# gSampler: General and Efficient GPU-based Graph Sampling for Graph Learning

Ping Gong[†¶§], Renjie Liu[‡§], Zunyao Mao[‡§], Zhenkun Cai[§], Xiao Yan[‡], Cheng Li[†¶]
Minjie Wang[§], Zhuozhao Li[‡]
[†]University of Science and Technology of China [‡]Southern University of Science and Technology
[§]AWS Shanghai AI Lab [¶]Institute of Artificial Intelligence, Hefei Comprehensive National Science Center

## Abstract

Graph sampling prepares training samples for graph learning and can dominate the training time. Due to the increasing algorithm diversity and complexity, existing sampling frameworks are insufficient in the generality of expression and the efficiency of execution. To close this gap, we conduct a comprehensive study on 15 popular graph sampling algorithms to motivate the design of *gSampler*, a general and efficient GPU-based graph sampling framework. gSampler models graph sampling using a general 4-step *Extract-Compute-Select-Finalize* (ECSF) programming model, proposes a set of *matrix-centric APIs* that allow to easily express complex graph sampling algorithms, and incorporates a data-flow intermediate representation (IR) that translates high-level API codes for efficient GPU execution. We demonstrate that implementing graph sampling algorithms with gSampler is easy and intuitive. We also conduct extensive experiments with 7 algorithms, 4 graph datasets, and 2 hardware configurations. The results show that gSampler introduces sampling speedups of 1.14-32.7× and an average speedup of 6.54×, compared to state-of-the-art GPU-based graph sampling systems such as DGL, which translates into an overall time reduction of over 40% for graph learning. gSampler is open-source at https://tinyurl.com/29twthd4.

***CCS Concepts:*** • **Software and its engineering** → **Massively parallel systems**; • **Computer systems organization** → *Neural networks*; • **Theory of computation** → *Graph algorithms analysis.*

***Keywords:*** Graph Neural Network, Graph Sampling, Graph Learning, Graphics Processing Unit

**ACM Reference Format:**
Ping Gong, Renjie Liu, Zunyao Mao, Zhenkun Cai, Xiao Yan, Cheng Li, Minjie Wang, Zhuozhao Li. 2023. gSampler: General and Efficient GPU-based Graph Sampling for Graph Learning. In *ACM SIGOPS 29th Symposium on Operating Systems Principles (SOSP '23), October 23–26, 2023, Koblenz, Germany.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3600006.3613168

## 1 Introduction

Nowadays, graph learning models, such as graph embedding (GE) [3] and graph neural networks (GNNs) [64], play a vital role in supporting important applications such as recommendation [56], fraud detection [24], and pharmacy [43] in many domains, ranging from finance [10, 50], e-commerce [45], medicine [37], to social networks [13].

Graph sampling is widely used to train graph learning models on large graphs [5, 14, 15, 18, 25, 56–62, 65]. In particular, given some *frontier* nodes, graph sampling extracts small subgraphs (called graph samples) from the input data graph to form mini-batches for training graph learning models. Graph sampling can be a significant bottleneck in this process, constituting up to 96.2% of the end-to-end training time as shown in Table 1. This is because the sampling time cannot be overlapped by training computation that is typically lightweight for graph learning and runs on fast GPUs. We are witnessing the increasing effort of moving graph sampling to GPU, which leads to significant speedups over CPU sampling. However, on-GPU sampling still remains the bottleneck for graph learning tasks.

To support sophisticated graph learning tasks, various graph sampling algorithms have been proposed, which are far more complex than conventional random walks. Designing scalable GPU-based graph sampling requires an in-depth understanding of these algorithms. Thus, we conduct a comprehensive study on 15 popular algorithms (listed in Table 2) in Section 2.1, which reveals that the complexity of graph sampling stems from the granularity for graph traversal and the tensor-oriented computation with diverse logic required to compute sampling probabilities.

The characteristics of graph sampling algorithms make it difficult to express and optimize them with existing systems, because these systems do not offer general APIs and holistic execution optimizations. First, they offer fine-grained APIs to operate on each node or edge, e.g., *vertex-centric* by NextDoor [19] and C-SAW [30], and *message passing*

[*]Xiao Yan and Cheng Li are the corresponding authors.
[*]Work done during Ping Gong's internship at AWS Shanghai AI Lab.

**Table 1.** The ratio of graph sampling time for training 3 popular graph learning models. The graph is Ogbn-Products, and see Section 5.1 for detailed settings. GraphSAGE uses the native implementations of PyG and DGL, while FastGCN and LADIES use open-source implementations with our optimizations.

| Framework | Sampling Hardware | Model | | |
|---|---|---|---|---|
| | | GraphSAGE | FastGCN | LADIES |
| PyG | CPU | 96.2% | - | - |
| DGL | CPU | 70.1% | 95.4% | 95.4% |
| DGL | GPU | 45.8% | 57.6% | 70.1% |

by DGL [46] and PyG [11], which are inconvenient for algorithms that require a global view of graph samples (e.g., LADIES [65]) or complex computation (e.g., PASS [57]). Second, existing systems lack a general representation of various graph sampling algorithms, precluding opportunities of jointly optimizing the entire sampling process. To tackle the limitations, DGL manually optimizes the implementations of a few algorithms with substantial development effort.

In this paper, we aim to build gSampler, a general and efficient GPU-based graph sampling framework for easily implementing a wide range of sampling algorithms with enhanced performance. First, we model a graph as a sparse matrix, and generalize graph sampling as an *ECSF* programming model, which decomposes graph sampling into four steps, i.e., *extract*, *compute*, *select*, and *finalize*. Each step can then be implemented via the combination of a set of *matrix-centric APIs*, which either filters out rows or columns of a given graph matrix, optionally according to certain conditions, or performs diverse tensor computation to obtain sampling probability. The ECSF model encompasses all existing graph sampling algorithms, maintains a global view across all the steps, and allows to interleave graph and tensor operations flexibly. Moreover, our matrix-centric APIs adopt Pythonic syntax to integrate with the most popular deep learning system PyTorch, leading to succinct implementations and easy deployment.

The unified matrix abstraction for graph and tensor operations enables general strategies to accelerate the execution of different graph sampling algorithms on GPU. In particular, we use a data-flow intermediate representation (IR) to model and transform user programs written with our matrix-centric APIs. Based on the IR, we propose three types of optimization passes that are tailored for graph sampling: (1) computation optimization passes including fusion and pre-processing eliminate redundant data movements and operators; (2) data layout selection passes determine the data storage formats that are the most efficient for the entire program; and (3) super-batch sampling batches the sampling tasks to improve GPU utilization.

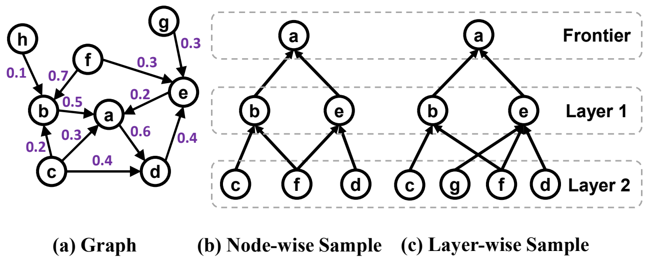


**Figure 1.** Example for graph sampling

We incorporate the matrix-centric APIs, IR, and all optimizations in gSampler, which has been further connected with DGL and PyG, two leading and widely adopted graph learning systems, for model training. We compare our gSampler with 5 state-of-the-art graph sampling systems, i.e., cuGraph [36], GunRock [49], SkyWalker [47], PyG [33] and DGL [7], and experiment with 7 graph sampling algorithms and 2 hardware configurations. The results demonstrate that gSampler outperforms baseline systems in all cases, with sampling speedups of 1.14-32.7×, where half of cases exceed 2× and the average speedup is 6.54×. We also measure the impacts of the optimization designs in gSampler, and the results suggest that they effectively accelerate execution. Finally, considering the entire training process for graph learning models, the efficient sampling delivered by gSampler translates into a substantial training acceleration, i.e., an overall time reduction of over 40%.

To summarize, we make the following contributions:

- We survey 15 popular sampling algorithms for graph learning, which reveals the unique characteristics of emerging algorithms and motivates us to model the sampling process with a 4-step ECSF model. This study also offers principles for proposing general programming models and system support for graph sampling.
- Based on the ECSF model, we propose matrix-centric APIs for graph sampling, which is general and user-friendly. The APIs together with the ECSF model and the data-flow IR not only lead to succinct and intuitive implementations but also enable holistic execution optimizations.
- We incorporate a suite of sampling-oriented optimizations for the execution of graph sampling algorithms on GPU, e.g., *computation optimizations*, *data layout selection*, and *super-batch sampling*, which effectively improve efficiency.
- We open-source gSampler and conduct an in-depth evaluation to compare with state-of-the-art sampling frameworks on representative sampling algorithms, graph learning models, and public graph datasets.

## 2 Background and Motivation

In this section, we introduce graph sampling algorithms for graph learning and discuss the design trends of these

**Table 2.** A summary of popular graph sampling algorithms. In each layer, the neighbors of some frontier nodes are sampled. *Node-wise* means that each frontier samples its neighbors independently while *layer-wise* means that sampling is conducted jointly among the neighbors of all frontiers. *Fanout (i.e., Fan)* is the number of neighbors to sample.

| Category | Bias | Fan | Algorithms | Description |
|---|---|---|---|---|
| **Node-wise** | Uniform | =1 | DeepWalk [31] | Vanilla rank walk, uniformly sample a neighbor of the frontier at each stee p |
| | | | GraphSAINT [59] | Conduct vanilla rank walk and induce subgraph according to sampled nodes |
| | | | PinSAGE [56], HetGNN [60] | Random walks following a meta-path (with node/edge types) or using restarts, select top-k visited neighbors as sampled nodes |
| | | >1 | GraphSAGE [15], VR-GCN [5] | Each frontier independently and uniformly samples *fanout* neighbors |
| | Static | >1 | SEAL [61], ShaDow [58] | Each frontier samples neighbors with uniform or PPR bias and then induce a subgraph using all the sampled nodes |
| | Dynamic | =1 | Node2Vec [14] | Random walk, a neighbor's bias is 1/q, 1/p or 1 based on the previous frontier |
| | | >1 | GCN-BS [25], Thanos [62] | Sampling bias of edges are updated with reward computed by bandit solvers |
| | | | PASS [57] | Sampling bias of edges are computed using trainable model parameters |
| **Layer-wise** | Static | >1 | FastGCN [4] | The sampling bias of a node is its degree |
| | Dynamic | | AS-GCN [18] | Sampling bias of edges are computed using trainable model updated by gradient |
| | | | LADIES [65] | The sampling bias of a node is the sum of its squared edge weights to the frontiers, edge weights of sampled subgraph are divided by sampling bias |

```
def dgl_normalize(g):
    g.edata[“e”] = g.edata[“e”] ** 2
    msg_fn = dgl.copy_e(‘e’, ‘e’)
    reduce_fn = dgl.sum(‘e’, ‘h’)
    g.update_all(msg_fn, reduce_fn)
    h = g.ndata[‘h’]
    return h / h.sum()
```

```
def matrix_normalize(A):
    h = (A ** 2).sum(axis=1)
    return h / h.sum()
```

**Figure 2.** Compute sampling bias for LADIES with DGL message passing APIs (left) and matrix abstraction (right)

algorithms. We also analyze the limitations of existing GPU-based graph sampling systems to motivate our design.

## 2.1 Graph Sampling Algorithms

Graph sampling usually involves several layers (i.e., steps for random walk) and executes layer-by-layer. Consider the toy graph in Figure 1(a) and the *graph sample* (i.e., a sampled subgraph) in Figure 1(b). The frontier node is $a$, and layer-1 samples among the in-neighbors of $a$ , i.e., $\{b, c, e\}$; if we select $b$ and $e$ as the layer-1 nodes, layer-2 samples among the in-neighbors of $b$ and $e$, i.e., $\{c, d, f, g, h\}$. We call the nodes whose neighbors are sampled *frontiers* (e.g., node $a$ for layer-1), and the sampled neighbors become the frontiers for the next layer (e.g., node $b$ and $e$ for layer-2).

In Table 2, we profile 15 popular graph sampling algorithms. We observe that these algorithms fall into two sampling patterns, namely, *node-wise sampling* and *layer-wise sampling*, and differ in how the frontiers compute sampling probability for their neighbors (i.e., *sampling bias*).

**Sampling patterns.** *Node-wise sampling* (e.g., GraphSAGE and PASS) conducts sampling for the frontiers independently. For example, in Figure 1(b), $b$ and $e$ each samples 2 of their in-neighbors and they both sample their common in-neighbor $f$. As a result, there are only 3 nodes in layer-2. *Layer-wise sampling* (e.g., FastGCN and LADIES) conducts sampling collectively among the neighbors of all frontiers. For example, in Figure 1(c), we first collect the neighbors of $b$ and $e$ as $\{c, d, f, g, h\}$ and then sample 4 nodes among them *without replacement*, i.e., one deliberately avoids choosing any member of the population more than once, which results in 4 nodes in layer-2.

After sampling, some algorithms conduct adjustments to prepare for training. For instance, SEAL and GraphSAINT induce a subgraph for the sampled nodes. LADIES adjusts the weights of the edges in the sampled subgraph for more accurate gradient estimation.

**Sampling bias.** A frontier can simply sample its neighbors with equal probability (i.e., uniform, GraphSAGE for instance). Unlike this, some algorithms assign static sampling bias for each edge/node, e.g., the node degree in FastGCN and personalized page rank (PPR) score in SEAL. The sampling bias can also be dynamic, for instance, Node2Vec determines the sampling bias of a neighbor according to its distance to the previous frontier. Both PASS and AS-GCN train a model to compute the sampling bias, and the model is updated using gradients from training. These model-driven sampling algorithms can involve complex tensor operations.

For node-wise sampling, a frontier simply normalizes the sampling bias among its neighbors to obtain the sampling probability. However, for layer-wise sampling, all frontiers need to jointly normalize the sampling bias of their neighbors. Assume that the graph sample in Figure 1(c) is constructed by LADIES, which uses squared edge weight as sampling bias. For layer-2, the candidate nodes are $\{c, d, f, g, h\}$, and they receive sampling bias from the layer-2 frontiers (i.e., $b$ and $e$). For example, the sampling bias of node $f$ is $B_f = w_{fb}^2 + w_{fe}^2 = 0.49 + 0.09 = 0.58$, where $w_{fb}$ and $w_{fe}$ are the weight of edges $fb$ and $fe$. With the sampling bias of the candidate nodes, LADIES computes their total sampling bias $B$ and uses it to normalize the sampling bias of

**Table 3.** Comparing representative GPU-based graph sampling systems with our system gSampler.

| System | Design Target | APIs |
|---|---|---|
| Gunrock, Medusa | Graph algorithms, e.g., PageRank, shortest path, and connected components | Vertex-centric, each frontier node processes a set of nodes and produces new frontiers |
| Nextdoor, C-SAW, Skywalker | Node-wise sampling without tensor computation, e.g., random walk and GraphSAGE | Vertex-centric, each frontier node samples its neighbors as new frontiers |
| DGL, PyG | Embedding mapping and neighbor embedding aggregation in GNN | Message passing, each node generates, passes, and aggregates messages along its edges |
| gSampler | Diverse graph sampling algorithms, with tensor computation and subgraph-level operations | Matrix-centric, a subgraph is a matrix that supports data extraction and computation |

each node (e.g., $B_f/B$). Thus, layer-wise sampling requires cross-frontier operations.

**Algorithm design trends.** We observe that graph sampling algorithms are moving towards tighter integration with graph learning models. This leads to the following two trends that are challenging for the design of a general and efficient graph sampling framework.

- **Graph view.** To better prepare the graph samples for training, the latest sampling algorithms involve operations such as layer-wise sampling, induced subgraph generation, and post-sampling edge weight adjustment, which require a complete view of the graph sample. In particular, layer-wise sampling allows to explicitly control the number of nodes in each layer while node-wise sampling exponentially increases the number of nodes with layer, which results in high training costs. LADIES adjusts edge weight across the sampled subgraph to reduce the variance in gradient estimation.
- **Compute-intensive.** Graph sampling can involve complex tensor computation. For example, PASS and AS-GCN compute sampling bias by training models. GCN-BS and Thanos use bandit solvers to update the sampling bias of each edge. This requires to interleave the graph operations (i.e., subgraph extraction and neighbor sampling) with tensor operations and jointly optimize them.

## 2.2 Limitations of Existing Solutions

We summarize representative GPU-based graph sampling solutions in Table 3. As mentioned above, sampling algorithms for graph learning are becoming increasingly complex and favor a complete view of the graph sample. However, most systems (e.g., Nextdoor [19], C-SAW [30], and Skywalker [47]) adopt *vertex-centric* APIs, where each frontier can only see its neighbors (i.e. local view). The vertex-centric APIs are designed primarily for random walks (which have only 1 frontier) and work well for simple node-wise sampling (which processes different frontiers independently). However, with a local view at each frontier, the vertex-centric APIs cannot handle operations that require a complete view of the graph sample. For instance, Nextdoor does not support cross-frontier bias normalization and only implements an approximate version of LADIES without bias computation. Moreover, these systems do not support tensor computation, which is required by model-driven sampling algorithms (e.g., PASS).

GNN training systems, such as DGL [46] and PyG [11], adopt *message passing* APIs, where each node generates/aggregates messages along its edges. Their APIs allow tensor computation on the nodes and edges, and are designed primarily for neighbor embedding aggregation in GNN. With a fine-grained view at each node/edge, the message passing often leads to complex implementations for graph sampling algorithms that require a complete view of the graph sample. For instance, as shown in Figure 2, using DGL, computing sampling bias for LADIES requires to first send the weights over the edges and then aggregate the weights. In contrast, treating the graph as an adjacency matrix (i.e., $A$) requires only two lines of codes (LoCs). Furthermore, employing DGL/PyG for programming intricate sampling algorithms requires users to go beyond its message passing APIs and introduce extra graph operations such as subgraph extraction or ID remapping across subgraphs. Finally, due to the lack of a unified abstraction, DGL and PyG implement and optimize the graph sampling algorithms case by case, which takes substantial effort and does not generalize to new algorithms.

## 2.3 Design Considerations for gSampler

To tackle the limitations of existing systems, we present the matrix-based abstraction, which provides a global view for graph sampling. This approach unifies fine-grained frontier, subgraph, and tensor views. Based on this abstraction, gSampler adopts matrix-centric APIs, which represent a subgraph as a matrix and conduct operations on the matrix for sampling. Our key observation is that both graph operations (e.g., extract neighbors and induce subgraph) and tensor operations (e.g., model mapping and bias computation) can be expressed as matrix operations. Thus, the matrix-centric

**Table 4.** Key operators provided by our matrix-centric APIs. We call $A$ a graph, subgraph, or matrix interchangeably. Our operator syntax is consistent with PyTorch matrix operations but mainly handle sparse matrix.

| Step | Operator | Interpretation |
|---|---|---|
| **Extract** | *A[:, columns], A[rows,:]* | Slice columns or rows from matrix $A$, return a sub-matrix (subgraph). |
| **Compute** | *A @ D* | Matrix multiply for sparse matrix $A$ and dense matrix $D$. |
| | *A.<op>(V, axis=0), <op>: add, sub, mul, div* | Broadcast operator between sparse matrix $A$ and vector $V$ on dimension *axis*, return a sparse matrix of the same shape as $A$. |
| | *A.sum(axis=0)* | Reduction operator on the dimension *axis* and return a vector. |
| | *A <op> D, A <op> v <op>: +, -, *, /, *** | Element-wise operators on sparse matrix $A$, possibly with a dense matrix $D$ or a number $v$, return a sparse matrix of the same shape as $A$. |
| **Select** | *A.individual_sample(K, probs)* | Sample $K$ neighbors for each frontier (i.e. column node) independently according to *probs* and return a sub-matrix of $A$. |
| | *A.collective_sample(K, node_probs)* | Sample $K$ out of matrix $A$'s row nodes according to *node_probs* and return a sub-matrix of $A$. |
| **Finalize** | *A.column(), A.row()* | Return the column or row node IDs of sparse matrix $A$. |

APIs support sampling algorithms that conduct diverse operations and allow to jointly optimize the entire sampling program. Moreover, with the global view for sampling, gSampler allows conducting graph-level processing conveniently with high-level matrix operations (e.g., the sampling bias computation in Figure 2), which solves the problem of fine-grained operations in existing systems. High-level matrix operations also allow us to condense common operations in graph sampling (e.g., extract subgraph and sample neighbor) and support them with built-in operators, which simplifies programming and improves usability.

## 3 Matrix-centric APIs for Graph Sampling

In this section, we introduce our matrix-centric APIs and *Extract-Compute-Select-Finalize* (ECSF) programming model for graph sampling, and show how to use them to easily express diverse graph sampling algorithms.

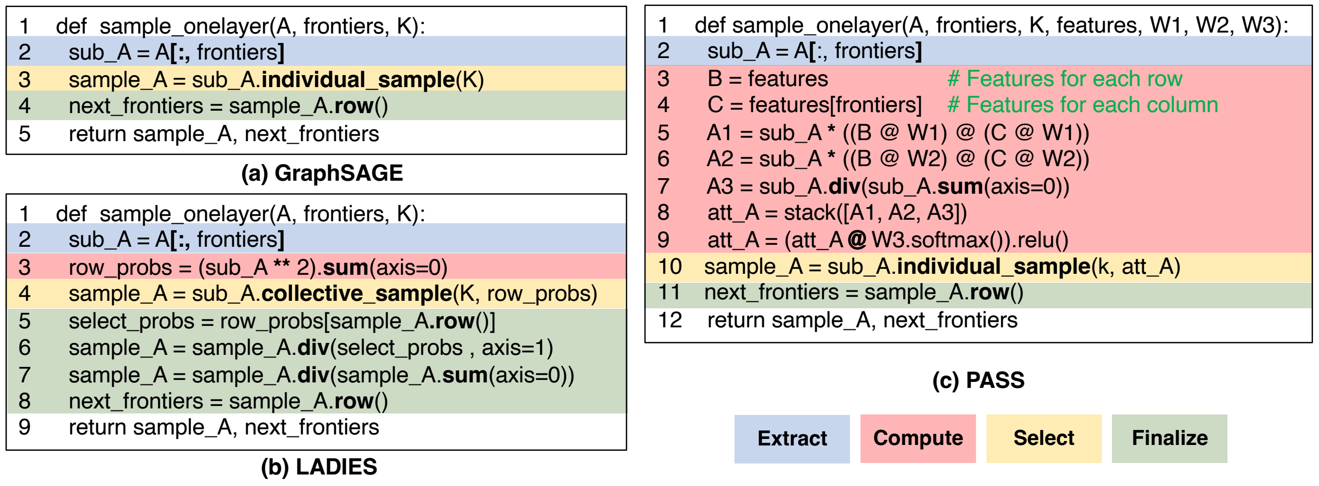
### 3.1 Matrix-centric APIs

For graph learning, the input data is a graph $G = (V, E)$, in which each node $v \in V$ (resp. edge $e \in E$) has a feature vector $f_v$ (resp. $f_e$) that records its attributes (e.g., the characteristic of a user node in social networks), and each edge $e$ may also have a weight $w_e$. We use an adjacency matrix $A$ to represent the graph topology, which is a sparse matrix [51] with shape $(|V|, |V|)$. For a node $v$, the row of the matrix (i.e., *A[v,:]*) keeps its out-going edges while the column (i.e., *A[:,v]*) keeps its in-coming edges. Matrix $A$ records the existence of edges if the graph is unweighted and keep edge weights if the graph is weighted. Conversely, we allow users to treat a matrix $A'$ as a graph, where each element $a_{ij} \in A'$ is the weight or sampling bias of an edge. Logically, the node (resp. edge) feature vectors are kept in an array and accessed via node (resp. edge) ids.

We observe that all existing graph sampling algorithms follow a 4-step *Extract-Compute-Select-Finalize* procedure, which can be conducted via matrix operations. Consider a layer of graph sampling, the *extract step* obtains the subgraph between the frontier nodes and their in-neighbors; the *compute step* calculates sampling probability for the candidate nodes; the *select step* samples the candidate nodes according to the sampling probability; and the *finalize step* determines frontiers for the next layer and adjusts the graph sample. We provide the operators in Table 4 to support the four steps and discuss them as follows.

**Extract step.** Given a list of node ids as *frontiers*, *sub_A = A[frontiers,:]* and *sub_A = A[:, frontiers]* extract the corresponding rows and columns from matrix $A$, which are essentially the out-neighbor and in-neighbor subgraphs for the *frontier* nodes. As graph sampling usually samples the in-neighbors of the frontier nodes, the extract step can be conducted by *sub_A = A[:, frontiers]*. If matrix A is of size $M \times N$ and there are $T$ frontier nodes, the output *sub_A* is of size $M \times T$.

**Compute step.** The compute step can be skipped for simple algorithms that conduct uniform sampling (e.g., vanilla random walk and GraphSAGE) but can be complex for advanced algorithms. Users can adopt our compute operators to conduct sampling bias computation, which share the same syntax as PyTorch operators to minimize semantic gap and learning overhead. Under the hood, we use sparse storage formats and computation optimizations for efficiency. For instance, consider matrix multiplication (*A @ D*), where the shape of matrix A and matrix D are $(N, M)$ and $(M, K)$, respectively. The resulting matrix shape is $(N, K)$, conforming to standard matrix multiplication. For efficiency, *A @ D* can be performed using sparse matrix multiplication (SpMM) [17]. Users can



```
1  def  sample_onelayer(A, frontiers, K):
2    sub_A = A[:, frontiers]
3    sample_A = sub_A.individual_sample(K)
4    next_frontiers = sample_A.row()
5    return sample_A, next_frontiers
```

```
1  def  sample_onelayer(A, frontiers, K):
2    sub_A = A[:, frontiers]
3    row_probs = (sub_A ** 2).sum(axis=0)
4    sample_A = sub_A.collective_sample(K, row_probs)
5    select_probs = row_probs[sample_A.row()]
6    sample_A = sample_A.div(select_probs , axis=1)
7    sample_A = sample_A.div(sample_A.sum(axis=0))
8    next_frontiers = sample_A.row()
9    return sample_A, next_frontiers
```

```
1  def sample_onelayer(A, frontiers, K, features, W1, W2, W3):
2    sub_A = A[:, frontiers]
3    B = features              # Features for each row
4    C = features[frontiers]   # Features for each column
5    A1 = sub_A * ((B @ W1) @ (C @ W1))
6    A2 = sub_A * ((B @ W2) @ (C @ W2))
7    A3 = sub_A.div(sub_A.sum(axis=0))
8    att_A = stack([A1, A2, A3])
9    att_A = (att_A @ W3.softmax()).relu()
10   sample_A = sub_A.individual_sample(k, att_A)
11   next_frontiers = sample_A.row()
12   return sample_A, next_frontiers
```

**Figure 3.** Implement the graph sampling algorithm by populating our ECSF interface with our matrix-centric APIs

also employ standard PyTorch operators to perform dense matrix/vector computations.

**Select step.** We provide two operators, namely, *individual_sample* and *collective_sample*, for node-wise and layer-wise sampling, respectively. The *individual_sample* operator independently samples $K$ neighbors for each frontier (i.e., column) of subgraph *sub_A* according to the sampling probability *probs*. Both *probs* and the return matrix have the same size as *sub_A*. The *collective_sample* operator samples $K$ of *sub_A*'s row nodes based on the sampling bias *node_probs*, which is a vector of the same dimension as the rows of *sub_A*, and returns a matrix that contains only the edges between the sampled row nodes and the frontiers (i.e. column nodes). Thus, the shape of *SubA.collective_sample()* is $K \times T$, where $T$ in the number of frontier. When *probs* or *node_probs* is omitted, we assume that the sampling bias on each edge is 1 and automatically aggregates the node sampling bias for *node_probs*. Moreover, we allow users to directly use sampling bias as *probs* and *node_probs* to skip normalization.

**Finalize step.** Following the above three steps, some other operations may be required, e.g., induce a subgraph, determine the frontiers of the next layer, and adjust edge weight to prepare for training. To support these operations, we provide two operators in Table 4, which returns elements of the sampled subgraph *sample_A*. For instance, *A.row()* and *A.column()* return the node IDs for the rows and columns of matrix A, which can be used to determine new frontiers. Note that even if A is a subgraph, the two operators still return the IDs of the nodes in the original graph, which avoids ID conversion for users. Users may also conduct other post-processing computations with our operators.

## 3.2 Example Use Cases

We demonstrate the generality and usability of our matrix-centric APIs and the ECSF programming model by expressing three representative graph sampling algorithms. In particular, GraphSAGE is a simple algorithm without the compute step, while the other two algorithms are more complex with bias computation. For simplicity, here, we focus on a layer of sampling as it is straightforward to stack multiple layers.

**GraphSAGE.** GraphSAGE conducts node-wise sampling, and each frontier samples its neighbors uniformly. In Figure 3(a), Line 2 extracts the subgraph between the frontiers and their in-neighbours. The compute step is skipped as GraphSAGE samples neighbors uniformly. Line 3 executes the *select* step, which employs the *individual_sample* operator to conduct node-wise sampling and select $K$ neighbors for each frontier node. To *finalize*, line 4 collects the row nodes of *sample_A* as the frontiers for the next layer. Note that if we set the number of neighbors to sample as $K$=*1*, GraphSAGE becomes a vanilla random walk.

**LADIES.** LADIES conducts layer-wise sampling and uses squared edge weights as sampling bias. In Figure 3(b), Line 2 *extracts* the subgraph between the frontiers and their in-neighbours. Line 3 executes the *compute* step, which conducts element-wise square for the edge weights of the extracted subgraph and aggregates the sampling bias of the candidate node by summing in the row dimension. Line 4 uses the *collective_sample* operator to conduct layer-wise sampling. To *finalize*, lines 5-7 normalize the edge weights for the layer to prepare the graph sample for training, and line 8 extracts the row nodes of *sample_A* as new frontiers.

**PASS.** PASS performs node-wise sampling but trains 3 projection matrices *W1*, *W2*, and *W3* to control the sampling bias. Figure 3(c) shows the matrix-centric implementation of PASS. Line 2 *extracts* a subgraph containing the frontiers and their in-neighbors. Lines 3-9 comprise the *compute* step, which is more complex than the above two algorithms. Lines 3-4 prepare dense matrices $B$ and $C$ that store feature vectors for the row and column nodes. Lines 5-6 use PyTorch dense matrix multiplication and element-wise sparse multiplication to compute two attention weights on the edges of *sub_A* (i.e., *A1* and *A2*) by using the projection matrices (i.e., *W1* and *W2*) to map the node features of the edges. Note that @ means matrix multiply in PyTorch. Line 7 normalizes the edge weights of *sub_A* by row, which is used as the third edge attention. Lines 8-9 concatenate the three attention weights and use project matrix *W3* to map the result to sampling bias. For the *select* step, line 10 employs node-wise sampling, while Line 11 prepares frontiers for the next layer to complete the *finalize* step.

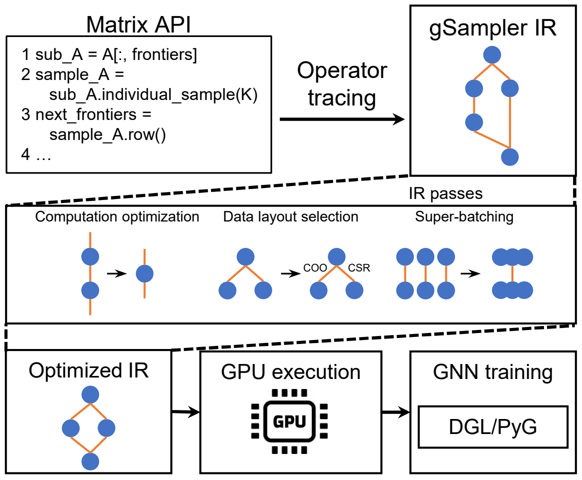
**Advantages of matrix-centric APIs.** From the examples, we can observe several advantages of our matrix-centric APIs. First, a global view of the graph sample and high-level matrix operations enable simple subgraph operations such as sampling bias aggregation and edge weight adjustment, which are complex to implement on the fine-grained vertex-centric or message passing APIs of existing systems. Second, different graph sampling algorithms share similar extract and select logic but differ significantly in sampling bias computation. We provide operators for typical extract and select patterns for good usability and enable users to customize bias computation with a complete set of matrix operators for good generality. Third, the semantic of our operators is consistent with PyTorch tensor operators, and users can express all operations (i.e., either graph or tensor) via matrix operators. This not only makes it easy to write and understand code with our APIs but also enables holistic optimizations for the entire sampling process as in the next section.

### 3.3 Expressiveness and Limitation

gSampler can express all 15 algorithms in Table 2 and has potentials to inspire new sample algorithm designs and facilitate easy integration of advanced graph sampling algorithms with modern graph learning frameworks. However, its expressiveness comes at the cost of some programming overhead for conventional sampling tasks that favor fine-grained or local views. That is, random walks are slightly more complex to express in gSampler than existing systems because they fit well with the conventional vertex-centric abstraction. For instance, while DeepWalk in C-SAW requires only 3 lines of code (LoCs), gSampler employs 10 LoCs. However, gSampler's inherent optimizations ensure that random walks achieve comparable efficiency with specialized frameworks. Therefore, the associated overhead is reasonable in return for heightened generality and expressiveness, particularly for intricate graph sampling algorithms.

## 4 The gSampler System

In this section, we firs provide an overview on how gSampler optimizes user code for GPU execution. Then, we discuss the core optimizations of gSampler for efficiency.



```
1 sub_A = A[:, frontiers]
2 sample_A =
    sub_A.individual_sample(K)
3 next_frontiers =
    sample_A.row()
4 ...
```

**Figure 4.** The execution workflow of gSampler

### 4.1 Overall Workflow

Figure 4 shows the execution workflow of gSampler. First, a user program written with our Pythonic matrix-centric APIs is parsed into an intermediate representation (IR), which is a data flow graph comprising computation operators as nodes and data dependencies as edges. Then, gSampler applies several optimization passes over the IR to improve execution efficiency. Finally, the optimized IR is translated into machine code for GPU execution.

In particular, gSampler conducts three types of IR optimization passes. First, computation optimization passes including fusion and pre-processing are applied to eliminate redundant data movements and operators (Section 4.2). Second, the data layout selection passes determine the data storage formats that are the most efficient for executing the program (Section 4.3). Finally, the super-batch sampling batches the sampling tasks to improve GPU utilization (Section 4.4).

Modeling graph sampling as a data-flow IR allows global optimizations that consider and benefit the entire sampling process, which are impossible in existing systems. For instance, gSampler conducts operator fusion and data layout selection that require tracking the dependencies between all operators in a program. In contrast, frameworks like Nextdoor and C-SAW only optimize a single operator (e.g., neighbor selection and bias computation), while GNN systems like DGL and PyG run graph sampling in eager mode without IR (i.e., one operator at a time). Moreover, the IR representation also allows to hide the optimization details from users and incorporate new optimizations as additional passes.

### 4.2 Computation Optimizations

**Operator fusion** combines adjacent operators in the IR to avoid materializing intermediate results [41] and is widely used to accelerate GPU execution. Considering the characteristics of our matrix-centric APIs and ECSF model, we tailor a

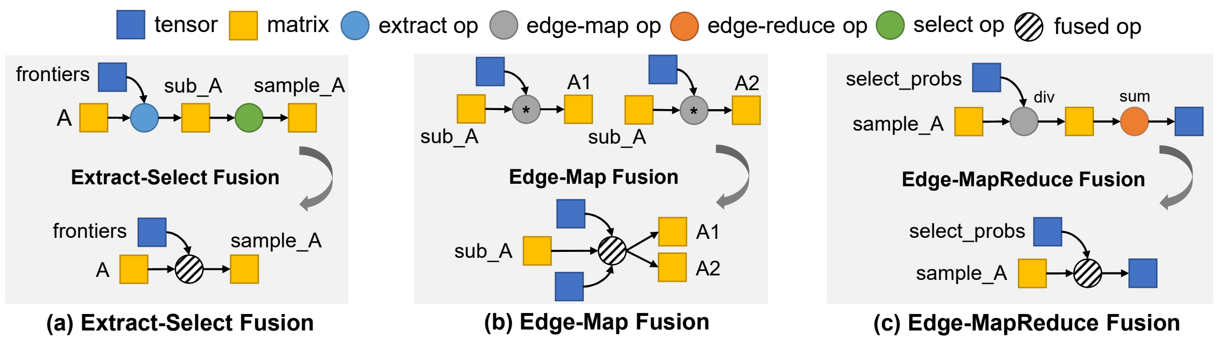


**Figure 5.** Operator fusion for the use case graph sampling algorithms in Section 3.2: (a) for line 2-3 in GraphsAGE, (b) for line 5-6 in PASS, and (c) for line 6-7 in LADIES case.

set of operator fusion rules that differ from those commonly utilized for DNNs and illustrate them in Figure 5.

For sampling algorithms that skip the compute step, gSampler applies *Extract-Select fusion*, which fuses the extract operator and select operator to avoid materializing the extracted subgraph. For example, in Figure 3(a),*sub_A = A[:, frontiers] & sub_A.individual_sample()* are fused. As shown in Figure 5(a), this allows to extract subgraph and sample neighbors in one pass.

For algorithms that conduct the compute step, gSampler designs operator fusion rules by using the fact that the compute operators are either *map* or *reduce* operators on the edge data of sparse matrices. In particular, we distinguish two types of operators: (1)*Edge-map* operator updates the scalars/vectors on the edges of matrix, e.g., *A.<op>(V, axis)*, where *V* is a dense tensor and *axis* is a number; and (2) *Edge-reduce* operator reduces data from the edges to the nodes and returns a dense tensor, e.g., *A.sum(axis)* and *A @ D*, where *D* is a dense tensor. For a user program, gSampler marks all compute operators on the sparse matrices as either *edge-map* operator or *edge-reduce* operator, and then apply the following fusion rules:

- *Edge-Map fusion.* Inspired by element-wise fusion [12], gSampler fuses consecutive edge-map operators that act on the same sparse matrix *A* into a single edge-map operator because they all process the edges of *A*. Figure 5(b) shows the edge-map fusion in PASS, i.e., the code in Figure 3(c).
- *Edge-MapReduce fusion.* For a pair of consecutive edge-map and edge-reduce operators that act on the same matrix *A*, the former updates the data on the edges while the latter aggregates the updated edge data onto the nodes. gSampler fuses them into a single edge-reduce operator to avoid writing the results of the edge-map operator to GPU global memory. Figure 5(c) shows the edge-MapReduce fusion in LADIES, i.e., the code in Figure 3(b).

While computation fusion is a general technique, its application in graph sampling presents new challenges. Deep learning compilers like TVM [6] and Tensorflow XLA [41] cannot fuse graph operators as they focus on dense computations. Graph compilers like Seastar [52] and Graphiler [53] are inadequate for graph sampling as they assume fixed graph structures, whereas graph sampling involves dynamic changes. In contrast, gSampler takes a different perspective, where we consider both the operators for graph sampling and the operators for bias computation. This allows optimizing them jointly within a single framework.

**Pre-processing.** gSampler pre-computes the variables that do not change before sampling, which fall in two cases. (1) An operator generates a constant value, e.g. computing the degree of each node in FastGCN and the PPR scores in SEAL. (2) When applied to a subgraph *sub_A*, an operator produces the same result on each edge as if applied to the entire graph *A*. In this case, gSampler executes the operator on *A* in advance and then substitutes the original operator node in the IR with an extract operator that reads the pre-computed results. For example, in LADIES, by pre-computing *M = A ** 2* on *A*, *sub_A = A[:, frontiers] and sub_A ** 2* are replaced with *sub_M = M[:, frontiers].*

**Other computation passes.** gSampler also adopts standard optimization passes to process the IR. For example, dead code elimination (DCE) [1] removes the operator nodes without outgoing edges. Common Subexpression Elimination (CSE) [1] keeps only one of the operator nodes that produces the same result.

## 4.3 Data Layout Selection

gSampler stores graphs and matrices using sparse formats, i.e., compressed sparse row (CSR), compressed sparse column (CSC), and coordinate list (COO) [51]. In particular, CSR stores the out-neighbors of each node consecutively; CSC stores the in-neighbors consecutively; COO stores the edge list. gSampler needs to choose efficient sparse formats for a user program but different operators have different preferences for the formats. We illustrate this phenomenon in Table 5 using LADIES as an example. In particular, CSC is

**Table 5.** Time cost (ms) of the operators in LADIES on different graph formats and the format conversion cost. The graph is Ogbn-Products.

| **Operator** | CSC | COO | CSR |
|---|---|---|---|
| *A[:, frontiers]* | 1.32 | 18.42 | 14.13 |
| *sub_A.sum(...)* | - | 0.86 | 0.55 |
| *sub_A.collective_sampling()* | 2.54 | 1.52 | 0.50 |
| format conversion | CSC2COO 0.38 | | COO2CSR 2.40 |

efficient for extracting in-neighbors but inefficient for collecting out-going edges while the opposite is true for CSR. gSampler may use different sparse formats for consecutive operators but this introduces a format conversion overhead, which should be balanced with the benefits of using efficient sparse formats for the operators. Similarly, during the sampling process, gSampler may remove isolated nodes without edges (i.e., compaction) to reduce the sizes of intermediate matrices but the gain should be balanced with the overhead of compaction. For example, as the extract step keeps the original row dimension, it could result in a matrix where a large number of isolated row nodes that do not connect to the frontiers. gSampler may remove these isolated nodes by paying the cost for global to local id conversion.

We introduce a data layout selection optimization pass to gSampler, which makes decisions on sparse format selection and graph compaction. Specifically, in our ECSF model, only the extract and select steps modify the graph structure, while the compute and finalize steps conduct computation or retrieve attributes of a graph. This means that operators of the compute and finalize steps just adopt the data layout of their upstream operators, and thus we only need to find optimal data output layout for the extract and select operators. Due to the small search space, brute force search can usually determine the optimal data layout in reasonable time (e.g., within 1 second), and the search time is amortized over many mini-batches.

## 4.4 Super-batch Sampling

Graph learning consumes the graph samples in mini-batch, and a small batch size is usually required for good model quality [21, 32, 54]. However, for graph sampling, a small batch size means lightweight workload and may lead to GPU under-utilization. We show such an example in Figure 6 for GraphSAGE and LADIES, where the time to go over all frontier nodes (i.e., an epoch) first reduces and then stabilizes with batch size, indicating that GPU can only be well utilized at large batch size.

For many graph sampling algorithms (except those that require model update, e.g., PASS), we can sample multiple mini-batches together. This super-batch sampling method

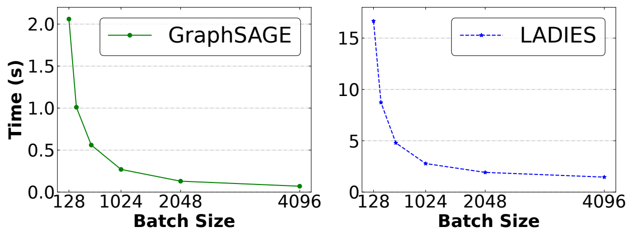


**Figure 6.** Sampling time in an epoch with different batch sizes. The graph is Ogbn-Products.

differs from a larger batch sampling task that merges duplicate nodes and edges during the sampling process and finally returns a single subgraph. Contrary, super-batch sampling refers to sampling multiple batch tasks simultaneously, where each batch's sampling process is independent of the others, even if there are duplicate edges or nodes between them, and the final results are multiple sampled subgraphs. gSampler ensures the correctness of super-batch sampling by mapping the graph nodes in each mini-batch to a different ID space such that the mini-batches do not interfere with each other during execution.

A naive implementation of the super-batch sampling is to create a super-batch version for each operator, requiring substantial engineering effort. To address this problem, we observe that the operators for all steps except *compute* are limited in number and similar across different algorithms. Thus, we provide a few dedicated super-batch operators for the *extract* and *select* steps to replace the corresponding operators in the IR. For example, we implement *segmented_collective_sample* to replace *collective_sample*, where the neighbors are sampled independently for each batch. This strategy does not apply to the *compute* step as the number of operators is large and their logic are diverse. As such, instead of changing the operators, we choose to properly construct large batch input. In particular, we concatenate the sliced matrices obtained by the *extract* step as the diagonal of a larger matrix, which is fed to the *compute* operators. This approach produces correct results and balances development effort for the non-compute operators and generality for the compute operators.

Although super-batch sampling improves efficiency, it takes extra GPU memory to store multiple mini-batches. Fortunately, many studies observe that GNN models are small and they do not use much GPU memory. To control the memory overhead, gSampler allows users to specify a memory budget for sampling and uses a grid search to determine the maximum super-batch size within the memory budget.

## 4.5 Implementation Details

We build gSampler on PyTorch [35] with 7.7K and 2.1K lines of code in C/C++/CUDA and Python, respectively. gSampler utilizes torch.fx [34] to generate the data-flow IRs and implement all IR optimizations. gSampler also leverages the

**Table 6.** The graphs used in the experiments.

| Dataset | Abbr. | \|V\| | \|E\| | Size (GB) |
|---|---|---|---|---|
| **Livejournal** | LJ | 5M | 69M | 1.1 |
| **Ogbn-Products** | PD | 2.5M | 126M | 3.4 |
| **Ogbn-Papers100M** | PP | 111M | 1.6B | 29 |
| **Friendster** | FS | 65M | 1.8B | 31 |

PyTorch GPU memory cache pool to reduce the overhead of GPU memory management. For heterogeneous graphs, each type of edges is modeled as a sparse matrix to conduct the same sampling workflow as homogeneous graphs. To support training, gSampler provides two APIs *to_dgl_graph* and *to_pyg_graph* to convert the produced sparse matrices produced into the graph formats required by DGL and PyG. By default, gSampler stores graphs in GPU memory for fast access. For large-scale graphs that exceed GPU memory, gSampler keeps them in CPU memory and accesses them via Unified Virtual Addressing (UVA) [29] in GPU kernels.

## 5 Evaluation

Our evaluation answers the following questions:

- *How does gSampler compare with state-of-the-art systems in terms of generality and efficiency for graph sampling algorithms?*
- *By making graph sampling efficient, how does gSampler accelerate graph learning tasks?*
- *How effective are the optimizations of gSampler?*

### 5.1 Experimental Setup

**Graphs and sampling algorithms.** We use the graphs in Table 6 for the experiments, which are obtained from OGB [16] and SNAP [38]. The PD and FS graphs are undirected, and we create two directed edges for each undirected edge. We randomly generate 128-dimension float feature vector for each node in the LJ and FS graphs as they do not come with node features. For the FS graph, we randomly sample 1% of its nodes as the frontier nodes for graph sampling. Among the 15 graph sampling algorithms in Table 2, we choose 7 representatives for the experiments, i.e., Deepwalk [31], Node2Vec [14], GraphSAGE [15], LADIES [65], AS-GCN [18], PASS [57] and ShaDow [58]. Among them, Deepwalk, Node2Vec, and GraphSAGE are relatively simple algorithms that conduct node-wise sampling without tensor computation. The other 4 algorithms are more complex, i.e., LADIES and AS-GCN conduct layer-wise sampling, PASS involves tensor computation, and ShaDow induces a subgraph using all the sampled nodes.

**Baselines.** For graph sampling, we compare gSampler with DGL [46], PyG [11], SkyWalker [47], Gunrock [49], and cuGraph [36]. Among them, DGL and PyG are two popular graph learning frameworks that also support graph sampling. Skywalker is a graph sampling framework with highly optimized GPU kernels for biased node-wise sampling. Gunrock is a general graph processing system on GPU that also supports unbiased node-wise sampling. cuGraph is a library from NVIDIA to support versatile graph functionalities on GPU. We do not include C-SAW [30] and NextDoor [19] as SkyWalker is reported to outperform them [47]. For the end-to-end time of graph learning tasks, we integrate gSampler with DGL for GNN training and compare with DGL and PyG.

DGL and PyG use CSC for the initial graph. gSampler does so because CSC is efficient in extracting incoming edges ($A[:, frontiers]$ in Table 5), which is the first step of graph sampling. cuGraph, GunRock, and SkyWalker use CSC, their best performing format, for the entire procedure.

**Experiment platform.** By default, we use an p3.16xlarge [2] instance on AWS with 64 vCPUs (Intel(R) Xeon(R) CPU E5-2686 v4 @ 2.30GHz) and 488GB memory, and utilize a single V100 GPU with 16GB memory. To test the influence of hardware, we also use a second experiment machine with T4 GPU, which also has 16 GB memory but smaller FLOPS than V100. We store the graphs on GPU memory when they fit in (i.e., LJ and PD) and keep them on CPU memory otherwise (i.e., PP and FS). The operating system is Ubuntu 20.04, and the softwares are CUDA 11.7 [28], PyTorch 2.0.0 [35], DGL 1.0.2 [7], PyG 2.3.0 [33], and cuGraph 23.02 [36].

**Performance metrics.** For graph sampling, we measure the sampling time for an epoch (called *sampling time* afterwards), which goes through all the frontier nodes in a graph once in mini-batches. For the 7 graph sampling algorithms, their hyper-parameters (number of layers, batch size and fanout) follow the author code of their original papers and examples released by well-known graph learning frameworks (i.e., DGL and PyG). We run 6 epochs for each configuration (i.e., system and algorithm) and report the average sampling time of the last 5 epochs, leaving the first epoch for warm-up. We do not report deviations in sampling time since they are within 1% across epochs.

### 5.2 Main Result: Sampling Time Comparison

We compare the sampling time of our gSampler with the baseline systems for the 3 simple graph sampling algorithms in Figure 7 and the 4 more complex graph sampling algorithms in Figure 8. For Figure 7, cuGraph cannot finish loading the PP graph in 10 hours. GunRock only implements GraphSAGE and cannot handle the large PP and FS graphs because it cannot use UVA. PyG can only run DeepWalk on GPU and does not support UVA. DGL has no GPU implementation for Node2Vec. For Figure 8, cuGraph, GunRock, and Skywalker do not support the complex graph sampling algorithms due to their vertex-centric APIs. PyG also has no GPU support for the complex algorithms and only provides CPU implementation for ShaDow. DGL does not have native

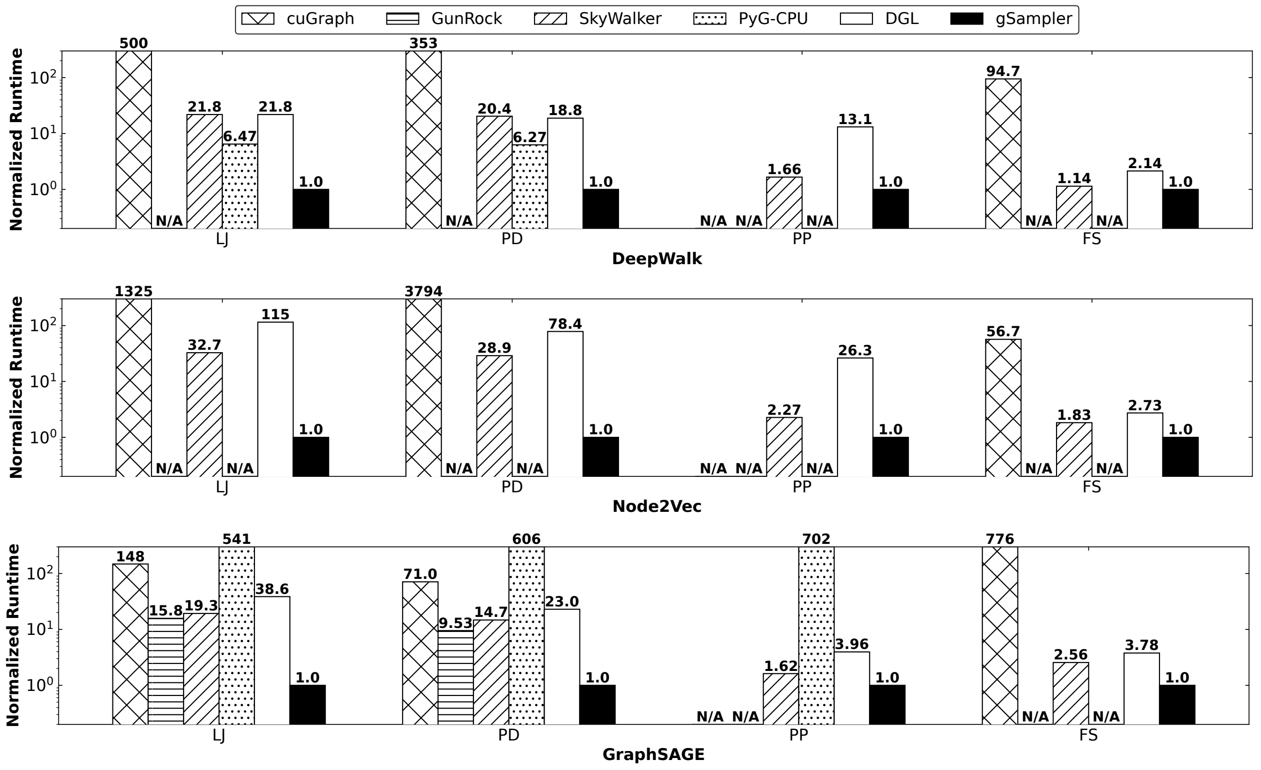


**Figure 7.** Normalized sampling runtime (gSampler is 1.0) and the baseline systems for the 3 simple graph sampling algorithms, DeepWalk, Node2Vec, and GraphSAGE. N/A means that the code is not available.

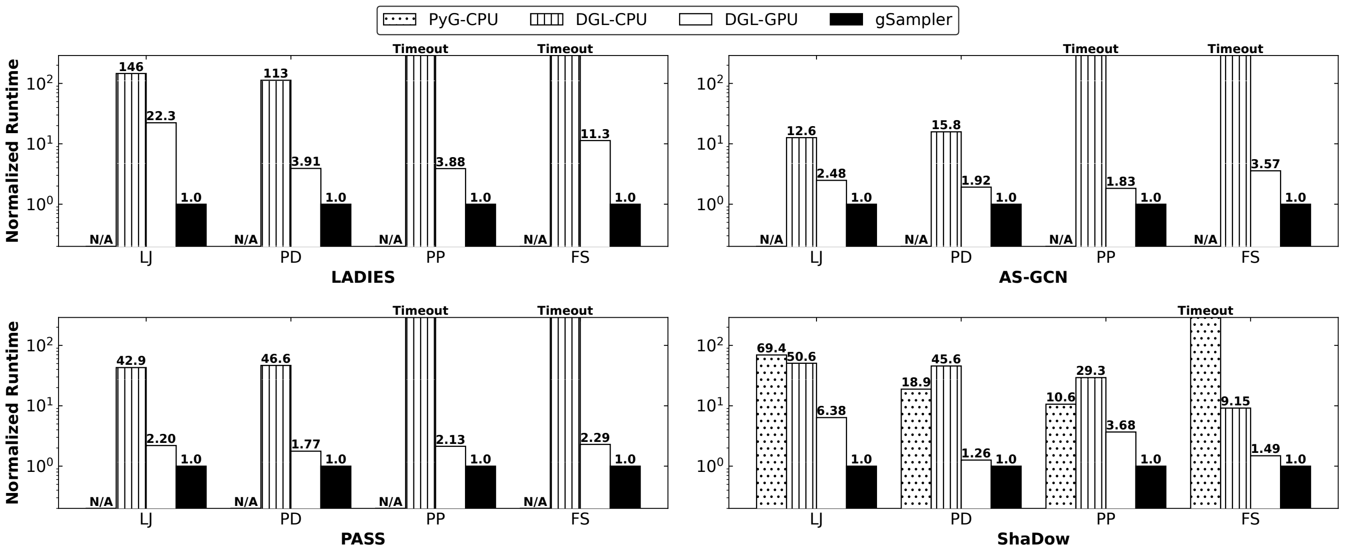


**Figure 8.** Normalized sampling time (gSampler is 1.0) and baseline systems for the 4 complex graph sampling algorithms, i.e., LADIES, AS-GCN, PASS, and ShaDow.

implementations for LADIES, AS-GCN, PASS, and ShaDow. For a fair comparison, since DGL is the best-performing baseline, we have made our best efforts to manually develop missing DGL implementations with optimizations. When using CPU, DGL cannot complete sampling in 10 hours for LADIES, AS-GCN, and PASS on the large PP and FS graphs. The large number of missing results and timeouts in Figure 7 and Figure 8 show the limited generality of existing systems. In fact, gSampler is the only system capable of running all 7 graph sampling algorithms on a GPU.

**Table 7.** The speedup of gSampler over the best performing baseline for each graph and algorithm.

| Algorithm | LJ | PD | PP | FS |
|---|---|---|---|---|
| **GraphSAGE** | 15.80 | 9.53 | 1.62 | 2.56 |
| **DeepWalk** | 6.46 | 6.27 | 1.66 | 1.14 |
| **Node2Vec** | 32.67 | 28.94 | 2.27 | 1.83 |
| **LADIES** | 22.31 | 3.91 | 3.88 | 11.31 |
| **AS-GCN** | 2.48 | 1.92 | 1.83 | 3.57 |
| **PASS** | 2.20 | 1.77 | 2.13 | 2.29 |
| **ShaDow** | 6.38 | 1.26 | 3.68 | 1.49 |

Figure 7 and Figure 8 show that our gSampler consistently outperforms all baselines across all sampling algorithms and datasets, suggesting that the optimizations of gSampler are general. In Table 7, we report the speedup of gSampler over the best-performing baseline for each case (i.e., graph and sampling algorithm). The results show that the speedup of gSampler is up to 32.67x and over 2x for 19 out of the 28 cases, and the average speedup is 6.54x. We also observe that graph sampling runs much faster on GPU than on CPU even when the large graphs are stored on CPU memory. For example, for GraphSAGE on the PP graph, gSampler and SkyWalker (on GPU) speed up PyG (on CPU) by 702x and 433x, respectively.

Figure 7 shows that for the 3 simple graph sampling algorithms, SkyWalker usually performs the best among the baselines. The speedup of gSampler over SkyWalker is more significant for the two small graphs than the two large graphs because accessing the large graphs via UVA becomes a bottleneck. gSampler also achieves larger speedup over SkyWalker for GraphSAGE than DeepWalk and Node2Vec. This is because GraphSAGE samples multiple neighbors for each frontier node while DeepWalk and Node2Vec are random walks and sample only 1 frontier node, and the heavier workload of GraphSAGE leaves more room for improvement. When PyG, DGL, and GunRock can run on GPU, they usually match or even outperform SkyWalker. However, cuGraph runs much slower than the other systems on GPU because it is inefficient for the mini-batch sampling of graph learning [9].

For the 4 complex sampling algorithms, Figure 8 shows that both DGL and gSampler run much faster than the CPU baselines. gSampler generally has larger speedup over DGL for LADIES than AS-GCN, PASS, and ShaDow because the later 3 algorithms have heavier tensor computation, which is well-optimized in DGL. Compared with the other datasets, gSampler usually has smaller speedup over DGL for the PD graph. This is because PD has the largest average degree, which results in heavier computation.

**Speedups on large-scale graphs.** In Figure 7 and Figure 8, for large-scale graphs like PP and FS, gSampler shows significant improvements even when the graph is stored in CPU memory and accessed through UVA. The reasons are two-folded. First, all steps of graph sampling except *Extract* run on the GPU, which benefit from the high memory bandwidth and compute parallelism of GPU. Second, graph sampling has skewed access to the nodes [27], and thus adjacency lists of the popular nodes are likely to be cached in the GPU, which reduces PCI-e traffic. Additionally, the improvements can be amplified when gSampler is combined with existing graph structure caching solutions [39, 63]. We leave this in the future work.

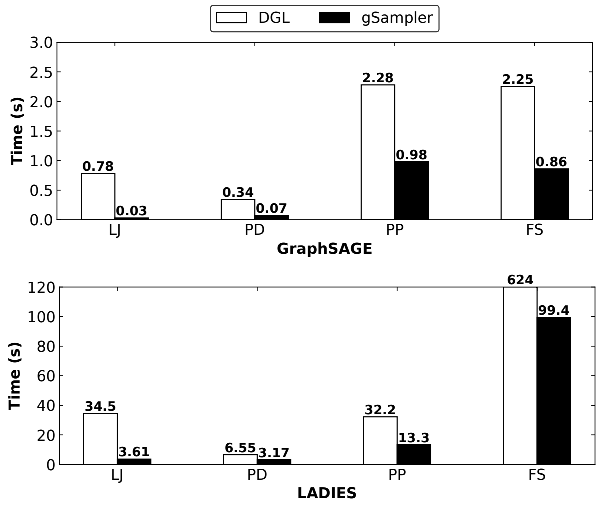


**Figure 9.** The sampling time on T4 GPU

**Results on T4.** We use GraphSAGE and LADIES as representatives of simple and complex graph sampling algorithms, respectively, and experiment them on T4 GPU to explore the influence of hardware. Note that the memory bandwidth and FLOPS of T4 are 30.0% and 51.6% of V100, respectively. The results in Figure 9 show that gSampler again achieves consistently shorter sampling time than DGL. For GraphSAGE, the speedup over DGL is up to 26x, and the average speedup is 8.9x for the 4 graphs. For LADIES, the average speedup over DGL is 5x. Compared with V100, the speedup of gSampler over DGL on T4 is generally smaller because T4 has smaller memory bandwidth and FLOPS than V100.

## 5.3 End-to-end Results for graph learning tasks

In Table 8, we report the end-to-end time and model accuracy for training GraphSAGE and LADIES on the PD graph when the model accuracy converges, i.e., accuracy stabilizes after training for a large number of epochs. Note that PyG does not support LADIES. The results show that gSampler yields almost the same convergence accuracy as DGL and PyG. This is because gSampler executes the same graph sampling logic as the baselines, and the small differences are caused by random initialization. Table 8 also shows that the more efficient graph sampling of gSampler reduces the end-to-end

**Table 8.** End-to-end performance for Graph Learning Tasks

| Algorithm | System | Time (s) | Accuracy (%) |
|---|---|---|---|
| **GraphSAGE** | gSampler | 226.21 | 90.48 |
| | DGL | 322.89 | 90.35 |
| | PyG | 13082.45 | 90.44 |
| **LADIES** | gSampler | 450.89 | 89.38 |
| | DGL | 808.94 | 89.39 |

model training time of DGL by 30.0% and 44.3% for GraphSAGE and LADIES, respectively. The large time reduction is because graph sampling takes up a significant portion of model training cost (45.8% for GraphSAGE and 70.1% for LADIES on DGL, see Table 1).

## 5.4 Breakdown Analysis

In Figure 10, we run GraphSAGE and LADIES on the PD and PP graphs to validate the effectiveness of our optimizations, i.e., computation optimization, data layout selection, and super-batch sampling. For the configurations without data layout selection, we greedily select the optimal sparse format for each operator without considering the conversion overheads, similar to the approach used in DGL.

**Plain implementation.** For GraphSAGE, gSampler w/o any optimization (denoted as 'P') is even better than DGL due to the more efficient operator implementation. Unlike this, for LADIES, gSampler outperforms DGL for PD graph but falls behind for PP graph. This is because the presence of more isolated nodes in PP graph hurts the performance.

**Computation optimizations.** gSampler's computation optimizations reduce sampling time by over 50% for both graphs, when training GraphSAGE. This is mainly because the *Extract-Select fusion* can reduce half of the GPU memory footprint by conducting *extract* and *select* in one pass. For LADIES, *Edge-MapReduce fusion* is the major computation optimization, which fuses the *edge-map divide* and *column-wise reduce* operators. The gain is limited because the two operators act on the sampled subgraph, which is already small.

**Data layout selection** is more effective for LADIES than GraphSAGE because LADIES involves more diverse compute and select operators, which prefer different data formats and thus require to balance operator efficiency and format conversion/compaction overhead. In contrast, gSampler chooses the same formats as the greedy strategy for GraphSAGE because this model only involves two operators (i.e., *A[:,columns]* and *A.individual_sample(K, probs)*), and they both prefer CSC. Moreover, for LADIES, the PP graph observes larger speedup than the PD graph because PP has more nodes than PD, which leads to higher format conversion overhead.

The DGL baseline uses the most efficient graph format for each operator and greedily converts CSC whenever required.

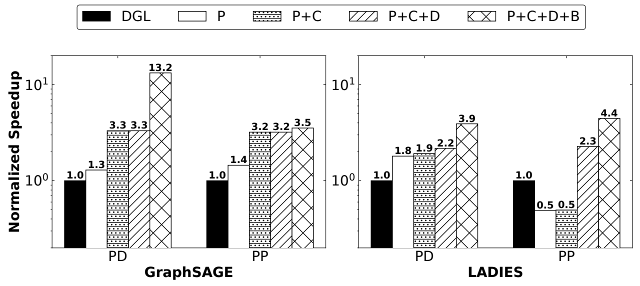


**Figure 10.** Speedups of the optimizations in gSampler on PD and PP graphs, normalized to DGL. **P** denotes plain execution without any optimization, **C** stands for computation optimizations, **D** means data layout optimizations, and **B** is super-batch sampling.

**Table 9.** GPU resource consumption comparison between gSampler and DGL for four complex sampling algorithms in Figure 8 on the PD graph. "Memory" refers to extra GPU memory usage, while "SM" indicates the utilization of streaming multiprocessors.

| Algorithm | System | Memory (GB) | SM (%) |
|---|---|---|---|
| **LADIES** | gSampler | 1.83 | 94.2 |
| | DGL | 0.19 | 37.4 |
| **AS-GCN** | gSampler | 0.07 | 36.0 |
| | DGL | 0.14 | 22.1 |
| **PASS** | gSampler | 0.17 | 56.6 |
| | DGL | 3.04 | 25.3 |
| **ShaDow** | gSampler | 1.65 | 98.0 |
| | DGL | 2.26 | 46.4 |

In contrast, gSampler jointly considers conversion overhead and operator efficiency (i.e., the overall sampling performance) to make format selection/compaction decisions. The case for LADIES on PP in Figure 10 shows that gSampler's cost-aware approach outperforms the greedy counterpart (i.e., the variants without 'D').

**Super-batch sampling.** For GraphSAGE, we observe that super-batch sampling is more effective for the PD graph than the PP because PP is stored in CPU memory and PCIe access becomes the bottleneck. Furthermore, this optimization generally leads to higher improvements for layer-wise sampling (e.g., LADIES) than node-wise sampling (e.g., GraphSAGE) because layer-wise sampling samples fewer neighbors in each layer, leading to lighter workload and lower GPU utilization.

## 5.5 Resource Consumption

Table 9 reports the GPU resource consumption by four complex sampling algorithms on the PD graph. We compare gSampler with DGL. gSampler exhibits higher GPU compute resource utilization, i.e., 1.62-2.52× of DGL, due to its holistic optimizations. This is consistent with the sampling speedups observed in Figure 8. Specifically, for LADIES and ShaDow, gSampler can effectively utilize 94.2% and 98.0% of GPU streaming processors, thanks to the super-batch sampling.

In addition, for most of cases, gSampler significantly reduces the extra GPU memory usage, compared to DGL. For instance, in cases of AS-GCN, PASS, and ShaDow, gSampler's memory usage is only 5.6-73.0% of that of DGL. Among the three algorithms, PASS with gSampler observes the least GPU memory usage, mostly because of the data layout optimization, which eliminates unnecessary formats and isolated nodes with their features. Unlike them, LADIES with gSampler demands an additional 1.64 GB memory than DGL. This is because extra space is required to store intermediate results of super-batch sampling. However, this amount of memory overhead is acceptable as it leads to an 77% increase in GPU compute resource consumption.

## 6 Related Work

**Graph learning.** There are two major classes of graph learning models, i.e., *graph embedding* [3, 14, 31, 60, 61] and *graph neural networks* (GNNs) [15, 22, 23, 42]. Graph embedding learns an embedding vector for each node in the graph and encourages proximal nodes to have similar embedding. To construct training samples, graph embedding uses various random walks, e.g., DeepWalk [31], Node2vec [14], Meta-path [8], and personalized PageRank (PPR) [44]. GNNs stack multiple *graph convolution* layers, where each node aggregates embedding from its neighbors and applies neural network models to obtain its embedding [64]. Many algorithms are designed to construct graph samples for GNN training, e.g., GraphSAGE [15], FastGCN [4], AS-GCN [18], and LADIES [65], which we have discussed in Section 2.

**CPU-based graph sampling.** Due to the importance of graph sampling, several systems are developed for CPU execution. In particular, KnightKing [20] adopts a walker-centric computation model for distributed random walks and uses rejection sampling to improve computation efficiency. FlashMob [55] observes that CPU cache miss in the major bottleneck for random walk and proposes optimizations to improve cache hit. ThunderRW [40] decomposes random walk into *gather-move-update* steps and interleaves the 3 steps to hide cache miss. ForkGraph [26] slices the graph into partitions that fit in CPU cache and shares the cache-resident partition among many random walks for cache efficiency. GraphWalker [48] stores large graphs on SSD and reduces disk access cost by scheduling graph partition loading. These works mainly consider random walks but graph sampling algorithms can be more diverse and complex. Moreover, graph sampling runs significantly slower on CPU than on GPU, and thus CPU-based graph sampling can easily become the bottleneck of graph learning [19].

**GPU-based graph sampling.** To execute graph sampling efficiently on GPU, Nextdoor proposes transit parallelism to share data access among different samples [19]. C-SAW adopts techniques like warp-centric parallelism to avoid thread divergence [30]. Skywalker implements an efficient alias sampling algorithm for GPU [47]. GPU-based graph processing systems, e.g., GunRock [49] and cuGraph [36], can also be utilized to conduct graph graph sampling. These systems adopt a vertex-centric model and do not support tensor operations, which render them unsuitable for graph sampling algorithms that require subgraph operations and complex computation. Moreover, they usually optimize a single operator and lack holistic optimizations from the entire graph sampling process. As graph learning systems, DGL [7] and PyG [33] implement and optimize the graph sampling algorithms case by case using their fine-grained message passing APIs, which takes substantial engineering effort and does not generalize for new algorithms. We propose matrix-centric APIs that is tailored for graph sampling algorithms and allows succinct implementations. We also optimize the entire graph sampling process with designs including intermediate representation, kernel fusion, and super-batch sampling.

## 7 Conclusions and Future Directions

This paper presents gSampler, a framework improves the generality and efficiency of GPU-based graph sampling to support graph learning. gSampler incorporates a set of expressive and user-friendly matrix-centric APIs, and a suite of optimizations for the efficient execution of graph sampling on GPU, which include intermediate representation, kernel fusion, data layout selection, and super-batch sampling. Experimental results demonstrate the easy-to-program and superior performance advantages over state-of-the-art baselines.

Promising future directions include: (1) exploit the skewed access of graph data to design smart caching strategies that improve efficiency for large graphs and (2) jointly utilize multiple GPUs on a machine to conduct graph sampling.

## Acknowledgments

We thank the anonymous reviewers, as well as our shepherd, Thomas Pasquier, for his guidance during the preparation of our camera-ready version. This work is supported in part by the National Natural Science Foundation of China under Grant No.: 62141216, 62172382 and 61832011, the University Synergy Innovation Program of Anhui Province under Grant No.: GXXT-2022-045, and the Guangdong Basic and Applied Basic Research Foundation under Grant No.: 2021A1515110067 and 2023A151 5010244. This work is also supported by our sponsor AWS Shanghai AI Lab.