantly different characteristics and bottlenecks than traditional graph traversals (*i.e.,* presence of complicated compute primitives as shown in Eq. 1). This calls for exploring a novel accelerator design for the random walk kernel. This design must focus on optimizing both the compute pipeline for long-latency arithmetic and floating point operations, and the memory system to speed up data-dependent loads for traversing sparse graph data structures (*e.g.,* [64]).

B. Incorporating New Tasks

While this work presents two important graph learning tasks used in several application domains, our framework can be easily extended to realize other tasks. For example, if a user wants to implement link property prediction (*i.e.,* predicting edge labels), Fig. 12 shows the modification of main source file that calls different pipeline stages. A user can re-purpose random walk and word2vec implementations by simply calling functions shown in lines 11 and 12. As the step of preparing classifier data is unique to each task, a user has to implement an appropriate data preparation step. Finally, a classifier containing neural network model, training, and testing loops can be incorporated by modifying already implemented modules in our framework.

IX. Conclusion

This paper presented high-performance implementations of two important graph learning tasks on continuous-time dynamic networks, optimized individually to run both on the CPU and GPU. We used a scalable *random walk*-based algorithm for learning node embeddings of a graph. Based on these implementations, we conducted an in-depth performance analysis from both algorithmic and hardware fronts. The algorithm-focused study presented a rich trade-off space between prediction accuracy and runtime complexity. The hardware-focused investigation analyzed different phases of the application to find their instruction type diversity, thread scalability, execution time breakdown, and execution bottlenecks. Based on these insights, we made recommendations to further optimize the workload performance for designers of algorithms, ML frameworks, GEMM library, compiler, and hardware. The proposed implementations will be open-sourced to the broader research community to encourage further investigation.

Acknowledgements

We thank the anonymous reviewers for their insightful feedback. The material is based on research sponsored by Air Force Research Laboratory (AFRL), Defense Advanced Research Projects Agency (DARPA), the National Science Foundation (NSF), and Army Research Office under agreement numbers FA8650-18-2-7864, HR0011-18-3-0007, HR0011-20-9-0017, CAREER Grant No. IIS 1845491, and Army Young Investigator Award No. W911NF1810397. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) or the U.S. Government. AMD, EPYC and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

References

[1] L. Backstrom and J. Leskovec, “Supervised random walks: predicting and recommending links in social networks,” in *Proceedings of the fourth ACM international conference on Web search and data mining*, 2011, pp. 635–644.

[2] H. Jeong *et al.*, “Lethality and centrality in protein networks,” *Nature*, vol. 411, no. 6833, pp. 41–42, 2001.

[3] S. Maslov and K. Sneppen, “Specificity and stability in topology of protein networks,” *Science*, vol. 296, no. 5569, pp. 910–913, 2002.

[4] B. Perozzi *et al.*, “Deepwalk: Online learning of social representations,” in *ACM SIGKDD*, 2014.

[5] W. Hamilton *et al.*, “Inductive representation learning on large graphs,” in *Advances in neural information processing systems*, 2017, pp. 1024–1034.

[6] G. H. Nguyen *et al.*, “Continuous-time dynamic network embeddings,” in *Companion Proceedings of the The Web Conference 2018*, 2018, pp. 969–976.

[7] T. N. Kipf and M. Welling, “Semi-supervised classification with graph convolutional networks,” *arXiv preprint arXiv:1609.02907*, 2016.

[8] P. Veličković *et al.*, “Graph attention networks,” *arXiv preprint arXiv:1710.10903*, 2017.

[9] F. Monti *et al.*, “Fake news detection on social media using geometric deep learning,” *arXiv preprint arXiv:1902.06673*, 2019.

[10] R. Ying *et al.*, “Graph convolutional neural networks for web-scale recommender systems,” in *ACM SIGKDD*, 2018.

[11] O. Litany *et al.*, “Deep functional maps: Structured prediction for dense shape correspondence,” *ICCV*, 2017.

[12] N. Choma *et al.*, “Graph neural networks for icecube signal classification,” in *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 2018, pp. 386–391.

[13] D. K. Duvenaud *et al.*, “Convolutional networks on graphs for learning molecular fingerprints,” in *Advances in neural information processing systems*, 2015, pp. 2224–2232.

[14] L. F. Ribeiro *et al.*, “struc2vec: Learning node representations from structural identity,” in *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, 2017, pp. 385–394.

[15] J. M. Stokes *et al.*, “A deep learning approach to antibiotic discovery,” *Cell*, 2020.

[16] M. Yan *et al.*, “characterizing and understanding gcns on gpu,” *IEEE Computer Architecture Letters*.

[17] Z. Zhang *et al.*, “Architectural implications of graph neural networks,” *IEEE Computer Architecture Letters*, vol. 19, no. 1, pp. 59–62, 2020.

[18] T. Baruah *et al.*, “GNNMark: A Benchmark Suite to Characterize Graph Neural Network Training on GPUs,” in *ISPASS)*, 2021.

[19] M. Yan *et al.*, “HyGCN: A GCN Accelerator with Hybrid Architecture,” in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020.

[20] T. Geng *et al.*, “AWB-GCN: A Graph Convolutional Network Accelerator with Runtime Workload Rebalancing,” in *MICRO*, 2020.

[21] J. Li *et al.*, “GCNAX: A flexible and energy-efficient accelerator for graph convolutional neural networks,” in *HPCA*, 2021.

[22] W. Hu *et al.*, “Open Graph Benchmark: Datasets for Machine Learning on Graphs,” *arXiv preprint arXiv:2005.00687*, 2020.

[23] K. Xu *et al.*, “How powerful are graph neural networks?” in *International Conference on Learning Representations*, 2019.

[24] J. Leskovec *et al.*, “Cascading behavior in large blog graphs: Patterns and a model,” *Society of Applied and Industrial Mathematics: Data Mining*, 2007.

[25] D. Jin *et al.*, “node2bits: Compact time-and attribute-aware node representations for user stitching,” in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, 2019.

[26] A. Pareja *et al.*, “Evolvegcn: Evolving graph convolutional networks for dynamic graphs,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 04, 2020, pp. 5363–5370.

[27] R. Trivedi *et al.*, “Dyrep: Learning representations over dynamic graphs,” in *International conference on learning representations*, 2019.

[28] A. Sankar *et al.*, “Dysat: Deep neural representation learning on dynamic graphs via self-attention networks,” in *Proceedings of the 13th International Conference on Web Search and Data Mining*, 2020.

[29] A. Grover and J. Leskovec, “Node2vec: Scalable feature learning for networks,” in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD ’16. New York, NY, USA: Association for Computing Machinery, 2016, p. 855–864. [Online]. Available: https://doi.org/10.1145/2939672.2939754

[30] J. Tang *et al.*, “Line: Large-scale information network embedding,” in *Proceedings of the 24th international conference on world wide web*, 2015, pp. 1067–1077.

[31] S. Cao *et al.*, “Grarep: Learning graph representations with global structural information,” in *Proceedings of the 24th ACM international on conference on information and knowledge management*, 2015, pp. 891–900.

[32] N. K. Ahmed *et al.*, “role2vec: Role-based network embeddings,” in *Proc. DLG KDD*, 2019.

[33] F. Scarselli *et al.*, “The graph neural network model,” *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 61–80, 2008.

[34] J. Zhou *et al.*, “Graph neural networks: A review of methods and applications,” *arXiv preprint arXiv:1812.08434*, 2018.

[35] P. Holme and J. Saramäki, “Temporal networks,” *Physics reports*, vol. 519, no. 3, pp. 97–125, 2012.

[36] C. Aggarwal and K. Subbian, “Evolutionary network analysis: A survey,” *CSUR*, 2014.

[37] P. Goyal *et al.*, “dyngraph2vec: Capturing network dynamics using dynamic graph representation learning,” *Knowledge-Based Systems*, vol. 187, p. 104816, 2020.

[38] J. Li *et al.*, “Attributed network embedding for learning in a dynamic environment,” in *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, 2017, pp. 387–396.

[39] C. C. Aggarwal *et al.*, “On clustering graph streams,” in *Proceedings of the 2010 SIAM International Conference on Data Mining*. SIAM, 2010, pp. 478–489.

[40] D. Le-Phuoc *et al.*, “Linked stream data processing engines: Facts and figures,” in *International Semantic Web Conference*. Springer, 2012.

[41] Y. Zuo *et al.*, "Embedding temporal network via neighborhood formation," in *ACM SIGKDD*, 2018.

[42] S. Kumar *et al.*, "Predicting dynamic embedding trajectory in temporal interaction networks," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019, pp. 1269–1278.

[43] M. Fey and J. E. Lenssen, "Fast graph representation learning with pytorch geometric," *arXiv preprint arXiv:1903.02428*, 2019.

[44] M. Wang *et al.*, "Deep graph library: Towards efficient and scalable deep learning on graphs," *arXiv preprint arXiv:1909.01315*, 2019.

[45] D. Zhang *et al.*, "AGL: a Scalable System for Industrial-purpose Graph Machine Learning," 2020.

[46] R. Zhu *et al.*, "Aligraph: A comprehensive graph neural network platform," *arXiv preprint arXiv:1902.08730*, 2019.

[47] L. Ma *et al.*, "Neugraph: parallel deep neural network computation on large graphs," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, pp. 443–458.

[48] J. Sun *et al.*, "GraphGrind: Addressing Load Imbalance of Graph Partitioning," in *Proceedings of the International Conference on Supercomputing*, ser. ICS '17, 2017, pp. 16:1–16:10.

[49] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," in *PPoPP*, 2013, pp. 135–146.

[50] J. Shun *et al.*, "Smaller and faster: Parallel processing of compressed graphs with ligra+," in *IEEE DCC*, 2015, pp. 403–412.

[51] K. Zhang *et al.*, "NUMA-aware graph-structured analytics," in *PPoPP*, 2015, pp. 183–193.

[52] S. Grossman *et al.*, "Making pull-based graph processing performant," in *PPoPP*, 2018, pp. 246–260.

[53] N. Sundaram *et al.*, "GraphMat: High performance graph analytics made productive," *Proc. VLDB Endow.*, 2015.

[54] Z. Peng *et al.*, "Graphphi: efficient parallel graph processing on emerging throughput-oriented architectures," in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2018.

[55] S. Hong *et al.*, "Green-Marl: A DSL for Easy and Efficient Graph Analysis," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012, pp. 349–362.

[56] M. S. Lam *et al.*, "Socialite: Datalog extensions for efficient social network analysis," in *IEEE International Conference on Data Engineering (ICDE)*, 2013, pp. 278–289.

[57] G. Gill *et al.*, "Abelian: A compiler for graph analytics on distributed, heterogeneous platforms," in *Euro-Par*, 2018, pp. 249–264.

[58] C. R. Aberger *et al.*, "Emptyheaded: A relational engine for graph processing," vol. 42, no. 4, Oct. 2017, pp. 20:1–20:44.

[59] K. Vora *et al.*, "Kickstarter: Fast and accurate computations on streaming graphs via trimmed approximations," in *ASPLOS*, 2017, pp. 237–251.

[60] Y. Zhang *et al.*, "Making caches work for graph analytics," in *2017 IEEE International Conference on Big Data (Big Data)*, 2017, pp. 293–302.

[61] S. Pai and K. Pingali, "A compiler for throughput optimization of graph algorithms on gpus," in *OOPSLA*, 2016, pp. 1–19.

[62] Y. Zhang *et al.*, "Graphit: A high-performance graph dsl," *OOPSLA*, vol. 2, pp. 121:1–121:30, Oct. 2018.

[63] ——, "Optimizing ordered graph algorithms with graphit," in *CGO*, 2020.

[64] N. Talati *et al.*, "Prodigy: Improving the memory latency of data-indirect irregular workloads using hardware-software co-design," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 654–667.

[65] S. Ainsworth and T. M. Jones, "Graph prefetching using data structure knowledge," in *ICS*, New York, NY, USA, 2016, pp. 39:1–39:11.

[66] A. Basak *et al.*, "Analysis and optimization of the memory hierarchy for graph processing workloads," in *HPCA*, Feb 2019, pp. 373–386.

[67] A. Segura *et al.*, "SCU: A GPU Stream Compaction Unit for Graph Processing," in *ISCA*, 2019, pp. 424–435.

[68] F. Khorasani *et al.*, "Scalable simd-efficient graph processing on gpus," in *PACT*. IEEE, 2015, pp. 39–50.

[69] S. Hong *et al.*, "Accelerating cuda graph algorithms at maximum warp," *Acm Sigplan Notices*, vol. 46, no. 8, pp. 267–276, 2011.

[70] T. J. Ham *et al.*, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *MICRO*, 2016, pp. 1–13.

[71] M. M. Ozdal *et al.*, "Energy efficient architecture for graph analytics accelerators," in *ISCA*, 2016, pp. 166–177.

[72] M. Yan *et al.*, "Alleviating irregularity in graph analytics acceleration: A hardware/software co-design approach," in *MICRO*. New York, NY, USA: ACM, 2019, pp. 615–628.

[73] S. Rahman *et al.*, "Graphpulse: An event-driven hardware accelerator for asynchronous graph processing," in *MICRO*, 2020, pp. 908–921.

[74] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *ISCA*, 2017, pp. 1–12.

[75] Y. Chen *et al.*, "DaDianNao: A Machine-Learning Supercomputer," in *MICRO*, 2014, pp. 609–622.

[76] S. Liang *et al.*, "EnGN: A High-Throughput and Energy-Efficient Accelerator for Large Graph Neural Networks," *IEEE Transactions on Computers*, 2020.

[77] H. Zeng and V. Prasanna, "Graphact: Accelerating gcn training on cpu-fpga heterogeneous platforms," in *International Symposium on Field-Programmable Gate Arrays*, 2020, pp. 255–265.

[78] T. Mikolov *et al.*, "Efficient estimation of word representations in vector space," *International Conference on Representation Learning*, 2013.

[79] Q. Li *et al.*, "Deeper insights into graph convolutional networks for semi-supervised learning," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, no. 1, 2018.

[80] M. Burtscher *et al.*, "A quantitative study of irregular programs on gpus," in *IISWC*. IEEE, 2012, pp. 141–151.

[81] S. Che *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," in *IISWC*, 2009, pp. 44–54.

[82] J. Deng *et al.*, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 2009, pp. 248–255.

[83] S. Beamer and others, "The GAP Benchmark Suite," in *arXiv:1508.03619 [cs.DC]*, 2015.

[84] T. Mikolov, "Word2vec implementation in C." [Online]. Available: https://github.com/tmikolov/word2vec

[85] T. M. Simonton and G. Alaghband, "Efficient and accurate word2vec implementations in gpu and shared-memory multicore architectures," in *HPEC*. IEEE, 2017, pp. 1–7.

[86] T. Nguyen, "Cuda implementation of cbow word2vec." [Online]. Available: https://github.com/cudabigdata/word2vec_cuda

[87] K. Hoste and L. Eeckhout, "Microarchitecture-independent workload characterization," *IEEE Micro*, vol. 27, no. 3, pp. 63–72, 2007.

[88] "Nvidia nsight compute cli." [Online]. Available: https://docs.nvidia.com/nsight-compute/NsightComputeCli/

[89] [Online]. Available: https://pytorch.org/cppdocs/

[90] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *AAAI*, 2015. [Online]. Available: http://networkrepository.com

[91] W. Cohen, "Enron email dataset," http://www.cs.cmu.edu/ enron/. Accessed in 2009.

[92] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[93] A. Paranjape *et al.*, "Motifs in temporal networks," in *WSDM*. New York, NY, USA: Association for Computing Machinery, 2017.

[94] J. Leskovec *et al.*, "Governance in social media: A case study of the wikipedia promotion process," in *AAAI Conference on Web and Social Media*, 2010.

[95] D. Xu *et al.*, "Spatio-temporal attentive rnn for node classification in temporal attributed graphs." in *IJCAI*, 2019.

[96] M. G. Preti *et al.*, "The dynamic functional connectome: State-of-the-art and perspectives," *Neuroimage*, 2017.

[97] NVIDIA Forum, "https://forums.developer.nvidia.com/t/some-questions-about-metrics-global-hit-rate-stall-constant-memory-dependency-etc-of-nvprof/63997."

[98] K. He *et al.*, "Deep residual learning for image recognition," in *IEEE conference on computer vision and pattern recognition*, 2016.

[99] G. Huang *et al.*, "Densely connected convolutional networks," in *IEEE conference on computer vision and pattern recognition*, 2017.

[100] A. Brahmakshatriya *et al.*, "Compiling graph applications for gpus with graphit," in *CGO*, 2021.

A. Artifact Appendix

A.1 Abstract

This document describes the process of acquiring and reproducing the experimental results presented in "A Deep Dive Into Understanding The Random Walk-Based Temporal Graph Learning." This paper is accepted for publication at 2021 IEEE International Symposium on Workload Characterization (IISWC 2021). The artifacts include implementations of link prediction and node classification algorithms both on CPU and GPU. To run these artifacts, a multi-threaded CPU and a GPU are required.

A.2 Artifact check-list (meta-information)

- **Algorithm:** link prediction and node classification
- **Compilation:** `g++ v7.5` for CPU and `nvcc v11.2` for GPU; cmake files included for compilation
- **Data set:** ia-email, wiki-talk, stackoverflow for link prediction, and dblp5, dblp3, and brain for node classification
- **Run-time environment:** CPU implementations should run natively. For GPU implementation, we provide a Docker image that resolves runtime dependencies
- **Hardware:** Multi-threaded CPU and an NVIDIA GPU
- **Execution:** Bash script for automatic compilation and execution
- **Metrics:** Prediction accuracy, execution time, GPU execution stalls
- **Output:** Terminal logs showing progress and execution time of different phases of the application along with prediction accuracy of an end-to-end task
- **Experiments:** Fig. 5, 6 (performance improvement of word2vec GPU implementation), Fig. 8 (accuracy-runtime complexity trade-off), Table III (execution time breakdown), Fig. 9 (Dynamic instruction breakdown of different kernels), Fig. 10 (performance scalability) and Fig. 11 (execution stall characterization on a GPU)
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** BSD 3-Clause License

A.3 Description

A.3.1 How to access

The artifact code base can be obtained by downloading from
`https://zenodo.org/record/5555384`.
A more updated version is maintained on GitHub here
`https://github.com/talnish/iiswc21_rwalk`.
The README files in the main and `data/` directories contain instructions to download open-source data sets. Additionally, README file in the main directory also contains instructions to run a Docker image for GPU.

A.3.2 Hardware dependencies

Although we have used an AMD CPU for evaluation, any commodity multi-core CPU should be adequate for running CPU implementations. Additionally, an NVIDIA GPU is required for running GPU implementations.

A.3.3 Software dependencies

We use PyTorch framework to execute the ML part of our workload. For our CPU implementations, a pre-compiled PyTorch binary (*i.e.,* `libtorch.so`) is dynamically linked during runtime. The REDME file in the main directory contains instructions on how to download and link this library. For the GPU version, the Docker environment already has this dependency resolved. Therefore, no additional software dependencies are necessary for GPU implementations.

A.3.4 Data sets

We use real-world temporal data sets [1–7] for evaluation. The datasets are obtained from the following websites:

- `http://snap.stanford.edu/data/index.html`
- `https://networkrepository.com/dynamic.php`
- `https://tinyurl.com/y67ywq6j`

In addition to real-world datasets, we provide scripts to generate synthetic Erdős-Renyi graphs using Python-based `networkx` library in `data/link_pred` directory.

A.4 Installation

For CPU implementation, the benchmark can run natively with a pre-compiled PyTorch binary.

Downloading and linking pre-compiled PyTorch binary. Download the latest C++ distribution of libtorch from
`https://pytorch.org/cppdocs/installing.html`.
Link the pre-compiled PyTorch binary to the benchmark compiling phase by replacing the `CMAKE_PREFIX_PATH` with the newly installed libtorch path in
`build_cpu/build_{linkpred/nodeclass}_run.sh`.

For GPU implementation, we provide the procedure to build and run the Docker image with the correct environment. Note that you may need to run the Docker commands as `sudo`.

Setting up Docker. Download Docker from
`https://docs.docker.com/get-docker/` and start the Docker daemon. Verify that Docker is working by running docker run -t hello-world.

Cloning the Artifact Repository. We have created Zenodo and GitHub repositories to clone our artifacts. The repository links are as follows.

- `https://zenodo.org/record/5555384`
- `https://github.com/talnish/iiswc21_rwalk`

Start the Docker Shell Prompt. We have uploaded the Docker image for GPU execution at `rwalklearn/iiswc21` in DockerHub. Users can run the Docker image using the following command:

```
sudo docker run --cap-add SYS_ADMIN --security-opt
seccomp=unconfined --gpus all -it --shm-size 8G -v
/your/directory/to/rwalk:/rwalk rwalklearn/iiswc21.
```

This step should start a shell prompt. This step provides all the environment and all tests can be run after `cd /rwalk` in the new shell prompt.

A.5 Preparing datasets

The datasets for link prediction and node classification need to be prepared separately. Link prediction only requires a temporal graph as the input dataset. Node classification, on the other hand, requires a temporal graph, and labeled training/validation/testing datasets as input. The following instructions detail how to prepare datasets.

Link prediction. This process is detailed in the README file in `data/link_pred/` directory. The real-world datasets used in this paper can be obtained from:

- `http://networkrepository.com/dynamic.php`
- `http://snap.stanford.edu/data/index.html#temporal`

First, download and unzip a dataset using `gunzip` or `unzip` command. Open the dataset file and make sure that there are no comment present in the dataset (typically they start with '#'). If there are text-based lines starting from '#', remove them. After this, each row of the dataset should have the following format:

- `<node_id node_id timestamp>`.

After this, preprocess the dataset using the following command.

- `python preprocess_dataset.py -i dataset-filename.ext #TODO: add dataset-filename.ext`

After running this command, the script will generate a dataset called `dataset-filename.wel` that is ready for link prediction. Note that each row of the prepared dataset will have the following format:

- `<node_id node_id norm_timestamp>`,

where `norm_timestamp` contains normalized timestamp between 0 and 1.

We also provide a script to generate synthetic datasets in the same directory. To generate a synthetic graph with `n` number of nodes, `e` number of vertices, using a seed `s`, run the following command:

- `python genereate_synthetic.py -n n -e e -s s`.

Node classification. The dataset for node classification includes a temporal graph, and labeled training/validation, and testing datasets. We place all of these files for one dataset with generic names in a directory. The name of the directory is representative of the dataset. For example, `data/node_class/dblp3` contains instructions to generate and store data for the `dblp3` dataset. The real-world input dataset for node classification are obtained from

- https://tinyurl.com/y67ywq6j

Download the datasets from the link above. The datasets have `.npz` file extension. Use `process_dataset.py` script to generate a temporal graph, and training/testing/validation datasets for node classification as follows.

- `python process_dataset.py -i DBLP3.npz`

This will create four files: (a) `tgraph.wel`, (b) `train.tsv`, (c) `valid.tsv`, and (d) `test.tsv`. By indicating the name of the dataset (same as the name of this directory), the application will automatically detect and use these four files.

A.6 Experiment workflow

Following instructions in A.4, a path to your local `libtorch.so` file should have been added in the following files:

- `build_cpu/build_linkpred_run.sh`
- `build_cpu/build_nodeclass_run.sh`

Additionally, add the appropriate dataset names in:

- `build_cpu/build_linkpred_run.sh`
- `build_cpu/build_nodeclass_run.sh`
- `build_gpu/build_linkpred_run.sh`
- `build_gpu/build_nodeclass_run.sh`

Finally, to run the benchmarks, execute the following commands (these will compile and run the benchmarks).

- Link prediction on CPU:
  `cd build_cpu/ && ./build_linkpred_run.sh`
- Node classificaiton on CPU:
  `cd build_cpu/ && ./build_nodeclass_run.sh`
- Link prediction on GPU:
  `cd build_gpu/ && ./build_linkpred_run.sh`
- Node classificaiton on GPU:
  `cd build_gpu/ && ./build_nodeclass_run.sh`

The profiling scripts using `nv-nsight-cu-cli` tool (for GPU implementation only) are also available by uncommenting the commands at the end of the corresponding run scripts.

A.7 Evaluation and expected results

After the runs are completed, the key information of each stage (random walk, word2vec, training, and testing) is shown in the terminal. The detailed output of each stage can also be documented by enabling the print flag of each stage.

The parameters in the link prediction/node classification tasks are tuned to the sweet spots discussed in Figure 8 in the paper. Sweeping the parameters should generate similar data points as shown in the Figure 8 in the paper. The major source of variability may come from the different hardware that executes the workload.

A.8 Experiment customization

We outline some customization that can apply to the benchmark. The parameter values can be easily altered by modifying the parameter files present in `build_{cpu/gpu}/params_files`.

Sweeping the parameters for individual stages. We provide the source code for enabling and tuning the parameters for individual stages (some parameters are only used in GPU instances). These parameters include the following,

- System-level: number of allocated threads, number of parallel data loaders
- Random walk: number of walks, maximum depth of each walk.
- Word2vec: dimension of generated embeddings, cacheline alignment (GPU), batch size of feature update (GPU), parallel reduction (GPU). (Word2vec GPU features can be tuned by changing the macro definition in `src_gpu/word2vec.cuh`.)
- Training and testing: number of epochs, hidden layer dimensions, learning rate, batch size of training, target accuracy.

Profiling metric collection. We use different tools to profile the CPU and GPU implementation of the random-walk based temporal graph learning. For CPU profiling, we use MICA to extract the instruction breakdown. This tool can be obtained from MICA Github repository. For GPU profiling, we use the `nv-nsight-cu-cli` to extract the instruction breakdown, characterization of GPU stalls, and other hardware performance counters (L2 cache hit rate, SM utilization, bandwidth utilization, etc). Users can define their own interested metrics by switching to the corresponding nsight profiler commands. The profiling instructions used to generate our plots are present in the last few commented lines of `build_gpu/build_{linkpred/nodeclass}_run.sh`

References

[1] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *AAAI*, 2015. [Online]. Available: http://networkrepository.com

[2] W. Cohen, "Enron email dataset," http://www.cs.cmu.edu/ enron/. Accessed in 2009.

[3] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[4] A. Paranjape *et al.*, "Motifs in temporal networks," in *WSDM*. New York, NY, USA: Association for Computing Machinery, 2017.

[5] J. Leskovec *et al.*, "Governance in social media: A case study of the wikipedia promotion process," in *AAAI Conference on Web and Social Media*, 2010.

[6] D. Xu *et al.*, "Spatio-temporal attentive rnn for node classification in temporal attributed graphs." in *IJCAI*, 2019.

[7] M. G. Preti *et al.*, "The dynamic functional connectome: State-of-the-art and perspectives," *Neuroimage*, 2017.