LightTraffic: On Optimizing CPU-GPU Data Traffic for Efficient Large-scale Random Walks

Yipeng Xing[1], Yongkun Li[1,2], Zhiqiang Wang[1], Yinlong Xu[1,2] and John C. S. Lui[3]
[1]*University of Science and Technology of China*, [2]*Anhui Province Key Laboratory of High Performance Computing*
[3]*Department of Computer Science and Engineering, The Chinese University of Hong Kong*
ypxing@mail.ustc.edu.cn, ykli@ustc.edu.cn, wzq666@mail.ustc.edu.cn, ylxu@ustc.edu.cn, cslui@cse.cuhk.edu.hk

***Abstract*—As a fundamental tool for graph analysis, random walk receives extensive attention in both industry and academia. For computing massive random walks, recent works show that GPUs provide a good option to accelerate the performance. However, due to the limited memory space of modern GPUs, it is infeasible to have both the graph data and walk index fully reside in GPU memory when running large-scale random walks. Thus, it necessitates an out-of-GPU-memory design, but this inevitably induces large amounts of CPU-GPU data transmission traffic and thus hinders the overall performance. In this paper, we develop LightTraffic, which optimizes the data transmission between CPU and GPU memory under the constraint of GPU memory capacity with various system designs, including a memory-efficient scheme for partition-based management and multiple scheduling techniques. LightTraffic is a fully out-of-GPU-memory design, so it supports running large-scale random walks on GPUs. Experiments on our prototype show that LightTraffic outperforms various state-of-the-art CPU-based in-memory systems which also support large-scale random walks. For example, compared to the CPU-based systems FlashMob and ThunderRW, which are highly optimized for random walks, LightTraffic achieves $1.7-5.0\times$ and $1.4-12.8\times$ performance speedup, respectively. It also achieves up to an order of magnitude speedup when compared to the GPU-based system Subway which also supports large-scale random walks with an out-of-GPU-memory design for graph data.**

I. Introduction

As a basic building block of graph analysis, random walk is widely used to support various applications, e.g., social network analysis [3], [16], [26], [54], [70] and recommender systems [8], [23], [57], [71]. Recent advances in graph machine learning, e.g., graph embedding [7], [14], [45] and graph neural networks [72], heavily use random walks to sample a large graph. As a result, random walks receive extensive efforts in academia to optimize random walk based system designs [15], [21], [25], [44], [53], [55], [58], [60], [68], [69].

As graphs become larger, to develop efficient random walk based graph systems, it is necessary to support massive random walks concurrently. For example, sampling a graph with billions of vertices may need to start millions or even billions of random walks, which is very common in big corporations such as Tencent [62], Pinterest [71] and Alibaba [57]. Thus, if the scale of random walks becomes large, running massive random walks is a heavy computation task. One option to alleviate the computation bottleneck is to leverage the abundant computing resources of GPU devices with thousands of cores and high memory access bandwidth. Multiple systems are also developed to support random walks on GPUs [15], [44], [58].

However, when the graph and walk scale become large, leveraging GPUs to accelerate massive random walks still faces multiple challenges, and one critical issue is the limited GPU memory capacity. To address this scalability issue, large efforts are made to develop out-of-GPU-memory graph processing systems [19], [34], [50], [51], [74], and their key idea is to leverage a partition-based processing framework. Specifically, they divide the graph into multiple partitions, and load the graph partitions iteratively into GPU memory for computation. We find that out-of-GPU-memory designs may incur a high data transmission traffic between CPU and GPU memory, because the required data is iteratively swapped multiple times. Besides, the bandwidth of the commonly used PCIe 3.0 connecting CPU memory and GPU memory is only 12 GB/s in practice, and it is much smaller than the bandwidth of memory bus connected to CPU/GPU cores. As a result, the high transmission cost limits the efficiency of random walks on GPUs. For example, GPU-based graph sampling system NextDoor [15] reported that running DeepWalk and PPR on GPUs performs even worse than CPU-based system KnightKing [69] on a single machine due to the high data transmission cost.

Despite that the partition-based method is a standard way to achieve out-of-GPU-memory processing, we argue that it is still challenging to optimize the data transmission between CPU and GPU memory. First, the partition of random walks changes during the computation, because a walk may move to other partitions when the state of the walk is updated. Under the constraint of GPU memory capacity, it is a common practice [39], [60] to also organize the computation tasks (e.g., random walks) in a partition-centric manner, and load only the related data into GPU memory when processing each partition. Thus, we need to keep track of the walks in each partition as walks move among different partitions. In particular, we need to efficiently handle their memory allocation as CUDA does not efficiently support growing or shrinking memory space by dynamic memory reallocation [65]. Furthermore, recording the state of walks costs a lot of memory space so it is necessary to save them in CPU memory and optimize the transmission. Also, we need to alleviate the random access and data contention problems during the insertion of updated walks because adjacent threads may write to different partitions and many threads may also try to write the same partition.

Second, a large portion of the loaded graph is usually useless for updating random walks. Generally, we use GPUs for memory-intensive tasks such as neural networks [46] where every element of the loaded data is accessed. However, for random walks, a walk visits only one neighbour of its current vertex, leaving most edges to be useless. Moreover, the required edges are only known during the random walks execution. Zero copy [37] is a GPU-specific feature to allow GPU cores to access data directly through PCIe at a cacheline granularity during execution. However, it is not scalable in comparison with the high-bandwidth GPU memory. So the key is to provide a design to balance the trade-off.

Third, the computation workload changes in different iterations and it is often insufficient to fully utilize the GPU. For most applications on GPUs, the computation time is constant when data is evenly divided. Thus, using a simple pipeline can easily achieve a steady speedup because the ratio of computation to transmission time is the same in each iteration [40]. However, the computation workload for random walks over a partition depends on the number of walks in the partition, rather than the size of the graph partition. As random walk tasks dynamically move among partitions, simply adopting a pipelined design to overlap computation and transmission would not provide much benefit if the computation time is relatively too small. Also, in an iteration, only the random walks in the currently selected partition are processed with the partition-based method. This makes the computation workload in an iteration insufficient to hide the transmission overhead. So the key to improving the pipeline efficiency is to accumulate and schedule enough computation tasks in an iteration so that we can hide the data transmission time as much as possible.

Moreover, adopting the partition-based method introduces stragglers which further exacerbate the above issues. GraphWalker [60] and GraSorw [25] reported that even when most walks finish their computation, it still needs many iterations to process the small number of unfinished stragglers. So in later iterations, more data of the loaded graph become useless, and the problem of limited pipeline efficiency due to inadequate computation workload in an iteration becomes more serious.

To address the above challenges, we propose LightTraffic, which supports massive random walks on GPUs with an efficient out-of-GPU-memory design for both graph data and walk index. Specifically, LightTraffic optimizes the data transmission traffic between CPU and GPU memory with multiple design efforts. We summarize our main contributions as follows.

- We develop a memory-efficient scheme to manage graphs and walks. We follow the partition-based idea to organize graph data, while for updated walk indexes, we use fine-grained batches with append-only writes. We manage memory with reserved pools and allocate space for graph partitions and walk index batches in GPU memory in a cached fashion, so as to efficiently realize dynamic memory reallocation when walks are traveling among partitions. We also reduce the data contention and random access overheads via two-level walk index caching.
- We develop an efficient pipeline with preemptive scheduling and selective scheduling to reduce the data transmission cost. Preemptive scheduling breaks the dataflow of vanilla partition-based framework by fully utilizing all computable walks with required data already cached on GPUs. Selective scheduling differentiates partitions in loading to maximize the utilization of cached data. In addition, we develop adaptive scheduling by leveraging zero copy in graph loading to optimize data transmission.
- We implement a prototype system LightTraffic and conduct extensive experiments to show its effectiveness and efficiency in running massive random walks on GPUs. Results show that LightTraffic achieves $1.7-5.0\times$ and $1.4-12.8\times$ performance speedup compared to the state-of-the-art CPU-based systems optimized for random walks, FlashMob and ThunderRW, and it also significantly outperforms out-of-memory GPU-based graph processing system Subway.

The source code is available at https://github.com/ustcadsl/LightTraffic.

II. Background and Motivation

We first introduce the necessary background on random walk and its computation on GPUs, then analyze the limitations of leveraging GPU in speeding up massive random walks. Finally, we motivate LightTraffic and summarize the design challenges.

A. Random Walk on GPUs

Random walk algorithms. A simple random walk over an undirected graph $G = (V, E)$, where V denotes the vertex set and E denotes the edge set, proceeds as follows. A walk w starts from a vertex, and moves to one neighbor with an equal probability in each step. It repeats the process until the termination condition is satisfied, e.g., it finishes the required number of walking steps, which we call the *walk length*. The simple random walk can be extended to handle weighted or heterogeneous graphs in real applications for graph sampling, e.g. rejection sampling and alias sampling [55].

We point out that random walk serves as a very basic function, and it supports many graph applications, which usually require to concurrently run a large number of walks. For example, graph embedding [7], [45], [62] often takes hundreds of epochs to converge, and each epoch requires to concurrently run $|V|$ walks to sample a path. To support these applications, we have to record the state of each walk. In particular, we use two variables to characterize the state of a walk at each step: the `current_vertex`, which records the vertex at which the walk stays, and the `walked_steps`, which denotes the number of steps being moved. We call these data *walk index*. To support applications based on random walks, we may need other application-specific states, e.g., we need a visit frequency per vertex for PageRank related algorithms and a unique identifier per walk for sampling.

Computation on GPUs. As concurrently running billions of random walks also consumes a lot of CPU resources, recent works also focus on leveraging GPUs to speed up the computation. To realize it, one choice is to keep the whole graph and all computing data, e.g., the walk index for random walks, in GPU memory [15], [44]. However, this approach is infeasible to handle huge graphs and large-scale random walks,

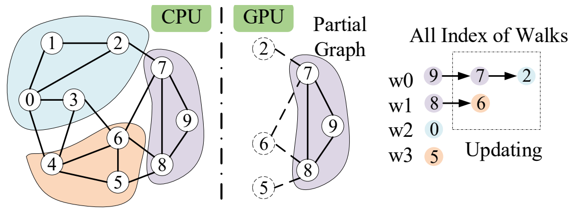


Fig. 1: Out-of-memory graph computation on GPUs. In the very beginning, we can update w_0 because v_9 belongs to the partial graph residing in GPU memory. After two steps, w_0 stays at v_2, but the partial graph in GPU memory does not contain all neighbors of v_2, so w_0 has to wait, and the computation resumes when another graph partition containing v_2 is loaded.

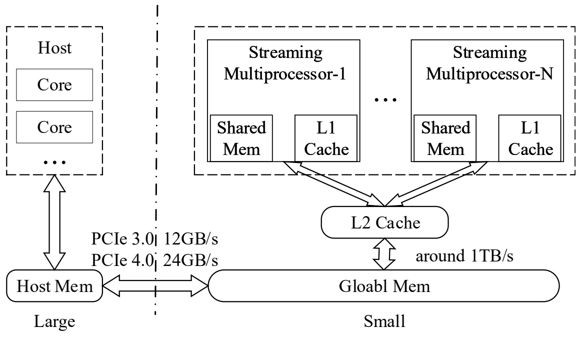


Fig. 2: GPU memory hierarchy.

due to the limited memory space on GPUs, e.g., the memory capacity is usually tens of gigabytes for commodity GPUs.

To address the scalability issue, state-of-the-art designs, e.g., C-SAW [44] and Subway [50], enable out-of-memory graph computation on GPUs. The key idea is to keep the full graph data in CPU memory and iteratively load only partial graph data into GPU memory for computation. Specifically, as illustrated in Figure 1, a graph is divided into multiple partitions, and they are transmitted to GPU for computing iteratively. In each iteration, only one graph partition is transmitted. Because we have only partial graph data in GPU memory, we are not able to continue updating a walk if it requires other parts of the graph not residing in GPU memory.

GPU-specific features. Note that the performance of computing on GPUs may be influenced by multiple GPU-specific features: (1) Memory hierarchy [17]. As illustrated in Figure 2, a GPU also has multi-layer caches. Each streaming multiprocessor has a private L1 cache and a shared memory with a latency of around 20 cycles, and these memory spaces are only accessible by threads in this streaming multiprocessor. Besides, all streaming multiprocessors share the L2 cache which has a latency of around 200 cycles. Both the L1 and L2 caches are controlled by the hardware, while the shared memory is programmable. (2) Zero copy [37]. CUDA provides memory access with zero copy, which allows the GPU programs to directly access the host memory at a cacheline granularity. It simplifies GPU programming and could be very helpful when GPU programs cannot tell which data is referenced before execution. However, random accesses through zero copy may significantly reduce the effective communication bandwidth between the host and GPU. (3) CUDA stream [42]. A stream is a sequence of operations performed in a given order, while operations in different streams can be interleaved. This allows to asynchronously overlap computation and transmission.

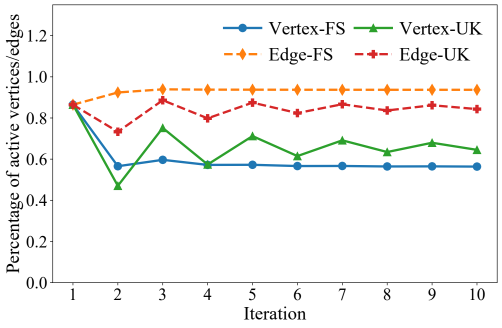


Fig. 3: Percentage of active vertices/edges in each iteration.

TABLE I: Time breakdown of running random walks on GPU.

Dataset	Computation	Transmission	Subgraph Creation
UK	11.2%	40.4%	48.4%
FS	2.0%	43.7%	54.3%

B. Limitations and Challenges

The above described partition-based approach enables processing large graphs on GPUs, but it still has multiple limitations to realize out-of-memory designs.

Inefficiency of graph data transmission. The partition-based approach involves iterative graph loading, introducing significant data transmission cost. Specifically, due to the randomness nature of random walks, it is very common to have only a small part of the loaded graph partition in GPU memory being really useful for updating the random walks. That is, many vertices and edges are unnecessarily transmitted with the partition-based iterative approach, and this introduces large transmission overhead. We point out that this limitation still exists even with the optimization of loading only the active subgraph [50], [74], which is generated by containing only the active vertices and active edges. A vertex is active if there is at least one walk currently staying at this vertex, and an edge is active if the source vertex of this edge is an active vertex. The reason is that when running massive walks, there is a large portion of active vertices, introducing large overhead if we dynamically generate active subgraph in each iteration. Besides, a walk visits only one neighbour of an active vertex in one step, so we still have many useless active edges. Thus, this optimization is not beneficial for running massive walks.

We run experiments to further justify the limitation. We run $2|V|$ random walks using the out-of-memory system Subway on FS and UK graph datasets (see §IV-A), and we enable the optimization of loading active graph to reduce data transmission overhead. As shown in Figure 3, we find that active vertices often account for a very large portion, e.g., around 60% vertices and 80% edges on UK are active in most iterations. In contrast, only about 3% of the loaded edges is actually used during the execution. As a result, it takes a very long time to generate the active subgraph in each iteration, and the transmission of active subgraph to GPU memory also incurs high overhead as shown in Table I. This experiment demonstrates the inefficiency of graph data transmission in existing partition-based designs for supporting out-of-memory random walks on GPUs.

Inefficiency of pipelining. To avoid GPU cores being idle and waiting for graph data transmission, many existing graph systems support overlapping the communication with computation. However, as the data transmission time is usually much higher than the computation time in an iteration when running random walks on GPUs, the pipeline efficiency is still very limited. To further demonstrate this limitation, we divide the graph of UK dataset into 128 MB partitions and run $2|V|$ walks with a length of 80, which is a common setting [68]. We find that loading a graph partition into GPU memory requires 10.4 milliseconds, while the highest computation time in an iteration is only 6.6 milliseconds using round-robin scheduling. Also, we find that most walks are moving to other partitions, but current partition-based method fails to continue the computation. Besides, without careful design, the loaded partition may be only useful for a few computation tasks. As a result, GPU cores are often at idle to wait for the graph data.

Inefficiency of walk index management. We find that all existing GPU-based random walk systems keep all the walks in GPU memory, this design consumes large GPU memory and thus limits the scale of random walks. For example, even recording only the state of each walk with `current_vertex` and `walked_steps` by using 8 bytes for each walk, it exceeds the GPU memory capacity for running $2|V|$ walks for the ClueWeb09 dataset [47]. Moreover, keeping all walks in GPU memory reduces the available memory for keeping graph data and thus increases the frequency of loading graph partitions. To support massive random walks, the only way is to run multiple rounds, and keep only a subset of walks in GPU memory in each round. However, it still results in high overhead because it requires more iterations to finish all random walks and thus introduces more traffic for transmitting graph partitions.

An alternative is to follow the idea in GraphWalker [60], which caches only a partial walk index in GPU memory for computation and evicts walks if needed. However, how to efficiently manage the cached walk index in GPU memory is challenging. To avoid scanning all walks, which introduces unnecessary transmission, we have to organize the walk index in a partition-based manner. However, it is unable to predict the memory space requirement for storing the walks belonging to a partition, because the number of walks residing in a given graph partition varies due to the dynamics of random walks. For example, some algorithms like PPR may initially start all walks at a single source vertex, so most walks are residing in one single partition in this case. Also, in an iteration, as all walks in the selected partition will be updated, the number of walks still belonging to this partition at the end of iteration drops to zero. Thus, the number of walks in a partition significantly varies. However, CUDA kernel function does not support dynamic memory reallocation during its execution like `realloc()` in C-type languages [65]. Thus, if we use a single consecutive memory space to manage walkers belonging to the same graph partition, e.g., the method used in C-SAW [44], the only choice is to reserve a large enough GPU memory space to keep the possible maximum number of walks for each partition so as to avoid memory overflow, but this reduces the memory utilization.

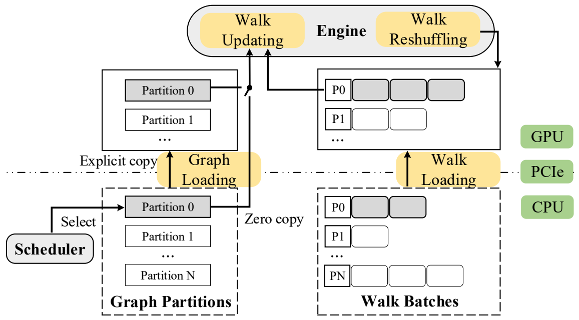


Fig. 4: Overall architecture of LightTraffic.

III. System Design

In this section, we present the design details of LightTraffic. We first introduce the overall idea, and then illustrate the details of each design component.

A. Overview

LightTraffic develops out-of-memory design as illustrated in Figure 4. The graph is divided into fixed-size partitions. The walks belonging to the same partition are grouped together and stored in multiple fixed-size small batches. To run random walks, LightTraffic adopts the partition-based approach, and in each iteration, the scheduler selects a partition, say partition i, to compute. First, we load graph partition i to GPU using *explicit copy* or *zero copy* (abbr. *graph loading*), and we also load the walk batches in partition i to GPU (abbr. *walk loading*). We skip the loading if the required data are already in GPU memory. Finally, the GPU starts multiple threads to compute all random walk tasks in partition i (abbr. *walk updating*). Note that after finishing the execution of all walks in a batch, these updated walks need to be inserted into the corresponding write frontier batches of other graph partitions (abbr. *walk reshuffling*). After finishing the execution of all walks over a partition, it enters into the next iteration if active walks still exist.

To optimize the random walk efficiency, LightTraffic develops multiple design optimizations. In particular, it designs a memory-efficient data organization for both graph data and walk index (see §III-B), and optimizes GPU memory management with a caching design and two-level caching for walk index to reduce the walk reshuffling overhead (see §III-C). LightTraffic also proposes a pipelined design to overlap data loading and computing, and develops preemptive scheduling and selective scheduling to improve the pipeline efficiency (see §III-D). It also leverages zero copy to deal with stragglers (see §III-E).

B. Data Organization

Memory pool reservation. To support out-of-memory processing, both the graph partition and walk index are fully stored in CPU memory, and only part of them are explicitly loaded by `cudaMemcpyAsync()` into GPU memory for computation in each iteration. We manage the memory spaces in GPU using

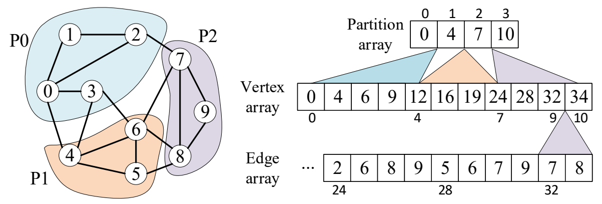


Fig. 5: Organization of graph data.

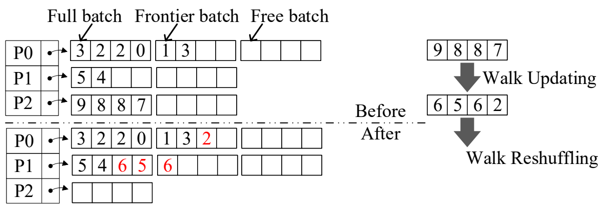


Fig. 6: Organization of walk index.

the idea of caching. We reserve the memory in both pools by allocating with `cudaMalloc()` in advance. Besides, the reserved memory in both pools are organized in unit of memory blocks, and the block size in the graph pool is the same as the partition size, while the block size in the walk pool is the same as the batch size. That is, the two pools operate independently as two caches without dynamic memory allocation. Specifically, we assume that the graph pool can cache at most m_g graph partitions, and the walk pool can cache at most m_w walks. Based on the cached design, if a walk batch is loaded into the walk pool, and there are already m_w walks being cached in the GPU memory, then an eviction of a walk batch is triggered. The graph pool operates in a similar way.

Graph data and partition. As illustrated in Figure 5, LightTraffic adopts the widely used CSR format for graph data storage, and it supports fast query of a given vertex's neighbors. Precisely, to find the neighbors of a vertex, we first access the vertex array to obtain the range in the edge array, and then access the edge array to get all neighbors.

LightTraffic adopts the partition-based approach to support out-of-memory processing for both graph data and walk index, and it uses static partitioning to avoid high overhead of dynamically creating active subgraphs. Specifically, LightTraffic adopts the range-based partition according to the order of vertices as shown in Figure 5. Specifically, we assign each vertex an identifier from 0 to $(|V|-1)$, and divide the vertices into multiple disjoint intervals. We assign an edge to a partition if its source vertex belongs to this partition. We create a graph pool to store graph partitions, and we ensure that every partition would not exceed the block size of the graph pool. The benefits of the above partition method are as follows. First, range partition makes the transmission of graph data in a contiguous order. Second, it is easy to make the size of partitions approximately fits any given size parameter. This can be realized by greedily expanding the vertex intervals until the size of graph partitions exceeds the pre-defined size. Third, given a vertex, we can efficiently find its corresponding partition using binary search.

Walk index. As illustrated in Figure 6, we use batches, which are small fixed-size arrays, as the basis for storing walk index. By default, the number of walks that are recorded in a batch is set as $16\times$ the number of GPU cores, such that the throughput of computing or transferring a full batch is large enough. Each batch must only contain walks belonging to the same partition so that we can always update walks in this batch given the corresponding graph partition. For ease of presentation, we say that a batch belongs to a partition, if all walks in the batch are currently staying in this partition.

As walks move among partitions, in order to efficiently insert and delete walks belonging to each partition, we organize all batches belonging to the same partition as a circular queue. Specifically, during computation, the walks in the head are fetched into GPU engine for processing in a batch basis. After the computation, this loaded batch is simply freed and all the updated walks are inserted into other batches according to their states. To handle walk insertion, the batch in the tail of each queue is called the write frontier batch, which receives the insertion of all updated walks belonging to this partition with append-only write. It is easy to load and evict walk batches between CPU and GPU memory. For example, to evict a walk batch from GPU memory to CPU memory, on the GPU side, we can simply fetch a batch from the given partition in GPU walk pool, then transfer the walk indexes in this batch, and finally free this batch in GPU memory. On the CPU side, we write the walk indexes evicted from GPU to a free batch and finally insert this batch to a given partition in CPU walk pool. Loading walk batches from CPU memory to GPU memory is similar to the above process.

Memory usage. We now analyze the memory usage of the walk pool in GPU memory. We keep only a frontier batch and a free batch for each graph partition in the walk pool. Suppose that we have P graph partitions in total, then the total number of frontier batches and free batches in GPU memory is $2P$. Thus, if the batch size is B, then the maximum memory size being wasted in the walk pool is just $(2P+1)B$. Besides, in order to to record the pointers to all batches, we maintain a circular queue for each partition with the length of $(\lceil \frac{S_w m_w}{B} \rceil + 2)$ where S_w is the size of the walk index and we cache at most m_w walks. We also need a queue to store free batches for allocation. Generally, we divide a large graph into hundreds of partitions (See Figure 17) and B is about 1 MB in our default setting. Thus, the space overhead is comparatively small and the size of the data structures in memory pool is only several megabytes.

C. Two-level Walk Index Caching

Recall that walks in the same batch are in the same partition. This invariant ensures that we can always update all walks in a batch with the corresponding graph partition. However, after these walks are updated, they may belong to different partitions. To preserve this invariant, we need to insert the updated walks into their corresponding frontier batches, and we call this process walk reshuffling.

We optimize the reshuffling process with two-level caching. The first-level cache uses the walk pool. Specifically, we cache

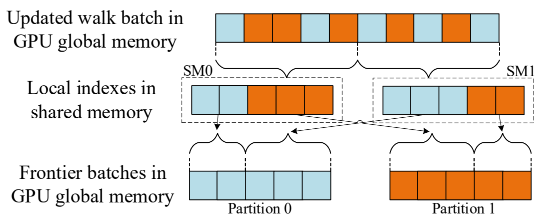

Updated walk batch in GPU global memory
Local indexes in shared memory
SM0
SM1
Frontier batches in GPU global memory
Partition 0
Partition 1

Fig. 7: Walk reshuffling with two-level caching.

all the write frontier batches in the walk pool, one for each graph partition. By doing this, updated walks are written to the frontier batches in the walk pool in GPU memory, so we avoid small updates to the walk index in CPU memory. It is possible that the remaining space of frontier batch is unable to keep all the updated walks, e.g., the partition 1 in Figure 6. To avoid memory overflow, we choose to reserve one free batch in the walk pool for each graph partition. If the frontier batch is full, then the reserved free batch becomes the frontier batch to store updated walks.

The second-level cache is designed to use the shared memory (see Figure 2 in §II). During walk reshuffling, each multiprocessor handles only a part of walks in the updated batch, i.e., the walk batch just being processed. To reduce data contention and random access overhead, we maintain a data structure called local index in the shared memory of each multiprocessor. As Figure 7 shows, walks belonging to the same partition are marked in the same color. To insert the updated walks to their corresponding frontier batches, we first insert the walks into the local index and sort them based on partitions, and then merge the results of each local index. Specifically, in each shared memory, we maintain an atomic local counter for each partition, which represents the number of walks in the local index belonging to the partition, and an inverted map, which records the mapping from the thread IDs to the current position in the updated walk batch, as well as the pairs of partition ID and offset which denote the target position in which each walk should be inserted.

Algorithm 1 presents the detailed workflow of walk updating and reshuffling. At Line 4, we assign a walk to a thread for walk updating. Note that writing to the frontier batches in global memory needs synchronization across multiprocessors, which must go through L2 cache and incur a relatively high latency. By using atomic local counters `localLen` in shared memory, synchronization is only needed at Line 12 of Algorithm 1, thus reducing the data contention overhead. Besides, by using inverted maps in shared memory, we coalesce the walks that are written to the same frontier batches to reduce random access overhead by assigning adjacent written addresses (denoted as (`part`, `pos`)) to adjacent threads. The inverted map sorts the written addresses, and it is built efficiently with the counting sort algorithm using the prefix sum of local counters [52].

D. Pipeline Design

To support out-of-memory processing for both graph data and walk index, loading graph partitions and walk batches still dominates the total execution time, due to the limited bandwidth between CPU and GPU memory. To improve the performance, LightTraffic leverages an efficient pipeline to reduce the data transmission overhead.

Algorithm 1 Kernel function of walk updating and reshuffling

```
1:  tid = blockDim * blockIdx + threadIdx ▷ global thread id
2:  i = threadIdx                          ▷ local thread id in SM
3:  if tid < batch.length() then
4:      batch[tid].update()                ▷ walk updating
5:      if batch[tid].isActive() then      ▷ insert to local index
6:          part = batch[tid].findPartition()
7:          pos = atomicAdd(&localLen[part], 1)
8:          invertedMap.add(part, pos, tid)
9:  invertedMap.sort()
10: if i < numPartition then ▷ get offset in walk pool to write
11:     offset[i] = atomicAdd(&globalLen[i], localLen[i])
12: if i < invertedMap.length() then       ▷ write to walk pool
13:     part, pos, j = invertedMap.get(i)
14:     globalIndex.add(part, pos + offset[part], batch[j])
```

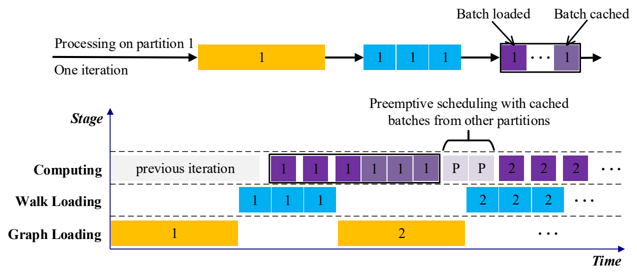


Fig. 8: Pipeline design.

To realize an efficient pipeline, as shown in Figure 8, we develop a 3-phase interleaving scheme by splitting one iteration into three phases: (1) graph loading, in which a graph partition is loaded from CPU memory to the graph pool, (2) walk loading, in which the corresponding batches belonging the loaded partition are loaded into the walk pool one by one; and (3) computing, in which the walks are processed based on the loaded graph partition. The intuitive way to realize pipeline is to interleave the three phases. Specifically, we start computation as long as one walk batch is loaded, and we enter into the next iteration to start loading new graph partitions as long as the loading phase finishes. Note that in the interleaved execution, when loading a walk batch, we may need to evict a cached walk batch from the walk pool to CPU memory. Eviction is fully integrated into our pipeline design because PCIe is full-duplex so that loading and eviction can be executed simultaneously without interference. We omit the eviction procedure in Figure 4, Figure 8 and Algorithm 2 for simplicity.

However, we find that simply interleaving the three phases is still inefficient to hide the transmission overhead. The main reason is that the loading phase takes much longer time than the computing phase. We further develop two optimizations: preemptive scheduling and selective scheduling.

Preemptive scheduling. One of the reasons why data transmission time dominates the total execution time is that the traditional partition-based method only computes tasks in a specific partition in an iteration for the consideration of data dependency. For example, as shown in Figure 8, when we finish computing random walks in partition 1, we are supposed to compute random walks in partition 2, but we have to wait for the transmission of graph partition 2 and thus the GPU cores are idle. However, we still have previously loaded graph partitions in GPU memory, and may also have some walks being reshuffled into these partitions, so these partitions have random walk tasks which can be immediately computed. We consider that tasks are in the sleeping state if they are supposed to run but currently waiting for data transmission, and define that tasks are in the ready state if their corresponding graph partition and walk index are both cached in GPU memory.

The key idea of preemptive scheduling is that instead of waiting for the random walk tasks in the current partition being loaded, we take advantage of the tasks with required data already being cached in GPU memory. Thus, we can immediately run these computable tasks belonging to other partitions in a preemptive way, that is, the tasks in the ready state preempt the tasks in the sleeping state. Specifically, in the phase of loading graph partition and walk batches, we find out the walk batches that can be immediately computed, e.g., the batches marked as P in Figure 8, then we dispatch these computable walk batches if we find GPU cores are idle. With preemptive scheduling, we can improve the GPU core utilization and thus improve the pipeline efficiency by leveraging the cached data already in GPU memory in the loading phase.

Selective scheduling. We also propose a selective scheduling policy by differentiating different graph partitions, and our goal is to maximize the computing load for each loaded graph partition so as to reduce the gap between the loading time and the computing time in each iteration, and this could further improve the pipeline efficiency. Specifically, in the first phase of loading a graph partition, we load the partition which has the largest number of walks. By doing this, the loaded graph partition can be used for executing more walks in the third phase. Besides, if there is no free space to cache the loaded graph partition in the graph pool, we overwrite the partition with the smallest number of walks. The rationale is that such a graph partition should have the lowest chance to be reused.

We also optimize the selection of walk batches. Specifically, when we need to compute a walk batch for preemptive scheduling, if there are some full batches whose corresponding graph partitions are already being kept in GPU memory, then we choose the batch whose corresponding graph partition contains the least walks. The rationale is that such a graph partition will be overwritten with high chance in the near future, so we try to finish all the walks belonging to this partition to avoid reloading the graph partition in the future. If no such a batch exists, then we select the batch containing the largest number of walks to amortize the cost of CUDA function call. We use the same way to evict walk batches.

In summary, our 3-phase interleaving scheme overlaps the computation with data transmission using the fine-grained walk batches. Furthermore, we develop preemptive scheduling and selective scheduling to help reduce the gap between the loading time and computing time by leveraging the cached data on GPUs. Preemptive scheduling allows some random walk tasks which are supposed to be processed in later iterations to be processed earlier, so it eliminates some iterations. Selective scheduling improves the utilization of the loaded partitions. They both help reduce the data transmission time and thus make the computation and transmission be well pipelined.

Algorithm 2 presents the detailed implementation. Note that existing systems [19], [44], [51] usually use the same CUDA stream to process both the loading and computing tasks of a partition to guarantee the ordering. However, this approach is hard to support preemptive scheduling and selective scheduling, because we require the information of the memory pool and need to dispatch walk batches from different partitions in a just-in-time manner. To address it, we create three CUDA streams to process the operations of computing, loading and eviction, respectively. We also add explicit synchronization between streams if data dependency exists.

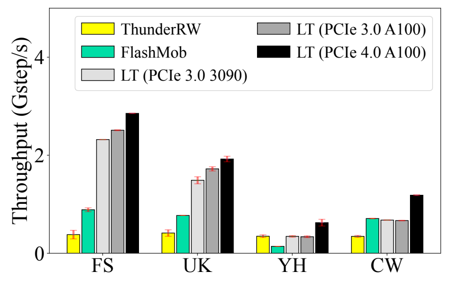
E. Adaptive Scheduling

With the pipeline design, we dispatch random walk tasks as many as possible while data are transmitted, so as to prevent GPU cores from being idle. However, we observe that in later iterations, only a few random walks, which we call the stragglers, are executed. In this case, we cannot find enough computation to hide the time of loading a whole graph partition.

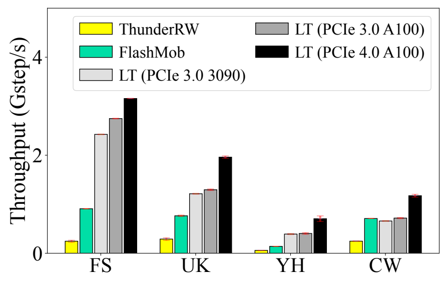
To address the straggler issue, we develop an adaptive scheduling policy by leveraging zero copy to reduce the graph loading overhead. Specifically, when the computing load is light for a graph partition, instead of explicitly loading the entire graph partition to GPU memory, we provide the address of the graph partition allocated by `cudaHostAlloc()` in host memory to the computation engine on GPU. During the

Algorithm 2 Scheduler of LightTraffic

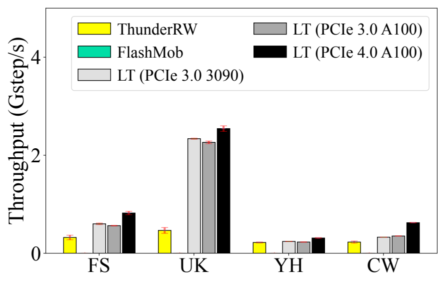
```
1:  while active walks exist do
2:      $i$ = scheduler.iteration()                          ▷ select a partition
3:      if !$G_{gpu}$.exists($i$) then                       ▷ graph loading
4:          if scheduler.shouldZerocopy($i$) then
5:              $G_{gpu}[i]$ = $G_{cpu}[i]$
6:          else
7:              $G_{gpu}[i]$ = copyAsync($G_{cpu}[i]$, loadStream)
8:      while busy(loadStream) do                            ▷ preemptive scheduling
9:          $j$ = scheduler.preemptive()
10:         kernelAsync($G_{gpu}[j]$, $W_{gpu}[j]$.poll(), compStream)
11:         cudaStreamSynchronize(compStream)
12:     for all batch in $W_{cpu}[i]$ do                     ▷ walk loading
13:         batch = copyAsync(batch, loadStream)
14:         cudaStreamSynchronize(loadStream)
15:         kernelAsync($G_{gpu}[i]$, batch, compStream)
16:     for all batch in $W_{gpu}[i]$ do                     ▷ computing
17:         kernelAsync($G_{gpu}[i]$, batch, compStream)
```



(a) Uniform Sampling



(b) PageRank



(c) Personalized PageRank

Fig. 9: Comparison with CPU-based random walk systems.

computation, the GPU device finds that the data are located in host memory, then it issues PCIe requests data in a cacheline.

One key issue of the adaptive scheduling is to decide when to use zero copy, and we use a simple analysis to decide the threshold. Note that when we decide to use zero copy, then the computation load should be very light, for ease of analysis, we assume that the computation time is negligible or can be fully overlapped by communication. With this approximation, we only have to consider the communication time, which is proportional to the transferred data size. For explicit copy, we explicitly transfer the whole graph partition, and we denote its size as S_p. For zero copy, we approximate the required data size by using an empirical formula αw, where w is the number of walks in the partition, and α denotes the average data size transferred using zero copy to finish the computation of one walk. Note that it is hard to accurately calculate α as we can not know how many steps each walk can move in one iteration, but we can empirically estimate it. We find that α is not a sensitive parameter, and it can be set as 256 bytes based on our experiments. Finally, based on the above empirical estimation, if the number of walks in a partition is small enough to satisfy $\alpha w < S_p$, then we use zero copy instead of explicit copy.

IV. Performance Evaluation

A. Setup

Testbed. By default, we conduct experiments on a single server equipped with two Intel(R) Xeon(R) Gold 5218R processors, 208 GB DRAM, and a single Nvidia GeForce RTX 3090 GPU with 24 GB memory, connected with PCIe 3.0. The programs run on Ubuntu 20.04 LTS and are compiled by nvcc 11.5 or g++ 9.4.0 with the -O3 flag.

Workload. Table II lists the graph datasets we considered where $d_{\max}$ is the largest vertex degree of a graph. It shows the statistics after preprocessing, which converts graphs into undirected ones, and removes self loops, duplicate edges and zero-degree vertices. We emphasize that UK, YH and CW are large graphs which cannot be entirely put into the GPU memory in our testbed. All the graphs can fit into DRAM. We define system throughput as the average number of processed steps per second. We run each experiment 5 times and report the average results. Except for the results of ThunderRW, the relative standard deviation is at most 5%.

TABLE II: Statistics of the graph datasets

Datasets	$\lvert V\rvert$	$\lvert E\rvert$	CSR Size	$d_{\max}$
LiveJournal(LJ) [24]	4.85 M	85.70 M	364 MB	20.33 K
Orkut(OR) [24]	3.07 M	234.4 M	917 MB	33.31 K
Twitter(TW) [24]	41.7 M	1.468 B	5.78 GB	3.00 M
FriendSter(FS) [63]	68.35 M	3.62 B	14.0 GB	5.21 K
UK-Union(UK) [4]	131.57 M	9.33 B	35.7 GB	6.37 M
Yahoo(YH) [66]	653.91 M	12.95 B	53.1 GB	653.91 M
ClueWeb09(CW) [47]	1.68 B	15.62 B	70.8 GB	6.44 M

Algorithms. We use three random walk based algorithms: Uniform sampling, PageRank [2], [43] and Personalized PageRank (PPR) [9], [43]. For PageRank and uniform sampling, walks are started uniformly at all vertices, and terminated when the number of walked steps reaches l. Note that PageRank adopts random walk with restart. That is, at each step, a walk has a probability p to restart at a random vertex, instead of uniformly selecting one of the neighbors of the current vertex. PPR starts all walks at the same source vertex. In our experiments, we select the source vertex that has the highest degree, and each walk terminates with probability p in each step. Besides the graph pool and walk pool, PageRank and PPR also store the visit frequency of each vertex in GPU memory. For uniform sampling, the walk index also records an application-specific state `walk_id` to specify the sampling path, and we do not store sampling paths in the GPU which executes random walks, and assume that paths are transferred to other GPUs like existing systems [67], [68]. For comparison with the state-of-the-art systems, we set $l = 80$ and $p = 0.15$, and set the number of walks as $2|V|$, which is a common setting [15], [25], [68]. Note that it is common to require a large number of samples, e.g., DeepWalk [45] samples $30|V|$ walks for convergence and metapath2vec [7] even requires to sample $1000|V|$ in their evaluations. Besides, Das Sarma et al. [6] infer that it needs $O(\frac{1}{\delta}|V|\log|V|)$ steps in total to make the relative error bounded by δ with high probability for PageRank.

B. Overall Performance

To evaluate the efficiency of LightTraffic, we first compare it with the state-of-the-art CPU-based systems ThunderRW [55] and FlashMob [68], as they are both highly optimized for random walks. We also compare LightTraffic with the GPU-based system Subway [50] which supports out-of-memory processing for graph data, as well as the GPU-accelerated random walk system NextDoor [15]. All experiments include the data transmission time. We do not compare with the recently proposed GPU-based graph system C-SAW [44], because C-SAW is not designed for running massive random walks and it runs out of GPU memory even when we try to run 100,000 walks. The reason is that C-SAW creates a large queue to store all walks for every step and every partition.

Comparison with ThunderRW and FlashMob. Figure 9 compares the performance of different random walk algorithms, and we also show the error bars of 95% confidence interval. FlashMob supports only fixed-length random walks, so the result of PPR algorithm using FlashMob is not available. For LightTraffic (abbr. LT), we consider both PCIe 3.0 and PCIe 4.0 by also using another platform equipped with a Nvidia Tesla A100 GPU connected with PCIe 4.0. To study the impact of PCIe bandwidth, we use the same A100 GPU to simulate different bandwidths so as to make sure other settings are the same. Precisely, we limit the bandwidth of PCIe 4.0 by transmitting data twice to simulate the same bandwidth as PCIe 3.0. We also limit to use the same amount of memory, i.e., 24 GB for the Tesla A100 GPU for fair comparison.

The results show that the time for loading graph may become the bottleneck due to the limited bandwidth of PCIe 3.0. However, LightTraffic still achieves up to 6.7× speedup when running fixed-length random walks on large graphs that cannot fit into GPU memory. For the small FS graph which can fit in the GPU memory, LightTraffic achieves at least 6.1× speedup because we only load graph partitions once. PPR is a variable-length random walk algorithm where the number of steps follows the geometric distribution. For PPR, LightTraffic achieves an average speedup of 2.0×. We note that the benefit of our system decreases in this case, because there are fewer walks in each graph partition due to the variable walk length, while we still have to continuously load the graph partitions for computation, incurring larger transmission overhead.

The results under PCIe 4.0 show that the increased bandwidth significantly improves the performance of LightTraffic. In this case, LightTraffic achieves $1.7-5.0\times$ and $1.4-12.8\times$ performance speedup over FlashMob and ThunderRW. We point out that as the size of the graph dataset keeps increasing, the data transmission traffic between the host and GPU also becomes larger, so using higher transmission bandwidth with PCIe 4.0 is effective to reduce the overall running time. However, as PCIe 4.0 is still much slower than memory, the data transmission cost is still critical. Note that new technology such as NVLink 2.0 can connect the GPU to CPU memory at a high bandwidth of 64 GB/s [32], [33], so it provides an opportunity to achieve better performance when using fast interconnects.

Comparison with Subway. We now compare with the state-of-the-art GPU-based system Subway, which supports out-of-memory processing. Figure 10 shows the speedup results when running PageRank and PPR on FS and UK. We omit the results of Uniform Sampling because the conclusion is similar. Besides, the results on large datasets of YH and CW are unavailable because Subway runs out of the host memory because of dynamically generating active subgraphs. Note that the total time of Subway includes the cost of dynamically generating active subgraphs.

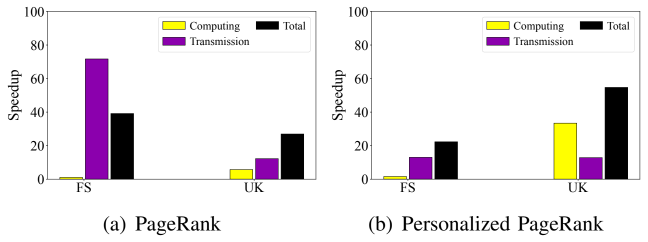


Fig. 10: Comparison with out-of-memory GPU-based system.

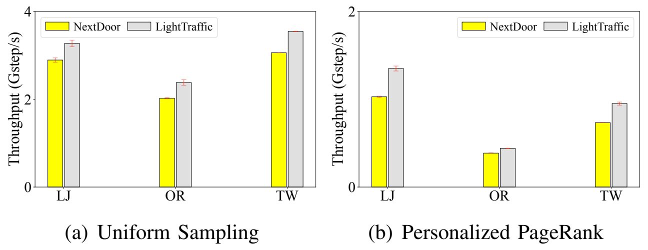


Fig. 11: Comparison with out-of-memory GPU-based system.

LightTraffic significantly outperforms Subway. Specifically, for PageRank, LightTraffic achieves 39.1× and 26.9× speedup in total time, 1.04× and 5.72× speedup in computing, 71.7× and 12.2× speedup in data transmission, on the datasets of FS and UK, respectively. For PPR, LightTraffic achieves 22.3× and 54.7× speedup in total time, 1.58× and 33.4× speedup in computing, 13.0× and 12.9× speedup in data transmission. The reason why LightTraffic speeds up the computation compared with Subway is as follows. Subway adopts the vertex-centric computation framework, i.e., it assigns all walks residing at the same vertex to a single thread, so it causes load imbalance among vertices due to the variable number of walks. LightTraffic achieves load balance by adopting the walk-centric computation model used by most random walk systems. Besides, benefited from the scheduling techniques, LightTraffic also significantly reduces the transmission cost.

Comparison with NextDoor. To demonstrate that LightTraffic has comparable performance with in-GPU-memory systems, we compare with the state-of-the-art GPU-accelerated random walk system NextDoor, and consider the same graph dataset as NextDoor so as to make all data fit into the GPU memory. As shown in Figure 11, LightTraffic still slightly outperforms NextDoor [15]. The performance gain comes from two optimizations. First, LightTraffic leverages an efficient pipeline to hide data transmission. Second, the two-level walk index caching also prevents the partition-based walk management from becoming a major overhead.

C. *Effectiveness of the Design Techniques*

Walk reshuffling. We first study the effectiveness of walk reshuffling in LightTraffic. Recall that walk reshuffling is to write the walks in the finished batch to the corresponding frontiers. We compare the design of two-level caching in LightTraffic with a basic method, which directly accesses the global memory in GPU to write the updated walks. We denote this baseline method as *direct write*. As shown in Figure 12, the

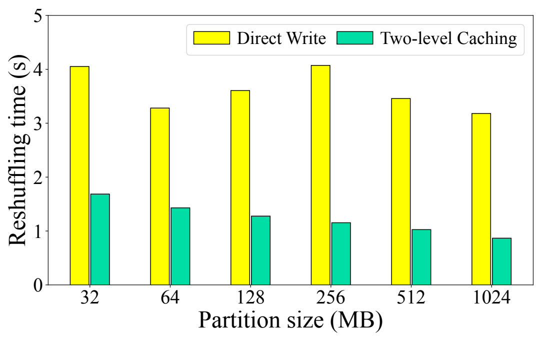


Fig. 12: Efficiency of walk reshuffling with two-level caching.

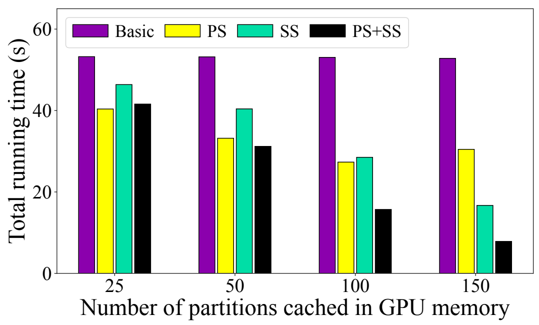


Fig. 13: Efficiency of pipeline design.

TABLE III: Impact of scheduling on data transmission.

	Baseline	PS	SS	PS+SS
Number of iterations	10670.8	6673.8	10513.6	6103.8
Number of explicit copies	8365.6	4222.2	4176.6	2380.4
Cache hit rate of graph pool	21.6%	36.7%	60.3%	61.0%

reshuffling time can be reduced by 73% at most. Besides, for large partition sizes, the reshuffling time is smaller, as there are fewer random writes in this case due to the reduced number of partitions. Note that without this optimization, the reshuffling time may even exceeds the walk updating time, and we will show it later (see Figure 17).

Pipeline design. We now study the effectiveness of the pipeline design. Recall that we develop two optimizations in LightTraffic: preemptive scheduling (abbr. PS) and selective scheduling (abbr. SS) (see §III-D). To show the effectiveness of the two optimizations, we consider a basic design, which uses round robin for loading graph partition and FIFO for eviction, and also allows to overlap the communication with computation between two successive iterations. Figure 13 shows the total running time by varying the number of graph partitions in GPU memory. The basic pipeline method does not leverage the cached data at all, because it considers only two successive iterations. With the two techniques PS and SS, the running time decreases when more partitions are cached in GPU memory.

Table III further shows the number of iterations and explicit copies when we cache 100 graph partitions in GPU. We find that both preemptive scheduling and selective scheduling optimize the data transmission. By dispatching random walks belonging to the partitions loaded in later iterations, preemptive scheduling reduces the number of iterations and thus reduces data transmission traffic. For selective scheduling, although it does not significantly reduce the number of iterations, it increases the cache hit ratio, and thus also reduces the data transmission traffic. As the two methods are orthogonal, combining them together in LightTraffic achieves higher speedup.

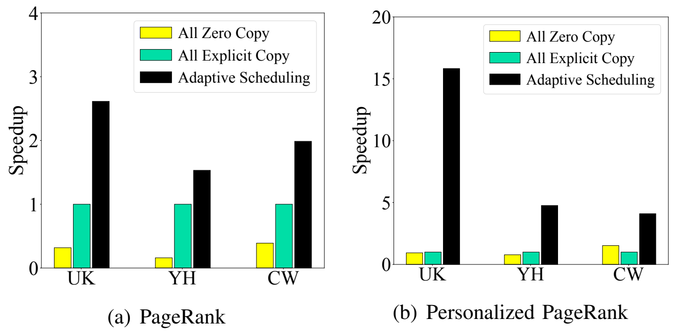


Fig. 14: Efficiency of adaptive scheduling.

Adaptive scheduling with zero copy. We now study the efficiency of the adaptive scheduling scheme, which leverages zero copy to handle the stragglers. For comparison, we consider three cases: (1) *All Zero Copy*: which always access graph data through zero copy during the computation and never loads full graph partitions to the GPU graph pool; (2) *All Explicit Copy*, which uses `cudaMemcpyAsync()` to load the whole graph partition from host memory if needed; (3) *Adaptive Scheduling*, which is the design used in LightTraffic (see §III-E). We consider two different algorithms PageRank and PPR. Note that PPR has variable walk lengths following geometric distribution, so it is more likely to encounter stragglers which are not finished after running many iterations. PageRank uses fixed walk length, and there still exists stragglers due to the asynchronous computation. We focus on the graphs which cannot fit into GPU memory. Figure 14 shows the speedup over the *All Explicit Copy* method. By balancing the trade-off between zero copy and explicit copy, adaptive scheduling achieves significant speedup compared with the simple scheme using only explicit copy or zero copy. Besides, we find that the speedup is more significant for PPR as the straggler issue becomes more critical due to the variable walk lengths.

Comparison with the approach using multiple rounds. To support running massive random walks on GPUs, another intuitive choice is to divide all walks into multiple sets, each of which contains a smaller number of walks that can fit into GPU memory, and then sequentially execute all sets of random walks in multiple rounds (see §II-B). To show the effectiveness of the out-of-memory walk index design in LightTraffic, we compare it with the multi-round baseline. Specifically, we aim to run 800 million random walks in total, and we consider three cases in which the GPU memory can only store 100 million, 200 million, and 400 million walks, so using the baseline approach needs to run 8, 4, and 2 rounds, respectively. For each case, we assume that LightTraffic caches the same number of walks in GPU memory for fair comparison, so walk eviction may be triggered in LightTraffic. We show the slowdown of the multi-round baseline, i.e., the ratio of the time using the baseline approach to that of using LightTraffic with the

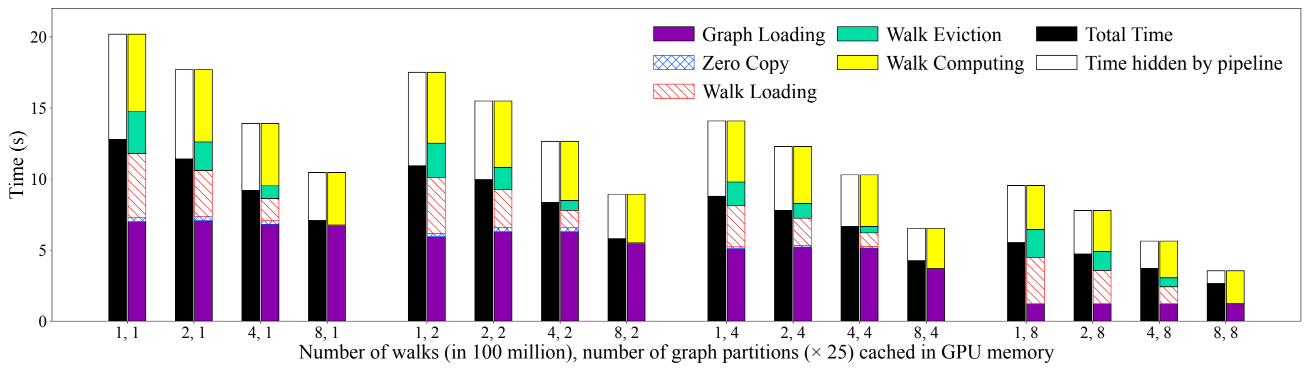


Fig. 15: Running time under different memory pool sizes.

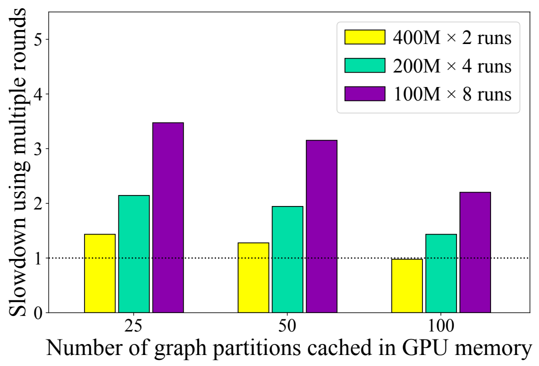


Fig. 16: Comparison with the baseline of using multiple rounds.

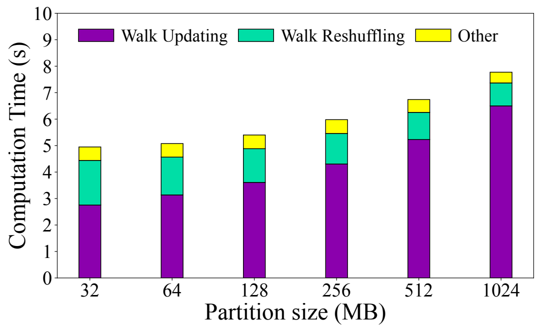


Fig. 17: Walk computing time under different partition sizes.

same memory capacity constraint. Figure 16 shows that using multiple rounds to support massive random walks incurs larger time cost, especially when the GPU can only store a small number of graph partitions. For example, it needs 3.5× time in the case where the GPU memory can only cache 25 graph partitions. This is because the multi-round approach incurs more times of graph partition loading. Moreover, the benefit of LightTraffic due to the out-of-memory walk index support is larger when the limit on GPU memory is more severe.

D. Sensitivity Analysis

Impact of GPU memory size. Figure 15 shows the impact of GPU memory size, and we vary the number of walks (in millions) and the number of graph partitions that can be cached in GPU memory. Under each setting, we show the time cost of each operation, including graph loading, walk loading, zero copy, walk eviction, walk computing on GPU cores, and the total running time. Note that the total running time is smaller than the sum of the time costed by each operation due to the pipeline design. The total running time is measured at the CPU side. The breakdown results are measured at the GPU side, and we set the walk length as 10 in this experiment so that each part of the breakdown results is visible in the figure. We show only the results of PageRank due to space limit.

The results show that graph loading and walk loading cost a significant amount of time in many cases, which may be even larger than the walk computing time. The results also demonstrate the effectiveness of the pipeline design because the total time is close to the maximum time of two stages: the walk computing stage and the data loading stage which includes graph loading, walk loading and zero copy. We can also find that given the same number of graph partitions being cached, caching more walks in GPU memory significantly reduces the running time. For example, in the case where 25 partitions are cached, the running time reduces from 12.8 seconds to 7.1 seconds when the cached number of walks increases from 100 million to 800 million. This justifies the necessity of efficient walk index management when designing out-of-memory systems. Also, by increasing the memory pool size, it is more likely to find a partition which has both the graph and full batch cached in GPU when we apply the preemptive scheduling, thus it improves the throughput of walk computing.

Impact of partition size. Figure 17 shows the walk computing time under different partition sizes. To be specific, for the walk computing time, we show the time breakdown, including the walk updating time, walk reshuffling time, and others. We see that the time spent on GPU cores for updating walks increases as the partition size increases, because using large partitions has poor locality of memory references [31], while the time of walk reshuffling decreases, as it reduces the time to search the corresponding partition. In addition, we find that the partition size is not a very sensitive parameter.

Scalability analysis. To demonstrate the scalability of LightTraffic, we first present a theoretical analysis on the throughput lower bound by considering an extreme case with very limited GPU memory. Suppose that none of the required data is saved in GPU memory and all w walks in the same partition move only one step, so in each iteration, the total transferred data size is $S_p + wS_w$, where S_p is the graph

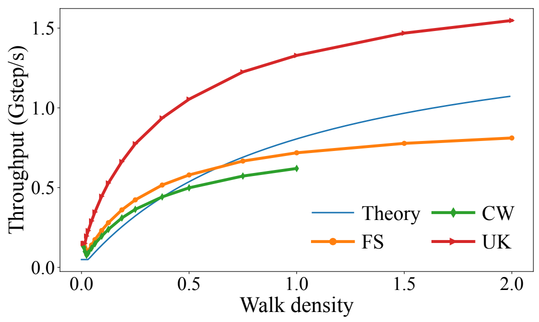


Fig. 18: Scalability analysis regarding walk density.

partition size and S_w is the walk index size of each walk. We assume that computing a walk is faster than transmitting a walk, so the computation time is hidden by pipeline. Thus, the total running time of one iteration is $\frac{S_p+wS_w}{B}$ where B is the transmission bandwidth, and the throughput is $\frac{wB}{S_p+wS_w}$. We define *walk density* $D = \frac{wS_w}{S_p}$, then the throughput is $\frac{B/S_w}{1+1/D}$. Also, when $D < \frac{S_w}{\alpha}$, zero copy is enabled to improve the throughput by avoiding the load of the whole graph partition. The above analysis implies that the throughout depends only on the walk density D regardless of the graph size, so LightTraffic can scale well for large graphs even with limited GPU memory.

To further demonstrate, we also run experiment by varying the walk density with a very restricted memory constraint where only 1 GB graph data and 1 GB walk index are allowed to be stored in GPU memory. In this case, less than 2% of graph is available for random walks for the large dataset CW. Figure 18 shows the theoretical estimation and the experimental results on both small and large datasets. The results of YH is not available because it has a high-degree vertex so storing its neighbors costs more than 1 GB memory. However, we could use smaller partitions by splitting the vertex [41]. The results conform with our theoretical analysis. That is, even with restricted memory, the throughput depends on the walk density rather than the graph size, so LightTraffic is scalable to process large graphs.

V. Related Work

CPU-based graph systems. Graph computing has received a lot of attentions. To address the scalability for processing large graphs, multiple disk-resident graph systems [1], [20], [22], [28], [35], [48], [49], [56], [73], [77] and distributed graph systems are also proposed [5], [12], [13], [30], [36], [76], and they mainly adopt the partition-based computation framework. Different from them, LightTraffic focuses on GPUs.

GPU-based graph processing. Many systems [18], [27], [29], [41], [61], [75] leverage the resourceful computing power of GPUs for accelerating graph computation. The key challenges for efficient graph computing on GPUs are the limited memory capacity and limited bandwidth between CPU and GPU. Prior works address the challenges by following the partition-based approach [11], [19], [34], [39], [50], [51], [74] and propose various optimizations, such as hybrid CPU-GPU co-processing [11], [34], loading only active subgraphs to optimize data transfer [50], [74], and designing with pipeline [19], [34], [39], [51], [74]. Recent works also optimize the memory access with zero copy [37], [38] and unified virtual memory [10], [59] to support out-of-GPU-memory graph processing. FaimGraph [64] uses efficient queuing structures to realize dynamic reallocation for in-memory dynamic graph analytics under the constraint of GPU memory capacity. Different from them, LightTraffic focuses on running massive random walks on GPUs with out-of-memory support.

Random walk systems. There have been increasing interests in random walks in recent years. Traditional graph systems mainly adopt the vertex-centric computation model which is not suitable for random walks. Because random walk is an embarrassingly parallel application, most random walk systems use walk-centric computation model. KnightKing [69] develops a distributed system for running massive random walks. DrunkardMob [21] and GraphWalker [60] adopt a single machine and propose disk-based random walk systems for handling large graphs that cannot fit into DRAM. Recent works [25], [53] also study the trade-off between time and space overheads of different sampling algorithms and improve the I/O utilization to support second-order random walks on large graphs. FlashMob [68] and ThunderRW [55] address the irregular memory access patterns, in particular, FlashMob [68] improves random walk at cache efficiency and ThunderRW [55] hides memory access latency via step-centric programming model and step interleaving. Recent efforts also try to enhance the parallelism using GPUs. C-SAW [44] and Skywalker [58] optimize inverse transform sampling and alias sampling on GPUs. NextDoor [15] leverages both caching and scheduling to improve random walk efficiency on GPUs. These random walk systems either use CPUs for computing or focus on optimizing the computation on GPUs, while LightTraffic focuses on optimizing the data transmission traffic when enabling out-of-memory processing on GPUs.

VI. Conclusion

In this paper, we developed an out-of-memory GPU-based system LightTraffic. LightTraffic supports out-of-memory management for both graph data and walk index, so it is able to run massive random walks over large graphs on a single GPU, LightTraffic mainly optimizes the data transmission traffic with multiple design optimizations to support fully out-of-memory processing. Experiments on our prototype demonstrate that LightTraffic outperforms existing CPU-based and GPU-based systems for running massive random walks. Despite that random walk has a very irregular memory access pattern, our results reveal that we can still improve the performance by exploiting the opportunities of intelligent pipeline and GPU-specific features such as zero copy and shared memory.

Acknowledgment

The work was supported in part by NSFC (62172382) and Youth Innovation Promotion Association CAS. The work of John C.S. Lui was supported in part by the RGC's GRF-14200321. Yongkun Li is USTC Tang Scholar, and he is the corresponding author.

References

[1] Z. Ai, M. Zhang, Y. Wu, X. Qian, K. Chen, and W. Zheng. Squeezing out all the value of loaded data: An out-of-core graph processing system with reduced disk I/O. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017.

[2] K. Avrachenkov, N. Litvak, D. Nemirovsky, and N. Osipova. Monte carlo methods in pagerank computation: When one iteration is sufficient. *SIAM J. Numer. Anal.*, 2007.

[3] L. Backstrom and J. Leskovec. Supervised random walks: Predicting and recommending links in social networks. In *Proceedings of the Fourth ACM International Conference on Web Search and Data Mining*, 2011.

[4] P. Boldi, M. Santini, and S. Vigna. A large time-aware graph. *SIGIR Forum*, 2008.

[5] R. Chen, J. Shi, Y. Chen, and H. Chen. Powerlyra: differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems*, 2015.

[6] A. Das Sarma, A. R. Molla, G. Pandurangan, and E. Upfal. Fast distributed pagerank computation. *Theoretical Computer Science*, 2015.

[7] Y. Dong, N. V. Chawla, and A. Swami. Metapath2vec: Scalable representation learning for heterogeneous networks. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2017.

[8] C. Eksombatchai, P. Jindal, J. Z. Liu, Y. Liu, R. Sharma, C. Sugnet, M. Ulrich, and J. Leskovec. Pixie: A system for recommending 3+ billion items to 200+ million users in real-time. In *Proceedings of the 2018 World Wide Web Conference*, 2018.

[9] D. Fogaras, B. Rácz, T. Sarlós, and K. Csalogány. Towards scaling fully personalized pagerank: Algorithms, lower bounds, and experiments. *Internet Mathematics*, 2005.

[10] P. Gera, H. Kim, P. Sao, H. Kim, and D. Bader. Traversing large graphs on gpus with unified memory. *Proc. VLDB Endow.*, 2020.

[11] A. Gharaibeh, L. Beltrão Costa, E. Santos-Neto, and M. Ripeanu. A yoke of oxen and a thousand chickens for heavy lifting graph processing. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, 2012.

[12] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed Graph-Parallel computation on natural graphs. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012.

[13] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: Graph processing in a distributed dataflow framework. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014.

[14] A. Grover and J. Leskovec. Node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016.

[15] A. Jangda, S. Polisetty, A. Guha, and M. Serafini. Accelerating graph sampling for graph machine learning using gpus. In *Proceedings of the Sixteenth European Conference on Computer Systems*, 2021.

[16] J. Jia, B. Wang, and N. Z. Gong. Random walk based fake account detection in online social networks. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2017.

[17] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza. Dissecting the nvidia volta gpu architecture via microbenchmarking. https://arxiv.org/abs/1804.06826, 2018.

[18] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan. Cusha: Vertex-centric graph processing on gpus. In *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing*, 2014.

[19] M.-S. Kim, K. An, H. Park, H. Seo, and J. Kim. Gts: A fast and scalable graph processing method based on streaming topology to gpus. In *Proceedings of the 2016 International Conference on Management of Data*, 2016.

[20] P. Kumar and H. H. Huang. G-store: High-performance graph store for trillion-edge processing. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016.

[21] A. Kyrola. Drunkardmob: Billions of random walks on just a pc. In *Proceedings of the 7th ACM Conference on Recommender Systems*, 2013.

[22] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: Large-Scale graph computation on just a PC. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012.

[23] S. Lee, S.-i. Song, M. Kahng, D. Lee, and S.-g. Lee. Random walk based entity ranking on graph for multidimensional recommendation. In *Proceedings of the Fifth ACM Conference on Recommender Systems*, 2011.

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

[25] H. Li, Y. Shao, J. Du, B. Cui, and L. Chen. An i/o-efficient disk-based graph system for scalable second-order random walk of large graphs. *Proc. VLDB Endow.*, 2022.

[26] R.-H. Li, J. X. Yu, X. Huang, and H. Cheng. Random-walk domination in large graphs. In *2014 IEEE 30th International Conference on Data Engineering*, 2014.

[27] H. Liu and H. H. Huang. Enterprise: breadth-first graph traversal on gpus. In *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015.

[28] H. Liu and H. H. Huang. Graphene: Fine-Grained IO management for graph computing. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, 2017.

[29] H. Liu and H. H. Huang. SIMD-X: Programming and processing of graph algorithms on GPUs. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019.

[30] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 2012.

[31] S. Lu, S. Sun, J. Paul, Y. Li, and B. He. Cache-efficient fork-processing patterns on large graphs. In *Proceedings of the 2021 International Conference on Management of Data*, 2021.

[32] C. Lutz, S. Breß, S. Zeuch, T. Rabl, and V. Markl. Pump up the volume: Processing large data on gpus with fast interconnects. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020.

[33] C. Lutz, S. Breß, S. Zeuch, T. Rabl, and V. Markl. Triton join: Efficiently scaling to a large join state on gpus with fast interconnects. In *Proceedings of the 2022 International Conference on Management of Data*, 2022.

[34] L. Ma, Z. Yang, H. Chen, J. Xue, and Y. Dai. Garaph: Efficient GPU-accelerated graph processing on a single machine with balanced replication. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017.

[35] S. Maass, C. Min, S. Kashyap, W. Kang, M. Kumar, and T. Kim. Mosaic: Processing a trillion-edge graph on a single machine. In *Proceedings of the Twelfth European Conference on Computer Systems*, 2017.

[36] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, 2010.

[37] S. W. Min, V. S. Mailthody, Z. Qureshi, J. Xiong, E. Ebrahimi, and W.-m. Hwu. Emogi: Efficient memory-access for out-of-memory graph-traversal in gpus. *Proc. VLDB Endow.*, 2020.

[38] S. W. Min, K. Wu, S. Huang, M. Hidayetoğlu, J. Xiong, E. Ebrahimi, D. Chen, and W.-m. Hwu. Large graph convolutional network training with gpu-oriented data communication architecture. *Proc. VLDB Endow.*, 2021.

[39] J. Mohoney, R. Waleffe, H. Xu, T. Rekatsinas, and S. Venkataraman. Marius: Learning massive graph embeddings on a single machine. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, 2021.

[40] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019.

[41] A. H. Nodehi Sabet, J. Qiu, and Z. Zhao. Tigr: Transforming irregular graphs for gpu-friendly graph processing. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018.

[42] Nvidia. Cuda c++ programming guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, 2022.

[43] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web, 1998.

[44] S. Pandey, L. Li, A. Hoisie, X. S. Li, and H. Liu. C-saw: A framework for graph sampling and random walk on gpus. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020.

[45] B. Perozzi, R. Al-Rfou, and S. Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD*

International Conference on Knowledge Discovery and Data Mining, 2014.

[46] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020.

[47] R. Rossi and N. Ahmed. Network repository. https://networkrepository.com, 2013.

[48] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015.

[49] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013.

[50] A. H. N. Sabet, Z. Zhao, and R. Gupta. Subway: Minimizing data transfer during out-of-gpu-memory graph processing. In *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020.

[51] D. Sengupta, S. L. Song, K. Agarwal, and K. Schwan. Graphreduce: Processing large-scale graphs on accelerator-based systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015.

[52] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for gpu computing. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, 2007.

[53] Y. Shao, S. Huang, X. Miao, B. Cui, and L. Chen. 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*, 2020.

[54] A. Sharma, J. Jiang, P. Bommannavar, B. Larson, and J. Lin. Graphjet: Real-time content recommendations at twitter. *Proc. VLDB Endow.*, 2016.

[55] S. Sun, Y. Chen, S. Lu, B. He, and Y. Li. Thunderrw: An in-memory graph random walk engine. *Proc. VLDB Endow.*, 2021.

[56] K. Vora. LUMOS: Dependency-Driven disk-based graph processing. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019.

[57] J. Wang, P. Huang, H. Zhao, Z. Zhang, B. Zhao, and D. L. Lee. Billion-scale commodity embedding for e-commerce recommendation in alibaba. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2018.

[58] P. Wang, C. Li, J. Wang, T. Wang, L. Zhang, J. Leng, Q. Chen, and M. Guo. Skywalker: Efficient alias-method-based graph sampling and random walk on gpus. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2021.

[59] P. Wang, J. Wang, C. Li, J. Wang, H. Zhu, and M. Guo. Grus: Toward unified-memory-efficient high-performance graph processing on gpu. *ACM Trans. Archit. Code Optim.*, 2021.

[60] R. Wang, Y. Li, H. Xie, Y. Xu, and J. C. Lui. 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)*, 2020.

[61] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens. Gunrock: A high-performance graph processing library on the gpu. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2016.

[62] W. Wei, Y. Wang, P. Gao, S. Sun, and D. Yu. A distributed multi-gpu system for large-scale node embedding at tencent. https://arxiv.org/abs/2005.13789, 2021.

[63] WeST. Friendster dataset. https://west.uni-koblenz.de/konect/networks/friendster, 2012.

[64] M. Winter, D. Mlakar, R. Zayer, H.-P. Seidel, and M. Steinberger. faimgraph: High performance management of fully-dynamic graphs under tight memory constraints on the gpu. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2018.

[65] M. Winter, M. Parger, D. Mlakar, and M. Steinberger. Are dynamic memory managers on gpus slow? a survey and benchmarks. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2021.

[66] yahoo. Yahoo webscope program. http://webscope.sandbox.yahoo.com, 2002.

[67] J. Yang, D. Tang, X. Song, L. Wang, Q. Yin, R. Chen, W. Yu, and J. Zhou. Gnnlab: A factored system for sample-based gnn training over gpus. In *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022.

[68] K. Yang, X. Ma, S. Thirumuruganathan, K. Chen, and Y. Wu. Random walks on huge graphs at cache efficiency. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021.

[69] K. Yang, M. Zhang, K. Chen, X. Ma, Y. Bai, and Y. Jiang. Knightking: A fast distributed graph random walk engine. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019.

[70] P. Yi, H. Xie, Y. Li, and J. C. Lui. A bootstrapping approach to optimize random walk based statistical estimation over graphs. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, 2021.

[71] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2018.

[72] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, and V. Prasanna. Graphsaint: Graph sampling based inductive learning method. In *International Conference on Learning Representations*, 2020.

[73] D. Zheng, D. Mhembere, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay. FlashGraph: Processing Billion-Node graphs on an array of commodity SSDs. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, 2015.

[74] L. Zheng, X. Li, Y. Zheng, Y. Huang, X. Liao, H. Jin, J. Xue, Z. Shao, and Q.-S. Hua. Scaph: Scalable GPU-Accelerated graph processing with Value-Driven differential scheduling. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020.

[75] J. Zhong and B. He. Medusa: Simplified graph processing on gpus. *IEEE Transactions on Parallel and Distributed Systems*, 2014.

[76] X. Zhu, W. Chen, W. Zheng, and X. Ma. Gemini: A Computation-Centric distributed graph processing system. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016.

[77] X. Zhu, W. Han, and W. Chen. GridGraph: Large-Scale graph processing on a single machine using 2-level hierarchical partitioning. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, 2015.