Optimizing GPU-based Graph Sampling and Random Walk for Efficiency and Scalability

Pengyu Wang, Cheng Xu, Chao Li, *Senior Member, IEEE,* Jing Wang, Taolei Wang, Lu Zhang, Xiaofeng Hou, *Member, IEEE* and and Minyi Guo, *Fellow, IEEE*

Abstract—Graph sampling and random walk algorithms are playing increasingly important roles today because they can significantly reduce graph size while preserving structural information, thus enabling computationally intensive tasks on large-scale graphs. Current frameworks designed for graph sampling and random walk tasks are generally not efficient in terms of memory requirement and throughput. Not to mention that some of them result in biased results. To solve the above problems, we introduce Skywalker+, a high-performance graph sampling and random walk framework on multiple GPUs supporting multiple algorithms. Skywalker+ makes four key contributions: First, it realizes highly paralleled alias method on GPUs. Second, it applies finely adjusted workload-balancing techniques and locality-aware execution modes to present a highly efficient execution engine. Third, it optimizes the GPU memory usage with efficient buffering and data compression schemes. Last, it scales to multi-GPU to further enhance the system throughput. Abundant experiments show that Skywalker+ exhibits significant advantage over the baselines both in performance and utility.

Index Terms—GPU graph sampling; graph random walk; scalability; throughput

1 INTRODUCTION

GRAPH has drawn great attention these years since graph can effectively model entities and their relations in the non-Euclidean space. Most of traditional graph processing algorithms mainly study the low-level information [1], [2], while graph learning algorithms adopt graph embedding to reduce target graphs to low-dimension vectors. The learned embedding can be used for the downstream tasks. It has been actively studied in recommendation system, e-commerce, and many other fields [3].

However, graph representation learning suffers time-consuming feature engineering. With multi-level optimizations, its overhead is still much higher than the cost of classical graph data traversal since graph learning requires capture graph features in multiple and deep levels. The ever-increasing size of graph data can be handled by graph sampling and random walk algorithms. These algorithms focus on local structural information and global information, respectively. Additionally, it enables the use of deeper and more intricate neural networks on massive graphs. Some graph learning algorithms (e.g., node2vec [4], DeepWalk [5] GraphSAGE [6], Para-GCN [7] and GraphSAINT [8]), which learn from sampled embedding, can approximate or even outperform directly learning from the intact graph. While helping large-scale graphs without sacrificing performance, these algorithms take a non-negligible amount of time to extract the embedding. As discussed in previous work [9], the sampling process for GraphSAGE training can take up to 82% of the total execution time. Therefore, accelerating graph sampling is of great significance for the research and adoption of GNN algorithms.

As sampling can be executed following the *vertex-centric* execution model [10], some works modified state-of-the-art graph processing frameworks to support sampling algorithms. For example, DrunkardMob [11] extends GraphChi [12] to support out-of-memory random walk capability while Deep Graph Library (DGL) [13] applies Gunrock [14] to perform graph sampling. While exhibiting some extent of utility and performance, these systems treat graph sampling same as traditional graph algorithms, ignoring its unique properties.

Specialized graph sampling frameworks have been proposed to maximize the overall sampling throughput both on CPU and GPU. KnightKing [15] is a distributed random walk system based on the *alias method* [16]. The system excels in graph sampling, but it requires building the alias table for all the vertices of the graph dataset. Recently, ThunderRW [17] introduces a step-interleaving technique, which switches among different walk queries to reduce the CPU pipeline stalls resulted from irregular memory access. Some works further apply CPU designs to GPU to leverage their massive computing power and high memory bandwidth. C-SAW applies *inverse transform sampling* (ITS) [18] method to select neighbours, while NextDoor adopts the rejection sampling technique from KnightKing. Although these frameworks shows superior performance compared with their CPU-based predecessors, they fail to implement most advanced sampling algorithms, having problems such as high time complexity and highly varied trial number.

Note that the above specialized system designs all have their limitations. While CPU-based frameworks manage to alleviate irregular memory success, the overall performance is limited compared with GPU-based ones due to lack of parallelism. On the other hand, GPU-based systems stuck in the problem of insufficient GPU memory. Once using host

- *Pengyu Wang and Cheng Xu contributed equally to this work.*
- *Pengyu Wang, Cheng Xu, Chao Li (corresponding author), Jing Wang, Taolei Wang, Lu Zhang and Minyi Guo are with the CSE Department, School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University, Shanghai, China.*
E-mail: wpybtw@sjtu.edu.cn, jerryxu@sjtu.edu.cn, lichao@cs.sjtu.edu.cn, jing618@sjtu.edu.cn, sjtuwtl@sjtu.edu.cn, luzhang@sjtu.edu.cn, guo-my@cs.sjtu.edu.cn.

memory, low PCIe bandwidth can be a severe bottleneck. One may choose to neglect high-degree vertices to improve performance [19], but it ends up getting inexact results.

In this paper we aim to achieve high-quality sampling/walking in an efficient and scalable way. We thoroughly investigate existing algorithms, frameworks, and systems for sampling. Prior works [1], [19] believe that the alias method is not suitable for GPU execution. Nevertheless, we believe the alias method is underestimated and has shown great potential for performance optimization. Our goal is to execute low-complexity sampling while amortizing the high-complexity preprocessing cost and preserving the sampling quality.

We introduce Skywalker+, an high-performance solution specialized for graph sampling and random walks on multiple GPUs exploiting both intra- and inter-instance parallelism while optimizing memory access and data locality. It provides usable interfaces and comprehensive support for all kinds of algorithms We have further extended our design to multiple GPUs, using balanced workload partition and locality-aware data management strategies to optimize the sampling throughput. Extensive experiments show that Skywalker+ exhibits significant advantage over the baselines while maintaining robustness on handling large graphs. We have open-sourced Skywalker+, which can be accessed online [1].

This work highlights the following contributions:

- We realize highly paralleled alias table construction on GPUs, making it practical for efficient SIMD execution for the first time.
- We propose an efficient execution engine supporting various sampling algorithms. It is capable of determining the proper execution mode as well as balancing the workload.
- We exploit the data locality in graph sampling by leveraging a locality-aware asynchronous execution strategy. It reorganizes the execution order so that the cached data is more likely to be reused.
- We introduce efficient buffering and data compression schemes to optimize the GPU memory usage, accelerating the execution process while cutting down the memory footprint.
- We extend the graph-friendly versatile sampler and alias table optimization to the multi-GPU domain. We devise efficient graph partition and GPU communication methods, which can effectively utilize multiple GPUs for better scalability.
- We put the above techniques together and present Skywalker+, a novel system executing various graph sampling algorithms with high computing and memory efficiency. Abundant experiments show that Skywalker+ exhibits significant advantage over the baselines both in performance and utility.

2 Background

2.1 Graph Sampling and Random Walk

We start from the terminology of graph sampling and random walk. Graph sampling and random walk are two algorithms widely used to extract small-size embedding form large graphs while preserving structural information. They can be considered as pro-processing processes for downstream graph learning tasks. Sampling and random walks can be regarded as embarrassingly parallel computing tasks, since they seldom update graph information. Specifically, sampling and random walks can be *unbiased (unweighted)* or *biased (weighted)*. The former selects the neighbors uniformly, while the latter computes the transition probability according graph property.

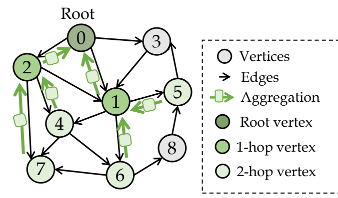


Fig. 1: Neighbor Sampling

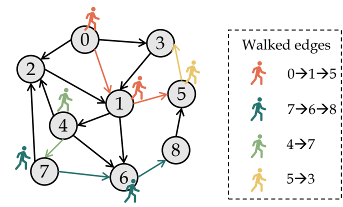


Fig. 2: Random walks

Graph Sampling. In graph sampling, several samplers start from one given root vertex and repeatedly selects several neighbors to generate a small sample graph from the large graph dataset, thus aggregating the neighbor information of the root vertex. The selection process can be either uniform (unbiased sampling) or based on a specific *transition probability distribution* (biased) generally calculated based on the weight of edges. A fixed number of neighbors are chosen for each layer when performing `Neighbor sampling`, as Figure 1 shows. `GraphSage` [6] is an *inductive* algorithm to learn graph embedding using `Neighbor sampling`, sampling k-hop neighbours. Hop indicates the distance of the target vertex to the seed vertex.

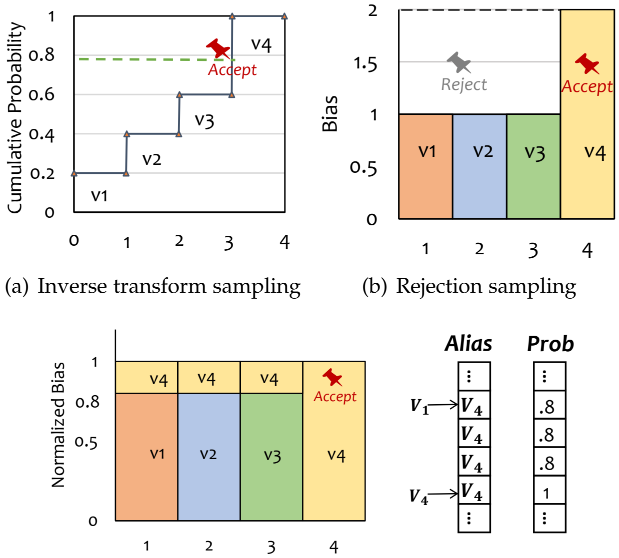
Random Walk. The random walk algorithms work similarly to sampling. Walkers repeatedly select one neighbor and moves to the selected vertices from their residing vertices until satisfying certain conditions, as Figure 2 shows. `Personalized PageRank` (`PPR`) [20] is a optimized version of PageRank [21]. `Deepwalk` [5] is an unbiased walk algorithm, while a later work [22] extends it to a biased version. Some dynamic algorithms further use runtime information to make decision. `Node2vec` [4] introduces the 2nd order random walk and defines two hyperparameters for the walking states.

Summary. Random walk algorithms can be seen as special cases of graph sampling that only select (at most) one vertex per step. They both select vertices based on the connectivity of graphs, leaving the opportunity to optimize them at the same time. For conciseness, we use sampling to refer to both the two algorithms. We term the vertices whose neighbors are to be selected as *transit vertices*.

2.2 Sampling Operations

For unbiased sampling, we can directly generate an random integral number to select the neighbour to be sampled in the next step of the current vertex. On the other hand, there are multiple neighbouring selecting methods when it comes to biased sampling, which are introduced next. We use an example where v_1 to v_4 are the adjacent vertexes of v_0 and have edge weight of 1, 1, 1 and 2 respectively to demonstrate the workflow of these methods.

1. https://github.com/wpybtw/Skywalker



(a) Inverse transform sampling (b) Rejection sampling

(c) Alias method

Fig. 3: Popular neighbour selection methods

Inverse Transformation Sampling (ITS) firsts computes the CDF on its preprocess stage. Suppose a vertex u has neighbours $v_1, v_2, \ldots, v_n$ with bias of edge e_{uv_i} being p_i. Then we can get the CDF as $p'_i = \sum_{j=1}^{i} p_j$. On the execution stage, ITS generates a uniform real number x in $[0, p'_n)$, then uses binary search to determine the smallest index i such that $x < p'_i$, and selects v_i. Figure 3(a) shows the CDF. If we generate a random number as 0.8, then v_4 should be the sampled vertex. The preprocess takes $O(n)$ time and space while the execution takes $O(\log n)$ time.

Rejection Sampling calculates $p^* = \max p_i$ on its preprocess stage. The execution stage takes two steps: (1) generates a uniform integer number x in $[1, n]$ and a uniform real number y in $[0, p*)$; and (2) if $y < p_x$, then select v_x; otherwise repeat step (1). The time complexity of preprocess and execution are $O(n)$ and $O(\frac{n \times p^*}{\sum p_i})$, respectively. Since it only needs to store the max bias p^*, the space complexity is $O(1)$. As shown in Figure 3(b), the rectangle areas covering (0,0) to (3,1) and (4,0) to (4,2) is the *envelope* and the sampling is accepted as long as it falls into these areas.

Alias method builds two tables on its preprocess stage: a probability table P and an alias table A. All biases are distributed into n groups of the same total bias. The vertices with biases above average fill up the gap of those with biases below average. As Figure 3(c) shows, v_4 uses its bias to help v_1, v_2, v_3 to form a normalized distribution. The normalized bias (i.e., $\{0.8, 0.8, 0.8, 1\}$) is then recorded in the probability table while the alias table records $\{v_4, v_4, v_4, v_4\}$. We then need to produce two random numbers in order to select one sample. While the first one chooses an index, the second one chooses the vertex of that index or its alias vertex stored in the alias table. The complexity of building an alias table for a vertex is $O(n)$; however, the cost of drawing sample once based on the table is $O(1)$. The complete probability table and alias table both have $|E|$ elements, where E is the number of edges in the target graph.

2.3 Limitations of the Prior Work

Multiple CPU-based frameworks have been proposed to accelerate graph sampling tasks. Knightking [15] leverages rejection sampling along with the alias method for lower sampling cost. ThunderRW [17] makes further optimization by alleviating irregular memory access. But the overall performances of these frameworks are limited compared to GPU-based frameworks, as GPUs far overwhelm CPUs in massive parallel computing.

While showing higher throughput, existing GPU-based frameworks fail to provide comprehensive support for graph sampling. For example, C-SAW (the latest work on GPU-based sampling) can easily produce *biased* results. It is because its open-sourced implementation [19] simply choose to skip the vertices with degrees over 8000 when computing *Cumulative Transition Probability Space* for the vertices. In addition, C-SAW applies simple binary search to draw samples, which leads to high time complexity compared with alias method ($O(log\ n)\ vs.\ O(1)$). Furthermore, the C-SAW implementation fails to optimize the storage of intermediate data and sampling results, storing them in space-consuming data structure. Accoding to our profiling results, it can only issue about 4000 in-memory random walkers at a time on a RTX 2080Ti with 11 GB memory.

Another key work is NextDoor [9]. It is based on rejection sampling, but it suffers from the problem of highly varied trial number. It can lead to severe slow down especially on large graphs. The biases of some hotspot vertices could be so large in this case that result in unexpectedly large average trail number.

2.4 Key Design Considerations

Once the alias table is built, the sampling process takes constant time as long as the graph bias does not change. This is significant as downstream GNN tasks always require multiple iterations of training, generating multiple sampling/walking queries on the same graph.

However, although alias methods shows superiority in execution stage, building the alias table appears to be the bottleneck especially for dynamic sampling algorithms such as `node2vec`. A natural way to speed up this procedure is leveraging parallel processors such as GPGPUs. However, alias table construction is considered to be problematic on GPUs [1], [19].

Wei et al. [23] found that their GPU implementation of alias method [1] performs even worse than the CPU version. Specifically, adopting the alias method on GPU platforms haves four challenges:

1) *The alias method is hard to parallelize on GPU.* Classical alias table construction method is executed largely in serial, making it non-trivial to map the alias table to SIMT-style GPUs.
Unpredictable logical branching would result in *warp divergence* if the serial portion of the method were naively mapped to each GPU thread.
2) *It is nontrivial to implement the application-aware execution engine.* The irregularity of graphs leads to extremely unbalanced workload distribution, creating severe stragglers. Besides, samplers may visit any

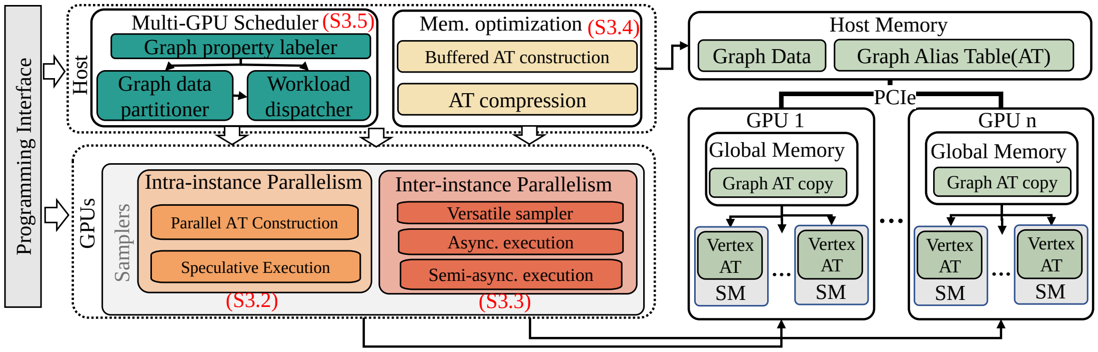


Fig. 4: Skywalker+ architecture.

part of the graph, which leads to frequent thrashing in both cache and on-board memory.

3) *It is difficult to effectively utilize the hybrid memory structure.* While processing many sampling instances concurrently is preferable, constructing an alias table for each vertex requires large memory space (detailed in §3.4.1). Therefore memory can be a scarce resource for graph sampling workloads.

4) *It is non-trivial to schedule workloads on multiple GPUs.* Since graph tends to have skewed distribution, simply dividing workload according to vertex can lead to inefficiency on different GPUs. Besides, it would suffer from massive long-latency accesses to remote data without careful arrangement.

3 SKYWALKER+

In this paper, we aim to address the aforementioned challenges and unleash the potential of the alias-method-based sampling on GPUs.

We propose Skywalker+, a highly efficient framework for random walk and graph sampling algorithms on heterogeneous computing platform. The optimization strategies of our design are multi-pronged, leading to significantly improved efficiency and scalability with multiple GPUs.

3.1 Design Overview

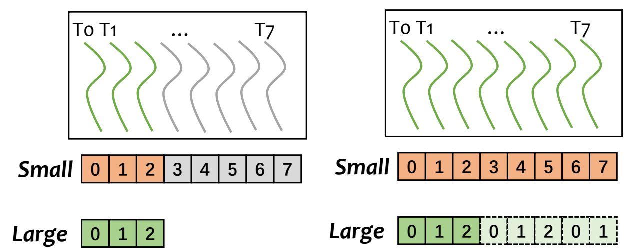
As shown in Figure 4, Skywalker+ is optimized along multiple dimensions. We revisit the sampling/random walk algorithm, devise the parallel execution engine, fine-tune memory usage, and enable multi-GPU scheduling.

1) At the algorithm level, we separate the samplers of Skywalker+ so the construction process of the alias table and sampling process of the graph can be done in parallel.

2) In terms of computing engine, we leverage a new execution model that can fully utilize the parallelism of independent sampling instances. It not only utilizes the high computing power of the SIMD architecture, but also exploits data locality of graph applications.

3) For memory management, we build memory hierarchy aware scheme that can reduce memory requirement for large graphs. Doing so allow us to further accelerate the entire graph sampling process.

4) Finally, at the cluster level, our multi-GPU scheduler distributes workloads evenly based on interleaved indices and leverages a heuristic strategy to instruct the execution.



(a) Direct mapping. (b) Speculative execution.

Fig. 5: Speculative execution of Skywalker+.

In total, Skywalker+ supprots a number of different workloads and execution modes. For example, it can constructs the alias table as an offline procedure. It can also work in a realtime mode, which is necessary for dynamic sampling tasks that cannot build the entire table beforehand without runtime information.

3.2 Tapping into Intra-instance Parallelism

To effectively run the alias method on GPUs, we parallelize the alias-table constructing algorithm while taking load balancing into consideration.

3.2.1 Parallel Table Construction Algorithm

Skywalker+ assigns many threads to compute the alias table of a single vertex in order to make use of the parallelism of the GPU. A *workgroup* is a collection of threads that cooperate to process nearby vertexes within a single thread warp/block. We use atomic operations so that threads in a workgroup can simultaneously put vertices into $Large$ and $Small$. Each thread in a workgroup handles one pair of large-/small- bias vertices individually.

If the proportion of vertices in $Large$ or $Small$ is uneven, the parallelism of the construction would be restricted. For example, we take into account a workgroup with eight threads in Figure 5(a). Only three vertices make up $Large$, but $Small$'s size is equal to or more than eight. Five threads will therefore be idle because only one large-bias vertex can be processed. Note that $Large$ and $Small$ do not necessarily have equal sizes, this circumstance occurs frequently.

Algorithm 1: Parallel alias table construction.

Input: $Large, Small, Prob, Alias$
Result: $Prob, Alias$
while $Large \neq \varnothing$ **do**
  if $ThreadIdx < Large.size$ **then**
    $IsMain = true$;
  else
    $IsMain = false$;
  $v_s = Small.pop()$;
  if $ThreadIdx == 0$ **then**
    $Large.size- = MIN(Large.size, GroupSize)$;
  $v_l = Large[ThreadIdx \mod Large.size]$;
  $oldP = atomicSub(Prob[v_l], (1 - Prob[v_s]))$;
  if $oldP - (1 - Prob[v_s]) < 0$ **then** `// Roll back`
    $AtomicAdd(Prob[v_l], (1 - Prob[v_s]))$;
    $Small = Small \cup v_s$;
  else `// Successful update`
    $Alias[v_s] = v_l$;
  if *IsMain* **then**
    if $Prob[v_l] > 1$ **then**
      $Large = Large \cup v_l$;
    else if $Prob[v_l] < 1$ **then**
      $Small = Small \cup v_l$;
end

3.2.2 Handling Complex Bias Distribution

To handle complex bias distribution, we devise a *speculative execution* mechanism. In the alias method, we can allow numerous small-bias vertices to aggressively consume the bias of large-bias vertices since the bias of vertices in $Large$ will eventually be dispersed into several buckets of vertices in $Small$. However, in the worst case, this could lead to negative bias value. The threads that consume excessive resources in this case should roll back their execution. Thus we further devise a parallel alias table construction algorithm to handle complex bias distribution.

The detailed parallel algorithm is shown in Algorithms 1. We illustrate how the algorithm works together with Figure 5(b). Eight threads in a workgroup work cooperatively. Each thread has a unique smaller-bias vertex to process. As for large-bias vertices, there are only three of them. Specifically, we term the threads with a local index less than the size of $Large$ as *main threads*. Main threads hold the ownership of the respective vertices, and are responsible for enqueuing that vertex to $Large$ or $Small$ at the end of the steps. Those non-main threads can also consume the bias of one large-bias vertex for its smaller-bias vertices even though they do not hold the ownership. Thus, thread t_0, t_1, t_2 are the main threads, holding the ownership of vertex v_{l_0}, v_{l_1} and v_{l_2}, respectively. All threads try to process their low-bias vertices using atomic operations, speculatively. If the resulted probabilities are valid, the speculative execution is succeeded. Otherwise, the corresponding threads withdraw their probability updates. Note that the atomic functions in Algorithms 1 return the original value before modification, following CUDA's semantic [24]. At the end of this step, main threads enqueue the large-bias vertices to $Large$ or $Small$ based on their current probability. In this way, the parallelism is improved. Algorithms 1 shows the parallel algorithm. Speculatively, all threads attempt to process their low-bias vertices using atomic operations. The execution of the speculation is successful if the probabilities that were generated are positive.

3.3 Tapping into Inter-instance Parallelism

This section explains how Skywalker+ makes use of the graph sampling inter-instance parallelism.

3.3.1 Versatile Sampler

We propose *versatile sampler*, a new execution model which enables GPU threads to engage in various levels of collaboration for alias table creation with the minimal overhead.

Our design features a multi-level load balancing technique for assigning GPU resources for vertices with various degrees of skewness. Specifically, it uses two key parameters, the *warp-processing threshold* and the *block-processing threshold*, to decide the various sizes of workgroups for one transit vertex. For example, several threads working in the same warp are used to build the alias table for *low-degree vertices* whose degrees are below the warp-processing threshold. Threads within a block collaborate to process *vertices* with degrees greater than the block-processing threshold.

The jobs in a queue are processed by GPU kernel threads that remain active during execution until there are no job left. Threads contained within one block of execution can switch between different modes if necessary. We devise collective samplers at the sub-warp level, warp level, and block level. We use shared memory to store sampler context. In contrast to prior execution model [25] where threads in one block independently complete the same task on several inputs, our versatile sampler allows threads in one block to switch between three working modes. As the burden of alias table formation varies greatly for each vertex, a distinct number of threads collaborate to process one task.

3.3.2 Semi-asynchronous Execution

The workload for each vertex is significantly skewed due to the irregularity of graphs, and therefore all the existing iterative-based GPU sampling frameworks can face significant performance overhead due to the straggler issue.

We use a *semi-asynchronous* [26] execution model, other than conventional synchronous one. In particular, the samplers in Skywalker+ separately handle each job after continuously requesting it from a per-depth global queue. One sampler advances to process jobs for the next depth without waiting for other samplers when the current depth has no jobs in the queue and just one sampler is available.

The execution flow of Skywalker+ is depicted in Figure 6: ① A subwarp-collective sampler is executed by each thread warp. The sampler builds the alias table and draws samples for the low-degree transit vertex. ② A sampler adds a high-degree transit vertex to a queue that holds high-degree jobs whenever it receives one. Our design momentarily saves mid-degree transit vertices in a per-SM queue. ③ Subwarp-collective samplers can form one warp-collective sampler to process jobs in the per-SM queue when the global job queue for the current iteration is empty. ④ Warp-collective samplers in a thread block gather together and become a single block-collective sampler when the per-SM queue is empty. ⑤ The transit vertices in the high-degree queue are processed by the block-collective sampler. ⑥ For

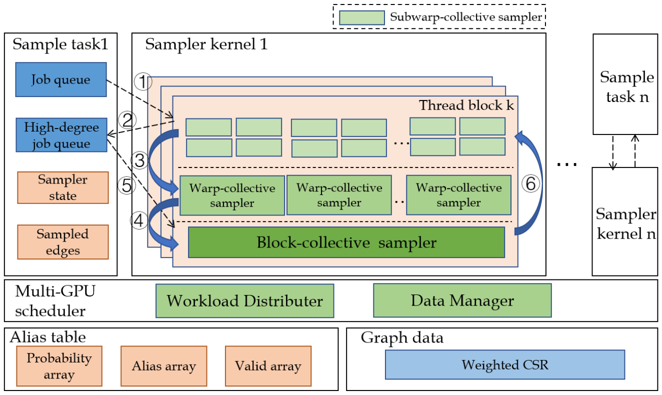


Fig. 6: Skywalker+'s semi-asynchronous execution flow.

the following iteration, the block-collective sampler changes back to subwarp-collective samplers.

3.3.3 Locality-aware Asynchronous Execution

When performing online sampling, the samplers need to access all the neighbors of transit vertices to construct alias tables. In this case, data locality is crucial for performance. Though the above semi-asynchronous execution style is efficient in load-balancing, it overlooks the locality existing in sampling just like other existing GPU-based method implementing the synchronous style. Vertices in large graphs are partitioned and processed successively, eliminating the chance of leveraging data locality. It can lead to frequent data thrashing, since data may be evicted from cache even it is to be used in the next iteration.

For example, suppose we need to sample vertex 0 and 4 in the example graph of Figure 7 in an iterative-based manner. The graph is divided into two parts so that each part can fit into a virtual cache. For the given sample results, the procedure is as follows: vertex 0 and 4 are in the frontier of the first iteration; to construct the alias table for vertex 0, Sampler0 loads partition A and then selects vertices 3 and 4; then, Sampler1 loads partition B and selects vertices 3 and 5; similarly, we construct the alias table and select neighbors for vertices 3, 4, 3 and 5. In this way, we need to repetitively load graph partitions to process each sampling task in iterative execution order, which is highly inefficient.

To address the above problems, Skywalker+ further introduces a locality-aware asynchronous execution strategy to explicitly exploit the locality existing in sampling. The main idea is to rearrange the order in processing the transit vertices based on their position in the graph so that the cached data is more likely to be reused.

Specifically, we partition the graph into subgraphs virtually based on vertex indices. Then, a locality-aware sampler creates one sub-frontier for each subgraph. During execution, tasks in the same sub-frontier are processed successively while the newly-generated sampling tasks enqueue into their corresponding sub-frontiers. The tasks from different sampling depths can be processed together. In other words, once a warp- or block- sampler constructs the alias table and selects several samples, the newly-generated sampling tasks can be processed immediately as long as they belong to the same subgraph. For example, as shown in the locality-aware execution of Figure 7, instead of moving to vertex 4 stored in another partition, the sampler first sample on vertex 2, thus eliminating an unnecessary subgraph switching. In this way, the data loaded in the cache is more likely to be reused and the number of loading data in memory is reduced eventually. On the other hand, if thread blocks do not get valid tasks from the current sub-frontier, they will immediately move on to the next sub-frontier and do not wait for other blocks.

3.3.4 Selecting Vertices

Skywalker+ choose vertices depending on the constructed alias table. Sampling algorithms generally use *sampling without replacement*. In other words, there shouldn't be any repetition in the vertices chosen for a single transit vertex. This leads to repeatedly sampling overhead, which stems from duplicate detection and resampling. Skywalker+ adopts several techniques to reduce this overhead for both offline and realtime workloads.

We use a bitmap for each transit vertex for duplicate detection. Each bit in the bitmap specifically specifies indicates whether a certain vertex has been chosen. To avoid selecting the same vertex again when selecting, threads employ atomic compare-and-swap operations. Besides, Skywalker+ aggressively lets excessive threads atomically increase a counter. With twice or more threads than actually needed vertexes (depending on a pre-set expand factor) launched to select one neighbour using atomicCAS to generate unique results, the warp is more likely to succeed in one trial. For high-degree vertices, collisions would be rather rare. For low-degree vertexes smaller than the expand factor, Skywaker+ directly submits all its neighbors.

With already constructed alias table, we can perform resampling with a constant $O(1)$ overhead. No repetitive alias table construction is needed since the graph bias does not change. Thus, Skywalker+ has significant advantage over other frameworks such as C-SAW which requires $O(log(d))$ operations to resample.

3.4 Memory-side Optimizations

This section explains how Skywalker+ optimizes memory access and lowers memory usage when creating alias tables.

3.4.1 Fast Alias Table Construction

To create the probability table and alias table ($Prob$ and $Alias$) for one vertex, we need to load all its neighbors in queues and process them with frequent enqueuing/dequeuing operations. Skywalker+ leverages shared memory of GPUs to further optimize the buffer. For example, each SM supports 1024 concurrent threads at most. Thus, each warp can be provisioned with around 1.5 KB shared memory. This means a warp-collective sampler can process nearly 100 elements using buffer shared memory, which is larger than a reasonable warp-processing threshold. For a block-collective sampler, shared memory in each SM alone is not sufficient for processing vertices with extremely high degrees. In this scenario, Skywalker+ splices shared memory and global memory for the buffer. In other words, the buffer falls back to global memory when the required buffer size is larger than the size of shared memory.

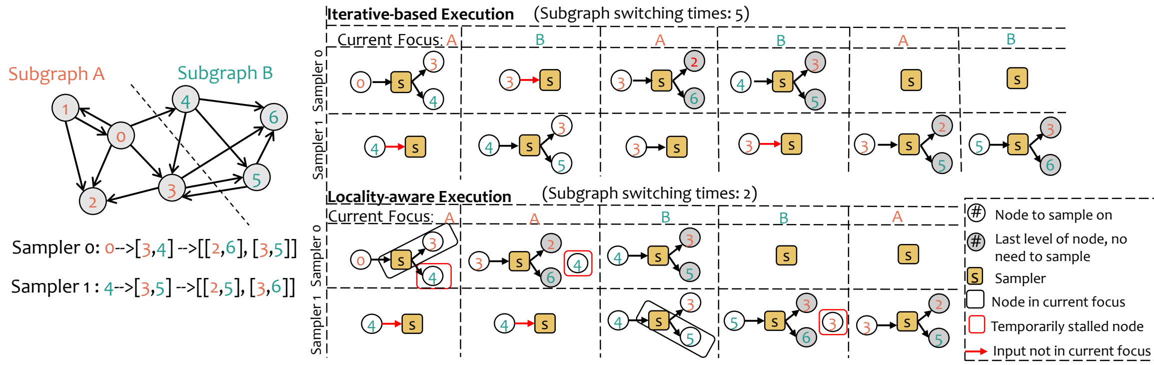


Fig. 7: Locality-aware Asynchronous Execution

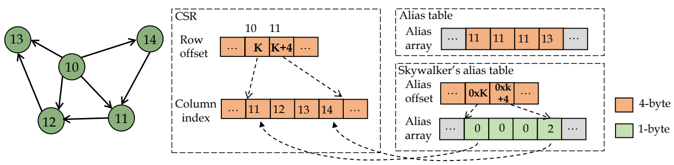


Fig. 8: Skywalker+ compresses the alias tables based on vertex offset.

In this way, Skywalker+ reduces the memory requirement to:

$$Mem_3 = K \times \#SM \times BlockPerSm \times MaxDegree.$$

Skywalker+ thus drastically lowers the amount of memory needed to generate an alias table by utilizing low-latency shared memory on the GPU.

3.4.2 *Table Storage Compression*

As discussed in section 2.2, the entire graph's alias table requires roughly twice as much storage space as the graph structure data. This severely limits the ability to sample huge graphs given the GPU's low memory capacity.

Skywalker+ adopts a compression technique on the alias table to solve the aforementioned problem. The original alias approach specifically maintains the neighbor vertices' indices, which span from 1 to $|V|$. Vertex indices must be identified using a 4-byte format for graphs with millions of vertices. Skywalker+ solves this issue by storing the offset in the alias table rather than the vertex indices. Skywalker+ can utilize 1-byte, 2-byte, or 4-byte format for the alias array, depending on the graph's maximum degree.

3.5 Scaling to Multi-GPU

To the best of our knowledge, prior work does not consider alias method based graph sampling on multiple GPUs. In this work we first investigate the scalability of our design and adapt our system to the multi-GPU environment. To fully utilize the parallelism of multiple GPUs, we adopt the following strategies to partition graphs, distribute workloads and manage data for higher system throughput.

3.5.1 *Locality-centric Graph Partition*

When handling large graphs, Skywalker+ first partitions the integrated graph so that the subgraphs can be accommodated to different GPUs respectively.

Graph partitioning with Multi-GPU. Skywalker+ partitions the workload by vertex indices instead of vertex number as table construction has $O(d)$ complexity for each vertex. Skywalker+ inspects the vertex index offset first and then partitions the workload so that the partitions have similar edges for better load-balance.

Alias table partitioning with Multi-GPU. Alias table composes of an alias array and a probability array, which is stored corresponding to the input order. It requires extra space and time if we want to rearrange alias tables. The integral alias table is stored in the host memory and accessed through PCIe if there lacks enough space in GPU memory.

3.5.2 *Balance-ensured Workload Distribution*

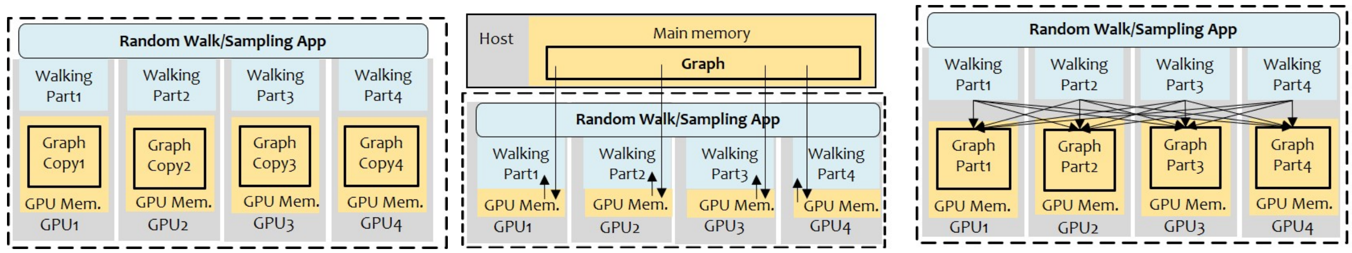
Workload distribution is a key factor in Multiple GPU computing since straggler GPUs can severely lower the overall performance. Skywalker+ distributes the workload evenly for alias table construction and sampling in both realtime and offline sampling scenarios to alleviate this problem.

Alias table construction with Multi-GPU. Each GPU is distributed with certain partition of graph data. Based on these data, the GPU builds the partial alias table. When GPUs finish its partitioned table construction, those partitioned alias tables are assembled together for further usage. Depending on the size of the graph and GPU memory capacity, Skywalker+ either gathers the partial alias table in GPU memory for lower access latency or assemble the partial alias table in host memory to avoid exhausting the GPU memory.

Sampling with Multi-GPU. Skywalker+ distributes the sampling instances evenly to different GPUs. As the computing of instances is independent, there is no need for communication or synchronization. Each GPU processes its assigned sampling instances independently. Although graphs could have skewed edge distribution, the workload for the GPUs are generally evenly distributed. It's because the time complexity for sampling is $O(1)$ for each vertex and Skywalker+ ensures that each partition has a similar amount of vertices.

3.5.3 *Latency-aware Data Management*

Data management is crucial for multiple GPU computing as GPU has a limited memory capacity. Due to the irregularity of graphs, the sampler/walker may access any locations of the graph. Thus, each sampler/walker must be able to



(a) In-GPU Execution (b) DMA Execution (c) Peer-to-Peer Execution

Fig. 9: Different execution modes on multiple GPUs.

access the whole graph data. Skywalker+ carefully places the necessary data to minimize the memory access latency.

Design Space Analysis. There are several options to allow GPU to access data larger than the GPU memory capacity, such as pinned host memory, Unified Memory (UM) and peer GPU memory. These options have different performance characteristics. Pinned host memory allows GPUs to directly access host memory through unified virtual address space, but fails to cache the hot pages. UM allows over-subscription of GPU memory, but can suffer from fault handing overhead [27]. Besides, directly applying UM among multi-GPU can lead to frequent data migration between host memory and devices, which further result in interface congestion. Peer-to-peer memory access enables GPUs to access remote graph data resident on other GPUs' memory, but it is only supported by limited GPU types. Skywalker+ carefully weighs these options and adapts them to suitable scenarios.

Heuristic Strategy. Skywalker+ devises a heuristic strategy taking both graph data characteristics and hardware capability into consideration. The methodology is to leverage memory with lower latency as much as possible. Figure 9 presents our various execution modes. ① Skywalker+ first inspects the graph to be sampled, and calculates the size of graph data, alias table and other runtime data (such as task queues and result). ② For small-sized graphs whose structure data and alias table is small enough to be integrally stored in a single GPU memory, Skywalker+ duplicates them to each GPU and let kernels access local-resident data as shown in Figure 9(a). ③ For large-size graphs whose graph data itself approaches or exceeds the total GPU memory, Skywalker+ allocates the graph directly in host memory. Then, the kernel accesses graph data through DMA and stores the runtime data locally as shown in Figure 9(b). ④ For medium-sized graphs, Skywalker+ partitions the graph into $\#GPU$ pieces, and dispatches different graph partitions to each GPU as shown in Figure 9(c). Users can also manually assign the strategy for Skywalker+ instead of the pre-set strategy.

4 Experimental Methodologies

In this section, we show the experimental methodologies to verify the superiority of Skywalker+.

Platform. We use a Linux server that has two 2.4 GHz Intel Xeon 6148 CPUs (20×2 physical cores in total) as the evaluation platform. Each CPU has 27.5 MB of L3 cache and 256 GB of main memory.

TABLE 1: Evaluated walk&sampling strategies.

Name	Sampling Method	Supported workload
Graphwalker [28]	Unbiased sampling.	Unbiased `PPR`.
KnightKing [15]	offline Alias method for static sampling; rejection sampling for dynamic sampling.	Biased/unbiased random walks.
C-SAW [19]	Inverse transform (ITS) sampling.	Biased sampling and `DeepWalk`.
NextDoor [9]	Rejection-sampling	Unbiased `node2vec / Sampling`. Biased `DeepWalk / PPR`.
ThunderRW [17]	ITS, rejection, and Alias method	Biased/unbiased.
Skywalker+	Alias method.	Biased/unbiased.

Our platform is well equipped with four RTX 2080Ti graphics cards. Each GPU has 4352 CUDA cores in total and 11GB of GDDR6 memory. We install Ubuntu with Linux kernel 4.15.0. We compile all programs using NVCC compiler version 11.0.167 (g++ version 7.5.0). For evaluation, we use nsight system v2021.5.2 and nsight compute v2022.1.1 to collect runtime information.

For CPU-based baselines, we let them use all 40 physical CPU cores to compute. The distributed systems Knightking [15] is also executed on a single machine. For comparison with the GPU-based framework, we use only one GPU.

Baseline Frameworks. To verify the superiority of Skywalker+ over existing approaches, we compare it to the most representative baselines as summarized in Table 1.

1) **GraphWalker** [28] is a CPU-based random walk system targeting single node. To maximize data access efficiency, it gives priority to the loaded subgraphs.
2) **KnightKing** [15] targets distributed system. For algorithms with static deviations, it uses the aliasing method, while for algorithms with dynamic deviations, it uses the rejection sampling method.
3) **Nextdoor** [9] is another representative GPU-based graph sampling approach. Its rejection sampling technique is the same as KnightKing.
4) **C-SAW** [19] is one of the most representative graph sampling and random walk system for GPU. It represents the ones that utilize the Inverse Transform Sampling method.
5) **ThunderRW** [29] represents the latest CPU-based graph walk engine. It supports multiple sampling methods on both biased and unbiased workloads.

TABLE 2: Evaluated graph datasets [30], [31]. The graphs are stored in weighted edgelist format in disk.

Graph Data	Abbr.	$\|V\|$	$\|E\|$	Max Degree	Size(GB)
web-Google	GG	0.9 M	5.1 M	456	0.07
Livejournal	LJ	5 M	69 M	20 K	1.4
Orkut	OK	3 M	117 M	33 K	2.0
Arabic-2005	AB	22 M	640 M	10 K	12
UK-2005	UK	39 M	936 M	5 K	18
Friendster	FS	65 M	1.8 B	3 K	35
SK-2005	SK	50 M	1.9 B	12 K	38
rmat27	RT	130M	3.2B	1700k	67

To implement the above baselines, we download their source code from GitHub and compile and run them on our evaluation platform. As shown in Table 1, some baselines may only support a portion of the workloads. For example, GraphWalker only supports unbiased walk `PPR` while the open-sourced C-SAW do not support `PPR`, `node2vec` and unbiased sampling. Additionally, because C-SAW only supports pre-allocating a fixed size buffer for each thread block, it ignores all vertices with degrees greater than 8000.

Workloads. We have selected three types of workloads that can represent the most common sampling and random walk algorithms in a wide range of application domains. This helps to verify the performance and effectiveness of Skywalker+ in various settings. To be specific, we use thses algorithms including `Deepwalk`, `PPR`, `node2vec` and `NeighborSampling`. As we mentioned above, some baselines may not support a portion of these algorithms due to their limited processing capabilities. We extend and improve them as much as possible to support more algorithms. Taking `NeighborSampling` as an example, we adopt the configuration from GraphSAGE [6] with the sampling depth as 2 and expansion factor (the number of neighbors to be sampled for one vertex) as $S_1 = 25$ and $S_2 = 10$. For `PPR`, we use 15% as determination probability. For `node2vec`, we set the hyper-parameters $p = 2.0$ and $q = 0.5$. We set 100 as the maximum length as adopted in previous works. For all algorithms, we perform sampling with batch size 40000, which is enough for most downstream applications. For our method Skywalker+, we evaluate the sampling algorithms in all possible execution modes and bith unbiased/biased.

Graph Dataset. As shown in Table 2, we use a variety of state-of-the-art datasets in our experiments. We totally consider 7 datasets used in two representative graph applications, i.e., social networks and web graph snapshots. Among them, LiveJournal (`LJ`), Orkut (`OK`) and Friendster (`FS`) are commonly used in social networks, and Web-Google (`GG`), UK-2005 (`UK`), Arabic-2005 (`AB`) and SK-2005 (`SK`) are the SOTA datasets used in web graph snapshot applications. We also mannualy create a 67GB rmat graph to evaluate our scalability for large graphs.

Implementation Details. `Node2vec` uses dynamic bias and is executed in online mode for Skywalker+ and Knightking. Other algorithms are executed in offline mode unless otherwise stated. To be mentioned, C-SAW simply skips all the vertices with degrees higher than 8000, resulting in biased result. Skywalker+ and other baselines follow the standard sampling method, which guarantees the quality of the results. We use a factor of the sampled edges to adjust the running time. We also change the batch size directly in ThunderRW's source code since it samples on all vertices by default, which is less flexible.

Metrics. We collect and demonstrate the runtime information for most of the tests. We do not include the time to initialize and load data from the disk. All datasets that are available can be loaded into the main memory entirely except for Graphwalker, which uses only host memory. We do not include the processing time for Graphwalker to repeatedly load graph segments. Although different input formats are fetched from disk, all frameworks eventually store graph data in memory in the same CSR format. Skywalker+ and other frameworks that utilize the alias method require additional space same to the size of graph data as the alias table for offline workloads. As for the GPU-based baselines, we collect their kernel execution time on GPU. Unless otherwise noted, we include the time to create the entire alias table as part of the preprocessing overhead for KnightKing and Skywalker+.

5 Results and Evaluations

In this section, we demonstrate the results and evaluate the performance of Skywalker+ from four aspects: 1) How fast can unbiased workloads run with Skywalker+? 2) How does Skywalker+ perform when dealing with workloads with static or dynamic biases? 3) How do the new optimizations affect performance? 4) How scalable is Skywalker+ when using multiple GPUs?

5.1 Performance on Unbiased Workload.

We begin with validating the performance of different methods i.e., Graphwalker, KnightKing, NextDoor, ThunderRW and Skywalker+ on the unbiased workload. The results are shown in Table 3. According to our observation, Skywalker+ outperforms all baselines in all test cases and can be adapted to a wide range of scenarios. To be fair, we only compare Skywalker+ with the baselines that support the unbiased model. In our comparison experiments, we do not use uniform bias as the unbiased workload. The unbiased model simply picks a random neighboring vertex, while the uniform bias must go through a complex preprocessing process, such as alias table construction.

Skywalker+ speeds up the execution time on `Deepwalk` by three orders of magnitude compared to Graphwalker; the result is two orders of magnitude on `PPR`. The key reason behind the performance improvement is that Graphwalker uses random walks to statically partition the graph, which involves significant additional coordination overhead. Instead, Skywalker+ is fully optimized for load balancing, data locality, memory access efficiency, and other factors.

Skywalker+ also respectively shows 641×, 142× and 4157× average speedup of improvement than knightking on `Deepwalk`, `PPR` and `node2vec`. Particularly, Skywalker+ achieves higher speedup on `node2vec` than the other algorithms. This is because the `node2vec` algorithm needs time to verify the connectivity of recently sampled vertices with previously sampled vertices. In this situation, the straggler thread would prevent all other threads from running. This problem is not present in Skywalker+ since different SMs are scheduled to work independently.

TABLE 3: Result of unbiased workloads. "O.O.M" indicates out-of-memory error and "_" indicates internal error

Algorithms	Frameworks	Task Runtime (ms)							Avg. Speedup of Skywalker+
		GG	LJ	OK	AB	UK	SK	FS	
`Deepwalk`	Graphwalker	_	172	_	338	721	1719	1739	5894
	ThunderRW	13	16	16	27	28	27	30	101
	Knightking	17	12	13	14	16	3	17	60
	Skywalker+	**0.44**	**1.25**	**0.22**	**0.15**	**0.48**	**0.21**	**0.1**	1
`PPR`	Graphwalker	_	29	_	40	30	73	109	641
	Knightking	3	16	20	16	23	1	16	142
	ThunderRW	7	4	4	5	5	18	17	94
	Skywalker+	**0.11**	**0.18**	**0.13**	**0.06**	**0.08**	**0.08**	**0.1**	1
`node2vec`	Knightking	1189	1382	202	1033	2323	946	122	4157
	ThunderRW	16	14	14	20	23	25	27	143
	Nextdoor	26	38	6.8	17.8	30.4	O.O.M	O.O.M	49.8
	Skywalker+	**1.11**	**1.98**	**0.45**	**0.11**	**1.04**	**0.07**	**0.07**	1
`Sampling`	Nextdoor	1.7	1.97	2	1.7	1.7	O.O.M	O.O.M	5.2
	Skywalker+	**0.3**	**0.73**	**1.24**	**0.22**	**0.21**	**0.3**	**0.3**	1

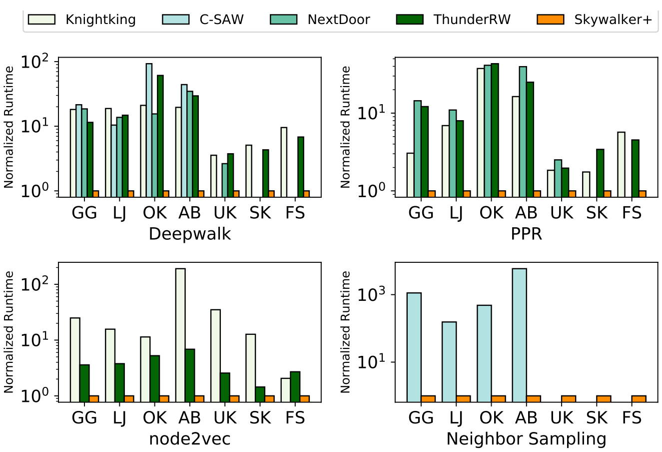


Fig. 10: Results of biased graph sampling (normalized to Skywalker+'s runtime).

TABLE 4: Speedup breakdown of different techniques

Technique	Speedup						
	GG	LJ	OK	AB	UK	SK	FS
w/ Speculative Execution	1.26	1.58	1.54	3.11	2.06	1.63	1.28
w/o Speculative Execution	1	1	1	1	1	1	1
w/ Semi-asynchronous Execution	2.35	1.55	1.99	5.20	1.49	2.82	1.10
w/o Semi-asynchronous Execution	1	1	1	1	1	1	1
w/ Data Locality-aware	2.18	2.64	2.89	2.71	2.69	2.77	2.29
w/o Data Locality-aware	1	1	1	1	1	1	1
Technique	**Space requirement for alias array**						
	GG	LJ	OK	AB	UK	SK	FS
w/ Compressed Alias Table	0.25	0.27	0.31	0.30	0.27	0.28	0.34
w/o Compressed Alias Table	1	1	1	1	1	1	1

Due to its high memory needs, NextDoor cannot process graphs larger than `UK`. Therefore, we only compare Skywalker+ to NextDoor on `UK` and `NeighborSampling`. The result shows that Skywalker+ speeds up `UK` and `NeighborSampling` respectively by 49.8× and 5.2× on average compared with NextDoor.

To compare with ThunderRW, we set the thread number as 20. It shows that Skywalker+ outperforms ThunderRW by 101×, 94× and 143× on `Deepwalk`, `PPR` and `node2vec` respectively. It is notable that the throughput improvement declines from 3× to 10× when ThunderRW samples a batch of vertices rather than all vertices. This is because ThunderRW's interleaving technique works well when the batch size is large. Its relatively constant start-up and shutdown time can surpass actual execution time when handling a small number of queries.

5.2 Performance on Biased Workloads.

In Figure 10, we demonstrate the result of the biased workloads. Compared with KnightKing, Skywalker+ achieves 3.6∼21×, 1.7∼38× and 2∼190× of performance improvement on `DeepWalk`, `PPR` and `node2vec`, respectively.

Skywalker+ achieves 10∼93× speedup on `DeepWalk` and 483∼5878× speedup on `NeighborSampling` compared with C-SAW. The reason behind the significant improvement is that Skywalker+ can utilize the CPU's main memory through the UM mechanism along with space-efficient designs. Therefore, well-designed UM has great potential in increasing the performance of graph sampling and random walk. Skywalker+ beats C-SAW mianly because Skywalker+ can utilize the precomputed alias tables.

Skywalker+ has a lower overhead for sampling multiple items without replacement. Firstly, *sampling by aliasing is more efficient* because it takes constant time. Secondly, *Skywalker+ has lower selection collision cost*. On Skywalker+, collision detection and resampling are substantially less expensive than they are on C-SAW. Thus, Skywalker+ shows higher speedup on `NeighborSampling` than `DeepWalk`. Compared to ThunderRW, Skywalker+ enables 3∼61× speedup on DeepWalk, 2∼43× speedup on `ppr` and 1.4∼7× speedup on `node2vec` respectively.

When it comes to Nextdoor, the latest sampling framework on GPU, Skywalker+ achieves 2.6∼35× speedup on `DeepWalk` and 2.5∼40× speedup on `PPR`. Besides, Skywalker+ show more comprehensive support for different sampling algorithms and large graphs.

5.3 Impact of Different Schemes

In Table 4, we validate the effectiveness of the core optimizations of Skywalker+. We conduct this experiment by analyzing the normalized results on `Deepwalk` with or without the proposed optimizations. We have the following observations. ***1) Speculative Execution.*** Skywalker+ uses speculative execution to speed up `Deepwalk` and `Neighbor sampling` by a factor of up to 3.1 and 1.6 times, respectively. ***2) Semi-asynchronous Execution.*** Using semi-asynchronous execution, Skywalker+ has up to 3.6× and 5.2× speedup on

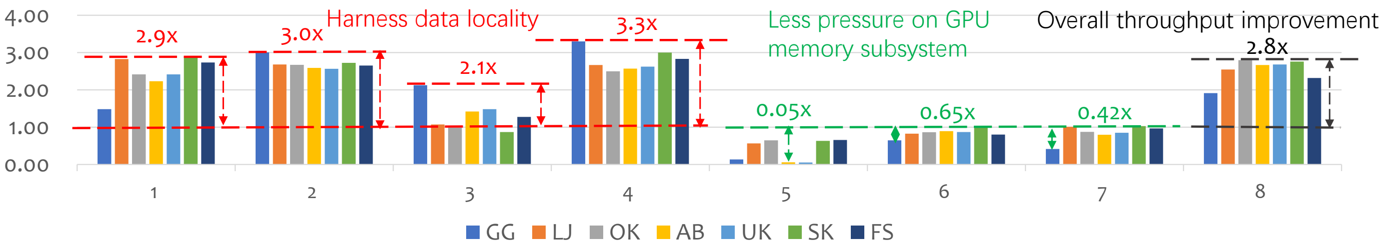
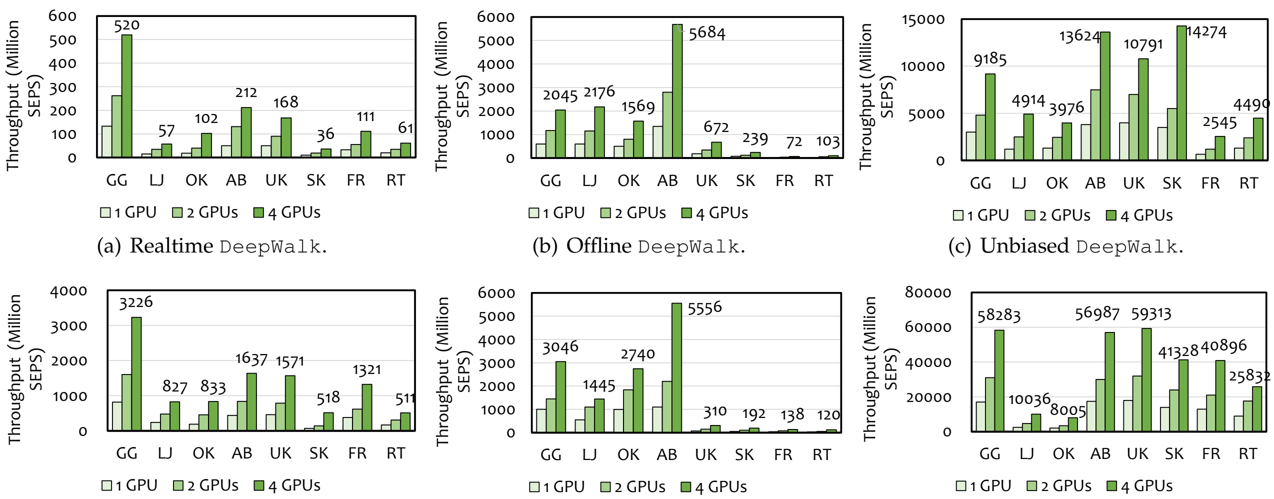


Fig. 11: The normalized profiling result of Skywalker+ over Skywalker on realtime sampling: 1) dram_throughput, 2) gld_throughput, 3) l1_hitrate, 4) IPC indicates data locality extent; 5) dram_utilization, 6) stall_mem_dep, 7) dram_transactions indicates GPU memory pressure; 8) throughput shows the overall system throughput.



(a) Realtime DeepWalk. (b) Offline DeepWalk. (c) Unbiased DeepWalk.

(d) Realtime sampling. (e) Offline sampling. (f) Unbiased sampling.

Fig. 12: Throughput on multiple GPUs. We only show the result for Skywalker+ as C-SAW's open-sourced implementation does not support multi-GPU execution. (a), (b) and (c) are realtime, offline or unbiased version for `Deepwalk`, respectively. Similarly, (d), (e) and (f) are three versions of `Neighbor Sampling`

`node2vec` and `Deepwalk`, respectively. ***3) Data Locality-aware Execution.*** Through data locality-aware execution, Skywalker+ achieves a speedup of 2.2∼2.9× on `Neighbour sampling` in online mode, respectively. ***4) Compressed Alias Table.*** On the evaluated graphs, the compressed alias array allows saving about 66% more spaces than the original ones. In this regard, Skywalker+ can handle large graphs with almost no overhead.

5.4 Data Locality-aware Execution Breakdown

In Figure 11, we evaluate the hardware characteristic of Skywalker+ and Skywalker. We validate the effectiveness of the locality-aware asynchronous execution technique. Compared with Skywalker, the normalized DRAM throughput of Skywalker+ increases up to 2.9×; global throughput improves to 2.6 to 3.0; L1 cache hit rate improves up to 2.1 except for SK; IPC improves to 2.5 to 3.3; On the other hand, normalized DRAM utilization ratio ranges from 0.05 to 0.66; the number of stalls due to memory dependency reduces down to 0.65; DRAM transactions reduces to 0.42 to 0.97 except for LJ and SK; As a result, the normalized sampling throughput is improved from 1.9 to 2.8. All the above results illustrate that the locality-aware asynchronous execution can effectively harness the locality while reducing the pressure on the GPU memory subsystem, thus improving the overall sampling throughput.

5.5 Multi-GPU Scalability

Figure 12 shows the overall sampling throughput achieved by Skywalker+ using 1 to 4 GPUs. Skywalker+ delivers up to 520 and 3226 million SEPS of throughput on `Deepwalk` and `Neighbor Sampling` for real-time workloads, respectively. The large difference in throughput on `Deepwalk` and `neighbor sampling` is due to the fact that `Deepwalk` needs to compute the alias table once for each newly sampled vertex while `neighbor sampling` samples 20 vertices using each computed alias table. For offline workloads, Skywalker+ respectively achieves up to 5.7 and 5.6 GSEPS (billions sampled edge per second) throughput on `Deepwalk` and `neighbor sampling`. As for unbiased workloads, Skywalker+ achieves up to 14.2 and 59.3 GSEPS throughput on `Deepwalk` and `Neighbor Sampling`.

6 Related Works

To the best of our knowledge, this paper provides the first extensive analysis of graph sampling and random walk on multiple GPUs. We adopt a multi-pronged approach and implement a highly efficient and scalable GPU graph sampling and random walk framework. The relevant prior work are summarized as bellow.

CPU-based Random Walk Systems. Similar to graph computing frameworks, DrunkardMob [11], leverages the popular vertex-centric computational model. KnightKing [15] is a distributed random walk system implementing on multi-core CPU. It applies alias method for static sampling algorithms, while implementing rejection sampling for dynamic sampling algorithms which are less suitable for alias method. We observe that the performance of the above designs are largely affected by the bias distribution. Similarly, GraphWalker [28] leverages GraphChi's out-of-core processing capability. While only supporting unbiased random walk, it features an asynchronous walk updating technique to optimize I/O. ThunderRW [17] is a framework that supports diverse graph sampling workloads. It reduces the CPU pipeline stall resulting from irregular memory access in graph processing with a step interleaving technique. This technique switches between different walk queries to hide the memory latency.

GPU-based Sampling Systems. There are several GPU-based graph sampling and random work frameworks. For example, C-SAW [19] introduces a parallel scan algorithm [32] to implement ITS to select neighbours. Unfortunately, its open-source implementation does not demonstrate how it is optimized for out-of-memory and multi-GPU sampling. Different from C-SAW, NextDoor [9] applies KnightKing's rejection sampling technique to GPU, achieving higher throughput. It introduces a parallel paradigm called *transit-parallelism*. It assigns transit vertices to thread blocks and assigns samples to threads. It also optimizes for memory coalescing.

Memory-constrained GPU Graph Computing. GPU has very limited memory for handling large graph dataset. Therefore, prior studies propose to divide large graphs into subgraphs and process them in a streaming manner [33]. Garaph [34] takes advantage of both CPU and GPU to process large graphs. Recently, a few works [2], [35] start to adopt NVIDIA's unified memory (UM) technique to oversubscribe GPU memory capacity, but often suffer from the problem of frequent page faults. HALO [36] introduces a graph reordering algorithm to speed up graph traversal with UM. Chen et.al [37] further applies a unified-memory-based hybrid processing for partition-oriented subgraph matching on GPU. Hum [38] tries to optimize the execution on unified memory. Subway [39] smartly chooses and transfers the subgraphs to process on the GPU.

Graph partitioning. Since it's common practice to first divide large graphs into subgraphs, extensive researches have been conducted on efficient and effective graph partitioning. Multiple works [40], [41] have leveraged both heuristic and machine learning methods to provide communication-efficient and workload-balanced partitions so that multiple workers such as GPUs can handle these partitions with less memory thrashing and performance straggler. Partition-centric processing model has also been applied [42] to take advantage of data-locality of the partitions and accelerate the whole processing. Hep [43] proposes a new partitioning algorithm which flexibly adapts its memory overhead by separating the edge set of the graph into two sub-sets and achieves improvements in both in-memory partitioning and streaming partitioning.

Multi-GPU. An application can be accelerated by multiple GPUs to achieve higher speedup. Groute [44] introduces asynchronous multi-GPU programming in a thin runtime environment. cuTS [45] develops a GPU-friendly trie-based data structure and a distributed sub-graph isomorphism algorithm to leverage the computing power of multiple GPUs. Case [46] constructs GPU tasks from CUDA programs in the compilation to guide task assignments.

Graph Compression. Besides the general *lossless* and *lossy* data compression techniques [47], [48], graph compression techniques [49], [50] compress the graph structure data. POCLib [51] proposes a near orthogonal processing method for compression. It mainly targets data analysis tasks such as *search* and *count* which require to scan or traversal the whole data region. However, alias tables are only looked up randomly for a few specific elements in a fine-grained granularity, which is significantly different.

7 Conclusion

We present Skywalker+, an novel and powerful system that supports various key random walk and graph sampling algorithms. We introduce a parallel algorithm for alias table construction on GPU and design an efficient locality-aware execution engine. We also apply specialized buffer and storage optimizations and further extend the system to multiple GPUs. Our design exhibits notable performance advantage over the state-of-the-art baseline systems on a variety of scenarios. Importantly, it shows strong robustness and capability on handling large graphs and provides comprehensive support for all kinds of algorithms and execution modes. It also presents satisfactory scalability on multiple GPUs.

References

[1] J. Shi, R. Yang, T. Jin, X. Xiao, and Y. Yang, "Realtime top-k personalized pagerank over large graphs on gpus," *Proc. VLDB Endow.*, vol. 13, pp. 15–28, 2019.

[2] P. Wang, L. Zhang, C. Li, and M. Guo, "Excavating the potential of GPU for accelerating graph traversal," in *2019 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2019, Rio de Janeiro, Brazil, May 20-24, 2019*, 2019, pp. 221–230. [Online]. Available: https://doi.org/10.1109/IPDPS.2019.00032

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

[4] 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, pp. 855–864.

[5] 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, pp. 701–710.

[6] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Advances in neural information processing systems*, 2017, pp. 1024–1034.

[7] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, and V. Prasanna, "Accurate, efficient and scalable graph embedding," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2019, pp. 462–471.

[8] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, and V. K. Prasanna, "Graphsaint: Graph sampling based inductive learning method," in *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. [Online]. Available: https://openreview.net/forum?id=BJe8pkHFwS

[9] A. Jangda, S. Polisetty, A. Guha, and M. Serafini, "Nextdoor: Gpu-based graph sampling for graph machine learning," *ArXiv*, vol. abs/2009.06693, 2020.

[10] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, 2010, pp. 135–146. [Online]. Available: http://doi.acm.org/10.1145/1807167.1807184

[11] A. Kyrola, "Drunkardmob: billions of random walks on just a pc," *Proceedings of the 7th ACM conference on Recommender systems*, 2013.

[12] A. Kyrola, G. E. Blelloch, and C. Guestrin, "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*, 2012, pp. 31–46. [Online]. Available: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/kyrola

[13] M. Wang, L. Yu, D. Zheng, Q. Gan, Y. Gai, Z. Ye, M. Li, J. Zhou, Q. Huang, C. Ma *et al.*, "Deep graph library: Towards efficient and scalable deep learning on graphs," *arXiv preprint arXiv:1909.01315*, 2019.

[14] Y. Wang, A. 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 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, San Francisco, CA, USA, February 7-11, 2015*, 2015, pp. 265–266. [Online]. Available: http://doi.acm.org/10.1145/2688500.2688538

[15] 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, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, T. Brecht and C. Williamson, Eds. ACM, 2019, pp. 524–537. [Online]. Available: https://doi.org/10.1145/3341301.3359634

[16] A. J. Walker, "An efficient method for generating discrete random variables with general distributions," *ACM Transactions on Mathematical Software (TOMS)*, vol. 3, no. 3, pp. 253–256, 1977.

[17] S. Sun, Y. Chen, S. Lu, B. He, and Y. Li, "Thunderrw: An in-memory graph random walk engine," *Proc. VLDB Endow.*, vol. 14, no. 11, pp. 1992–2005, 2021. [Online]. Available: http://www.vldb.org/pvldb/vol14/p1992-sun.pdf

[18] S. Olver and A. Townsend, "Fast inverse transform sampling in one and two dimensions," *arXiv: Numerical Analysis*, 2013.

[19] S. Pandey, L. Li, A. Hoisie, X. S. Li, and H. Liu, "C-saw: A framework for graph sampling and random walk on gpus," 2020.

[20] D. Fogaras and B. Rácz, "Towards scaling fully personalized pagerank," in *WAW*, 2004.

[21] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking : Bringing order to the web," in *WWW 1999*, 1999.

[22] M. Cochez, P. Ristoski, S. P. Ponzetto, and H. Paulheim, "Biased graph walks for rdf graph embeddings," *Proceedings of the 7th International Conference on Web Intelligence, Mining and Semantics*, 2017.

[23] Z. Wei, X. He, X. Xiao, S. Wang, S. Shang, and J.-R. Wen, "Topppr: Top-k personalized pagerank queries with precision guarantees on large graphs," *Proceedings of the 2018 International Conference on Management of Data*, 2018.

[24] Nvidia, "Programming Guide :: CUDA Toolkit Documentation," 2022. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[25] K. Gupta, J. A. Stuart, and J. Owens, "A study of persistent threads style gpu programming for gpgpu workloads," *2012 Innovative Parallel Computing (InPar)*, pp. 1–14, 2012.

[26] H. Kung, "Synchronized and asynchronous parallel algorithms for multiprocessors," 1976.

[27] R. Landaverde, T. Zhang, A. K. Coskun, and M. C. Herbordt, "An investigation of unified memory access performance in CUDA," in *IEEE High Performance Extreme Computing Conference, HPEC 2014, Waltham, MA, USA, September 9-11, 2014*, 2014, pp. 1–6. [Online]. Available: https://doi.org/10.1109/HPEC.2014.7040988

[28] R. Wang, Y. Li, H. Xie, Y. Xu, and J. Lui, "Graphwalker: An i/o-efficient and resource-friendly graph analytic system for fast and scalable random walks," in *USENIX Annual Technical Conference*, 2020.

[29] S. Sun, Y. Chen, S. Lu, B. He, and Y. Li, "Thunderrw: An in-memory graph random walk engine," *Proc. VLDB Endow.*, vol. 14, no. 11, pp. 1992–2005, 2021. [Online]. Available: http://www.vldb.org/pvldb/vol14/p1992-sun.pdf

[30] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," in *12th IEEE International Conference on Data Mining, ICDM 2012, Brussels, Belgium, December 10-13, 2012*, 2012, pp. 745–754. [Online]. Available: https://doi.org/10.1109/ICDM.2012.138

[31] P. Boldi and S. Vigna, "The WebGraph framework I: Compression techniques," in *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*. Manhattan, USA: ACM Press, 2004, pp. 595–601.

[32] A. Grimshaw and D. Merrill, "Parallel scan for stream architectures," 2012.

[33] M. 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, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, 2016, pp. 447–461. [Online]. Available: http://doi.acm.org/10.1145/2882903.2915204

[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 2017, Santa Clara, CA, USA, July 12-14, 2017.*, 2017, pp. 195–207. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/ma

[35] 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.*, vol. 18, no. 2, 2021.

[36] P. Gera, H. Kim, P. Sao, H. Kim, and D. Bader, "Traversing large graphs on GPUs with unified memory," *Proceedings of the VLDB Endowment*, vol. 13, no. 7, pp. 1119–1133.

[37] J. Chen, Q. Wang, Y. Gu, C. Li, and G. Yu, "Unified-memory-based hybrid processing for partition-oriented subgraph matching on GPU," vol. 25, no. 3, pp. 1377–1402. [Online]. Available: https://doi.org/10.1007/s11280-021-00952-w

[38] J. Jung, D. Park, Y. Do, J. Park, and J. Lee, "Overlapping host-to-device copy and computation using hidden unified memory," ser. PPoPP '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 321–335. [Online]. Available: https://doi.org/10.1145/3332466.3374531

[39] A. H. N. Sabet, Z. Zhao, and R. Gupta, "Subway: minimizing data transfer during out-of-gpu-memory graph processing," in *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, 2020, pp. 12:1–12:16. [Online]. Available: https://doi.org/10.1145/3342195.3387537

[40] F. Sheng, Q. Cao, H. Jiang, and J. Yao, "Grabi: Communication-efficient and workload-balanced partitioning for bipartite graphs," ser. ICPP '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3404397.3404453

[41] M. Tanaka, K. Taura, T. Hanawa, and K. Torisawa, "Automatic graph partitioning for very large-scale deep learning," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2021, pp. 1004–1013.

[42] C. Tian, L. Ma, Z. Yang, and Y. Dai, "Pcgcn: Partition-centric processing for accelerating graph convolutional network," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2020, pp. 936–945.

[43] R. Mayer and H.-A. Jacobsen, "Hybrid edge partitioner: Partitioning large power-law graphs under memory constraints," in *Proceedings of the 2021 International Conference on Management of Data*, ser. SIGMOD '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1289–1302. [Online]. Available: https://doi.org/10.1145/3448016.3457300

[44] T. Ben-Nun, M. Sutton, S. Pai, and K. Pingali, "Groute: Asynchronous multi-gpu programming model with applications to large-scale graph processing," *ACM Trans. Parallel Comput.*, vol. 7, no. 3, jun 2020. [Online]. Available: https://doi.org/10.1145/3399730

[45] L. Xiang, A. Khan, E. Serra, M. Halappanavar, and A. Sukumaran-Rajam, "Cuts: Scaling subgraph isomorphism on distributed multi-gpu systems using trie based data structure," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '21. New York, NY, USA:

Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3458817.3476214

[46] C. Chen, C. Porter, and S. Pande, "Case: A compiler-assisted scheduling framework for multi-gpu systems," in *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 17–31. [Online]. Available: https://doi.org/10.1145/3503221.3508423

[47] J. Gilchrist, "Parallel data compression with bzip2," in *Proceedings of the 16th IASTED international conference on parallel and distributed computing and systems*, vol. 16, no. 2004. Citeseer, 2004, pp. 559–564.

[48] S. T. Klein and Y. Wiseman, "Parallel lempel ziv coding," *Discrete Applied Mathematics*, vol. 146, no. 2, pp. 180–191, 2005.

[49] G. Buehrer and K. Chellapilla, "A scalable pattern mining approach to web graph compression with communities," in *Proceedings of the 2008 International Conference on Web Search and Data Mining*, 2008, pp. 95–106.

[50] F. Claude and G. Navarro, "Fast and compact web graph representations," *ACM Transactions on the Web (TWEB)*, vol. 4, no. 4, pp. 1–31, 2010.

[51] F. Zhang, J. Zhai, X. Shen, O. Mutlu, and X. Du, "Poclib: A high-performance framework for enabling near orthogonal processing on compression," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 2, pp. 459–475, 2022.

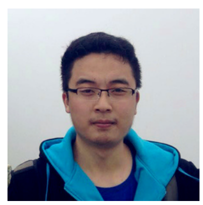
Pengyu Wang is a PhD. candidate at Shanghai Jiao Tong University, China. His research interests include systems and architectures for graph processing and graph neural network.

Cheng Xu is a MS candidate in Shanghai Jiao Tong University, China and is working toward the Ph.D degree. His current research interests include computer architecture design for graph and AI applications.

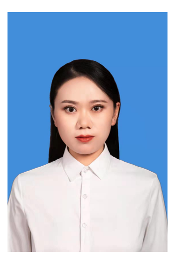
Chao Li is a professor with tenure in the Department of Computer Science and Engineering, Shanghai Jiao Tong University (SJTU). His primary research area is system architecture design with an emphasis on energy-efficient, high-performance computers of large scale.

Jing Wang is a Ph.D. candidate in Shanghai Jiao Tong University, China. Her current research interests include computer architecture, disaggregated memory, and graph processing.

Taolei Wang is a Ph.D. candidate at Shanghai Jiao Tong University, China. His current research area includes computer architecture, disaggregated memory and cloud computing.



Lu Zhang received the BS degree from the Northwestern Polytechnical University in 2016. He is working toward the PhD degree in the Department of Computer Science and Engineering, Shanghai Jiao Tong University. His research interests include edge computing, network function virtualization and serverless computing.



Xiaofeng Hou received her Ph.D. degree from Shanghai Jiao Tong University in 2020. She is currently a Post-doctoral Fellow at AI Chip Center for Emerging Smart Systems (ACCESS) and at Hong Kong University of Science and Technology. Her research mainly focuses on hardware-software co-design, power/energy management for computing system of different sizes. She received the IEEE ICCD Best Paper Award in 2018. She is a IEEE member.

Minyi Guo is a chair professor in the Department of Computer Science and Engineering of Shanghai Jiao Tong University (SJTU), China, and was the department head from 2009 to 2019. His research area includes parallel and distributed Processing, compilers, cloud computing, pervasive computing, software engineering, embedded systems; green computing, etc.