FlowWalker: A Memory-efficient and High-performance GPU-based Dynamic Graph Random Walk Framework

Junyi Mei[1], Shixuan Sun[1], Chao Li[1], Cheng Xu[1], Cheng Chen[2], Yibo Liu[1], Jing Wang[1], Cheng Zhao[2], Xiaofeng Hou[1], Minyi Guo[1], Bingsheng He[3], Xiaoliang Cong[2]
[1]Shanghai Jiao Tong University, [2]ByteDance Inc., [3]National University of Singapore
meijunyi@sjtu.edu.cn,sunshixuan@sjtu.edu.cn,lichao@cs.sjtu.edu.cn,jerryxu@sjtu.edu.cn
chencheng.sg@bytedance.com,liuyib@sjtu.edu.cn,jing618@sjtu.edu.cn,zhaocheng.127@bytedance.com
hou-xf@cs.sjtu.edu.cn,guo-my@cs.sjtu.edu.cn,hebs@comp.nus.edu.sg,congxiaoliang@bytedance.com

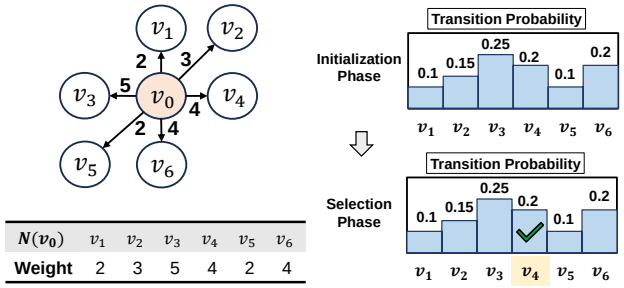
ABSTRACT

Dynamic graph random walk (DGRW) emerges as a practical tool for capturing structural relations within a graph. Effectively executing DGRW on GPU presents certain challenges. First, existing sampling methods demand a pre-processing buffer, causing substantial space complexity. Moreover, the power-law distribution of graph vertex degrees introduces workload imbalance issues, rendering DGRW embarrassed to parallelize. In this paper, we propose FlowWalker, a GPU-based dynamic graph random walk framework. FlowWalker implements an efficient parallel sampling method to fully exploit the GPU parallelism and reduce space complexity. Moreover, it employs a sampler-centric paradigm alongside a dynamic scheduling strategy to handle the huge amounts of walking queries. FlowWalker stands as a memory-efficient framework that requires no auxiliary data structures in GPU global memory. We examine the performance of FlowWalker extensively on ten datasets, and experiment results show that FlowWalker achieves up to 752.2×, 72.1×, and 16.4× speedup compared with existing CPU, GPU, and FPGA random walk frameworks, respectively. Case study shows that FlowWalker diminishes random walk time from 35% to 3% in a pipeline of ByteDance friend recommendation GNN training. The source code of FlowWalker can be found at https://github.com/junyimei/flowwalker-artifact.

1 INTRODUCTION

Random walk (RW) is a practical approach to extract graph information and is widely used in real-world applications such as social network analysis [14], recommendation systems [41], and knowledge graphs [22]. Take the friend recommendation in Douyin (a popular social media developed by ByteDance) as an example. In the recommendation graph, vertices represent users, and edges depict diverse user interactions such as co-liking, co-favoring, etc. RW is used to generate random walk sequences serving the Graph Neural Network (GNN) [31, 38, 51, 52] tasks for personalized friend recommendations. However, the computational demands of RW are substantial. For instance, on a recommendation graph snapshot with 227 million users and 2.71 billion edges, RW takes up to 3.5 hours, contributing to 35% of the end-to-end training duration. Since recommendation graphs are undergoing frequent updates, ensuring the prompt completion of the RW tasks becomes vital for maintaining service quality. Consequently, there is an urgent need to accelerate RW computations.

Recognizing the significance of the problem, researchers have conducted comprehensive studies [35, 45, 49] to parallelize RW on multi-core CPUs. Some works modify state-of-the-art graph processing frameworks to support RW algorithms, but they treat RW the same as traditional graph algorithms and ignore its unique properties [16, 18, 42]. Thus specialized graph sampling frameworks have been proposed to maximize the overall sampling throughput. For instance, GraphWalker [45] introduces a partition-based method for out-of-core computation. ThunderRW [35] optimizes cache utilization to enhance in-memory computation. These frameworks work well in *static graph random walk* (SGRW) such as DeepWalk [29], where the transition probability remains constant. Specifically, they execute SGRW in two phases: 1) a preprocessing phase that computes the transition probability table for each vertex, and 2) a computation phase that runs random walk queries. As shown in Figure 1, this approach greatly diminishes the sampling cost [35] by avoiding the initialization of the probability table at every step. However, this preprocessing strategy cannot process *dynamic graph random walks* (DGRW), where transition probabilities are dynamically determined during runtime as in Node2Vec [9] and MetaPath [36]. As a result, the computational complexity surges in DGRW as each step requires scanning the neighbors to calculate the transition probability table. For instance, ThunderRW can execute DeepWalk on the previously discussed recommendation graph in approximately 150 seconds with the preprocessing strategy; however, it exceeds an eight-hour time limit when running Node2Vec.



(a) An example graph. **(b) Sampling a neighbor of v_0.**

Figure 1: The procedure for sampling a neighbor of v_0.

Recently, DGRW has gained popularity over SGRW due to its ability to capture temporal structure relations (i.e., the state of each query), rendering it a more powerful tool [9, 37]. Researchers have turned to GPU acceleration to enhance DGRW performance leveraging their high-bandwidth on-board memory and massive

computing power. For example, C-SAW [27] parallelizes *inverse transform sampling* [26] on GPU, and Skywalker [43] proposes a GPU-based *alias table sampling* [40] method. However, we uncover several fundamental limitations in existing GPU-based frameworks that lead to significant performance constraints.

First, these frameworks require extensive memory space to facilitate the query execution. They necessitate an $O(d)$ memory buffer to store the transition probability table for each query, where d denotes the degree of the vertex that is being sampled. Since dynamic memory allocation can be costly, they opt to pre-allocate a buffer with $O(d_{max})$ size where d_{max} denotes the maximum vertex degree in the graph. This approach can consume vast amounts of memory, especially when dealing with real-world graphs characterized by significant skewness. In the case where d_{max} in *twitter* reaches 3×10^6, a buffer size of around 11.45 MB is required for every single query. Though GPUs offer powerful computing capabilities, the limited memory space restricts concurrent parallelism (i.e., queries processed simultaneously) and reduces available space for graph data.

Second, these frameworks disregard the load imbalance issue emanating from both workload and hardware characteristics. The workload at each step is governed by the vertex degree, and the degree skewness among vertices can lead to workload imbalance. Besides, despite that RW is embarrassingly parallel, the concurrent execution capability of modern GPUs, which can support tens of thousands of threads, exacerbates the load imbalance problems among computing resources. While C-SAW overlooks these concerns, Skywalker handles sampling tasks of varying degrees with warps or blocks, which leads to burdensome memory costs as well as communication overhead.

In this paper, we introduce **FlowWalker**, a GPU-based DGRW framework that performs fast sampling at minimal memory cost. We design a *sampler-centric* computation model, which abstracts the computation from the hardware perspective. Under this model, an RW application is conceptualized as a set of discrete sampling tasks, where each task aims to randomly select a vertex from a specified vertex set. The GPU threads are systematically organized into a collection of samplers, which efficiently process these tasks. This abstraction narrows the RW computation down to two crucial problems: 1) devising efficient samplers; and 2) formulating an effective scheduling mechanism that assigns tasks to the samplers according to workload characteristics.

Inspired by sampling on streams, we design a parallel sampling method based on the *reservoir sampling* technique [39]. This method is tailored for GPU optimization and is sufficiently adapted to handle vertices with varied degrees. Our design significantly reduces the space complexity of handling a sampling task from $O(d)$ to $O(1)$, thereby facilitating the concurrent execution of a substantial number of tasks. Coupled with efficient samplers, we develop a high-performance processing engine based on a multi-level task pool that distributes tasks among the samplers. Benefiting from its sampler design and processing engine, FlowWalker attains notable memory efficiency, with no auxiliary data structures in the global memory to streamline computation. Thereby, FlowWalker effectively tackles the challenges of limited query concurrency and load imbalance, optimizing the utilization of computational resources.

We showcase the generality of FlowWalker by implementing four representative algorithms, including DeepWalk [29], PPR [8], Node2Vec [9], and MetaPath [36]. We compare performance against ThunderRW [35], the state-of-the-art CPU-based framework; Skywalker [43], a GPU-based approach; DGL [42], the widely used GNN framework; and LightRW [37], the state-of-the-art FPGA-based approach. We conduct extensive experiments on ten real-world graphs, the size of which scale from millions to billions. Experiment results show that 1) FlowWalker stands as the sole GPU-based solution that able to support all of the four algorithms above; 2) FlowWalker consistently completes all test cases within a time frame of 2.2 hours, achieving up to 752.2× speedup over competitors, whereas DGL, LightRW, ThunderRW and Skywalker either exceed an eight-hour limit or encounter memory constraints; and 3) FlowWalker has negligible memory cost by getting rid of auxiliary data structures. In summary, we make the following contributions in this paper:

- We introduce FlowWalker, a memory-efficient and high-performance GPU-based random walk framework, which leverages a sampler-centric computation model.
- We propose an efficient parallel sampling method for GPU based on reservoir sampling. This method greatly diminishes the space complexity, thereby substantially accelerating the sampling process.
- We develop a concise scheduling mechanism to efficiently channel a vast number of fine-grained tasks through samplers of different granularities. This mechanism enhances overall efficiency and adaptability.

Paper Organization. Section 2 introduces backgrounds. Section 3 gives an overview of the system. Sections 4 and 5 elaborate on the sampling method and computation engine, respectively. Section 6 details our experiment as well as case study. Section 7 concludes this paper.

2 BACKGROUND

We introduce the preliminary and the background related to our work in this section.

2.1 Graph Random Walk

Let $G = (V, E)$ denote a directed graph where V is the set of vertices and E is the set of edges. Given a vertex $v \in V$, $N(v)$ is the neighbor set of v and $d(v)$ is the degree, i.e., $|N(v)|$. Given an edge $e(u, v) \in E$, $w(u, v)$ and $l(u, v)$ represent its weight and label respectively.

Algorithm 1 presents a common RW computation paradigm. An RW algorithm has a set $\mathbb{Q}$ of random walk queries. A query Q begins at a start vertex. At each step, Q randomly selects a neighbor u of the current residing vertex $Q.cur$ and moves to it. The operation is performed in two phases: 1) the *initialization phase* calculates the *transition probability* $p(u)$ for each neighbor u; and 2) the *selection phase* randomly picks a neighbor given the distribution. Q records the walk sequence in $Q.seq$ and stops until meets a specified condition, for example, $Q.seq$ reaches a length threshold. The outputs are the query sequences. Assume that the current residing vertex is $Q.cur = v$. The selection of a neighbor involves sampling u from $N(v)$ based on a transition probability $p(u)$, determined by a weight function f applied to the edge $e(v, u)$. For instance, if we

Algorithm 1: Random Walk Computation Paradigm

Input: a graph G and a set $\mathbb{Q}$ of RW queries;
Output: the sequence of each query $Q \in \mathbb{Q}$;

```
1 for Q ∈ ℚ do
2     do
          /* The initialization phase.                    */
3         foreach u ∈ N(Q.cur) do
4             Calculate u's transition probability p(u);
          /* The selection phase.                         */
5         Select a u ∈ N(Q.cur) given p(u) and add it to Q.seq;
6     while Stop(Q) is false;
7 return ℚ.seq;
```

define $f(e(v,u)) = w(v,u)$, then $p(u) = \frac{w(v,u)}{\sum_{u' \in N(v)} w(v,u')}$, which is a normalized value. To simplify the presentation, we refer to the transition probability $p(u)$ as the relative chance (e.g., the edge weight $w(v,u)$) of u being selected without normalization in the subsequent discussions.

Graph random walk algorithms are broadly divided into two categories based on the transition probability property: *static graph random walk* (SGRW) and *dynamic graph random walk* (DGRW). In SGRW applications like DeepWalk and PPR, the transition probability is fixed throughout the computation. This allows for calculating values in a pre-processing stage (as discussed in Section 1), which significantly reduces computational complexity by eliminating the initialization phase in Algorithm 1. In contrast, the transition probability of DGRW relies on the query states and requires determination during runtime. Consequently, the initialization is postponed to the computation step. Next, we will introduce two representative DGRW algorithms.

MetaPath [36] is a widely used algorithm for representation learning in heterogeneous networks [7]. Within MetaPath, an edge label schema $l_1 \rightarrow \ldots \rightarrow l_i \ldots \rightarrow l_k$ constrains the walk sequence $Q.seq$ of a random walk query. Specifically, the labels of adjacent vertices in the sequence must align with the schema, i.e., $l(Q.seq[i], Q.seq[i+1]) = l_i$. Suppose the current residing vertex is $Q.cur = v$, where v is the i-th vertex in $Q.seq$. The transition probability for selecting a neighbor $u \in N(v)$ is defined by Equation 1. The weighted version of MetaPath incorporates the edge weight into the calculation by multiplying it with the transition probability $p(u)$.

$$p(u) = \begin{cases} 1, & \text{if } l(v,u) = l_i, \\ 0, & \text{otherwise.} \end{cases} \tag{1}$$

Node2Vec [9] is a second-order RW algorithm, where the transition probability is dependent on the last visited vertex. Assuming that $Q.cur$ is v, then the transition probability $p(u)$ for selecting a neighbor $u \in N(v)$ is governed by Equation 2, in which v' represents the last visited vertex before v and $dist(v',u)$ denotes the distance between v' and u. a and b are two hyperparameters that modulate the random walk behavior. Similar to MetaPath, the edge weight $w(v,u)$ can be factored into the computation by multiplying it with the computed transition probability $p(u)$.

$$p(u) = \begin{cases} \frac{1}{a}, & \text{if } dist(v',u) = 0, \\ 1, & \text{if } dist(v',u) = 1, \\ \frac{1}{b}, & \text{if } dist(v',u) = 2, \\ 0, & \text{otherwise.} \end{cases} \tag{2}$$

In addition to Node2Vec and MetaPath, methods such as Hetespaceywalk [10] exemplify the application of DGRW. Representation learning methods on Heterogeneous Information Networks (HINs) [19, 34] are typically grounded in DGRW, necessitating consideration of label information—akin to the MetaPath approach. DGRW is also used for similarity measurement [21, 47] and community detection [1, 6]. In ByteDance, there are massive graphs with vertex labels such as users, videos, and advertisement items. Taking the advertisement recommendation scenario in Douyin as an example, we need to generate random walk sequences for each user and advertisement item based on specific meta-paths, such as user-item-user. Subsequently, the embeddings are trained to serve as inputs for the recommendation models. The practical necessity for dynamic walk algorithms in real-world business scenarios has motivated us to commence work on FlowWorker.

2.2 Sampling Methods

In the context of our study, sampling is the process of selecting a vertex u from a neighbor set $N(v)$ based on the transition probability distribution. Different frameworks implement this operation through various sampling methods. ThunderRW, for example, offers *inverse transform sampling* (ITS) [26], *rejection sampling* (RJS) [30], and *alias table sampling* (ALS) [12, 40], allowing users to choose the method most suitable for the algorithm's property. C-SAW [27] and Skywalker [43] utilize ITS and ALS, respectively. However, both methods require an $O(d)$-sized memory buffer to store the transition probability, which, as discussed in Section 1, consumes substantial memory and can lead to significant performance issues. Contrastingly, RJS requires only $O(1)$ space to store the maximum transition probability, employing a "trial-and-error" selection approach. However, this method comes with its drawbacks: the non-deterministic running time of randomized selection is heavily affected by the underlying probability distribution, and the process leads to numerous random memory accesses. These factors make RJS challenging to implement efficiently on GPUs.

Contrary to other methods, *reservoir sampling* (RS) [4, 39] is tailored for sampling streaming data. As outlined in Algorithm 2, RS operates on a vertex sequence S with length n. W_P maintains the prefix sum of weights, and *selected* stores the index of the vertex chosen from S. Upon encountering a vertex at position i, RS updates W_P and generates a random number. If this number is smaller than the transition probability $\frac{W[i]}{W_P}$, RS updates the *selected* index accordingly (Line 4). Ultimately, RS returns the last selected vertex. Notably, the space complexity of RS is $O(1)$, and the time complexity is $O(d)$ given a neighbor set $N(v)$ with d vertices as the input. While both ITS and ALS require only a single random number, RS necessitates generating a random number for each element. Although this might pose a challenge for CPUs, it is well-suited for GPUs, which offer ample computational resources.

Algorithm 2: Sequential Weighted Reservoir Sampling

Input: a vertex sequence S, the corresponding weight sequence W, the sequence length n;
Output: a vertex sampled from S based on W;

```
W_P ← 0, selected ← 0;
for i ← 1 to n do
    W_P ← W_P + W[i];
    if RANDOM(0,1) < W[i]/W_P then selected ← i;
return S[selected];
```

2.3 GPU-based Random Walk Frameworks

Researchers have proposed several works to accelerate RW applications using GPUs. NextDoor [13] is a graph sampling framework utilizing the RJS sampling method. It adopts the offline computation mode, which calculates the maximum weight for a neighbor set during the pre-processing stage. When executing random walk queries, NextDoor only performs the selection phase of the sampling. Therefore, NextDoor cannot support variant DGRW applications. Note that NextDoor implements unweighted Node2Vec by choosing the maximum value from $(1, \frac{1}{a}, \frac{1}{b})$ to bypass the initialization phase. The implementation cannot be generalized to weighted Node2Vec and other DGRW applications such as weighted MetaPath. During runtime, NextDoor follows the BSP [5] model, advancing all queries by a single step at a time. NextDoor, which can sample multiple vertices from a neighbor set, ensures load balance by allocating threads according to the number of sampling results.

Distinct from NextDoor [13], C-SAW [27] supports DGRW and employs the ITS sampling method. It adopts a query-centric computation model, assigning each query to a warp and executing them synchronously in a step-by-step fashion using the BSP [5] model. Although C-SAW optimizes ITS for GPUs to speed up computations, it falls short in supporting queries with variable walk lengths, such as PPR, due to its synchronized execution approach.

Skywalker [43, 44] parallelizes the ALS sampling methods and optimizes memory access by compressing alias tables. To address the load imbalance caused by varying vertex degrees, Skywalker employs versatile samplers tailored to vertices with different degrees. To further mitigate load imbalance among thread blocks, it introduces a queue to distribute queries across blocks. As queries can have different lengths, the space complexity of the queue is $O(L_{max} \times |\mathbb{Q}|)$ where L_{max} is the maximum length of queries.

Despite these advancements, both C-SAW and Skywalker possess foundational limitations, as discussed in Section 1, which restrict their efficiency in handling large graphs. Besides, frameworks like GraSS [50] focus on graph compression. This technique is complementary to our work and can be integrated with FlowWalker to further minimize memory usage.

2.4 Other Related Works

Given the critical role of Random Walk (RW) applications, numerous studies have focused on optimizing CPU-based graph random walk frameworks. NosWalker [46], DrunkardMob [17], and GraphWalker [45] are designed to handle graphs that exceed available memory. ThunderRW [35] optimizes in-memory computation by improving cache utilization. KnightKing [49] and FlashMob [48] are distributed frameworks that address communication and memory bandwidth utilization. Nevertheless, all these frameworks are optimized for SGRW, though some of them (e.g., ThunderRW) can execute DGRW. Additionally, research efforts have been made to optimize memory usage for random walks on both static and streaming graphs [28, 33].

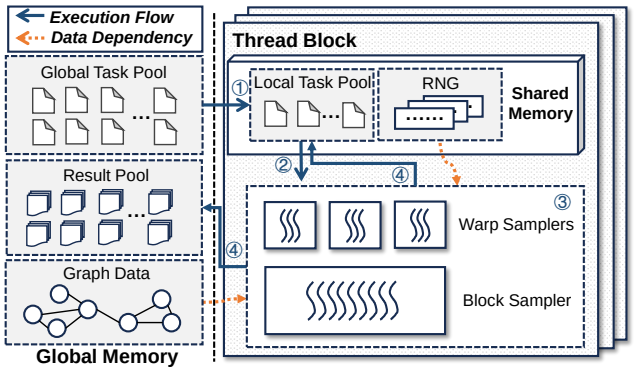


Figure 2: System Design Overview of FlowWalker. The execution flow is organized as follows: ① Thread blocks fetch tasks from the global task pool into its local task pool. ② Tasks are dispatched to the appropriate sampler based on the vertex degree. ③ Warp and block samplers execute the sampling tasks. The process necessitates graph data stored in the global memory and random number generators (RNG) stored in the shared memory. ④ The query states in the local task pool are updated and the sampling results are recorded.

Recently, Tan et al. [37] introduce an FPGA-based approach to accelerate DGRW. They develop a parallel reservoir sampling method on FPGAs, akin to Algorithm 3. Despite the similarities, the fundamental differences in the underlying hardware architectures set our approaches apart. LightRW's emphasis lies in customizing hardware to optimize pipeline execution and memory access during sampling. In contrast, GPU architectures are fixed, with threads grouped into thread blocks at runtime. Our approach involves a meticulous exploration of the design space to adapt reservoir sampling to the unique demands of GPU workloads and hardware characteristics. The inherent distinctions in hardware architectures influence our respective sampling algorithms, system designs, and research focuses.

3 AN OVERVIEW OF FLOWWALKER

Computation Model. Different from the query-centric model, we propose the *sampler-centric* model that abstracts the computation from the hardware perspective. Specifically, an RW application consists of massive random walk queries each of which is a sequence of steps. A step performs one sampling operation, which selects a neighbor from the neighbor set of the current residing vertex and updates the query. Therefore, an RW application can be viewed as a set of sampling tasks. The computation on GPUs is to organize threads to a set of samplers to perform these sampling tasks efficiently until all queries are complete.

System Design. Based on the sampler-centric model, we design FlowWalker, a memory-efficient and high-performance GPU-based

DGRW framework. We propose a parallel reservoir sampling method that can perform the sampling with $O(1)$ memory cost. Besides, an efficient computation engine is implemented to guide global task scheduling and computation inside a thread block.

Figure 2 gives an overview of our system design. In FlowWalker, a thread block is an independent worker whose threads are organized into samplers with different parallelism. Particularly, given a set of sampling tasks, a thread block adopts a two-stage execution scheme to handle variant workloads among these tasks. In the first stage, threads are organized into warp samplers (i.e., a warp works as a sampler) to process small tasks. In the second stage, all threads in the same thread block form a block sampler (i.e., a block works as a sampler) to handle large tasks. A multi-level task pool based dynamic scheduling mechanism is adopted to keep load balance among computing resources. A thread block has a local task pool that maintains the queries assigned to it. Once a query is completed, it will fetch a new query from the global task pool. The fine-grained scheduling method requires no communication and synchronization among blocks and achieves good load balance. Additionally, it gets rid of auxiliary data structures in the global memory, and a small amount of intermediate data can be held inside the shared memory, which is a type of fast-speed GPU memory. In terms of APIs, our framework adheres to the conventions established by prior works [27, 35, 43, 45]. Therefore, we omit the details for brevity.

Benefiting from the designs mentioned above, FlowWalker is able to perform memory-efficient sampling with no data structures stored in the global memory to assist the execution. This significantly benefits GPU-based RW because GPUs have abundant computing resources but limited memory space. We will introduce the sampling method and the engine in Sections 4 and 5, respectively.

4 SAMPLING METHOD

Under the sampler-centric abstraction, sampling is the key operation in RW applications. As discussed in Section 2, existing methods [26, 30, 40] have severe performance issues on GPUs due to the large memory consumption of the intermediate data. Inspired by stream processing, we model the problem of choosing a neighbor as that of sampling an element from a stream. Therefore, we can adopt reservoir sampling (RS) to reveal the memory consumption issue because RS does not maintain a state for each element.

4.1 Direct Parallel Reservoir Sampling

Given a sequence S of n vertices, the corresponding weights W and a group of k threads (e.g., a warp), our goal is to parallel reservoir sampling which selects a vertex v from S based on W. Moreover, we want to keep k threads having coalesced memory access patterns to fully utilize GPUs. Recall that reservoir sampling scans S along the sequence order with the probability of replacing the selected vertex with v_i as $\frac{w_v}{W_i}$ where v_i is the ith vertex in S, w_v is the weight of v, and $W_i = \sum_{j=1}^{i} W[j]$ (i.e., the sum of weights of vertices before v_i in S). If v_i is picked, then we replace the selected vertex with v_i. Reservoir sampling returns the last selected vertex as the sampling result.

A straightforward idea of parallelization is to sample a vertex from k consecutive vertices in parallel in each iteration and repeat until all vertices are processed. We call this method the *direct parallel reservoir sampling* (DPRS) algorithm. Algorithm 3 depicts the details. In a certain iteration (Lines 4-11), we first read weights from W for k vertices in parallel with thread j holding value $W_L[j]$. Next, we compute the prefix sum W_P for the k values in parallel. w_B maintains the sum of weights in previous iterations, i.e. vertices from $S[1]$ to $S[i{\times}k]$. Therefore, thread j selects the vertex $S[j{+}i{\times}k]$ with the probability $\frac{W_L[j]}{W_P[j]+w_B}$ (Line 9). We then set the selected index to the maximum value in C (i.e., the maximum sequence index selected by these k threads) and update w_B (Lines 10-11). Finally, we return the sampled vertex given the index (Line 12). Note that returning $S[0]$ denotes that no vertex is selected, for example, no label can match the constraint in MetaPath.

Example 4.1. Figure 3 presents a running example of DPRS where $n = 6$ and $k = 3$. At Iteration 1, threads T_{1-3} first load weights of v_{1-3} in parallel and then compute their prefix sum. After that, they perform the selection independently. For example, T_1 sets the selected index C to 1 since the random number value $r = 0.5$ is less than $\frac{W_L}{W_P+w_B} = 1.0$. At the end of the iteration, DPRS performs a parallel reduction to get the last selected index (i.e., the maximum C among T_{1-3}), which is the selected item at this iteration. Additionally, DPRS sets w_B to 10, which is the W_P value held by T_3. DPRS continues its computations until all elements have been processed. The result is 4 and the selected vertex is v_4. The parallel sampling order is equivalent to the order of S.

Analysis. Given the vertex $v = S[j + i \times k]$, thread j updates the selected vertex with the probability of $\frac{w_v}{\sum_{m=1}^{j+i\times k} W[m]}$. The max operation keeps the algorithm to return the last picked vertex. Therefore, Algorithm 3 intuitively has the same logic as Algorithm 2 though it runs in parallel, and Proposition 1 holds.

Proposition 1. *Given a sequence S of vertices and the corresponding weight sequence W, Algorithm 3 picks v with the probability $\frac{w_v}{\sum W}$ where w_v is the weight of v.*

Next, we analyze the time cost of Algorithm 3. Suppose that the cost of obtaining $W[i]$ is α, that of communication among threads is β, and that of random number generation is γ. In Algorithm 3, Line 6 accesses global memory, Lines 7 and 10 perform the parallel collective operations among k threads, and Line 9 computes a random number in each thread. Therefore, the cost at one iteration is $\alpha + 2 \times \beta \log k + \gamma$. The time cost of the algorithm is $\lceil \frac{n}{k} \rceil \times (\alpha + 2 \times \beta \log k + \gamma)$. The time complexity is $O(\frac{n}{k} \times \log k)$, and the speedup over Algorithm 2 is $O(\frac{k}{\log k})$.

Finally, we discuss the space complexity of Algorithm 3. In addition to storing S and W, we do not maintain a state for each vertex, while each thread only requires several local variables (W_P, C, and w_B, etc.). Therefore, the space complexity of the algorithm is $O(k)$ and that for one thread is $O(1)$.

4.2 Zig-Zag Parallel Reservoir Sampling

Although DPRS accesses the global memory in a coalesced pattern, we find that DPRS can have performance issues when processing long vertex sequences. Specifically, a GPU thread group has a limited number of threads, for example, a warp has 32 threads. Consequently, given a long vertex sequence (e.g., millions of vertices),

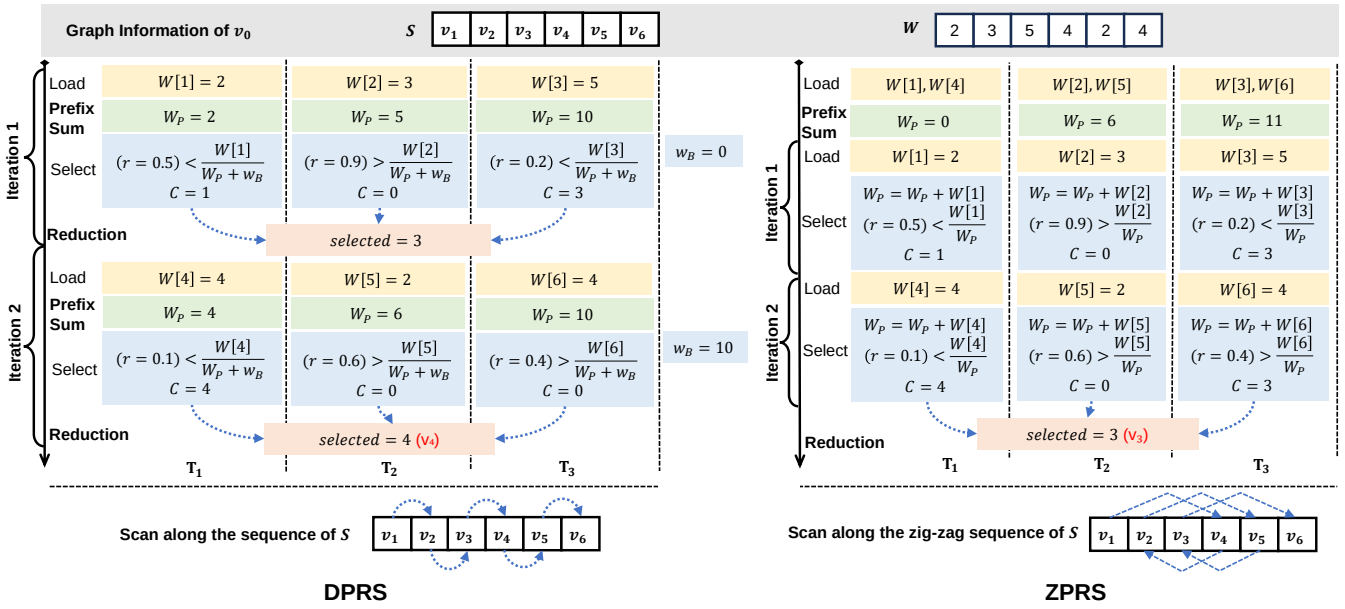


Figure 3: The comparison of DPRS and ZPRS on sampling a neighbor of v_0 in Figure 1a using three threads. DPRS scans W once, but the number of collective operations depends on the number of iterations. ZPRS performs two collective operations only, but scans W twice. Logically, DPRS scans along the sequence of S, whereas ZPRS scans in a zig-zag order of S.

Algorithm 3: Direct Parallel Reservoir Sampling(DPRS)

Input: a vertex sequence S, the corresponding weight sequence W, the sequence length n and k threads;
Output: a vertex sampled from S based on W;

1 **parallel for** $j \leftarrow 1$ *to* k **do**
2 $C[j] \leftarrow 0, W_L[j] \leftarrow 0, W_P[j] \leftarrow 0$;
3 $w_B \leftarrow 0$;
4 **for** $i \leftarrow 0$ *to* $\lceil \frac{n}{k} \rceil - 1$ **do**
5 **parallel for** $j \leftarrow 1$ *to* k **do**
6 $W_L[j] \leftarrow W[j + i \times k]$;
7 $W_P \leftarrow$PARALLEL_INCLUSIVE_PREFIX_SUM(W_L, k);
8 **parallel for** $j \leftarrow 1$ *to* k **do**
9 **if** $RANDOM(0,1) < \frac{W_L[j]}{W_P[j]+w_B}$ **then** $C[j] \leftarrow j + i \times k$;
 /* Get the maximum value in C. */
10 $selected \leftarrow$PARALLEL_REDUCTION(C, k);
11 $w_B \leftarrow w_B + W_P[k]$;
12 **return** $S[selected]$;

DPRS frequently performs parallel collective operations that incur expensive costs due to communication overhead among threads. As real-world graphs have vertices with large degrees and processing these vertices dominates the random walk cost, the performance issue degrades the computation speed.

To solve the problem, we design the *zig-zag parallel reservoir sampling* (ZPRS), which not only has coalesced memory access patterns but also reduces the number of parallel collective operations. In particular, different from DPRS scanning and sampling vertices along the order of S, ZPRS scans vertices along the order but samples in a zig-zag order S'. Algorithm 4 presents the details. S can be divided into k sets where $S_j = \{v_m \in S | m \mod k = j\}$. We first compute the weight sum for vertices in S_j and store the value to $W_L[j]$ (Lines 3-5). Next, we compute the exclusive prefix sum on W_L such that $W_P[j] = \sum_{m=1}^{j-1} \sum_{v \in S_m} w_v$. After that, thread j replaces the selected vertex with v in the probability $\frac{w_v}{W_P[j]}$. To pick the last sampled vertex, we select the last item that is greater than 0 in C in parallel (Line 11).

Example 4.2. Figure 3 presents a running example of ZPRS where $n = 6$ and $k = 3$. Threads T_{1-3} first load six weights in parallel at two iterations and then calculate the exclusive prefix sum. As this procedure is simple, we omit the details of the two iterations and directly show W_P values. After that, T_{1-3} performs the sampling independently. For example, at Iteration 1, T_3 first loads $W[3]$ and then sets the selected index C to 3 because the random number $r = 0.2$ is less than $\frac{W[3]}{W_P} = 0.31$. After processing all elements, T_{1-3} performs a parallel reduction to get the last C value such that $C > 0$. The result is 3 and the selected item is v_3. The parallel sampling order is equivalent to along a zig-zag order of S.

Analysis. First, we prove Proposition 2 based on the correctness of Algorithm 2, which is proved in the technical report.

Proposition 2. *Given a sequence S of vertices and the corresponding weight sequence W, Algorithm 4 picks v with the probability $\frac{w_v}{\sum W}$ where w_v is the weight of v.*

Proof. Consider a sequence S of n elements with corresponding weights W and k threads. Define S_i as a sub-sequence of S such that $S[j] \in S_i$ if $j \mod k = i$ for $1 \leqslant j \leqslant n$, and set $S_k = S_0$. This construction yields a new sequence $S' = (S_1, S_2, \ldots, S_k)$ and its associated weight sequence W'. As shown in Lines 7-10, each thread i processes S_i independently. Given $v = S_i[j]$, thread i

Algorithm 4: Zig-Zag Parallel Reservoir Sampling(ZPRS)

Input: a vertex sequence S, the corresponding weight sequence W, the sequence length n and k threads;

Output: a vertex sampled from S based on W;

1 **parallel for** $j \leftarrow 1$ *to* k **do**
2 $C[j] \leftarrow 0, W_L[j] \leftarrow 0, W_P[j] \leftarrow 0$;
3 **for** $i \leftarrow 0$ *to* $\lceil \frac{n}{k} \rceil - 1$ **do**
4 **parallel for** $j \leftarrow 1$ *to* k **do**
5 $W_L[j] \leftarrow W_L[j] + W[j + i \times k]$;
6 $W_P \leftarrow$PARALLEL_EXCLUSIVE_PREFIX_SUM(W_L, k);
7 **for** $i \leftarrow 0$ *to* $\lceil \frac{n}{k} \rceil - 1$ **do**
8 **parallel for** $j \leftarrow 1$ *to* k **do**
9 $W_P[j] \leftarrow W_P[j] + W[j + i \times k]$;
10 **if** $RANDOM(0, 1) < \frac{W[j+i\times k]}{W_P[j]}$ **then** $C[j] \leftarrow j + i \times k$;

/* Get the last item greater than 0 in C. */

11 $selected \leftarrow$PARALLEL_REDUCTION(C, k);
12 **return** $S[selected]$;

replaces its current selected vertex with a probability $\frac{W'[j]}{\sum_{l=1}^{j} W'[l]}$. Line 11 ensures that the element chosen by thread i is replaced by the selection of thread j if $i < j$. Consequently, parallel processing mirrors serial sampling along S'. By Proposition 1, each element $S'[i]$ is selected with probability $\frac{W'[i]}{\sum W'}$. So Proposition 2 holds. □

We next analyze the time cost of Algorithm 4. Compared with DPRS, ZPRS only requires two collective operations (Lines 6 and 11). In contrast, ZPRS scans the weight sequence twice (Lines 3-5 and 7-10). Therefore, the time cost of ZPRS is $\lceil \frac{n}{k} \rceil \times (2\times\alpha+\gamma)+2\times\beta \log k$. The time complexity is $O(\frac{n}{k} + \log k)$ and the speedup over the sequential method is $O(k \times (1 - \frac{k \log k}{n+k \log k}))$. When processing long sequences, ZPRS has a better speedup than DPRS and generally runs much faster than DPRS in practice, because modern GPUs have a big bandwidth and a large cache, e.g., A100 has 1.5-2 TB/s bandwidth and 40 MB L2 cache. But for the cases where the transition probability requires an expensive computation (i.e., α is high), ZPRS can run slower than DPRS in practice because it has to calculate the probability for each element twice. Experiment results in Section 6.3 confirm our analysis. The space complexity of ZPRS is $O(k)$, which is the same as DPRS.

4.3 Implementation

Both DPRS and ZPRS access global memory in a coalesced pattern. In their implementation, we focus on reducing the cost of collective operations β and that of random number generation.

In principle, both DPRS and ZPRS can be executed in parallel with any number of threads. However, in practice, modern GPUs manage threads with warps, blocks, and grids. Moreover, they only support efficient communication and synchronization for warps and blocks. Due to this constraint, we implement the warp and block samplers, which execute with one warp and one block, respectively. The parallel collective operations have been extensively studied [11, 15, 23, 32]. In our implementation, we use CUB [24] to conduct the prefix sum and reduction operations. Variables such as C, W_L, and W_P can be held with a register, and the collective calculation merely requires a shared memory buffer.

The cuRAND library [25] generates a random number by updating a `curandState`, which is a C struct containing a small integer array to record the generator state. As both DPRS and ZPRS generate a random number for each vertex in S, a simple method is to maintain an array of `curandState` for the warp (or block) with each thread having one state. However, this leads to uncoalesced global memory accesses. To resolve the issue, we transform the array of structures into a structure of arrays to optimize the memory access pattern. Similar to NextDoor [13], we store this structure in shared memory to further accelerate the computation. The optimization can bring up to 20.3× speedup in our experiment in Section 6.3. Investigating the efficient generation of massive random numbers (e.g., each thread has a random number generator) on GPUs constitutes a compelling topic for future study.

5 FLOWWALKER ENGINE

An RW application consists of massive random walk queries and each query is a sequence of walking steps. Steps from different queries can be processed independently, while steps from the same query have dependency. Under the sampler-centric computation model, threads in GPUs are organized into samplers and each step is a task unit. Specifically, given a step of a query, a sampler updates the query by selecting a neighbor of the current residing vertex. To process these tasks efficiently, we encounter two challenges caused by the workload and hardware properties. First, the workload of a step is determined by the degree of the current residing vertex. Due to degree skewness among vertices, workloads among different tasks are imbalanced. Second, although an RW application is embarrassingly parallel, modern GPUs support tens of thousands of threads executing concurrently, which leads to load imbalance issues among computing resources. Additionally, the communication and synchronization cost on GPUs is expensive.

In this section, we design an efficient walking engine on the top of our parallel reservoir samplers. In this engine, thread blocks are independent workers. Given a set of tasks, a thread block processes them by organizing its threads into different-level samplers (i.e., samplers with different threads) to handle variant workloads. Moreover, we design an effective scheduling mechanism based on multi-level task pools to keep load balance among workers. In the following, we will introduce the computation in a thread block, and then we will elaborate on the scheduling mechanism. Finally, the time and memory cost will be discussed.

5.1 Computation

To address workload imbalance, we can organize thread blocks to warp and block samplers and assign tasks to different thread blocks based on their degrees. However, under the query-centric model, a query needs to move between different thread blocks frequently. As the communication and synchronization cost among blocks is very expensive in GPUs, this approach can incur significant overhead. Therefore, instead of moving queries among different blocks, FlowWalker sticks a query to a thread block and processes tasks with variant workloads.

Figure 4 presents the computation in thread blocks. Each thread block has a *local task pool* P_L that maintains the queries assigned to it. An element in P_L stores the status of a query Q, which has

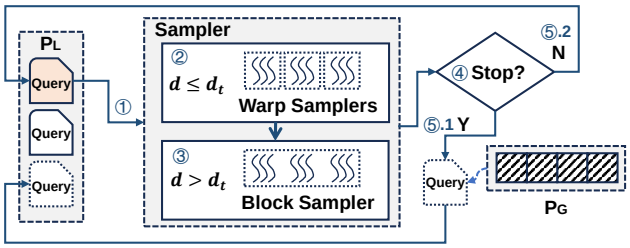


Figure 4: Computation in a thread block. A query will not be evicted from a thread block until stop conditions are met. Tasks are processed in two stages. First, warp samplers process tasks in which the degree of the current residing vertex is no greater than d_t. Then, the block sampler processes the remaining tasks. After sampling, if one query meets the stop conditions, a new query will be fetched from the global task pool (P_G) and added to the local task pool (P_L) (Step ⑤.1). Otherwise, we update the query state in P_L (Step ⑤.2).

the current residing vertex v, the degree $d(v)$, the location of $N(v)$, the location of the result sequence $Q.seq$, and the length $|Q.seq|$ of the sequence. P_L resides in shared memory because it is frequently accessed, while $N(v)$ and $Q.seq$ are stored in the global memory.

At the first stage, the thread block forms $\frac{|T|}{32}$ warp samplers to process the small tasks, the degrees $d(v)$ of which are no greater than a threshold d_t. $|T|$ denotes the number of threads in a block. As the warp is the basic scheduling unit in GPUs and executes independently, these samplers process small tasks in P_L concurrently. Note that for the cases where the number of small tasks is less than warp samplers, the strategy still works well in modern GPUs because 1) the idle samplers incur a negligible cost, and 2) multiple thread blocks run concurrently on an SM to fully utilize hardware resources. After completing small tasks, the thread block forms a block sampler to process the remaining tasks one by one.

After the two stages, we store sampling results in the global memory and update the query status in P_L. If a query stops, we will get a new query from the global task pool, which will be introduced in the next subsection. In summary, queries in P_L are processed iteratively and move one step at one iteration. A query will be processed in a specific block once it is fetched into the local task pool. This can eliminate the communication and synchronization costs among blocks. Moreover, the two-stage execution scheme processes tasks with variant workloads efficiently.

5.2 Scheduling

A simple method to handle massive queries is to evenly assign queries among workers (i.e., thread blocks). The static scheduling method works well on CPUs [35, 45]. However, we find that it can incur performance issues on modern GPUs because 1) GPUs have much higher parallelism than CPUs; and 2) thread block scheduling is transparent to users and certain thread blocks can start much later than others. To address this issue, we design a simple and effective dynamic scheduling method that cooperates with the two-stage computation scheme.

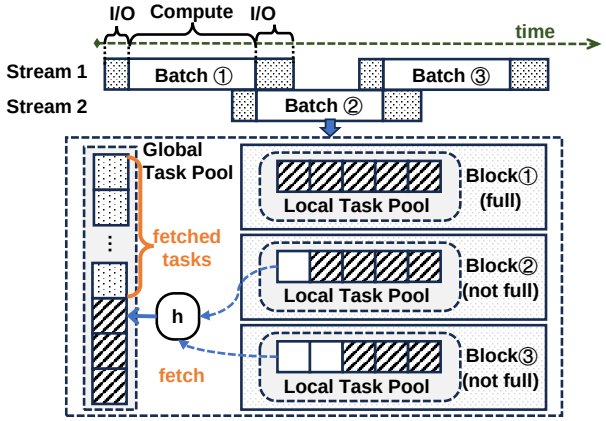


Figure 5: Queries are grouped into batches which execute alternatively in two CUDA streams. h refers to the head pointer of the global task pool. Thread blocks fetch tasks in a preemptive way if they have empty slots.

Figure 5 describes the dynamic scheduling strategy. We have a global task pool P_G, which keeps all queries in the device. Particularly, P_G is an array where an element is the start vertex of a query. Correspondingly, the result pool is the array storing the query sequence with the size as $|P_G| \times L_{max}$ where L_{max} is the maximum length of a query. The result sequence of a query is stored continuously. As discussed in Section 5.1, thread blocks execute independently. Upon finding that there are empty slots in the local task pool, they will fetch queries from the global task pool to fill these empty slots. The thread blocks fetch tasks from the head of P_G in a preemptive manner. The concurrent accesses are supported by an atomic integer pointing to the first available queries in the pool. A thread block gets a query by increasing the integer atomically. The local task pool size is very small compared with the number of queries. Therefore, fine-grained scheduling can keep load balance among thread blocks to fully utilize computing resources. We do not adopt any work-stealing techniques because a query takes a short time and the communication and synchronization cost among thread blocks is expensive.

The number of queries residing on GPU is constrained by the result pool size. For the cases where the results exceed the result pool size, we process them in multiple batches. Specifically, we divide queries into multiple batches such that the result sequences of each batch can be held by the result pool. To overlap the GPU I/O time with computation time, we adopt the classical ping-pong buffer technique and process batches alternatively with two CUDA streams. The number of queries in a batch is determined by Equation 3 where M represents the total GPU memory size and M_G is the memory allocated for the graph. M_v is the memory required to store a single vertex. The overarching strategy aims to fully utilize available GPU memory for the result pool to minimize batch processing. Notice that: 1) the equation includes a division by two as a ping-pong buffer employs two alternating buffers; and 2) $L_{max} + 1$ includes the memory allocated for the start vertex for each query (i.e., the global task pool P_G). In summary, FlowWalker is capable of handling scenarios where the result sequence exceeds the available GPU memory.

$$|P_G| = \lfloor \frac{M - M_G}{2 \times (L_{max} + 1) \times M_v} \rfloor \quad (3)$$

5.3 Analysis and Comparison

In the following, we analyze the cost of FlowWalker and compare it with C-SAW and Skywalker, two GPU-based systems.

Memory Consumption. The input is a graph G and start vertices of queries $\mathbb{Q}$, and the output is the result sequence for each query. Their memory consumption is compulsory for all competing frameworks. Thus, we focus on the memory consumption for auxiliary data structures. The global task pool of FlowWalker is based on the array storing start vertices of walkers, which has no extra memory consumption, and the local task pool resides in the shared memory. Moreover, both warp and block samplers do not consume any global memory. Therefore, FlowWalker has no auxiliary data structures consuming the global memory.

In contrast, both C-SAW and Skywalker need an auxiliary data structure with $O(d_{max})$ to serve one query. This incurs expensive memory overhead for large graphs. Additionally, Skywalker uses a task pool with the memory consumption of $O(L_{max} \times |\mathbb{Q}|)$ to keep load balance among thread blocks. In summary, FlowWalker is memory-efficient, which brings two advantages: 1) FlowWalker can support larger graphs; and 2) the number of queries that can be processed simultaneously by FlowWalker is determined by computing resources, whereas that of C-SAW and Skywalker is limited by the available memory space.

Time. We first compare the time cost of processing one step of a query. As analyzed in Section 4, the time complexity of moving one step of a query using ZPRS is $O(\frac{d}{k} + \log k)$, while that of C-SAW is $O(\frac{d}{k} \times \log k + \log d)$ where d is the degree of $Q.cur$. Skywalker uses the alias table sampling method to perform sampling. Although its time complexity is $O(\frac{d}{k} + \log k)$, the practical performance is slow due to the complex alias table building process.

Next, we compare their techniques for keeping load balance. C-SAW can process a query with a warp only and uses a static scheduling method, which ignores both the load imbalance among tasks and thread blocks. Skywalker can adopt the parallelism based on degrees. However, Skywalker schedules queries among thread blocks with a global queue at each step. Consequently, each step requires a pop and a push operation, which incurs expensive overhead. And the queue consumes a large amount of memory space as discussed above. NextDoor assigns a single thread to a sampling function. This design ignores the variance of neighbor set sizes. Moreover, NextDoor operates in a BSP manner [5], advancing all queries by one step per iteration. This approach, however, may lead to two issues: 1) overhead from global synchronization, especially with queries of varying lengths such as PPR; and 2) the necessity to materialize all query results.

Under the sampler-centric model, FlowWalker handles variant tasks with different samplers and uses the multi-level task pool based scheduling strategy to keep load balance efficiently and effectively. Particularly, thread blocks can fetch a query by an atomic incremental operation, and a query sticks to the block until it is completed, which requires no communication and synchronization overhead among blocks. In our experiments, we show that FlowWalker runs much faster than its counterparts. Additionally, FlowWalker stands out as the only solution capable of handling cases where the result sequence exceeds available GPU memory.

Table 1: The detailed statistics of graphs.

Dataset	Name	$\|V\|$	$\|E\|$	d_{max}	Size(GB)
com-youtube	YT	1.1 M	6 M	28K	0.05
cit-patents	CP	3.8 M	33 M	793	0.26
Livejournal	LJ	4.8 M	86 M	20K	0.66
Orkut	OK	3.1 M	234 M	33K	1.76
EU-2015	EU	11 M	522M	399K	3.93
Arabic-2005	AB	23 M	1.1B	576K	8.34
UK-2005	UK	39 M	1.6B	1.7M	11.82
Twitter	TW	42 M	2.4 B	3M	18.08
Friendster	FS	66 M	3.6 B	5K	27.16
SK-2005	SK	51 M	3.6 B	8.5M	27.16

6 EXPERIMENTS

In this section, we conduct extensive experiments to evaluate the performance of FlowWalker.

6.1 Experimental Setup

We study five frameworks in the experiments. **DGL** [1] **[42]** is a widely used GNN framework. **LightRW**[2] **[37] (LRW)** is a FPGA-based DGRW framework. **ThunderRW**[3] **[35] (TRW)**, which is the state-of-the-art CPU-based framework, **Skywalker**[4] **[43] (SW)**, which is a GPU-based framework, and **FlowWalker (FW)**, which is the GPU framework proposed in this paper. ThunderRW executes with the ITS sampling method, which achieves the optimal performance in the online computation mode. We also contemplated using *C-SAW*[5]. However, it encounters memory issues when handling more than 10^5 queries. Therefore we exclude it from our experimental baselines. We do not involve NextDoor because it can only support the offline computation mode as discussed in Section 2.3.

Implementation and Experiment Environments. *FW* is implemented with ~6000 lines of CUDA code. The experiments of *DGL*, *SW*, and *FW* are conducted on a Linux server equipped with the 40 GB A100 GPU. It contains 108 streaming multiprocessors (SMs) each of which has 64 FP32 cores. The shared memory size of each SM is configured to 100 KB. The PCIe type is PCI-E 4.0 × 16, and the maximum bandwidth is 31.5GB/s. The server is equipped with one Intel(R) Xeon(R) Silver 4310 CPU and 256GB host RAM. We test *TRW* on a Linux server equipped with one Intel Xeon Platinum 8336C CPU, which has 16 physical cores with hyper-threading enabled. The size of the host RAM is 128 GB. *LRW* is tested on HACC@NUS[6] with an AMD Alveo U250 FPGA. We use NVCC of version 11.6, g++ of version 9.4.0 and the optimization flag `-O3` for compilation.

Datasets and Workloads. We select a variety of real-world graphs from different fields such as social networks, citations, and

[1] https://github.com/junyimei/dgl
[2] https://github.com/Xtra-Computing/LightRW
[3] https://github.com/Xtra-Computing/ThunderRW
[4] https://github.com/wpybtw/Skywalker
[5] https://github.com/concept-inversion/C-SAW
[6] https://xacchead.d2.comp.nus.edu.sg/

websites. The detailed statistics are listed in Table 1. YT, CP, LJ, OK, and FS are downloaded from Stanford SNAP [20], and EU, AB, UK, TW, and SK are from LAW [2, 3]. We have data sizes ranging from tens of megabytes to tens of gigabytes (with weight). To keep consistent with previous work [35, 49], we generate a real number randomly from an interval $[1, 5)$ as the edge weight and an integer from the interval $[0, 4]$ as the edge label.

We study DeepWalk, PPR, Node2Vec, and MetaPath in the experiments. For DeepWalk, we set the target depth to 80. For PPR, we set the stop probability to 0.2. For Node2Vec, we set the target length to 80, $a = 2.0$ and $b = 0.5$. For MetaPath, we set the schema to $(0, 1, 2, 3, 4)$. We issue a query from every vertex in the graph for DeepWalk, Node2Vec, and MetaPath. For PPR, $|V|$ queries start from the same vertex. We set the vertex to that with the maximum degree in G. In detailed evaluation, we follow the settings of DeepWalk and set the number of queries to 10^6 because *SW* frequently encounters performance issues and has no valid experiment results for comparison. For the comparison purpose, all applications, including SGRW are executed in the dynamic manner. As a result, the results on SGRW may diverge from those reported in previous papers [35, 43], which are obtained with static mode.

FW executes Node2Vec with DPRS, while the other three applications with ZPRS. *DGL* implements Node2Vec on CPUs, while the other three applications on GPUs. *SW* does not support MetaPath because it cannot handle labeled graphs. *LRW*, the FPGA-based framework, currently supports Node2Vec and MetaPath only. *TRW* and *FW* implement all these four applications.

Metrics. The *execution time* refers to the total time required for computation, excluding the time spent on loading the graph data into GPUs. The results are averaged through three runs. *OOT* signifies that the method exceeds the time limit, which is set as 8 hours for our experiments, while *OOM* indicates a memory overflow. For a more comprehensive analysis, we employ *NVIDIA Nsight Compute* to profile GPU *memory consumption*.

Parameters. *FW* requires two hyperparameters: the local task pool size $|P_L|$, and the degree threshold d_t. $|P_L|$ dictates the number of queries that a thread block can hold, and d_t serves as the threshold for selecting between the warp sampler and block sampler. We empirically tune their values and set $|P_L|$ and d_t to 64 and 1024, respectively, across our experiments. *FW* achieves a good performance on the settings. Due to space limits, we include a detailed evaluation of hyperparameter impacts in the technical report.

6.2 Overall Comparison

Table 2 showcases the overall comparison of execution times across different frameworks. Notably, *FW* is the only method capable of completing all test cases. In contrast, *DGL*, *LRW*, *TRW*, and *SW* struggle with larger graphs, encountering either time-out (OOT) or memory overflows (OOM). Specifically, *FW* finishes all cases within merely 2.2 hours. Among scenarios where all five frameworks succeed, *FW* achieves remarkable speedups. Compared with *DGL* on GPU, the maximum speedup is 92.2×, while this number is 315.8× for *DGL* on CPU (executing Node2Vec). *FW* reaches up to 16.4×, 752.2× and 72.1× speedup compared to *LRW*, *TRW* and *SW* respectively, underscoring its superior performance.

FW takes considerably longer time to process the UK, TW, and SK graphs compared to other datasets, while *DGL*, *LRW*, *TRW*, and *SW* often fail to complete within the time limit for these graphs. This increased time is attributed to the high degree of skewness in these graphs, as indicated in Table 1. High-degree vertices are visited more frequently, thereby dominating the processing time. These results underscore the importance of employing different levels of samplers for vertices with varying degrees. Despite its large size, the FS graph is processed relatively quickly due to its sparsity. Although both DeepWalk and Node2Vec have the same target length, the execution time on Node2Vec is longer than that on DeepWalk because the cost of calculating the transition probability of Node2Vec is higher than that of DeepWalk.

FW eliminates the need for auxiliary data structures for each query's sampling, thereby reducing the space cost per query from $O(d_{\max})$ to $O(1)$, where $d_{\max}$ is the maximum degree of a graph. This efficiency enables *FW* to support large graphs and a substantial number of concurrent queries. *FW* also exhibits superior performance on smaller graphs due to the improvement of scheduling and sampling methods. We evaluate these techniques in Section 6.3. In summary, *FW* surpasses existing CPU, GPU, and FPGA frameworks in DGRW performance and is capable of efficiently handling large graphs.

6.3 Detailed Evaluation

In this subsection, we have a detailed evaluation of the performance of *FW*. Due to space limitations, some evaluations such as the comprehensive ablation study are provided in the technical report.

Memory Consumption. Table 3 presents a comparison of memory consumption between *FW* and *SW* across different datasets, with query sizes $|\mathbb{Q}| = 10^6$ and $|\mathbb{Q}| = 10^7$. *SW* can exceed GPU memory capacity due to its use of unified virtual memory (UVM). The "extra" memory usage (**E**) is calculated by subtracting the dataset size from the total memory consumption.

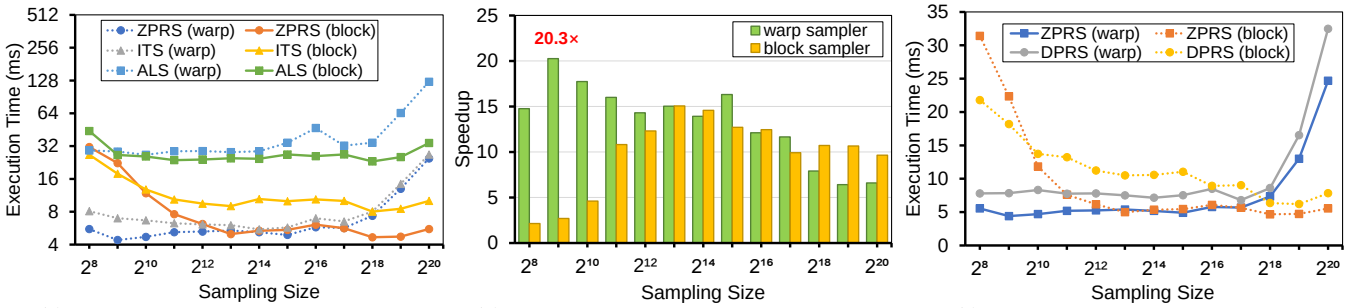
Remarkably, the extra memory consumption of *FW* remains consistent across all graph sizes, whereas *SW* exhibits a marked increase in memory use for larger graphs. This stability is attributable to the design of *FW*. *FW* minimizes per-query memory usage from $O(d)$ to $O(1)$, which is independent of graph size. It requires no auxiliary data structures in the global memory to support the execution. In contrast, *SW* requires a buffer of size $O(d_{max})$ for each query and has a large task queue for load balance.

For $|\mathbb{Q}| = 10^6$, query sequences occupy approximately 309 MB of memory, with a 32-bit integer representation for each vertex. For $|\mathbb{Q}| = 10^7$, this figure rises to 3090 MB. Beyond storing query sequences, *FW* uses no additional memory for auxiliary data structures. These findings confirm two key points: 1) existing GPU frameworks struggle with significant memory consumption issues, and 2) *FW* excels in memory efficiency.

Evaluation of Sampling Methods. We assess the performance of ZPRS, ITS, and ALS on GPUs by sampling a cumulative 2GB of elements, partitioned into tasks of varying sampling sizes. "Sampling size" refers to the number of elements involved in a single sampling operation, and all tasks within a single workload share the same sampling size. Figure 6a reveals that ITS on warp performs comparably to ZPRS, while ZPRS on block outperforms ITS on block.

Table 2: The overall comparison on execution time (seconds).

	Dataset	YT	CP	LJ	OK	EU	AB	UK	TW	FS	SK
DeepWalk	**DGL**	0.93	**0.30**	1.25	1.84	68.11	3492.19	OOM	OOM	OOM	OOM
	TRW	6.90	3.81	14.28	20.86	739.97	3298.71	OOT	OOT	496.52	OOT
	SW	7.82	3.20	21.89	28.88	431.61	1410.01	OOT	OOT	OOM	OOT
	FW	**0.45**	0.42	**0.95**	**0.99**	**17.40**	**59.86**	**736.52**	**2674.25**	**24.26**	**1509.83**
PPR	**DGL**	1.03	0.29	2.76	2.91	138.20	7728.80	OOM	OOM	OOM	OOM
	TRW	7.50	0.52	20.17	21.66	1900.78	3591.19	OOT	OOT	56.67	OOT
	SW	4.10	0.85	10.85	11.70	690.55	1763.33	OOT	OOT	OOM	OOT
	FW	**0.23**	**0.10**	**0.74**	**0.69**	**32.60**	**82.29**	**1041.55**	**897.61**	**3.83**	**2797.56**
Node2Vec	**DGL**	273.71	132.65	428.92	583.50	15988.82	OOT	OOT	OOM	OOM	OOM
	TRW	66.69	28.65	260.63	553.65	5936.80	23042.37	OOT	OOT	27329.18	OOT
	SW	40.39	12.07	134.23	130.38	1065.75	2498.27	OOT	OOT	OOM	OOT
	LRW	12.68	7.13	18.16	24.70	758.57	2771.56	OOM	OOM	OOM	OOM
	FW	**0.89**	**0.44**	**1.86**	**2.60**	**50.64**	**192.31**	**2044.09**	**7514.67**	**65.51**	**4688.86**
MetaPath	**DGL**	0.04	0.09	0.13	0.10	1.67	35.17	376.55	OOM	OOM	OOM
	TRW	0.22	0.42	2.43	13.32	121.96	2144.53	OOT	OOT	202.27	OOT
	LRW	0.11	0.19	0.36	0.61	9.13	40.24	422.24	OOM	OOM	OOM
	FW	**0.01**	**0.02**	**0.05**	**0.07**	**0.65**	**2.85**	**37.45**	**132.62**	**0.98**	**74.36**



(a) Comparison of ZPRS, ITS, and ALS. (b) Impact of RNG optimization on ZPRS. (c) Comparison of ZPRS and DPRS.

Figure 6: Detailed evaluation of ZPRS and DPRS: warp and block indicate task processing at warp and block levels, respectively.

Table 3: Memory usage (GB). T refers to the total memory consumption, and E refers to the extra memory consumption (subtracting the size of dataset from T).

Dataset	$\|\mathbb{Q}\| = 10^6$				$\|\mathbb{Q}\| = 10^7$			
	FW		SW		FW		SW	
	T	E	T	E	T	E	T	E
YT	0.35	0.31	2.07	2.02	3.10	3.05	18.41	18.36
CP	0.57	0.30	2.08	1.81	3.31	3.05	18.42	18.15
LJ	0.96	0.30	2.62	1.95	3.71	3.05	18.96	18.29
OK	2.06	0.30	3.81	2.04	4.81	3.05	20.15	18.38
EU	4.24	0.31	8.63	4.66	6.99	3.06	24.97	21.00
AB	8.64	0.30	14.32	5.90	11.39	3.05	30.66	22.24
UK	12.12	0.30	26.50	14.5	14.87	3.05	42.83	30.87
TW	18.38	0.30	41.60	23.4	21.13	3.05	57.93	39.70
FS	27.46	0.30	29.01	1.61	30.21	3.05	45.35	17.95
SK	27.47	0.31	90.99	63.6	30.22	3.06	107.3	79.98

This discrepancy arises because ITS necessitates frequent collective operations, which are more efficiently executed on warps than on blocks. ALS lags behind its counterparts due to the complex alias table construction. Recall that ZPRS has a space complexity of $O(1)$, while ITS has a space complexity of $O(d)$. Our results demonstrate that ZPRS outperforms existing samplers without auxiliary data structures.

Unlike ITS, ALS, and RS, the performance of Rejection Sampling (RJS) depends on the underlying probability distribution. We observe that on less biased distributions, RJS can surpass RS at some sampling sizes due to its lower initialization cost. However, as the distribution grows more biased, RJS's performance significantly deteriorates. This variability can impact the stability of performance. Detailed results are presented in the technical report.

Figure 12 highlights the substantial speedup achieved through optimizing random number generation (RNG) in ZPRS. These results underscore both the necessity and effectiveness of RNG optimization in ZPRS, as each element requires the generation of a random number. The observed speedup for ZPRS when processed on blocks is minimal for small sampling sizes because small tasks do not fully utilize the hardware capabilities. Conversely, speed gains on warps are limited for large sampling sizes, as processing extended sequences on warps does not maximize memory bandwidth utilization.

For the same reason, both ZPRS and DPRS on warps run faster than on blocks for small sampling sizes but slower for large sizes in

Figure 6c. When the sampling size is larger than 2^9, DPRS lags behind ZPRS for both warp and block samplers due to communication costs between threads.

Ablation Study. We conduct an ablation study to analyze the contributions of each individual technique to the overall speedup. Initially, we implement a baseline version of *FW* with DPRS, RNGs stored in global memory, and a basic static scheduler. This setup is referred to as **FW**. Subsequently, we enhance **FW** by optimizing RNG, which we denote as **FW + RNG**. Following this, we replace DPRS with ZPRS, marked as **FW + ZPRS**. Finally, we integrate dynamic scheduling, labeled as **FW + DS**.

The speedup of *SW* on DeepWalk with 10^6 queries against LJ, EU, and TW is illustrated in Figure 7. The data indicates that *FW* achieves a speedup range of 2.1× to 6.1× over *SW* without any optimizations. The optimized shared-memory RNG contributes an additional 2.5× to 4.4× speedup. The adoption of ZPRS further results in a speedup of 1.1× to 2.0×. Lastly, the implementation of dynamic scheduling offers an additional 1.1× to 2.3× speedup. These findings affirm the efficacy of each technique introduced in our paper. It is noteworthy that ZPRS, despite being a basic operator, contributes a significant 1.1× to 2.0× speedup to the overall system performance. The effect of dynamic scheduling is relevant to the degree skewness of the graph. This is the reason that the speedup of **FW + DS** on TW is smaller than EU and LJ. We will elaborate on this in the technical report, as well as the ablation study results of all datasets and applications.

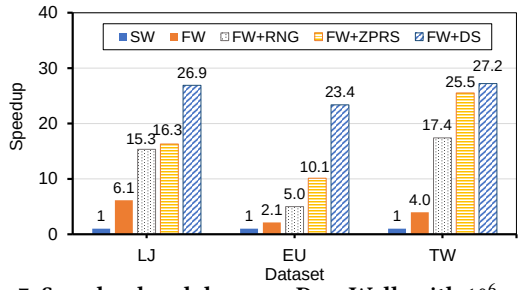


Figure 7: Speedup breakdown on DeepWalk with 10^6 queries. The value is normalized to SkyWalker (*SW*).

6.4 Case Study

GNNs are important for ByteDance operations, spanning video recommendations, friend suggestions, and fraud detection. In this case study, we focus on Douyin friend recommendation scenarios. The utilized framework is a business-specific adaptation of Graph-Learn [52], running on a CPU-only cluster of 20 machines with 560 cores. The RW phase in the training is to perform DeepWalk where Graph-Learn executes in the dynamic mode. The test graph comprises 227 million vertices and 2.71 billion edges.

Figure 8 breaks down the execution time for a single training epoch. The process is composed of several key components: data loading, random walk generation, and embedding learning. Completing one epoch takes nearly 10 hours, subdivided into data loading (0.25 hours), random walk (RW) generation (3.49 hours), and network training (6.32 hours). RW occupies 35% of the total processing time. If more advanced RW algorithms like Node2Vec are used, RW can consume much more time, as evidenced by DeepWalk vs. Node2Vec in Table 2. We do not include Node2Vec in the case study since Graph-Learn cannot support it.

As shown in Figure 8, FlowWalker reduces the RW time to merely 13 minutes (3% of the total cycle time), offering significant efficiency gains. On the other hand, ThunderRW requires more than 10 hours on a single machine. Skywalker is omitted from the comparison because it encounters a memory failure. These findings highlight the compelling performance advantages of FlowWalker.

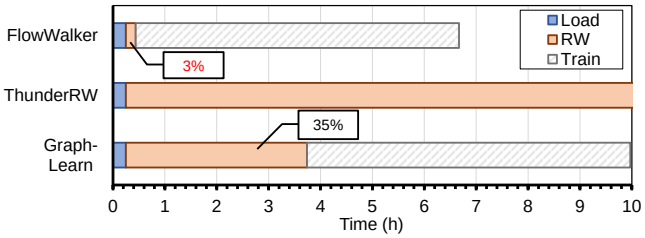


Figure 8: Time breakdown of training one epoch.

7 CONCLUSION

In this paper, we propose FlowWalker, a memory-efficient and high-performance GPU-based framework for dynamic graph random walks. We develop DPRS and ZPRS, two parallel reservoir sampling algorithms to perform fast sampling with no extra global memory and pre-processing cost. We implement a GPU walking engine to process a massive number of walking queries based on the sampler-centric paradigm. The effectiveness of FlowWalker is evaluated through a variety of datasets, and the results show that FlowWalker achieves up to 752.2× speedup on four representative random walk applications. At last, the case study reveals that FlowWalker can reduce the time cost of dynamic random walk from 35% to 3% of the GNN training pipeline.

REFERENCES

[1] Paolo Boldi and Marco Rosa. 2012. Arc-community detection via triangular random walks. In *2012 Eighth Latin American Web Congress*. IEEE, 48–56.

[2] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks. In *Proceedings of the 20th international conference on World Wide Web*, Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar (Eds.). ACM Press, 587–596.

[3] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*. ACM Press, Manhattan, USA, 595–601.

[4] Min-Te Chao. 1982. A general purpose unequal probability sampling plan. *Biometrika* 69, 3 (1982), 653–656.

[5] Thomas H Cormen and Michael T Goodrich. 1996. A bridging model for parallel computation, communication, and I/O. *ACM Computing Surveys (CSUR)* 28, 4es (1996), 208–es.

[6] Xiaoheng Deng, Genghao Li, Mianxiong Dong, and Kaoru Ota. 2017. Finding overlapping communities based on Markov chain and link clustering. *Peer-to-peer Networking and Applications* 10 (2017), 411–420.

[7] Yuxiao Dong, Nitesh V Chawla, and Ananthram Swami. 2017. metapath2vec: Scalable representation learning for heterogeneous networks. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*. 135–144.

[8] Dániel Fogaras, Balázs Rácz, Károly Csalogány, and Tamás Sarlós. 2005. Towards scaling fully personalized pagerank: Algorithms, lower bounds, and experiments. *Internet Mathematics* 2, 3 (2005), 333–358.

[9] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*. 855–864.

[10] Yu He, Yangqiu Song, Jianxin Li, Cheng Ji, Jian Peng, and Hao Peng. 2019. HeteSpaceyWalk: A Heterogeneous Spacey Random Walk for Heterogeneous Information Network Embedding. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management* (Beijing, China) *(CIKM '19)*. Association for Computing Machinery, New York, NY, USA, 639–648. https://doi.org/10.1145/3357384.3358061

[11] W. Daniel Hillis and Guy L. Steele. 1986. Data Parallel Algorithms. *Commun. ACM* 29, 12 (dec 1986), 1170–1183. https://doi.org/10.1145/7902.7903

[12] Lorenz Hübschle-Schneider and Peter Sanders. 2022. Parallel weighted random sampling. *ACM Transactions on Mathematical Software (TOMS)* 48, 3 (2022), 1–40.

[13] Abhinav Jangda, Sandeep Polisetty, Arjun Guha, and Marco Serafini. 2021. Accelerating graph sampling for graph machine learning using GPUs. In *Proceedings of the Sixteenth European Conference on Computer Systems*. 311–326.

[14] Yong-Yeon Jo, Myung-Hwan Jang, Hyungsoo Jung, and Sang-Wook Kim. 2018. A High-Performance Graph Engine for Efficient Social Network Analysis. In *Companion of the The Web Conference 2018 on The Web Conference 2018, WWW 2018, Lyon , France, April 23-27, 2018*, Pierre-Antoine Champin, Fabien Gandon, Mounia Lalmas, and Panagiotis G. Ipeirotis (Eds.). ACM, 61–62. https://doi.org/10.1145/3184558.3186929

[15] Peter M Kogge and Harold S Stone. 1973. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE transactions on computers* 100, 8 (1973), 786–793.

[16] Aapo Kyrola. 2013. DrunkardMob: billions of random walks on just a PC. *Proceedings of the 7th ACM conference on Recommender systems* (2013).

[17] Aapo Kyrola. 2013. Drunkardmob: billions of random walks on just a pc. In *Proceedings of the 7th ACM conference on Recommender systems*. 257–264.

[18] Aapo Kyrola, Guy E. Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-Scale Graph Computation on Just a PC. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*. 31–46. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/kyrola

[19] Ni Lao and William W Cohen. 2010. Relational retrieval using a combination of path-constrained random walks. *Machine learning* 81 (2010), 53–67.

[20] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[21] Xueting Liao, Yubao Wu, and Xiaojun Cao. 2019. Second-Order CoSimRank for Similarity Measures in Social Networks. In *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*. 1–6. https://doi.org/10.1109/ICC.2019.8761899

[22] Xin Lv, Yuxian Gu, Xu Han, Lei Hou, Juanzi Li, and Zhiyuan Liu. 2019. Adapting Meta Knowledge Graph Information for Multi-Hop Reasoning over Few-Shot Relations. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019, Hong Kong, China, November 3-7, 2019*, Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan (Eds.). Association for Computational Linguistics, 3374–3379. https://doi.org/10.18653/v1/D19-1334

[23] Pedro J. Martín, Luis F. Ayuso, Roberto Torres, and Antonio Gavilanes. 2012. Algorithmic strategies for optimizing the parallel reduction primitive in CUDA. In *2012 International Conference on High Performance Computing & Simulation (HPCS)*. 511–519. https://doi.org/10.1109/HPCSim.2012.6266966

[24] NVIDIA. 2022. CUB Documentation. https://nvlabs.github.io/cub/index.html, Last accessed on 2023-6-25.

[25] NVIDIA. 2023. CUDA Toolkit Documentation, cuRAND. https://docs.nvidia.com/cuda/curand/index.html, Last accessed on 2023-6-25.

[26] S. Olver and A. Townsend. 2013. Fast inverse transform sampling in one and two dimensions. *arXiv: Numerical Analysis* (2013).

[27] Santosh Pandey, Lingda Li, Adolfy Hoisie, Xiaoye S Li, and Hang Liu. 2020. C-SAW: A framework for graph sampling and random walk on GPUs. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.

[28] Serafeim Papadias, Zoi Kaoudi, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. 2022. Space-efficient random walks on streaming graphs. *Proceedings of the VLDB Endowment* 16, 2 (2022), 356–368.

[29] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. 701–710.

[30] Christian Robert and George Casella. 2013. Monte Carlo statistical methods. In *Springer Science & Business Media*.

[31] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2009. The Graph Neural Network Model. *IEEE Transactions on Neural Networks* 20, 1 (2009), 61–80. https://doi.org/10.1109/TNN.2008.2005605

[32] Shubhabrata Sengupta, Aaron Lefohn, and John D Owens. 2006. A work-efficient step-efficient prefix sum algorithm. (2006).

[33] Yingxia Shao, Shiyue Huang, Xupeng Miao, Bin Cui, and Lei Chen. 2020. Memory-aware framework for efficient second-order random walk on large graphs. In *Proceedings of the 2020 ACM SIGMOD international conference on management of data*. 1797–1812.

[34] Baoxu Shi and Tim Weninger. 2016. Discriminative predicate path mining for fact checking in knowledge graphs. *Knowledge-based systems* 104 (2016), 123–133.

[35] Shixuan Sun, Yuhang Chen, Shengliang Lu, Bingsheng He, and Yuchen Li. 2021. ThunderRW: An In-Memory Graph Random Walk Engine. *Proc. VLDB Endow.* 14, 11 (2021), 1992–2005. http://www.vldb.org/pvldb/vol14/p1992-sun.pdf

[36] Yizhou Sun and Jiawei Han. 2012. Mining heterogeneous information networks: a structural analysis approach. *SIGKDD Explor.* 14, 2 (2012), 20–28. https://doi.org/10.1145/2481244.2481248

[37] Hongshi Tan, Xinyu Chen, Yao Chen, Bingsheng He, and Weng-Fai Wong. 2023. LightRW: FPGA Accelerated Graph Dynamic Random Walks. *Proc. ACM Manag. Data* 1, 1, Article 90 (may 2023), 27 pages. https://doi.org/10.1145/3588944

[38] Alok Tripathy, Katherine Yelick, and Aydin Buluc. 2023. Distributed Matrix-Based Sampling for Graph Neural Network Training. *arXiv preprint arXiv:2311.02909* (2023).

[39] Jeffrey S Vitter. 1985. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)* 11, 1 (1985), 37–57.

[40] Alastair J. Walker. 1977. An Efficient Method for Generating Discrete Random Variables with General Distributions. *ACM Trans. Math. Softw.* 3, 3 (1977), 253–256. https://doi.org/10.1145/355744.355749

[41] Jizhe Wang, Pipei Huang, Huan Zhao, Zhibo Zhang, B. Zhao, and D. Lee. 2018. Billion-scale Commodity Embedding for E-commerce Recommendation in Alibaba. *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (2018).

[42] Minjie Wang, Lingfan Yu, Da Zheng, Quan Gan, Yu Gai, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Chao Ma, et al. 2019. Deep graph library: Towards efficient and scalable deep learning on graphs. *arXiv preprint arXiv:1909.01315* (2019).

[43] Pengyu Wang, Chao Li, Jing Wang, Taolei Wang, Lu Zhang, Jingwen Leng, Quan Chen, and Minyi Guo. 2021. Skywalker: Efficient Alias-Method-Based Graph Sampling and Random Walk on GPUs. In *30th International Conference on Parallel Architectures and Compilation Techniques, PACT 2021, Atlanta, GA, USA, September 26-29, 2021*. IEEE, 304–317. https://doi.org/10.1109/PACT52795.2021.00029

[44] Pengyu Wang, Cheng Xu, Chao Li, Jing Wang, Taolei Wang, Lu Zhang, Xiaofeng Hou, and Minyi Guo. 2023. Optimizing GPU-based Graph Sampling and Random Walk for Efficiency and Scalability. *IEEE Trans. Comput.* (2023).

[45] Rui Wang, Yongkun Li, Hong Xie, Yinlong Xu, and John CS Lui. 2020. {GraphWalker}: An {I/O-Efficient} and {Resource-Friendly} Graph Analytic System for Fast and Scalable Random Walks. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 559–571.

[46] Shuke Wang, Mingxing Zhang, Ke Yang, Kang Chen, Shaonan Ma, Jinlei Jiang, and Yongwei Wu. 2023. NosWalker: A Decoupled Architecture for Out-of-Core Random Walk Processing. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 466–482.

[47] Yubao Wu, Yuchen Bian, and Xiang Zhang. 2016. Remember where you came from: on the second-order random walk based proximity measures. *Proceedings of the VLDB Endowment* 10, 1 (2016), 13–24.

[48] Ke Yang, Xiaosong Ma, Saravanan Thirumuruganathan, Kang Chen, and Yongwei Wu. 2021. Random walks on huge graphs at cache efficiency. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 311–326.

[49] Ke Yang, Mingxing Zhang, Kang Chen, Xiaosong Ma, Yang Bai, and Yong Jiang. 2019. KnightKing: a fast distributed graph random walk engine. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, Tim Brecht and Carey Williamson (Eds.). ACM, 524–537. https://doi.org/10.1145/3341301.3359634

[50] Hongbo Yin, Yingxia Shao, Xupeng Miao, Yawen Li, and Bin Cui. 2022. Scalable Graph Sampling on GPUs with Compressed Graph. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management* (Atlanta, GA, USA) *(CIKM '22)*. Association for Computing Machinery, New York, NY, USA, 2383–2392. https://doi.org/10.1145/3511808.3557443

[51] Dalong Zhang, Xin Huang, Ziqi Liu, Jun Zhou, Zhiyang Hu, Xianzheng Song, Zhibang Ge, Lin Wang, Zhiqiang Zhang, and Yuan Qi. 2020. AGL: A Scalable System for Industrial-Purpose Graph Machine Learning. *Proc. VLDB Endow.* 13, 12 (aug 2020), 3125–3137. https://doi.org/10.14778/3415478.3415539

[52] 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.

APPENDIX

A DIFFERENCES WITH EXISTING MEMORY REDUCTION STRATEGIES

FlowWalker effectively eliminates the need for large auxiliary data structures in graph random walk processing. This approach enables the simultaneous execution of a vast number of random walks, fully leveraging GPU computing power. In contrast, GraSS [50] focuses on graph compression. However, when processing graph random walk queries, the significant overhead from per-query auxiliary data structures restricts the number of concurrent queries, leading to suboptimal utilization of computing resources. Essentially, graph compression techniques like GraSS are complementary to our work. They can be integrated with FlowWalker to further minimize memory usage.

Training GNN on a large graph usually involves mini-batch method. Mini-batch training effectively reduces memory consumption by limiting the number of vertices in each batch. However, the sampling process for generating mini-batches in GNN, due to its stochastic nature, often requires operating on the entire graph [38]. Additionally, several prominent graph learning frameworks, such as AGL [51], adopt a two-stage processing approach. This involves initial sampling followed by mini-batch training, a method that ensures reproducibility of embedding queries and training results. The first stage entails executing a large number of random queries on the graph. It is pertinent to note that our work is primarily aimed at enhancing the efficiency of random walk queries.

B CORRECTNESS PROOF OF ALGORITHM 2

As the correctness proof of Algorithm 4 depends on the correctness of Algorithm 2, we first prove the correctness of Algorithm 2. The algorithm selects the element $S[i]$ with the probability $p(i) = p(i) = \frac{W[i]}{\sum_{j=1}^{n} W[j]}$ given a sequence S with n elements and the corresponding weight sequence W. We prove this using the constructive method.

Base Case. The algorithm apparently works for $n = 1$.

Induction Assumption. Given a sequence with n elements and an arbitrary integer i where $n > 1$ and $1 \leqslant i \leqslant n+1$, we assume that $S[i]$ is selected with a probability of $\frac{W[i]}{\sum_{j=1}^{n} W[j]}$.

Inductive Step. Give a sequence with $n+1$ elements and an arbitrary integer i where $1 \leqslant i \leqslant n$, want to prove that $S[i]$ is selected with the probability of $\frac{W[i]}{\sum_{j=1}^{n+1} W[j]}$.

As shown in Line 4 in Algorithm 2, $S[n+1]$ is chosen with a probability of $\frac{W[n+1]}{\sum_{j=1}^{n+1} W[j]}$. Thus, the current selected element has a probability of $1 - \frac{W[n+1]}{\sum_{j=1}^{n+1} W[j]} = \frac{\sum_{j=1}^{n} W[j]}{\sum_{j=1}^{n+1} W[j]}$ to stay (not be replaced by element $n+1$). According to the inductive assumption, for the elements $S[i]$ positioned before $S[n+1]$ in the sequence, $S[i]$ has the probability of $\frac{W[i]}{\sum_{j=1}^{n} W[j]}$ to be the currently selected vertex. Then, all these elements $S[i]$ has a probability of $\frac{W[i]}{\sum_{j=1}^{n} W[j]} \times \frac{\sum_{j=1}^{n} W[j]}{\sum_{j=1}^{n+1} W[j]} = \frac{W[i]}{\sum_{j=1}^{n+1} W[j]}$ to be the selected element after processing $S[n+1]$. Thus, all the $n+1$ elements has the probability of $\frac{W[i]}{\sum_{j=1}^{n+1} W[j]}$ to be selected.

The correctness of Algorithm 2 is proved. The correctness proof of ZPRS is detailed in Section 4.2.

C SUPPLEMENTARY EVALUATION

C.1 Comparison with Rejection Sampling

In this section, we discuss and evaluate the performance of rejection sampling (RJS). Given a neighbor set $N(v)$ of a vertex v, RJS samples a vertex u from $N(v)$ in two phases. The initialization phase calculates $p_{max} = \max_{u \in N(v)} p(u)$ where $p(u)$ is the selection probability of u. Subsequently, the selection phase has two steps: 1) randomly select a vertex u from $N(v)$; and 2) randomly generate a real number p in $[0, p_{max})$. If $p < p(u)$, then u is the sampling result. Otherwise, RJS repeats the selection phase. The time complexity of initialization is $O(d(v))$, and that of selection is $O(\mathbb{E})$ where $\mathbb{E} = \frac{d(v) \times p_{max}}{\sum p(u)}$.

We empirically compare the performance of reservoir sampling (RS) with RJS under different distributions in Figure 9. Specifically, we generate the weights using log-normal distribution with the mean value μ as 0 and the standard deviation σ varying from 1 to 3. The sampling size is the number of elements in a single operation. When $\sigma = 1$, we can see that RS is faster than RJS on small sampling sizes (e.g., 2^8). This is because the selection phase of RJS incurs expensive overhead. However, RS is slower on large sampling sizes (at most 2.6×) because RS needs to generate a random number for each element that dominates the cost. When $\sigma = 2$ and $\sigma = 3$, the performance of RJS significantly degrades because the selection phase needs to be repeated many times. In contrast, the performance of RS is steady and significantly outperforms RJS (up to 39.6×).

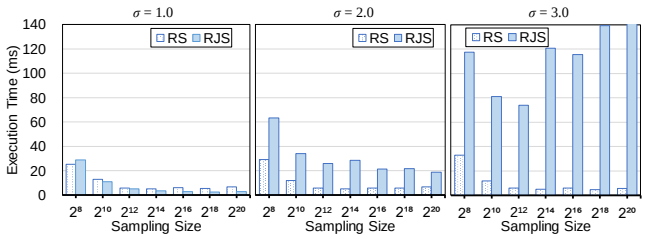
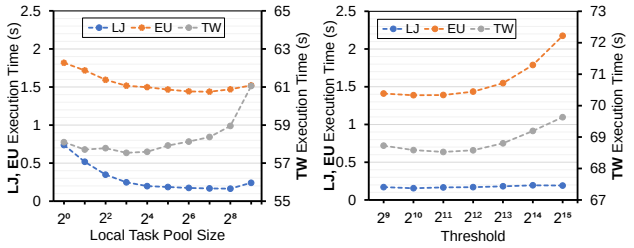


Figure 9: Comparison of RS and RJS on different weight distributions with varying sampling sizes.

In summary, the running time of RJS is non-deterministic and heavily depends on the probability distribution. This can affect the system stability given the complex real-world scenarios, e.g., Table 4 and Table 5 present experiment results on weighted Node2Vec and weighted MetaPath with different distributions. Nevertheless, an interesting research direction is to dynamically select the sampling method given the input. However, this requires an efficient and effective adaptive sampling method selection mechanism at runtime, which we will leave as a future work.

C.2 Impact of Hyperparameters

FlowWalker requires two hyperparameters: the local task pool size $|P_L|$, and the degree threshold d_t. $|P_L|$ dictates the number of queries that a thread block can hold. A small $|P_L|$ would underutilize the



(a) Local task pool size. (b) Warp-block threshold.

Figure 10: Impact of hyperparameters on the performance.

Table 4: Comparison of RS and RJS performance in weighted Node2Vec across various weight distributions. Values are derived using a log-norm generator with standard deviations ranging from 1 to 3.

	$\sigma = 1$		$\sigma = 2$		$\sigma = 3$	
Dataset	**RS**	**RJS**	**RS**	**RJS**	**RS**	**RJS**
YT	**0.94**	1.02	**0.87**	3.40	**0.76**	8.11
CP	**0.42**	0.80	**0.42**	1.44	**0.41**	1.81
LJ	**1.84**	2.81	**1.72**	8.46	**1.67**	17.71
OK	**2.51**	3.62	**2.55**	12.11	**2.50**	30.15
EU	49.87	**38.66**	**49.75**	156.25	**48.97**	615.80
AB	192.06	**133.79**	**190.54**	434.28	**186.18**	1825.56
UK	2060.20	**1251.38**	2061.07	**1743.77**	**2057.86**	9429.69
TW	7570.73	**4727.40**	7594.72	**7303.92**	**7600.89**	OOT
FS	**64.33**	100.22	**64.09**	360.87	**63.80**	819.82
SK	4717.55	**2834.26**	**4691.10**	4913.34	**4647.53**	23776.17

Table 5: Comparison of RS and RJS performance in weighted MetaPath across various label distributions. Values are derived using a log-norm generator with standard deviations ranging from 1 to 3.

	$\sigma = 1$		$\sigma = 2$		$\sigma = 3$	
Dataset	**RS**	**RJS**	**RS**	**RJS**	**RS**	**RJS**
YT	**0.01**	0.01	**0.01**	0.04	**0.01**	0.09
CP	**0.02**	0.02	**0.02**	0.03	**0.02**	0.03
LJ	**0.04**	0.05	**0.04**	0.12	**0.04**	0.22
OK	**0.06**	0.09	**0.06**	0.25	**0.06**	0.52
EU	**0.63**	0.69	**0.62**	3.52	**0.62**	12.06
AB	**1.54**	2.63	**1.51**	13.03	**1.50**	51.59
UK	77.80	**24.22**	78.59	**54.19**	**78.08**	348.52
TW	87.72	**85.72**	**86.71**	227.87	**86.53**	1448.48
FS	**1.32**	1.73	**1.32**	4.42	**1.31**	7.28
SK	**40.93**	48.41	**40.35**	172.70	**39.92**	912.44

computational resources, while a large $|P_L|$ would lead to coarse-grained task fetching and may aggravate load imbalance. Besides, as GPUs have limited shared memory sizes, a large $|P_L|$ can exceed the shared memory limitations. Following this guideline, we can tune $|P_L|$ by varying it from 1 to $|T|$ where T is the number of threads in a block. Figure 10a displays the experimental results with $|P_L|$ varying from 1 to 512. It is observed that the execution time remains relatively stable within the range of $[2^5, 2^8]$, but exhibits a slight increase beyond this interval. Consequently, the performance of FlowWalker demonstrates robustness to changes in $|P_L|$, leading us to set 64 as the default value for all datasets.

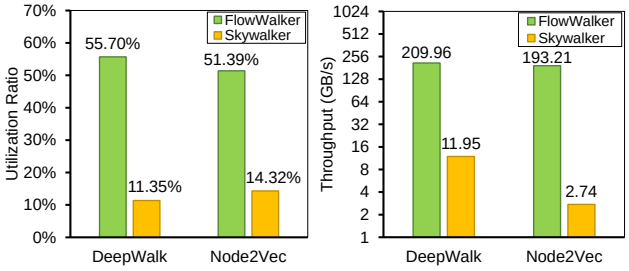
The parameter d_t serves as the threshold for selecting between the warp sampler and block sampler. To assess the impact of this threshold on performance, we conducted micro-benchmarking experiments. The results are depicted in Figure 10b. Our observations reveal that the execution time remains consistent when d_t ranges from 2^9 to 2^{12}, showing a slight increase for values of d_t larger than this range. Consequently, FlowWalker exhibits robust performance relative to variations in d_t. Based on these findings, we have chosen 1024 as the default value for d_t.

In summary, the performance of FlowWalker is robust with respect to the hyperparameters. Users can adhere to the default settings for optimal performance. In light of the reviewer's comments, we expand our discussion in Section 6.1 in the revision to include guidelines on parameter settings and their impacts.

C.3 GPU Resource Utilization

We analyze GPU resource utilization using *NVIDIA Nsight Compute*. Figure 11 showcases the performance metrics of DeepWalk and Node2Vec on the LJ dataset. As evidenced in Figure 11a, *FW* boasts substantially higher SM (Streaming Multiprocessor) utilization compared to *SW*, highlighting superior parallelism of *FW*. Additionally, Figure 11b reveals that *FW* enjoys a significantly higher memory bandwidth, which is attributed to its efficient coalesced memory access.

For *FW*, both the SM utilization and memory bandwidth are marginally lower for Node2Vec than for DeepWalk. This is due to the binary search operations required for transition probability calculation, which lead to some random memory accesses. Despite this, *FW* still outperforms *SW* in overall resource utilization, demonstrating the efficacy of our approach.



(a) Maximum SM utilization. (b) Memory bandwidth.

Figure 11: Comparison of GPU resource utilization on LJ.

C.4 RNG Performance Evaluation

Figure 12 highlights the substantial speedup achieved through optimizing random number generation (RNG) in ZPRS. These results underscore both the necessity and effectiveness of RNG optimization in ZPRS, as each element requires the generation of a random number. The observed speedup for ZPRS when processed on blocks is minimal for small sampling sizes because small tasks do not fully utilize the hardware capabilities. Conversely, speed gains on warps are limited for large sampling sizes, as processing extended

sequences on warps does not maximize memory bandwidth utilization.

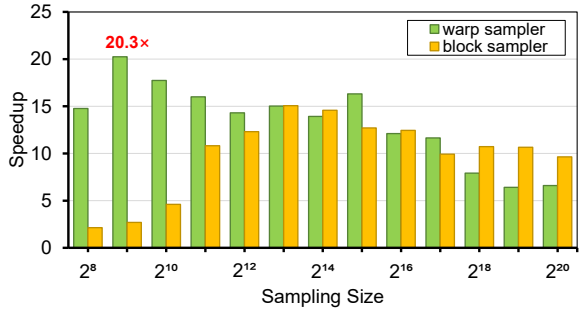


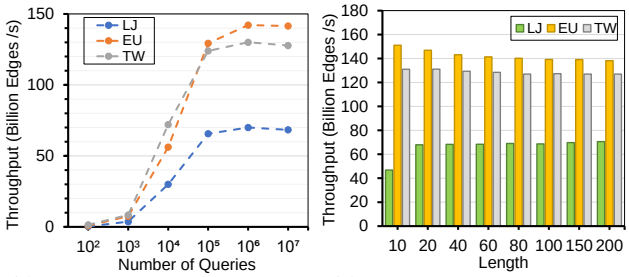
Figure 12: Impact of RNG optimization on ZPRS.

C.5 Scalability Evaluation

We assess the scalability of FlowWalker by examining throughput under varying numbers of walking queries and walking lengths. Specifically, throughput is defined as the number of edges processed per second. We opt for this metric over the number of vertices processed to avoid bias introduced by degree skewness. By default, we conduct 10^6 walking queries starting from randomly selected vertices with a walking length of 80.

Figure 13a illustrates how throughput varies as the number of queries changes from 10^2 to 10^7. The throughput is suboptimal at low query counts because the workload is insufficient to fully utilize the computational capacity of the GPUs. It plateaus at around 10^6 queries, indicating strong scalability in relation to the number of queries.

In Figure 13b, the throughput stabilizes when the query length exceeds 20, confirming the system scalability with respect to query length. During these experiments, we observed significantly higher throughput on the TW and EU datasets compared to LJ. This can be attributed to the degree distribution of tasks: over 97% of tasks on LJ have small degrees. Consequently, the system efficiency is lower when processing a large volume of tasks that each involves scanning only a short sequence of neighbors, as opposed to EU and TW having many tasks that each scan a longer sequence.



(a) Varying number of queries. **(b) Varying length of queries.**

Figure 13: Throughput of *FW* with varying walker number and query length.

C.6 Ablation Study

We conduct the ablation study on all datasets and four applications, assessing the impact of each technique individually. Notice that the original Skywalker code has bugs leading to fewer sampling steps. We add some thread fence to eliminate this problem[7] in this section. However, the Skywalker we used as the baseline in the paper is in its original state without any modification[8]. The findings reveal that ZPRS contributes to speedups of 1.1×-2.8×, while the implementation of DS leads to speedups of 1.03×-19.0×. Notably, DPRS is utilized by default in Node2Vec to mitigate the transition probability computation overhead, resulting in DPRS outperforming ZPRS by 1.1× to 1.7×. This outcome validates our cost analysis for DPRS and ZPRS, underscoring the significance of both techniques. Specifically, on five billion-scale graphs—AB, UK, TW, FS, and SK—ZPRS and DS facilitate up to 2.8× and 19.0× speedup, respectively, showcasing their efficiency on large-scale graph datasets. The results demonstrate the effectiveness of the techniques proposed in this paper.

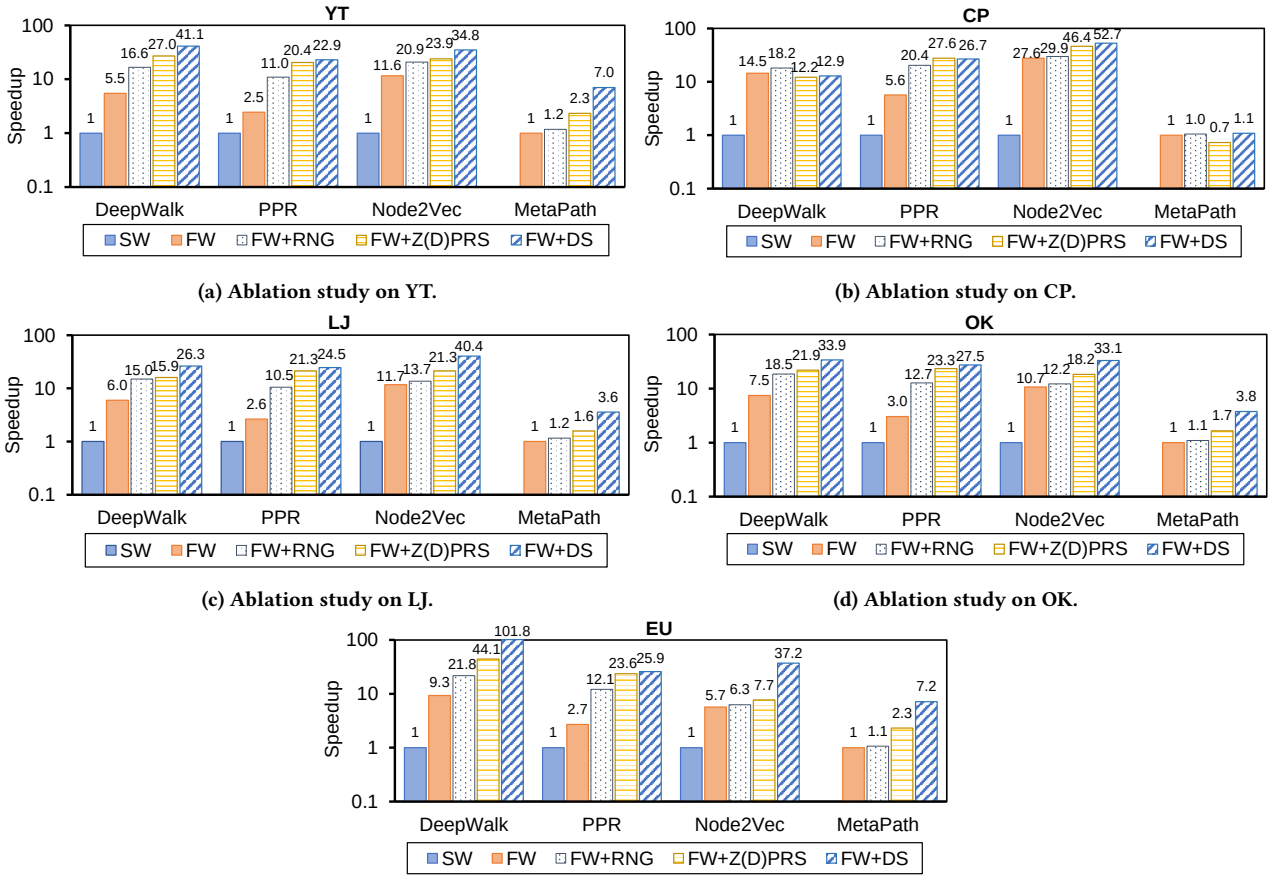
We develop a baseline version of FlowWalker (**FW**), featuring DPRS, RNGs in global memory, and a simple static scheduler. This configuration is identified as **FW**. We then improved **FW** by optimizing RNG storage, creating the variant **FW + RNG**. Next, we substituted DPRS with ZPRS. For Node2Vec, we incorporate ZPRS in **FW**, and replace it with DPRS in this step. Therefore we denote this configuration as **FW + Z(D)PRS**. Finally, we incorporated dynamic scheduling, producing **FW + DS**. For Node2Vec experiments, **FW** initially uses ZPRS.

Figure 14 and Figure 15 depict the results of all the datasets listed in Table 1. The results are normalized to Skywalker (SW). Specifically, *for MetaPath, we normalize the results to **FW** as SW does not support MetaPath.* According to the data, the proposed optimization methods in FlowWalker are able to enhance performance in the majority of scenarios (146 of 150 cases), and FlowWalker (**FW+DS**) outperforms SW on all cases. The performance varies across different datasets and different applications. We broadly summarize several general patterns as follows.

The performance of **FW** surpasses SW on all datasets without any optimizations, including the shared-memory RNG. And the speedup is substantial on many datasets. Especially for the application Node2Vec, the baseline **FW** can provide up to 27.6× speedup, which is much higher than the speedup from RNG (up to 1.8×), DPRS (up to 1.7×), and dynamic scheduling (up to 18.2×). This demonstrates the effectiveness of reservoir sampling.

The optimized RNG improves performance in all scenarios, but it is not the main contribution to the enhancement. The performance gain with RNG of PPR is higher than the other three applications, with a range of 3.6× to 4.8× speedup. This is because the PPR queries start from the vertex with the highest degree in the experiment setting. The average amount of random numbers generated is higher than the other applications. In terms of MetaPath, RNG contributes up to 1.3× speedup to the overall performance. This is because in MetaPath, edge labels have to match the given pattern. FlowWalker only needs to generate random numbers for the matched edges.

[7] https://github.com/junyimei/Skywalker
[8] https://github.com/wpybtw/Skywalker



(a) Ablation study on YT.

(b) Ablation study on CP.

(c) Ablation study on LJ.

(d) Ablation study on OK.

(e) Ablation study on EU.

Figure 14: The results of ablation study on small and medium graphs. For DeepWalk, PPR, and Node2Vec, the results are normalized to Skywalker (SW). For MetaPath, we normalize the results to FW as SW does not support MetaPath.

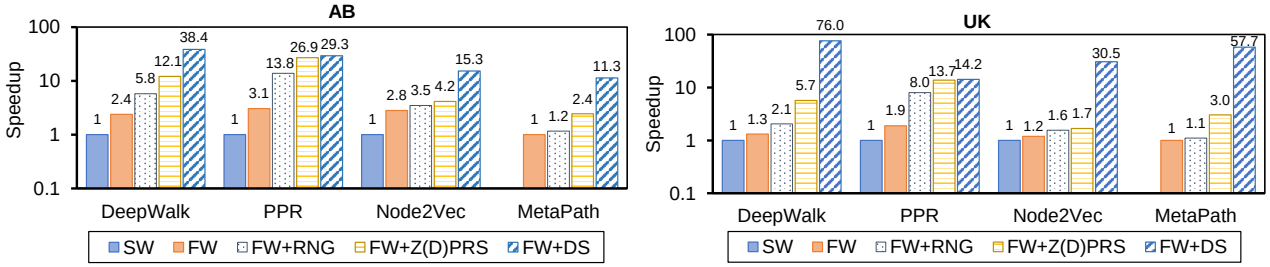
ZPRS contributes 1.1× to 2.8× speedup to the overall system performance. For Node2Vec, DPRS provides 1.1× to 1.7× speedup. The improvement is non-trivial for a primitive operator.

Dynamic scheduling enhances the system in most cases (149 of 150 cases), especially on skewed datasets. EU, AB, and UK greatly benefit from the dynamic scheduling method. For the highly skewed dataset UK, the speedup reaches at most 19.0×. The performance rise of dynamic scheduling is relevant to the workload. For example, in our experiment setting of PPR, all queries start from the same vertex with the highest degree, and there is a minor discrepancy between the workloads. Therefore the speedup of PPR is not as high as other applications, and for dataset CP, the performance even drops by 3%. Despite this, dynamic scheduling is still an effective optimization approach.

Next, we showcase the ablation study results on five billion-scale graphs and analyze the dynamic scheduling performance on TW. The graphs include AB (Figure 15a), UK (Figure 15b), TW (Figure 15c), FS (Figure 15d), and SK (Figure 15e). ZPRS (DPRS in Node2Vec) facilitates up to 2.1×, 2.8×, 1.7×, 1.7×, and 1.9× improvement for five datasets respectively. The results demonstrate the effectiveness of ZPRS on the billion-scale graphs. Dynamic scheduling offers up to 4.7×, 19.0×, 1.4×, 1.2×, and 2.4× speedup on these datasets.

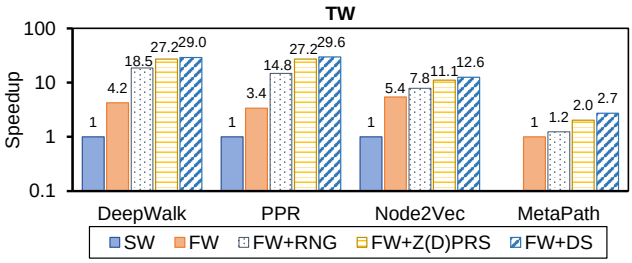
As discussed above, the effectiveness of dynamic scheduling varies among different datasets, depending on the graph degree distribution. Take the datasets UK and TW as two examples. Figure 16a and Figure 16b depict the CDF curves of their degree distributions. The x-axis in the graph represents the degree distribution, for example, 10% stands for the vertices with the least 10% degrees. And the y-axis depicts the vertex numbers. UK is a highly skewed graph with 80% of the vertices having the lowest 10% degrees, and the vertices with the highest 20% degrees take up only 3.4% of all the nodes. Compared with UK, the degree distribution of TW is more uniform, with 80% vertices having degrees less than 28%, and 14% vertices fall into the degree range of 80% - 100%. The skewed workload brought by skewed degree distribution in UK incurs significant performance rise of dynamic scheduling. In contrast, the speedup for TW is marginal due to the uniform workload.

The optimization techniques can lead to negative speedup in a minority of scenarios (4 of 150 cases). ZPRS incurs a performance drop when computing DeepWalk on CP (about 30%) and FS (about 10%). This is because CP and FS are sparse graphs, with a maximum degree of 793 and 5214 respectively. This number is much smaller

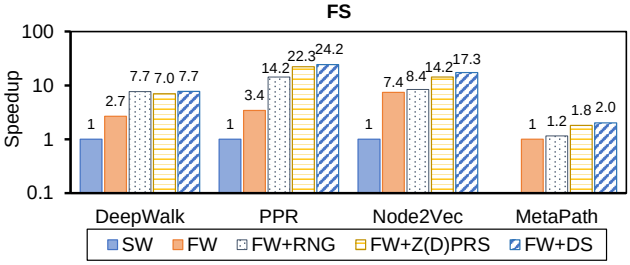


(a) Ablation study on AB.

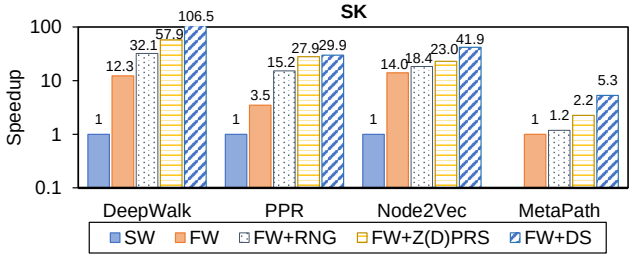
(b) Ablation study on UK.



(c) Ablation study on TW.

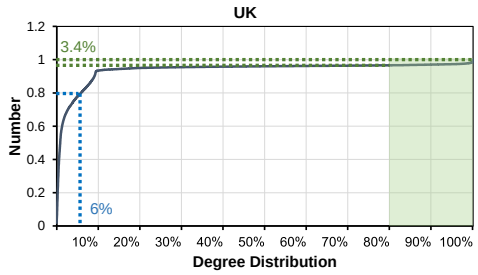


(d) Ablation study on FS.

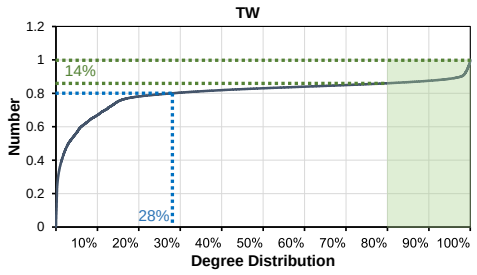


(e) Ablation study on SK.

Figure 15: The results of ablation study on five billion-scale graphs.



(a) Degree CDF of UK.



(b) Degree CDF of TW.

Figure 16: The degree distribution differs in UK and TW.

than the other graphs. As shown in Figure 6c, the performance of DPRS is better than ZPRS when the vertex degree is small. The case is the same for MetaPath on CP. Besides, as described above, dynamic scheduling slightly degrades the performance with 3% when performing MetaPath on CP. Since dynamic scheduling incurs additional overhead compared with the static method. The performance gain in other cases can offset the overhead, but for dataset CP, which is a small and sparse graph, dynamic scheduling degrades the performance.

In summary, the optimization methods in FlowWalker are able to enhance performance across the majority of scenarios. Combining all optimizations, FlowWalker significantly outperforms its counterpart in all cases. It is noteworthy that ZPRS contributes 1.1× to 2.8× speedup to the overall system performance. In particular, ZPRS

contributes 1.7×-2.8× speedup on billion-scale graphs. The improvement is non-trivial for a primitive operator. Moreover, the speedup varies across different datasets and applications. Each acceleration technique possesses a unique zone of superiority, manifesting divergent outcomes across various contexts. Consequently, within FlowWalker, the incorporation of each optimization approach is deemed essential.