GENTI: GPU-powered Walk-based Subgraph Extraction for Scalable Representation Learning on Dynamic Graphs

Zihao Yu*
Nanyang Technological University
zihao.yu@ntu.edu.sg

Ningyi Liao*
Nanyang Technological University
liao0090@e.ntu.edu.sg

Siqiang Luo
Nanyang Technological University
siqiang.luo@ntu.edu.sg

ABSTRACT

Graph representation learning is an emerging task for effectively embedding graph-structured data with learned features. Among them, Subgraph-based GRL (SGRL) methods have proven better scalability and expressiveness for large-scale tasks. The core challenge of applying SGRL to dynamic graphs lies in accommodating the extraction of subgraphs to evolving data with efficient computation. To address the efficiency bottleneck, we propose GENTI, a GPU-oriented SGRL algorithm for dynamic graphs. Our approach mainly improves the critical subgraph extraction stage by disentangling it into two phases, namely neighbor sampling and subgraph gathering, which are respectively performed on CPU and GPU in an asynchronous fashion. The design favorably eliminates the dependence of feature learning on subgraph extraction, and is capable of exploiting the GPU batch processing ability to remarkably boost computations throughout the pipeline. Dedicated data structures are designed for efficiently managing the dynamic graph storage and conforming efficient subgraph operations. Extensive empirical results on various real-world dynamic graphs show that GENTI achieves up to 30× faster in subgraph extraction time than the state-of-the-art walk-based methods and up to 26× acceleration in overall learning time, while maintaining comparable prediction performance. In particular, it is able to complete learning on the largest available graph of 1.3 billion edges within 24 hours, while all other baselines exhibit prohibitive overhead.

1 INTRODUCTION

Graph representation learning (GRL) has recently garnered substantial attention for effectively understanding graph-structured data in modeling entities and their relationships, especially for complicated and large-scale graphs in real-world applications, such as recommender systems [50] and network modeling [39]. GRL aims to utilize neural networks to retrieve information from the graph structure and map into low-dimensional representations, which is then used to generate predictions for downstream tasks such as node classification and link prediction [10].

To obtain structural representations, i.e., features from the graph, conventional studies for GRL can be broadly categorized into two types [2, 17]. Approaches *based on graph neural networks* (GNNs) generate representations by normal GNN iterations of propagating and updating neighboring messages [9, 18, 41], which suffer from the loss of relative intra-node knowledge and limited expressiveness [4, 6]. Alternatively, *subgraph-based GRL* (SGRL) methods embed features from a specified area around nodes of interest, namely subgraphs. Applying neural network models to the substructure instead of the whole graph favorably captures the underlying structural information [44, 49, 51]. Previous studies show that compared to GNNs, SGRL preserves expressiveness and offers better scalability for GRL tasks emphasizing graph structure, such as link prediction [21, 35, 45]. From the perspective of SGRL system, its learning process exhibits a joint utilization of CPU and GPU devices as shown in Fig. 1. Typically, CPU is utilized to compute the subgraph extraction and feature generation. Afterwards, the feature data is learned by a neural network on the GPU in batches.

However, most existing GRL approaches are tailored for static graphs and overlook the dynamic perspective. In realistic scenarios such as financial transactions and recommender systems [23, 40], graphs are actively evolving with frequent updates such as the establishment of new connections. With the aim of transferring to dynamic graphs, recent SGRL models [15, 39] propose to utilize temporal variants of subgraph processing techniques and achieve promising accuracy in the link prediction task.

Challenges of Dynamic SGRL. Despite their prominent algorithmic solutions, the dynamic SGRL scheme calls for dedicated system designs with unique challenges: the subgraph data in extraction requires **streaming updates** under the evolving graph structure, entailing a significant computation and communication cost [40, 48]. As subgraph extraction is prerequisite for feature learning, it becomes the **workload bottleneck** between CPU and GPU devices. Correspondingly, graph **data structure** needs to be specifically redesigned for enhanced efficiency when undergoing dynamic amendments and subgraph sampling operations. These system level issues pose piratical difficulties when applying the advanced algorithms to real-world scenarios, especially when computational resources are limited.

Figure 1 illustrates the time consumption of the three-stage pipeline of representative dynamic SGRL methods CAW [39] and NeurTWs [15] in our experiment. It can be observed that subgraph extraction and feature generation executed on CPU constitute the majority of GRL overhead, and further delays GPU computation. The sequential execution also results in low CPU and GPU device utilization. Additionally, in our evaluation, both current methods are prohibitive on the largest available dynamic graph MAG with 1.3 billion edges, demanding prolonged learning time over 24 hours.

Our Contribution. In this paper, we propose GENTI, a **G**PU-effici**EN**t SGRL on continuous-**TI**me dynamic graphs. GENTI highlights the adaptation of GPU to better balance the workload and improve efficiency. We primarily focus on the subgraph extraction procedure in light of its pivot role for affecting both efficacy and efficiency within the SGRL pipeline. To this end, algorithms, data structures, and operations related to this stage are thoroughly refined for GPU processing and dynamic updates to be applicable on billion-edge graphs. GENTI is useful in enabling the application of

*Both authors contributed equally to this research.

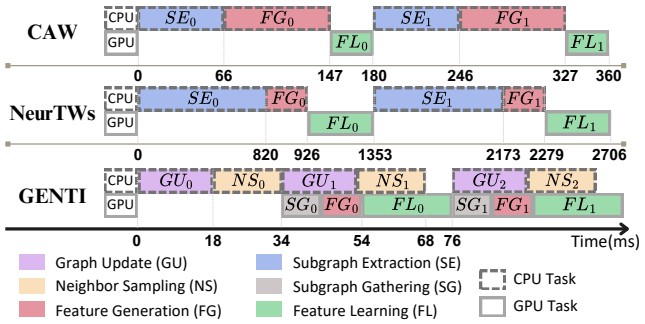


Figure 1: Experimental comparison on execution time of our GENTI pipeline against SGRL methods CAW [39] and NeurTWs [15]. Each stage is either on the CPU or GPU device. GENTI divides the subgraph extraction bottleneck into asynchronous CPU neighbor sampling and GPU subgraph gathering to facilitate balanced and concurrent workloads. Graph update procedures are not shown in CAW and NeurTWs as they are preprocessed in an extra stage, while an initial graph update and sampling stage is presented in GENTI.

SGRL to large-scale realistic graphs with acceptable computational power and running time.

To harness GPU acceleration and address imbalanced device workload, we devise a novel subgraph extraction scheme by dissecting it into two phases respectively for neighbor sampling and subgraph gathering. The first phase samples neighborhoods from the graph structure by CPU and loads the data onto GPU. Then, subgraph gathering by performing random walks is completely conducted on GPU thanks to its parallelism. A memory-efficient pool is innovatively crafted to cache the sampled subgraph on GPU and support gathering in batches. As shown in Figure 1, our overall subgraph extraction scheme ensures that it is entirely asynchronous to feature processing, effectively distributing the workload between the two devices and preventing the pipeline from being blocked by the CPU computation bottleneck. Moreover, we perform dedicated enhancements on the data structure for graph storage in consideration of the dynamic SGRL pipeline, powering it with capabilities including $O(1)$ updating and search-free sampling.

To summarize, we have made the following contributions:

- **Algorithm:** We propose GENTI as a scalable SGRL algorithm on dynamic graphs with GPU-oriented designs. We decouple the resource-intensive subgraph extraction stage to be separately conducted on CPUs and GPUs, which enables full GPU utilization and offers improved efficiency.
- **System:** We design the data structure for maintaining subgraphs on GPU with improved memory complexity and fast batch processing ability. We also enhance the graph storage to render it applicable for scalable and dynamic updating and sampling.
- **Experiment:** We conduct comprehensive experiments to evaluate the efficiency and effectiveness of GENTI on 7 real-world dynamic graphs with up to millions/billions of nodes/edges. Concerning the comprehensive pipeline and specifically the subgraph extraction stage, our approach respectively achieves up to 30× and 26× speedups in running time when compared to state-of-the-art SGRL algorithms.

2 RELATED WORKS

2.1 Subgraph-based GRL

GRL methods based on subgraphs have been extensively applied to a variety of tasks on graph-structured data [12, 19, 24, 25, 50, 54]. The intuition of SGRL is to extract and learn from a subgraph around queried nodes to emphasize local structural information and enhance model expressiveness. As depicted in Fig. 2, common SGRL methods adopt a pipeline of extracting subgraphs for each queried node, generating features from subgraph structure, and lastly learning predictions for the given task [44, 45, 49, 51].

Being the core of SGRL, subgraph extraction has been accentuated through the design of samplers, which can be further divided into two categories. *Walk-based samplers* [29, 43, 49, 51] generate a certain number of random walks starting from each seed node and encode such structural information as features. In comparison, *metric-based samplers* [1, 19, 37, 44] rely on graph metrics such as personalized PageRank (PPR) [14] for assessing the proximity between given nodes. Previous evidence shows that the latter nonetheless struggle to capture intricate subgraph motifs and achieve relatively suboptimal performance in downstream tasks [44]. Therefore, our SGRL algorithm focuses on utilizing and improving the walk-based sampler for subgraph extraction.

2.2 GRL on Dynamic Graphs

In order to learn from graphs with dynamic updates, GRL methods are equipped with techniques for obtaining and handling features for both temporal and topology information [16]. *GNN-based* approaches [46, 52, 53] commonly exploit Recurrent Neural Networks (RNNs) structures [5, 28] to deal with the sequential data. However, these methods necessitate accessing the entire graph data stored in RAM to update structural features, resulting in frequent synchronization delays during the GPU training pipeline.

Subgraph-based designs [22, 26] introduce specific metrics to sample representative subgraphs and embed temporal information. In particular, *SGRLs featuring walk-based samplers* represented by CAW [39] and NeurTWs [15] have the merit to extend random walks into the temporal dimension and efficiently retrieve graph dynamics. Intuitively, most recent updates of the graph have greater impacts on the prediction result. Hence, they tend to only extract and model relevant nodes being closest to the current timestamp, which greatly reduce the overhead in both structural and temporal domains. These dynamic solutions entails an extra round of graph update computation for the whole time range, which significantly impedes their deployment with streaming updates.

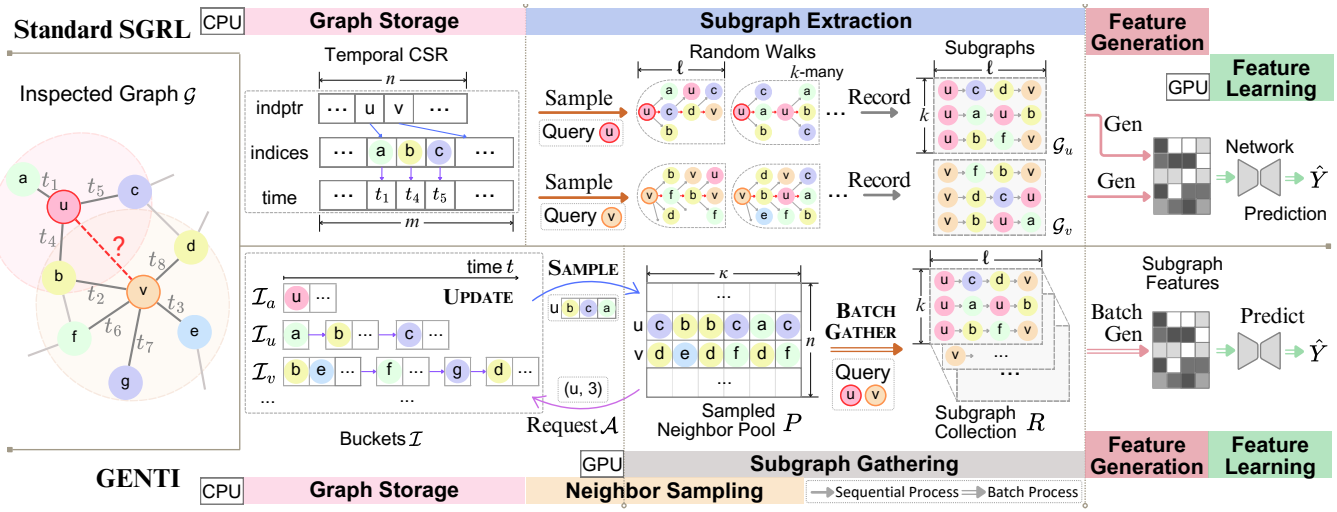


Figure 2: Framework overview of GENTI and standard SGRL methods. Conventionally, the three-stage SGRL pipeline, including subgraph extraction, feature generation, and feature learning, is conducted in a sequential manner. GENTI divides the critical subgraph extraction stage into asynchronous neighbor sampling and subgraph gathering, which enables the representation learning stages to commence in batches on GPU without blockage. The overlap of neighbor sampling and subgraph gathering stages indicates that they can be performed simultaneously in actual execution. Moreover, GENTI introduces a bucketing scheme to enhance dynamic graph storage by supporting efficient update and sampling.

3 PRELIMINARIES

3.1 Dynamic Graphs

At a specific timestamp, a dynamic graph is denoted as $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ and $\mathcal{E}$ are node and directed edge sets with sizes $n = |\mathcal{V}|$ and $m = |\mathcal{E}|$, respectively. Dynamic graphs can be classified into two categories, namely *Continuous-Time Dynamic Graphs* (CTDGs) and *Discrete-Time Dynamic Graphs* (DTDGs), depending on their representation with regard to time intervals [16]. We mainly focus on CTDGs as they are more expressive in depicting temporal information and are commonly used in SGRL tasks [7, 16].

For dynamic graphs, the graph components of nodes and edges are changing overtime, which are knows as events. A CTDG can be constructed by a series of edge additions and deletions events. The graph $\mathcal{G}$ can thus be represented by an event sequence $\mathcal{S} = \{(u, v, t, op)\}$. Each event implies that the edge from node u to v experiences the operation denoted by op at timestamp t. The operation op can either be insertion or deletion of the edge (u, v).

3.2 Temporal Walk-based Sampling

Walk-based SGRL employs the random walk procedure to sample connected nodes and construct the subgraph. The temporal random walk serves as an extension tailored for dynamic graphs, facilitating learning from the temporal dimension. A temporal random walk at time t is a traversal of $\mathcal{G}$ that initiates from the source node $u \in \mathcal{V}$ and, at each step, advances to a randomly selected out-neighbor of the current node. Generating the subgraph for a node of interest usually requires k independent temporal random walks. At each step of these walks, the destination nodes are selected by the Weighted Neighbor Sampling (WNS) technique [15, 39]. At time t, the weighted neighbor set of node u is defined as $\mathcal{N}_u = \{(v, \tau, w) \mid (u, v) \in \mathcal{E}, \tau < t\}$. Each node v is an out-neighbor of u indicated by a specific $\tau < t$. In other words, the edge (u, v) is added to the graph at time τ and maintained at time t. The associated sampling weight is determined by $w = \exp(\tau/\alpha)$, where $\alpha > 0$ is a hyper-parameter controlling the effect of temporal significance. The current degree of node u is denoted as $d_u = |\mathcal{N}_u|$.

Different from previous GRL methods that directly implement WNS by performing sampling from the entire neighborhood, we here strategically derive an equivalence between WNS and the technique for dynamic weighted set sampling (DWSS) [8, 27, 31, 47]. The DWSS problem aims to perform k independent sampling with replacement from a dynamic set of elements based on corresponding weights. Corresponding to our context, it is equivalent to the WNS problem from the node neighborhood $\mathcal{N}_u$.

Zhang et al. [47] proposes a bucketing scheme [8, 27] for efficiently solving the DWSS problem. Candidate elements are allocated into r buckets in total based on their weights, where the b-th bucket is $\mathcal{I}_{u,b} = \{(v, \tau, w) \mid (v, \tau, w) \in \mathcal{N}_u, \exp(b) \le w < \exp(b+1)\}$, and the bucket index $0 \le b \le r$. Fetching an element from the buckets follows a rejection sampling scheme FETCH: when assessing an element v from bucket $\mathcal{I}_{u,b}$, the sampler either accepts it with a probability of $w/\exp(b+1)$, or rejects it and repeats the process until an element is accepted. [47] proves that the total complexity for sampling k elements can be bounded by $O(\log d_u + k)$.

4 THE FRAMEWORK OF GENTI

In this section, we introduce GENTI by highlighting the GPU-oriented and streaming update aspects of our method. First, we present the overall framework of GENTI in Section 4.1 by introducing the relationship between different phases and data structures. Then, we describe the designs related to graph storage, decoupled sampling, and subgraph gathering in the following sections.

4.1 GPU-powered Learning Pipeline

Figure 2 displays the overview of GENTI compared to typical walk-based SGRL methods [15, 39]. Among the three consecutive stages of the SGRL pipeline—subgraph extraction, feature generation, and feature learning—the last one usually occurs on GPU devices, enjoying efficient batch processing. In contrast, the former two stages must be sequentially computed by the CPU. During subgraph extraction, canonical SGRL performs random walk sampling directly from the stored graph structure. The frequent graph access results in a relatively high cache miss ratio. This stage hence becomes the efficiency bottleneck in the SGRL deployment as shown in Figure 1, even though its theoretical time complexity appears to be low.

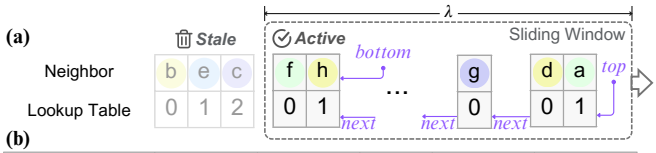
To mitigate the imbalanced workload, we intend to exploit parallel computation and propose disentangling the subgraph extraction into two independent phases, one for sampling and the other for gathering. By introducing new data structures and addressing their dependency, the two stages can be executed in parallel on CPU and GPU, respectively. In this way, we successfully transfer all subsequent stages onto the GPU for batch processing. Our GPU-efficient pipeline of GENTI is shown in Algorithm 1. The tensor P maintains the candidate pool for subgraph extraction, with each row $P[u]$ representing the sampled neighbors of node u. For each timestamp, we construct both the source and destination nodes of current queries as a single batch $\mathcal{U}_t$ and perform the batch *subgraph gathering* algorithm BSGather to construct the collection of subgraphs R for all queried nodes based on current pool P. The usage of P is recorded in requests $\mathcal{A}$, and the pool is updated on demand by calling *neighbor sampling* algorithm Sample in a separate thread. Simultaneously, the graph storage is updated according to temporal events, and the prediction of the query batch is composed by learning from the generated features.

Among the operations, the Sample and Update are processed by CPU, while all other stages are conducted by GPU utilizing the power of batch computation. It is noteworthy that the two major data structures, i.e., the graph storage on RAM and the sampled neighbor pool on GPU, are updated asynchronously. Therefore, the subgraph learning on GPU is ensured to receive the up-to-date data to accommodate the dynamic graph changes. Meanwhile, the workload is considerably balanced to prevent cross-device blockage.

Algorithm 1: GENTI

Input: Graph $\mathcal{G}(\mathcal{V}, \mathcal{E}, X)$, queries $\{(u, v, \tau)\}$
Output: Predictions $\hat{Y}$

for *each* $u \in \mathcal{V}$ **do** $P[u] \leftarrow$ Sample(u, κ)
for *each timestamp* t **do**
  $\mathcal{U}_t \leftarrow \{u, v \mid \tau = t\}$
  Append $\mathcal{U}_t$ with negative queries if required
  $R, \mathcal{A} \leftarrow$ BSGather$(P, \mathcal{U}_t)$
  for *each* $(u, k_u) \in \mathcal{A}$ **do** *in separate thread*
    $P[u] \leftarrow$ Sample(u, k_u)
  end
  $\mathcal{G} \leftarrow$ Update$(\mathcal{U}_t)$
  $\hat{Y} \leftarrow$ PREDICT(R, X)
end
return $\hat{Y}$



Bucket	$\mathcal{I}_{u,0}$	$\mathcal{I}_{u,1}$		$\mathcal{I}_{u,r-1}$	$\mathcal{I}_{u,r}$
Timestamp	$[0, \alpha)$	$[\alpha, 2\alpha)$		$[(r-1)\alpha, r\alpha)$	$[r\alpha, (r+1)\alpha)$
Sample Weight	$[0, e^1)$	$[e^1, e^2)$		$[e^{r-1}, e^r)$	$[e^r, e^{r+1})$
Random Number	$\varnothing$	$\{z_4\}$		$\{z_3\}$	$\{z_1, z_2\}$
Fetch Amount	0	1		1	2
Sampled Neighbor	$\varnothing$	f		g	a d

Figure 3: (a) Bucket-based graph storage $\mathcal{I}_u$ for maintaining the neighbor set of a node u in temporal order. Two pointers are maintained to track the top and bottom indices of the current sliding window, ensuring that only buckets inside it are active, and stale ones are removed. (b) Sampling is performed by successively fetching nodes from corresponding buckets to construct the sampled neighborhood.

4.2 Bucket-based Dynamic Graph Storage

Conventional SGRL stores graph on RAM by sparse matrix structures such as the compressed sparse row (CSR) format [44, 53]. We however note that it is not optimized for performing walk-based sampling, as walking on the graph requires iteratively accessing neighboring nodes. This process renders a time complexity of $O(k\ell d_{max})$ when generating k-many ℓ-length random walks for each source node, where d_{max} is the maximum degree of visited nodes [36]. In addition, live update on the graph structure is also of prohibitive overhead due to the continuous layout. As a consequence, dynamic SGRLs [15, 39] have to preprocess all the timestamps and maintain the entire temporal information, which inevitably causes computation and memory redundancy.

We mainly look into two abilities of the dynamic storage scheme: (1) efficient sampling for the neighborhood of given nodes; (2) online update of edge addition and deletion. Inspired by the DWSS process introduced in Section 3.2, we employ the bucketing scheme as our graph storage on RAM. As illustrated in Figure 3, we store out-neighbors of each node u by a linked list of non-empty buckets $\mathcal{I}_{u,b}$. Inside each bucket, we also maintain a lookup table for fast accessing the individual element. The index b of a bucket indicates its time interval and determines the temporal sampling weight $w_{u,b} = \sum_{(v,\tau,w)\in\mathcal{I}_{u,b}} w$. The total weight of all buckets with respect to node u is $w_u = \sum_b w_{u,b}$. Each traversal of the linked list starts from the most recent non-empty bucket with the largest index r. Denote the hyperparameter representing duration length of a bucket as α and the current timestamp as t, there is $r = \lfloor t/\alpha \rfloor$.

Bucket number λ. An inherent merit of the bucketing storage is that, it is naturally sorted in temporal order, and only a certain number of buckets representing most recent updates are frequently accessed for sampling at the current timestamp. According to [47], the desired samples are likely to exist with high probability in buckets with index range $r - 2\lceil \ln d_u \rceil \le b \le r$. Therefore, we can apply a sliding window scheme that only maintains $\lambda = 2\lceil \ln d_u \rceil$ buckets in this time frame and progressively drops those stale ones.

As a result, the memory usage in RAM remains $O(m+\lambda)$, as each edge is added only once and is deleted permanently upon expiration.

Sampling. As elaborated in Section 3.2, we innovatively utilize DWSS to implement the walk-based sampling. Instead of conducting k independent sampling, our storage scheme is capable of acquiring k samples *at once* as shown in Algorithm 2. To achieve this, we generate k random numbers uniformly within the overall weight range $z \sim U(0, w_u)$. The amount of samples drawn from each bucket is decided by the amount of numbers z falling in the corresponding bucket-wise weight range. Therefore, we are able to sample all k neighbors of the given node in a single traverse of buckets through the maintained pointers. As the number of buckets is $\lambda = 2\lceil \ln d_u \rceil$, the total time complexity of SAMPLE is $O(\log d_u + k)$.

Figure 3 illustrates a running example for the sampling request $\mathcal{A} = (u, 4)$ enquiring $k = 4$ neighbors of source node u. At the current timestamp, the index range of active buckets is $1 \le b \le r$. The time range and weight range of each bucket can be directly derived from its index, and $k = 4$ random numbers are generated

Algorithm 2: SAMPLE

Input: Storage $\mathcal{I}$, source node u, number of samples k
Output: Updated neighbor pool $P[u]$

$\mathcal{Z} \leftarrow \{z_1, \cdots, z_k \mid z_j \sim U(0, w_u),\ 1 \le j \le k\}$
$b \leftarrow r,\ w_{sum} \leftarrow w_u$
while $\mathcal{Z} \neq \emptyset$ **do**
  $w_{sum} \leftarrow w_{sum} - w_{u,b}$
  $\mathcal{Z}_b \leftarrow \{z \in \mathcal{Z} \mid z \ge w_{sum}\},\ \mathcal{Z} \leftarrow \mathcal{Z} \setminus \mathcal{Z}_b$
  for *i from* 1 *to* $|\mathcal{Z}_b|$ **do**
    $v \leftarrow \texttt{FETCH}(\mathcal{I}_b)$
    $c_{u,tail} \leftarrow (c_{u,tail} + 1) \mod \kappa$
    $P[u, c_{u,tail}] \leftarrow v$
  end
  $b \leftarrow \mathcal{I}_{u,b}.next$
end
return P

Algorithm 3: UPDATE

Input: Storage $\mathcal{I}$, update events $\mathcal{S} = \{(u, v, t, op)\}$
Output: Up-to-date storage $\mathcal{I}$

for *each* $(u, v, t, op) \in \mathcal{S}$ **do**
  $b \leftarrow \lfloor t/\alpha \rfloor$
  if *op* = *'insert'* **then**
    if $\mathcal{I}_{u,b} = \emptyset$ **then** insert $\mathcal{I}_{u,b}$
    $\mathcal{I}_{u,b} \leftarrow \mathcal{I}_{u,b} \cup \{v\}$
    $w \leftarrow \exp(t/\alpha),\ w_{u,b} \leftarrow w_{u,b} + w$
  end
  if *op* = *'delete'* **then**
    $\mathcal{I}_{u,b} \leftarrow \mathcal{I}_{u,b} \setminus \{v\}$
    if $\mathcal{I}_{u,b} = \emptyset$ **then** remove $\mathcal{I}_{u,b}$
    $w_{u,b} \leftarrow w_{u,b} - \exp(t/\alpha)$
  end
  remove $\mathcal{I}_{u,b'}$ for $b' < r - 2\lceil \ln d_u \rceil$
end
return $\mathcal{I}$

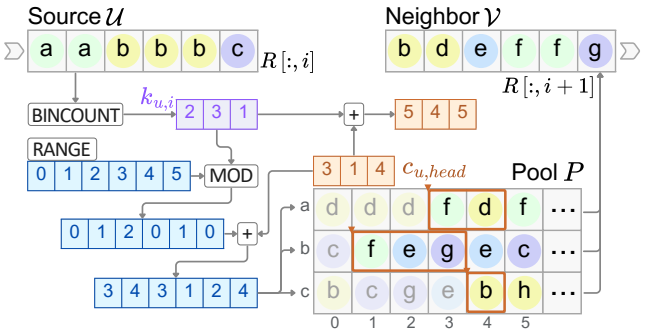


Figure 4: Example of one step of BSGATHER by computing indices and accessing the sampled neighbor pool. All operations are implemented on the GPU via batch processing.

accordingly. Among them, both z_1 and z_2 are within the weight range of $\mathcal{I}_{u,r}$, while z_3 and z_4 correspond to $\mathcal{I}_{u,r-1}$ and $\mathcal{I}_{u,1}$, respectively. We perform 2, 1, 1 times of retrieval FETCH from these buckets by successively accessing them in the linked list. Eventually, four neighbors are sampled and loaded into the pool $P[u]$.

Update. The dynamic UPDATE operation can be found in Algorithm 3. Thanks to the temporal order of our bucket storage, the target bucket corresponding to incoming and stale edges can be immediately indexed based on the timestamp. The lookup table then locates and manages the specific neighbor within the bucket to perform neighbor addition or deletion. At the end, stale buckets are removed since they are rarely accessed in the future. This is implemented by accessing the bucket with the lowest index in the linked list and setting its pointer to nil, marking its data as deleted. Overall, this scheme completes a single update event in $O(1)$ time.

4.3 Sampled Neighbor Pool

The sampled neighbor pool is a novel data structure we design to expedite the subgraph extraction, which serves as an intermediate storage for the sampled neighbors on GPU and supports asynchronous gathering and updating. The pool is denoted as tensor $P \in \mathbb{R}^{n \times \kappa}$, where κ is the width of the pool. Each row of the tensor $P[u]$ represents a queue containing out-neighbors with respect to source node u. Two indices $c_{u,head}, c_{u,tail}$ maintain the current access points of BSGATHER and SAMPLE processes, respectively.

During subgraph extraction, the separate thread SAMPLE loads the sampling results onto GPU and append them to the end of the queue in a streaming fashion. In contrast, the GPU process BSGATHER consumes neighbors stored in P to construct subgraphs. Pool elements positioned between the two pointers $c_{u,head}, c_{u,tail}$ are available for the following subgraph construction since they are freshly sampled and have not been accessed by previous gathering. The disentangled read and write operations towards the pool prevent BSGATHER from waiting for neighbor sampling to provide the updated data and thereby enjoys better throughput.

Pool width κ. The width κ of queues in the pool has the following requirements: (1) it should be as small as possible to save GPU memory usage; (2) it should be large enough to accommodate the neighbor consumption by BSGATHER for applicable cases. As described in Section 4.4, the gathering procedure carries out k-many ℓ-length random walks for each query node. A naive selection is

thence $\kappa = k\ell$, resulting in a memory expense of $O(nk\ell)$ as in [44, 45] for the entire static graph.

By referring to the cache strategy in the two-pass streaming random walk [3], we are able to reduce the overhead with a tightened bound for κ. We distinguish the storage schemes into two types based on the consumption status of pool elements. *Heavy* nodes are source nodes being likely to visit more than κ neighbors. Hence we directly access all the neighbors with size d_u of these nodes during walks. On the contrary, candidate neighbors for *light* nodes are handled by the pool since they will not exhaust the κ-length queue during gathering. [3] derive the following lemma:

Lemma 1 ([3]). *Under the boundary $\kappa \sim O(\sqrt{\ell})$, the sum of outgoing degrees of all heavy nodes u is bounded by $\sum_u d_u \leq O(n\sqrt{\ell})$.*

In our implementation, we set $\kappa = k\sqrt{\ell}$ as the pool width. Consequently, the total GPU memory usage for all nodes in the graph, including both heavy and light ones, has a complexity of $O(nk\sqrt{\ell})$.

4.4 Batch Subgraph Gathering

In this section, we highlight the GPU-efficient batch subgraph gathering BSGather as illustrated in Algorithm 4. It returns a shape $|\mathcal{U}_t| \times k \times \ell$ tensor R containing subgraphs formed by k-many ℓ-length random walks with respect to $|\mathcal{U}_t|$ query nodes. The tensor R can be directly used by the subsequent feature generation and learning pipeline, thereby boosts the GPU batch processing.

According to the selective caching strategy adapted in Section 4.3, in each walk step, we split the current nodes into heavy and light multisets $\mathcal{U}_{heavy}$ and $\mathcal{U}_{light}$ by comparing the pointers $c_{u,head}$ and $c_{u,tail}$. Different from [3] requiring an individual round of walk to identify heavy nodes, our splitting scheme of the multiset can be instantly completed in $O(1)$ time. We then perform discriminative sampling for these nodes as one step of walk. For heavy source nodes, we gather all their neighbors directly from the graph structure $\mathcal{I}_u$. For light nodes, the sampled neighbor pool can be employed to provide $k_{u,i}$ number of active neighboring nodes based on the multiplicity $k_{u,i}$, i.e., occurrence of source node u in the multiset.

Algorithm 4: BSGather

Input: Storage $\mathcal{I}$, sampled neighbor pool P, walk number k, walk length ℓ, seed nodes $\mathcal{U}_t$
Output: Random walks R of shape $|\mathcal{U}_t| \times k \times \ell$, requests $\mathcal{A}$

Construct multiset $\mathcal{V}$ by repeating k times for each $u \in \mathcal{U}_t$
for *i from* 1 *to* ℓ **do**
  $\mathcal{U}_{heavy} \leftarrow \{u \mid u \in \mathcal{V}, c_{u,head} = c_{u,tail}\}$
  $\mathcal{V}_{heavy} \leftarrow \{v \mid v \in \mathcal{I}_u, u \in \mathcal{U}_{heavy}\}$
  $\mathcal{U}_{light} \leftarrow \mathcal{V} \setminus \mathcal{U}_{heavy}, \mathcal{V}_{light} \leftarrow \emptyset$
  for *each identical* $u \in \mathcal{U}_{light}$ **do**
    $k_{u,i} \leftarrow \texttt{BINCOUNT}(u, \mathcal{U}_{light})$
    $\mathcal{A} \leftarrow \mathcal{A} \cup \{(u, k_{u,i})\}$
    $\mathcal{V}_{light} \leftarrow \mathcal{V}_{light} \cup P[u, c_{u,head} : (c_{u,head} + k_{u,i})]$
    $c_{u,head} \leftarrow (c_{u,head} + k_{u,i}) \mod \kappa$
  end
  $\mathcal{V} \leftarrow \mathcal{V}_{heavy} \cup \mathcal{V}_{light}, R[:, :, i] \leftarrow \mathcal{V}$
end
return R, $\mathcal{A}$

All the sampled neighbors are combined and recorded as the i-th step random walk result R constituting the subgraph, and initiate the next step. After the ℓ-length walk is finished, nodes accessed during the process are marked as stale and are accordingly updated by sending requests to the Sample thread. This is to ensure each pool element is used once so that the walks for subgraph extraction are mutually independent. Since there are ℓ iteration steps handling k-many random walks, the total complexity of BSGather is $O(k\ell)$.

Note that for each walk step i, all operations regarding the k walks are performed in batches so that the GPU computation power is fully utilized. An example for conducting a BSGather step is provided in Figure 4. We first calculate the occurrence $k_{u,i}$ of each individual element in current node set $\mathcal{U}$, which is then used to update the head pointer and generate neighbor indices, both by batch addition operation. As all of the nodes belong to the light set, the algorithm constructs $\mathcal{V}$ by accessing entries in the pool P based on the indices corresponding to source nodes.

5 EVALUATION

In this section, we empirically evaluate the proposed framework GENTI, targeting the following major questions:

- **Q1.** Can GENTI provide prediction performance comparable to other GRL methods for CTDGs?
- **Q2.** What is efficiency gain of GENTI in the subgraph extraction stage and overall training time?
- **Q3.** How does our pipeline design affect the workloads on CPU and GPU, and can they be fully parallelized?
- **Q4.** How does the performance of GENTI vary with changes in sampling settings including the random walk number, length, and the neighbor pool size?

5.1 Experimental Setup

Task and Metric. We evaluate CTDG representation learning by the common task of future link prediction in both transductive and inductive settings [22], as well as node classification in the transductive setting [52]. In the *transductive* training, temporal links between all nodes in the graph are observed up to a specific timestamp. The model is tested by predicting the existence of remaining links after the time point. The *inductive* link prediction task involves predicting links associated with nodes that were not observed in the training node set. For both settings, model performance is assessed using average precision (AP) on the test set following previous GRL works [22, 38, 39], while results for other

Table 1: Statistics of dynamic graph datasets including the number of nodes, temporal edges, the dimension of node features, edge features, and timespan.

Dataset	$\lvert\mathcal{V}\rvert$	$\lvert\mathcal{E}\rvert$	f_v	f_e	T
UCI-MSG	1,899	59,835	0	0	180 days
Wikipedia	9,227	157,474	172	172	30 days
Reddit	10,984	672,447	172	172	30 days
SuperUser	194,085	1,443,339	172	0	2773 days
Wiki-Talk	1,140,149	7,833,140	172	0	2320 days
Tgbl-Comment	994,790	44,314,507	172	0	1848 days
GDELT	16,682	191,290,882	413	186	2 days
MAG	121,751,665	1,297,748,926	768	0	1826 days

Table 2: Comparison of SGRL methods on small CTDGs. We present model performance metrics including transductive average precision (%), inductive average precision (%), total training time (s) , and the number (#) of convergence epochs. The best and second-best results in each column are marked with bold and underlined fonts, respectively. Particularly, "TLE" indicates a time limit exceeded exception that one epoch of model training exceeds 24 hours.

Model	UCI-MSG			Wikipedia			Reddit		
	Trans AP	Induct AP	Time (#)	Trans AP	Induct AP	Time (#)	Trans AP	Induct AP	Time (#)
JODIE	80.27 ± 0.1	71.64 ± 0.6	431(12)	95.16 ± 0.4	93.13 ± 0.5	1985(18)	95.83 ± 0.3	93.20 ± 0.4	8320.4(12)
TGAT	60.25 ± 0.3	75.27 ± 2.3	689(25)	94.26 ± 0.1	92.88 ± 0.3	2428(29)	97.80 ± 0.2	96.08 ± 0.3	11138(24)
TGN	78.91 ± 0.1	75.47 ± 0.1	507(21)	98.58 ± 0.1	98.05 ± 0.1	1839(26)	98.66 ± 0.1	97.55 ± 0.1	8152(26)
APAN	84.02 ± 0.3	83.14 ± 0.5	**266(25)**	96.41 ± 0.5	96.06 ± 0.4	1352(21)	98.50 ± 0.2	97.62 ± 0.7	7728(9)
Zebra	92.74 ± 0.2	91.16 ± 0.3	483(31)	98.63 ± 0.1	98.65 ± 0.1	1329(32)	98.73 ± 0.1	98.42 ± 0.1	6207(25)
D-DGNN	90.41 ± 0.1	89.72 ± 0.1	14467(30)	99.16 ± 0.3	98.54 ± 0.2	15173(30)	**98.93 ± 0.2**	98.56 ± 0.1	50342(30)
CAW	95.33 ± 0.3	95.19 ± 0.2	1488(8)	99.18 ± 0.1	99.34 ± 0.1	3720(5)	98.80 ± 0.1	98.99 ± 0.1	30912(8)
NeurTWs	**95.46 ± 0.3**	95.70 ± 0.2	44064(12)	99.17 ± 0.1	99.32 ± 0.1	65448(9)	98.32 ± 0.2	98.05 ± 0.1	TLE
GENTI	95.36 ± 0.3	**95.82 ± 0.3**	394(8)	**99.18 ± 0.1**	**99.37 ± 0.1**	**739(8)**	98.87 ± 0.1	**99.18 ± 0.1**	**5890(8)**

Table 3: Comparison of representative SGRL methods on large CTDGs. We present the same performance metrics as Table 2, but the training time are measured in hours. Particularly, "OOM" denotes the out-of-memory error, and "TLE" indicates the time limit exceeded exception.

Model	SuperUser			Wiki-Talk			Tgbl-Comment		
	Trans AP	Induct AP	Time (#)	Trans AP	Induct AP	Time (#)	Trans AP	Induct AP	Time (#)
APAN	89.63 ± 0.5	86.32 ± 0.4	5.18(16)	TLE	TLE	TLE	TLE	TLE	TLE
Zebra	93.34 ± 0.3	97.63 ± 0.2	0.92(21)	95.25 ± 0.1	97.56 ± 0.3	4.65(18)	91.32 ± 0.5	95.15 ± 0.3	18.4(10)
CAW	93.12 ± 0.2	97.36 ± 0.4	10.9(5)	95.37 ± 0.1	98.07 ± 0.3	73.3(10)	TLE	TLE	TLE
GENTI	**94.05 ± 0.2**	**98.27 ± 0.3**	**0.66(8)**	**95.53 ± 0.1**	**98.58 ± 0.2**	**3.70(8)**	**93.51 ± 0.3**	**95.61 ± 0.2**	**12.3(6)**

Table 4: Evaluation of GENTI on the largest CTDGs. We provide identical performance metrics as detailed in Table 3, along with the corresponding memory overhead (GB) on both GPU and RAM. Specifically, the baselines encountered either out-of-memory errors or exceeded time limits, hence we exclude them from this table.

Dataset	Trans AP	Induct AP	Time (#)	Memory	
				GPU	RAM
GDELT	98.12 ± 0.5	99.53 ± 0.1	52(8)	6.3	67.2
MAG	94.88 ± 0.4	99.45 ± 0.1	192(10)	22.8	188

Table 5: Dynamic node classification on CTDGs. ROC AUCs (%) are exhibited for Wikipedia and Reddit, while F1-Micros (%) are displayed for GDELT.

	Wikipedia	Reddit	GDELT
JODIE	81.37	70.91	11.25
TGAT	85.18	60.61	10.05
TGN	88.33	63.78	10.04
APAN	82.54	62.00	11.89
D-DGNN	**89.81**	67.53	25.49
GENTI	88.77	**73.15**	**32.82**

metrics such as accuracy and AUC are also identical. Particularly, GENTI are able to apply for various other tasks such as high-order patterns identification with minor modifications regarding to the input nodes. We exclusively concentrate on assessing node classification and link prediction tasks in this study, as there are currently no dynamic datasets accessible for other tasks.

Datasets. We conduct experiments on 8 real-world dynamic datasets, including 3 small-scale ones: UCI-MSG [32], Wikipedia [39], and Reddit [39]; 5 large-scale ones: SuperUser [22], Wiki-Talk [22], Tgbl-Comment [13], GDELT [53] and MAG [53]. The statistics of datasets are presented in Table 1. For each graph in Table 1, we chronologically partitioned it into a training set (70%), a validation set (15%), and a test set (15%), following the setting outlined in [39]. In the inductive setting, we randomly select 10% of nodes and exclude the corresponding temporal links from the training sets to test the model performance.

Baselines. We extensively evaluate 8 state-of-the-art GRL methods applicable on CTDGs with varying categories and designs. They are: (1) GNN-based: JODIE [20], TGAT [42], TGN [34], APAN [38]; (2) Metric-based: Zebra [22], D-DGNN [52]; (3) Temporal walk-based: CAW [39], NeurTWs [15]. We mainly utilize their released source code and training configuration with the best prediction performance, and perform evaluation on our platform. For D-DGNN, we incorporate its preprocessing cost into the training time in our experiment to take update efficiency into account.

Hyperparameters. For a fair comparison, GENTI keep consistency in network hyperparameter, encoder, and decoder settings with [15, 39] except fixing the encoder to correctly generate prediction on non-attribute graphs. We set the batch size to 32 for small-scale datasets, 128 for SuperUser, Wiki-Talk and Tgbl-Comment,

and 512 for MAG, as an effort to reproduce baselines on most of the datasets without causing the out-of-memory error.

Environment. Experiments are conducted on a server with 192GB RAM, two 28-core Intel Xeon CPUs (2.2GHz), and an NVIDIA RTX A5000 GPU (24GB memory). Our asynchronous SGRL pipeline is implemented by the PyTorch 2.0 multiprocessing interface [33].

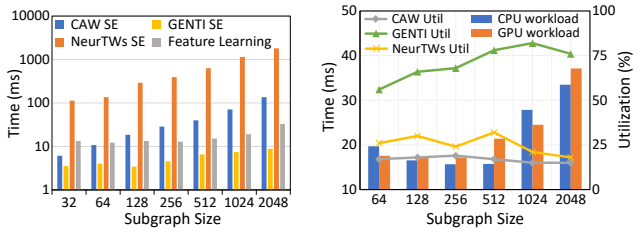
5.2 Main results

Tables 2 to 4 present our key results of model performance in the future link prediction task on 8 real-world dynamic graphs. As most baselines suffer from prolonged training time on larger graphs, we only select representative ones with applicable efficiency to perform evaluation on large CTDGs in Tables 3 and 4. Besides, the result in node classification are presented in Table 5. We conduct a comparative analysis of GENTI against the baselines in terms of both effectiveness and efficiency.

Prediction Accuracy. For GNN-based methods, the average precision is similar to the evaluation in previous studies [22] since we inherit the same experimental settings. It is worth noting that GNN-based methods exhibit relatively inferior performance due to the constraint of their aggregation scheme, which marks the superiority of subgraph-based approaches. Among SGRL methods, models with walk-based designs tend to outperform metric-based ones with fewer epochs in most cases, particularly on non-attribute graphs, where the ability to learn graph structural information is pivotal. This superiority is brought by the employment of their expressive structural encoding technique that utilizes the power of temporal random walk. Moreover, CAW and NeurTWs stand out for requiring fewer epochs to achieve model convergence due to their anonymization encoding strategy, which enhances their generalization capabilities. In our approach, we focus on enhancing the subgraph extraction stage of walk-based methods while adhering to their random walk encoding techniques. Therefore, we achieve nearly equivalent model convergence speed and prediction accuracy in link prediction task. In addition, it can be observed from Table 5 that GENTI also achieves comparable effectiveness to best baseline models in node classification task. Its subgraph-based scheme demonstrates unique usefulness in graphs with complex structures such as Reddit and GDELT. We deduce that its secondary performance on Wikipedia is caused by the weaker importance of graph topology information in the node classification task, where node attributes is essential.

As a result, GENTI successfully achieves comparable or better prediction performance in all settings as our implementation yields equivalent subgraph extraction and feature learning results and guarantees the model efficacy.

Efficiency and Scalability. According to Table 2 and Table 3, GENTI consistently achieves either optimal or suboptimal total training time across all datasets compared to other GRL methods. We highlight that GENTI exhibits notable improvements in the running time of each training epoch when compared to other walk-based methods, especially on large-scale graphs. Compared to its predecessor CAW, it realizes approximately 3∼26× speedups across all datasets. Benefiting from the GPU-oriented design that addresses the pipeline bottleneck and boosts the subgraph extraction, GENTI is the first walk-based method to complete training within 4 hours



(a) Subgraph Extraction (SE) **(b) Device Usage**

Figure 5: (a) Breakdown running time of the subgraph extraction stage of GENTI and walk-based SGRL methods with different subgraph size on Reddit. Overhead of the common feature learning stage is also plotted. Note that the time axis is on a log scale. (b) Running time of CPU and GPU computations in the GENTI pipeline, along with a comparison of average GPU utilization against baselines.

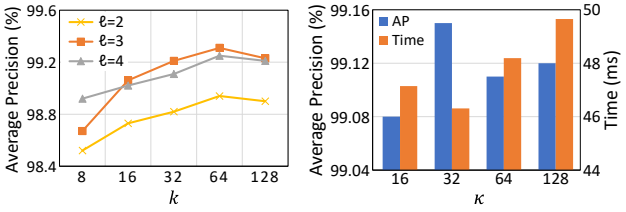
for million-size graphs and accomplish a single training epoch within 24 hours for billion-size graphs. In addition, as displayed in Table 4, the memory overhead can be controlled to a reasonable level when training on large-scale graphs. Conversely, other baselines face challenges related to either time constraints or memory issues.

Improvement of Subgraph Extraction. We specifically look into the GENTI improvement on the subgraph extraction stage that comprises the majority of SGRL learning overhead. Figure 5(a) illustrates the separate run times of the subgraph extraction stage among walk-based methods. The feature learning time is also plotted for reference. The result highlights the scalability of GENTI, as it achieves significantly shorter subgraph extraction latency and is less sensitive to the increase of subgraph size. When the subgraph size is large, we observe an approximately 30× acceleration compared to the best SGRL counterparts. GENTI is thus more scalable on billion-size graphs thanks to the larger applicable batch size for computation. Additionally, we examine the CPU and GPU workload in GENTI pipeline and compare device utilization between GENTI and other walk-based SGRL methods, as shown in Figure 5(b). It can be observed that workload between the two devices is generally balanced, and the average device utilization remains at up to 80% even with a large subgraph size. In comparison, the CPU workload becomes a bottleneck in the pipelines of CAW [39] and NeurTWs [15] when extracting large subgraphs, leading to low device utilization. As a result, our design of separating subgraph extraction into two independent phases contributes to balanced workloads and facilitates parallelism of the streaming pipeline.

5.3 Effect of Parameters

In this section, we investigate the impact of important GENTI parameters on both efficiency and prediction performance to evaluate our method design and provide guidance on the parameter choice. Due to the page limit, we mainly discuss representative results of transductive experiments on the Wikipedia dataset.

Walk Number k. The value of k determines the extent of neighborhood information extracted by SGRL for feature generation and learning. Figure 6(a) displays the effect of k on GENTI transductive average precision. The prediction precision is positively related to the value of k when $k \leq 64$. We deduce the optimal performance



(a) Walk Parameters **(b) Pool Width**

Figure 6: (a) Impact of the random walk number k and length ℓ on GENTI prediction accuracy on Wikipedia. (b) Impact of the pool width κ on GENTI prediction accuracy and time when $k = 32$ and $\ell = 4$.

when $k = 64$ indicates that, generating such k-many random walks is adequate for constructing the expressive subgraph of each seed node, while a larger number will further introduce redundancy and negatively affect feature learning. As a general conclusion, choosing a small k is ideal when embedding graphs with low-diversity interaction patterns. Otherwise, a larger value can be applied for sufficiently sampling the nodes and representing the subgraph.

Walk Length ℓ. A larger value of ℓ implies using a longer random walk to form a subgraph, allowing for the exploration of more distant information in the whole graph. According to Figure 6(a), when ℓ is set to 3, we achieve the best performance in the future link prediction task, which is a general observation across all datasets in our experiment. Intuitively, this preference indicates that the local neighbors are greater importance compared to those nodes at longer distances. We believe that a generally small ℓ can be employed in scenarios of a similar graph structure and prediction task.

Pool size κ. The pool size κ can impact both prediction accuracy and algorithmic efficiency since it decides the heavy-light splitting scheme in BSGather. A small κ can save GPU footprint but results in numerous heavy nodes that cannot be processed efficiently in a single batch. A larger κ tends to introduce more update and sampling workload on CPU and undermines the balanced workload between the two devices. According to our evaluation in Figure 6(b), when walking with $k = 32$ and $\ell = 4$, the optimal value with regard to both accuracy and efficiency is achieved by $\kappa = 32$. Comparing to the original selection $\kappa = k\sqrt{\ell}$ in Section 4.3, it is considerably smaller with a 4× reduction in memory overhead. The observation suggests that, heavy nodes are uncommon in real-world graphs, and the sampled neighbor pool scheme is competent as random walks rarely run out of its items. In practice, κ can be empirically set to a relatively low value to save GPU memory budget.

6 CONCLUSION

In this paper, we propose GENTI, a novel algorithm designed for scalable subgraph-based graph representation learning on dynamic graphs. GENTI decouples the subgraph extraction stage, commonly the bottleneck of SGRL methods, into two asynchronous phases and boosts GPU utilization. In specific, GPU processing incorporates the efficient BSGather for subgraph gathering in batches and subsequent feature generation and learning. CPU is solely responsible for maintaining the dynamic graph structure with Sample and Update operations in a streaming manner. We conduct extensive experiments on various datasets to demonstrate the scalability of GENTI in subgraph extraction and overall graph learning. GENTI achieves up to 26× faster training time than the state-of-the-art walk-based SGRL methods, while maintaining comparable prediction performance. Specifically, it is efficient to process the billion-scale graph MAG within 24 hours, which is a generally acceptable overhead compared with current solutions that fail to accomplish one training epoch with the same time expense.

Limitation and prospect. Although our GENTI achieves a significant improvement on the SGRL with *walk-based* sampling, its performance degrades when integrating with *metric-based methods*. This is because complicated metric computations require more advanced algorithmic design to divide the workloads, necessitating explorations into parallel strategies at the CUDA kernel level. To enhance the model's capability, future work may be devoted to addressing SGRL issues with various metric-based sampling methods [11, 22, 30].

Moreover, to boost its adaptability, extending GENTI to a distributed graph storage scenario also presents a challenging direction. This extension may introduce additional communication costs, potentially leading to imbalanced workloads. In detail, graph update and sampling requests are distributed across different machines, meaning that an exception from any instance can block the entire training pipeline. Consequently, strategies such as error detection and resolution are imperative to ensure the smooth operation of the training pipeline.

Fortunately, GENTI is naturally compatible with the aforementioned extensions. As displayed in Fig. 5(b), the GPU workload gradually surpasses the CPU workload as the subgraph size increases. This difference in workload allows additional strategies to be implemented on the host side without disrupting the overall training efficiency.

REFERENCES

[1] Emily Alsentzer, Samuel Finlayson, Michelle Li, and Marinka Zitnik. 2020. Subgraph neural networks. *Advances in Neural Information Processing Systems* 33 (2020), 8017–8029.

[2] Fenxiao Chen, Yun-Cheng Wang, Bin Wang, and C-C Jay Kuo. 2020. Graph representation learning: a survey. *APSIPA Transactions on Signal and Information Processing* 9 (2020), e15.

[3] Lijie Chen, Gillat Kol, Dmitry Paramonov, Raghuvansh Saxena, Zhao Song, and Huacheng Yu. 2021. Near-optimal two-pass streaming algorithm for sampling random walks over directed graphs. *International Colloquium on Automata, Languages and Programming (ICALP)* (2021).

[4] Zhengdao Chen, Lei Chen, Soledad Villar, and Joan Bruna. 2020. Can graph neural networks count substructures? *Advances in neural information processing systems* 33 (2020), 10383–10395.

[5] Jeffrey L Elman. 1990. Finding structure in time. *Cognitive science* 14, 2 (1990), 179–211.

[6] Jiarui Feng, Yixin Chen, Fuhai Li, Anindya Sarkar, and Muhan Zhang. 2022. How powerful are k-hop message passing graph neural networks. *Advances in Neural Information Processing Systems* 35 (2022), 4776–4790.

[7] Shubham Gupta and Srikanta Bedathur. 2022. A survey on temporal graph representation learning and generative modeling. *arXiv preprint arXiv:2208.12126* (2022).

[8] Torben Hagerup, Kurt Mehlhorn, and J Ian Munro. 1993. Maintaining discrete probability distributions optimally. In *Automata, Languages and Programming: 20th International Colloquium, ICALP 93 Lund, Sweden, July 5–9, 1993 Proceedings 20*. Springer, 253–264.

[9] William L Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive Representation Learning in Large Attributed Graphs. In *30th Advances in Neural Information Processing Systems*.

[10] William L Hamilton, Rex Ying, and Jure Leskovec. 2017. Representation learning on graphs: Methods and applications. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* (2017).

[11] Guanhao Hou, Qintian Guo, Fangyuan Zhang, Sibo Wang, and Zhewei Wei. 2023. Personalized PageRank on Evolving Graphs with an Incremental Index-Update Scheme. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–26.

[12] Kexin Huang and Marinka Zitnik. 2020. Graph meta learning via local subgraphs. *Advances in neural information processing systems* 33 (2020), 5862–5874.

[13] Shenyang Huang, Farimah Poursafaei, Jacob Danovitch, Matthias Fey, Weihua Hu, Emanuele Rossi, Jure Leskovec, Michael Bronstein, Guillaume Rabusseau, and Reihaneh Rabbany. 2024. Temporal graph benchmark for machine learning on temporal graphs. *Advances in Neural Information Processing Systems* 36 (2024).

[14] Glen Jeh and Jennifer Widom. 2003. Scaling personalized web search. In *Proceedings of the 12th international conference on World Wide Web*. 271–279.

[15] Ming Jin, Yuan-Fang Li, and Shirui Pan. 2022. Neural temporal walks: Motif-aware representation learning on continuous-time dynamic graphs. *Advances in Neural Information Processing Systems* 35 (2022), 19874–19886.

[16] Seyed Mehran Kazemi, Rishab Goel, Kshitij Jain, Ivan Kobyzev, Akshay Sethi, Peter Forsyth, and Pascal Poupart. 2020. Representation learning for dynamic graphs: A survey. *The Journal of Machine Learning Research* 21, 1 (2020), 2648–2720.

[17] Shima Khoshraftar and Aijun An. 2022. A survey on graph representation learning methods. *arXiv preprint arXiv:2204.01855* (2022).

[18] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *5th International Conference on Learning Representations*.

[19] Lecheng Kong, Yixin Chen, and Muhan Zhang. 2022. Geodesic graph neural network for efficient graph representation learning. *Advances in Neural Information Processing Systems* 35 (2022), 5896–5909.

[20] Srijan Kumar, Xikun Zhang, and Jure Leskovec. 2019. Predicting dynamic embedding trajectory in temporal interaction networks. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*. 1269–1278.

[21] Pan Li, Yanbang Wang, Hongwei Wang, and Jure Leskovec. 2020. Distance encoding: Design provably more powerful neural networks for graph representation learning. *Advances in Neural Information Processing Systems* (2020).

[22] Yiming Li, Yanyan Shen, Lei Chen, and Mingxuan Yuan. 2023. Zebra: When Temporal Graph Neural Networks Meet Temporal Personalized PageRank. *Proceedings of the VLDB Endowment* 16, 6 (2023), 1332–1345.

[23] Xiao Liu, Shunmei Meng, Qianmu Li, Lianyong Qi, Xiaolong Xu, Wanchun Dou, and Xuyun Zhang. 2023. SMEF: Social-aware Multi-dimensional Edge Features-based Graph Representation Learning for Recommendation. In *Proceedings of the 32nd ACM International Conference on Information and Knowledge Management*.

[24] Xin Liu, Haojie Pan, Mutian He, Yangqiu Song, Xin Jiang, and Lifeng Shang. 2020. Neural Subgraph Isomorphism Counting. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 1959–1969.

[25] Yunyu Liu, Jianzhu Ma, and Pan Li. 2022. Neural Predicting Higher-Order Patterns in Temporal Networks. In *Proceedings of the ACM Web Conference 2022*. 1340–1351.

[26] Yuhong Luo and Pan Li. 2022. Neighborhood-aware scalable temporal network representation learning. In *Learning on Graphs Conference*. PMLR.

[27] Yossi Matias, Jeffrey Scott Vitter, and Wen-Chun Ni. 2003. Dynamic generation of discrete random variates. *Theory of Computing Systems* 36 (2003), 329–358.

[28] Long Short-Term Memory. 2010. Long short-term memory. *Neural computation* 9, 8 (2010), 1735–1780.

[29] Gaspard Michel, Giannis Nikolentzos, Johannes F Lutzeyer, and Michalis Vazirgiannis. 2023. Path neural networks: Expressive and accurate graph neural networks. In *International Conference on Machine Learning*. PMLR, 24737–24755.

[30] Dingheng Mo and Siqiang Luo. 2021. Agenda: Robust personalized PageRanks in evolving graphs. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*. 1315–1324.

[31] Frank Olken and Doron Rotem. 1995. Random sampling from databases: a survey. *Statistics and Computing* 5 (1995), 25–42.

[32] Pietro Panzarasa, Tore Opsahl, and Kathleen M Carley. 2009. Patterns and dynamics of users' behavior and interaction: Network analysis of an online community. *Journal of the American Society for Information Science and Technology* 60, 5 (2009), 911–932.

[33] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[34] Emanuele Rossi, Ben Chamberlain, Fabrizio Frasca, Davide Eynard, Federico Monti, and Michael Bronstein. 2020. Temporal graph networks for deep learning on dynamic graphs. *arXiv preprint arXiv:2006.10637* (2020).

[35] Balasubramaniam Srinivasan and Bruno Ribeiro. 2020. On the equivalence between positional node embeddings and structural graph representations.

[36] Nishil Talati, Di Jin, Haojie Ye, Ajay Brahmakshatriya, Ganesh Dasika, Saman Amarasinghe, Trevor Mudge, Danai Koutra, and Ronald Dreslinski. 2021. A Deep Dive Into Understanding The Random Walk-Based Temporal Graph Learning. In *2021 IEEE International Symposium on Workload Characterization (IISWC)*.

[37] Komal Teru, Etienne Denis, and Will Hamilton. 2020. Inductive relation prediction by subgraph reasoning. In *International Conference on Machine Learning*. PMLR, 9448–9457.

[38] Xuhong Wang, Ding Lyu, Mengjian Li, Yang Xia, Qi Yang, Xinwen Wang, Xinguang Wang, Ping Cui, Yupu Yang, Bowen Sun, et al. 2021. Apan: Asynchronous propagation attention network for real-time temporal graph embedding. In *Proceedings of the 2021 international conference on management of data*. 2628–2638.

[39] Yanbang Wang, Yen-Yu Chang, Yunyu Liu, Jure Leskovec, and Pan Li. 2021. Inductive representation learning in temporal networks via causal anonymous walks. *International Conference on Learning Representations* (2021).

[40] Cheng Wu, Chaokun Wang, Jingcao Xu, Ziwei Fang, Tiankai Gu, Changping Wang, Yang Song, Kai Zheng, Xiaowei Wang, and Guorui Zhou. 2023. Instant Representation Learning for Recommendation over Large Dynamic Graphs. *IEEE 39th International Conference on Data Engineering (ICDE)* (2023).

[41] Felix Wu, Amauri Souza, Tianyi Zhang, Christopher Fifty, Tao Yu, and Kilian Weinberger. 2019. Simplifying graph convolutional networks. In *International conference on machine learning*. PMLR, 6861–6871.

[42] Da Xu, Chuanwei Ruan, Evren Korpeoglu, Sushant Kumar, and Kannan Achan. 2020. Inductive representation learning on temporal graphs. *International Conference on Learning Representations* (2020).

[43] Pei-Kai Yeh, Hsi-Wen Chen, and Ming-Syan Chen. 2023. Random walk conformer: learning graph representation from long and short range. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 37. 10936–10944.

[44] Haoteng Yin, Muhan Zhang, Jianguo Wang, and Pan Li. 2023. SUREL+: Moving from Walks to Sets for Scalable Subgraph-based Graph Representation Learning. *Proceedings of the VLDB Endowment* 16, 11 (2023), 2939–2948.

[45] Haoteng Yin, Muhan Zhang, Yanbang Wang, Jianguo Wang, and Pan Li. 2022. Algorithm and System Co-design for Efficient Subgraph-based Graph Representation Learning. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2788–2796.

[46] Jiaxuan You, Tianyu Du, and Jure Leskovec. 2022. ROLAND: graph learning framework for dynamic graphs. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 2358–2366.

[47] Fangyuan Zhang, Mengxu Jiang, and Sibo Wang. 2023. Efficient Dynamic Weighted Set Sampling and Its Extension. *Proceedings of the VLDB Endowment* 17, 1 (2023), 15–27.

[48] Jiasheng Zhang, Jie Shao, and Bin Cui. 2023. StreamE: Learning to Update Representations for Temporal Knowledge Graphs in Streaming Scenarios. In *Proceedings of the 46th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 622–631.

[49] Muhan Zhang and Yixin Chen. 2018. Link prediction based on graph neural networks. *Advances in neural information processing systems* 31 (2018).

[50] Muhan Zhang and Yixin Chen. 2019. Inductive matrix completion based on graph neural networks.

[51] Muhan Zhang, Pan Li, Yinglong Xia, Kai Wang, and Long Jin. 2021. Labeling trick: A theory of using graph neural networks for multi-node representation learning. *Advances in Neural Information Processing Systems* 34 (2021), 9061–9073.

[52] Yanping Zheng, Zhewei Wei, and Jiajun Liu. 2023. Decoupled Graph Neural Networks for Large Dynamic Graphs. *Proceedings of the VLDB Endowment* (2023).
[53] Hongkuan Zhou, Da Zheng, Israt Nisa, Vasileios Ioannidis, Xiang Song, and George Karypis. 2022. Tgl: A general framework for temporal gnn training on billion-scale graphs. *Proceedings of the VLDB Endowment* (2022).
[54] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. 2019. Aligraph: A Comprehensive Graph Neural Network Platform. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2094–2105.