DSP: Efficient GNN Training with Multiple GPUs

Zhenkun Cai[1], Qihui Zhou[1], Xiao Yan[2], Da Zheng[3], Xiang Song[3], Chenguang Zheng[1], James Cheng[1], George Karypis[3]
[1]Department of Comptuer Sicence and Engineering, The Chinese University of Hong Kong
[2]Department of Computer Science and Engineering, Southern University of Science and Technology
[3]Amazon Web Services
{zkcai,qhzhou,cgzheng,jcheng}@cse.cuhk.edu.hk,yanx@sustech.edu.cn,{dzzhen,xiangsx,gkarypis}@amazon.com

Abstract

Jointly utilizing multiple GPUs to train graph neural networks (GNNs) is crucial for handling large graphs and achieving high efficiency. However, we find that existing systems suffer from *high communication costs* and *low GPU utilization* due to improper data layout and training procedures. Thus, we propose a system dubbed *Distributed Sampling and Pipelining* (DSP) for multi-GPU GNN training. DSP adopts a tailored data layout to utilize the fast NVLink connections among the GPUs, which stores the graph topology and popular node features in GPU memory. For efficient graph sampling with multiple GPUs, we introduce a *collective sampling primitive* (CSP), which pushes the sampling tasks to data to reduce communication. We also design a *producer-consumer-based pipeline*, which allows tasks from different mini-batches to run congruently to improve GPU utilization. We compare DSP with state-of-the-art GNN training frameworks, and the results show that DSP consistently outperforms the baselines under different datasets, GNN models and GPU counts. The speedup of DSP can be up to 26x and is over 2x in most cases.

CCS Concepts: • **Computing methodologies → Machine learning**; **Parallel computing methodologies**.

Keywords: graph neural networks, model training, GPU

1 Introduction

Graph neural networks (GNNs) are a class of models designed specially for graph data and have many variants include GCN [19], GAT [37], GraphSAGE [14], and etc. They achieve outstanding performance for many graph tasks such as node classification [11, 15, 33], node clustering [34] and link prediction [20, 44], and thus are widely used in various domains including e-commerce [32], social networks [23, 40] and bioinformatics [8, 12]. *Sampling-based training* [6, 7, 22, 36], which consumes graph sample (i.e., a sub-graph sampled from the neighborhood of a seed node) in mini-batches to update the GNN model, is widely used due to its high efficiency and good model accuracy.

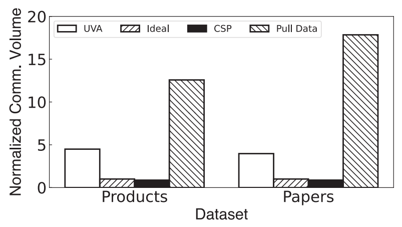


Figure 1. The communication volume of different graph sampling methods when using 8 GPUs (normalized by *Ideal*)

Early systems, such as GraphLearn [47] and Euler [1], train GNNs with CPU, and thus their training speed is limited by the computation power of CPU. Later, PyG [9], DGL [39] and ByteGNN [45] jointly utilize CPU and GPU by conducting graph sampling on CPU and transferring the graph samples to GPU for computation. However, CPU-based graph sampling is slow and the cost of transferring the graph samples via PCIe is high. Nextdoor [16], C-SAW[29] and DGL-GPU (i.e., DGL with GPU sampler feature) conduct sampling entirely on GPU and incorporate kernel optimizations for graph sampling. But they require the graph data to fit in a single GPU and thus cannot handle large graphs. As a GPU server usually has multiple GPUs nowadays and practical graphs can be large [38, 41], scaling GNN training to multiple GPUs is beneficial by jointly utilizing the computation and storage powers of the GPUs. To this end, Quiver [2] and DGL-UVA (i.e., DGL with GPU UVA sampling feature [35]) [39] store the graph topology in CPU memory and cache node features in GPU memory. Each GPU conducts graph sampling independently and accesses the graph topology (and out-of-cache node features) through PCIe using UVA. However, we find that they suffer from two problems that harm efficiency.

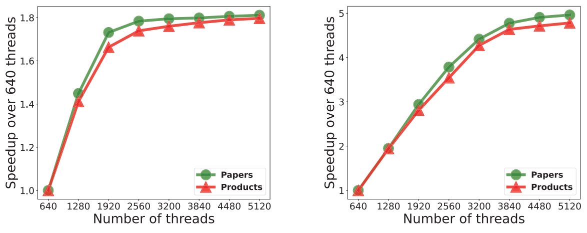
High communication costs. For graph sampling, Quiver and DGL-UVA read the sampled graph nodes from CPU memory via PCIe. This results in *read amplification* (i.e., the amount of read data is larger than requested) because the

*The first two authors contribute equally, Xiao Yan is corresponding author.

PPoPP '23, February 25-March 1, 2023, Montreal, QC, Canada
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0015-6/23/02...$15.00
https://doi.org/10.1145/3572848.3577528

Table 1. Aggregate bandwidth (GBps) of NVLinks and PCIe on a DGX-1 GPU server when different GPUs are used [26]

	1-GPU	2-GPU	4-GPU	8-GPU
PCIe	32	32	64	128
NVLink	0	100	400	1200



(a) Graph sampling **(b)** Feature loading

Figure 2. Execution speed for the graph sampling and feature loading kernels of Quiver when changing physical threads. One GPU with 5120 physical threads is used.

minimum PCIe request size is 50 bytes [24] (32 bytes for payload and 18 bytes for package header). As shown in Figure 1, the communication volume of UVA sampling is much larger than a hypothetical ideal case that only fetches the needed data. To make matters worse, UVA sampling conducts communication on slow PCIe while the NVLink connections among the GPUs are usually much faster according to Table 1. Moreover, Quiver and DGL-UVA only allow GPUs with direct NVLink connections to access each other for cached node features. Given that NVLink bandwidth is much larger than PCIe, reading node features from a remote GPU *without* direct NVLink connection (via multi-hop forwarding) may still be faster than from CPU memory via PCIe.

Low GPU utilization. Quiver and DGL-UVA cannot fully utilize the GPUs as they execute the kernels for different tasks sequentially. This is partially because the kernels for GNN training are lighter than those for ordinary neural networks. For instance, Figure 2 shows that the running time of both graph sampling and feature loading stabilizes before using all physical threads on a GPU. The problem becomes more severe when multiple GPUs are utilized because (i) the workload of each GPU becomes lighter and (ii) the kernels are more likely to be blocked by the irregular memory access on graph data (UVA or global memory access).

By tackling the two problems, we architect the DSP system for efficient sampling-based GNN training with multiple GPUs. To reduce communication costs, DSP adopts a tailored data layout to fully utilize the fast NVLink [28] connections among the GPUs. Specifically, the graph structure is partitioned into well-connected graph patches, and each patch is stored on one GPU. This allows graph sampling to conduct all communication over fast NVLink and avoids the read amplification problem of PCIe. Using the remaining memory, each GPU caches different node features and all GPUs form a large aggregate cache that is shared over NVLink to reduce the access to CPU memory via PCIe for node features.

For each mini-batch of GNN training, DSP uses three tasks, i.e., a *sampler*, a *loader* and a *trainer*. The sampler produces graph samples for a set of seed nodes. The loader loads the node features for the graph samples. The trainer conducts computation on the graph samples to update the model. For the sampler, we propose a *collective sampling primitive* (CSP), where the GPUs jointly conduct sampling on a partitioned graph. CSP pushes sampling tasks on each graph node to its resident GPU instead of pulling the adjacency lists. This *task push* paradigm has significantly lower communication cost than pulling data from remote GPUs (see Figure 1) [1] because usually only a small fraction of a node's neighbors are sampled. CSP also allows to fuse the small sampling kernels on each GPU for efficient execution and is general in expressing different graph sampling methods. To improve GPU utilization, we observe that the sampler, loader and trainer of different mini-batches have no data dependencies, and thus can run concurrently. As such, we design a *training pipeline* based on producer-consumer queues, where the tasks run asynchronously by producing data for the queues or consuming data from the queues. With the pipeline, the communication kernels can cause deadlocks as they have different launching order on the GPUs. We resolve the deadlocks by coordinating the kernel launching order of the GPUs.

We conduct extensive experiments to compare DSP with state-of-the-art GNN training systems including PyG [9], DGL [39], DGL-UVA and Quiver [2]. The results show that DSP consistently outperforms the baselines across different datasets, GNN models and GPU counts. The speedup over the baselines can be an order of magnitude and is over 2x in most cases. DSP also achieves better scalability than the baselines. We also conducted detailed profiling to evaluate the designs of DSP. The results show that CSP is much more efficient than the CPU-based and UVA-based sampling designs in existing works, and the speedup can be up to 20x. The training pipeline effectively improves GPU utilization and can reduce the epoch time by more than 1.5x.

In summary, we made the following contributions.

- We observe that existing systems suffer from high communication cost and low GPU utilization when conducting sampling-based GNN training with multiple GPUs.
- We propose CSP, which is the first to conduct graph sampling jointly with multiple GPUs. CSP is general in expressing different graph sampling schemes and efficient by using fast NVLinks and pushing tasks to data.
- We observe that the tasks of different GNN training mini-batches are independent and design a producer-consumer based pipeline to overlap the tasks for GPU utilization.

[1] CSP uses less communication than *Ideal* because all accesses to graph topology are remote for *Ideal* but CSP has some local accesses on each GPU.

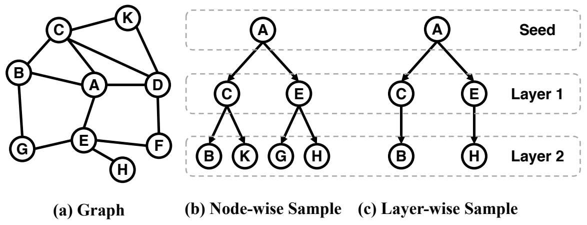


Figure 3. An illustration of graph sampling, the seed node is A, the number of layers is 2, and the fan-out vector is [2,2].

- We tailor the data layout to fully utilize the fast NVLinks and implement the DSP system, which significantly outperforms state-of-the-art systems for GNN training.

2 Sampling-based Training for GNNs

In this part, we introduce background on sampling-based training for GNNs to facilitate further discussion.

Consider a graph $G = (V, E)$, where each node $v \in V$ has a *feature vector* h_v^0 (e.g., author profiles for each author node in a citation network). A K-layer GNN generates an embedding vector h_v^K for each node v by aggregating information from v's neighbors via K layers. In the k^{th} ($k \geq 1$) layer, the GNN model can be expressed as

$$h_v^k = \text{AGGREGATE}^k(h_u^{k-1}, \forall u \in \mathcal{N}(v) \cup v; w^k), \qquad (1)$$

where set $\mathcal{N}(v)$ contains the neighbors of node v in the graph, h_u^{k-1} is the embedding of node u in the $(k-1)^{\text{th}}$ layer, and w^k is the model parameter of the k^{th} layer aggregate function. For the 1st layer, h_u^0 is the feature vector of node u that comes with the input graph. To train the GNN model, the output embedding h_v^K of node v is compared with a ground-truth embedding y_v to compute gradient for the model parameters. By unfolding Eq. (1), it can be observed that computing h_v^K requires the layer-$(K-1)$ embedding of v's 1-hop neighbors, which in turn depend on the layer-$(K-2)$ embedding of v's 2-hop neighbors. Thus, training with node v requires all K-hop neighbors of v. The complexity is high as a node can have many neighbors in real world power-law graphs and the number of involved nodes increase quickly with hops.

For large graphs, *sampling-based training* (called training hereafter for conciseness) is widely used due to its high efficiency and good accuracy [7, 22, 36]. Specifically, training is conducted in mini-batches, and in each mini-batch, gradient is computed using the output embedding of some nodes (called seed nodes for the batch) rather than all nodes in the graph. Moreover, when computing the output embedding for a seed node v, instead of using all of v's K-hop neighbors, a sub-graph is sampled from v's K-hop neighborhood (called graph sample) to reduce complexity. In particular, for a K-layer GNN, the neighbors of the seed node are sampled in the first layer, then neighbors of the first layer samples are sampled in the second layer; the process continues until reaching layer-K, which forms a subgraph of tree structure.

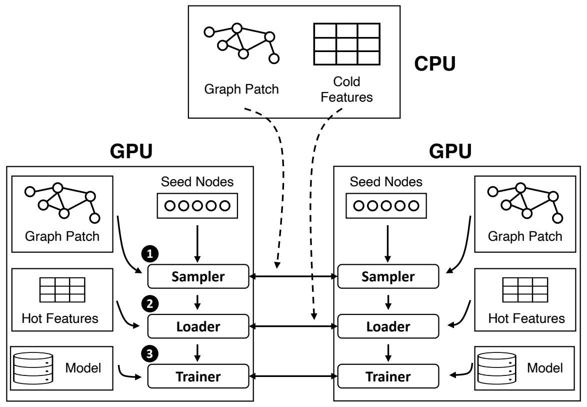


Figure 4. The overall architecture of DSP.

Figure 3 shows examples for two popular graph sampling methods, *node-wise sampling* [14] and *layer-wise sampling* [10], for a 2-layer GNN. The *fan-out vector* controls the number of nodes to be sampled in each layer. Node-wise sampling is conducted independently for each node in the same layer, and a fan-out vector of [2,2] means that for both layers, each node should sample two neighbors. Layer-wise sampling is conducted jointly for all nodes in the same layer, and a fan-out vector of [2,2] means that in both layers, the total number of sampled nodes should be 2. Sampling on a node can be either *unbiased* (i.e., the neighbors are sampled uniformly) or *biased* (i.e., each node has a weight and the neighbors are sampled according to the weight). Note that for a graph sample, the feature vectors of all its nodes are required to compute the output embedding for the seed node.

Feature caching. The node feature vectors usually have a high dimension and may not fit in GPU memory for large graphs. As a result, some node feature vectors need to be stored on CPU memory and fetched to GPU on demand. Existing works observe that some nodes are accessed much more frequently than others during GNN training [18, 21, 25] and propose to cache these hot nodes in GPU memory to reduce feature fetching cost [18, 21, 25]. Several criteria are introduced to select hot nodes, such as large in-degrees, PageRank scores, and reverse PageRank scores.

3 The DSP Framework

In this part, we introduce the data layout and training workflow of DSP. We assume that DSP uses multiple GPUs in the text but note that DSP can also work with a single GPU.

3.1 Data Layout

As shown in Figure 4, DSP partitions the graph topology into patches and stores one patch on each GPU, where each patch contains some nodes and their adjacency lists. We use the graph partitioning algorithms in METIS [17] to minimize the number of cross-patch edges and balance the sizes of the patches. This is because DSP requires each GPU to handle nodes in its own patch, and minimizing edge crossing reduces cross-GPU communication. For the node feature vectors, we cache as many hot nodes as possible in GPU memory and store the other nodes (called cold nodes) in CPU memory and access them using the UVA technique. DSP uses large in-degrees to select hot nodes by default and is compatible with other criteria. Different from Quiver [2] and DGL-UVA [25], which replicate cached feature vectors across different GPU groups, DSP uses a *partitioned cache*, where each GPU caches different feature vectors.

The data layout of DSP considers utilizing fast NVLink and data locality. As shown in Table 1, the NVLink bandwidth among GPUs is much larger than the PCIe bandwidth between GPUs and CPU. (i) By partitioning the graph topology over the GPUs, graph sampling only involves communication over NVLink and does not need to access CPU memory. This is feasible because the graph topology is small even for large graphs (e.g., a graph with one billion edges only takes about 8 GB). Note that DSP can also handle large graph patches by storing the hot nodes in GPU memory and the other nodes in CPU memory (accessed via UVA). (ii) With the partitioned feature cache, more node features are cached in the aggregate memory of all GPUs, which increases the amount of data that can be accessed via NVLink. Empirically, we observe that even the multi-hop NVLink communication across GPU groups (i.e., relayed via an intermediate GPU) is faster than UVA access to CPU memory. (iii) Finally, the graph topology, cached node features and sampling seed nodes are co-partitioned among the GPUs for data locality.

3.2 Training Procedure

DSP maintains the same bulk synchronous parallel (BSP, which means that an iteration should observe all model updates made in its preceding iterations) training semantics as Quiver and DGL-UVA. A data parallel paradigm is utilized, where each GPU works on different graph samples for a mini-batch. As shown in Figure 4, each GPU runs three workers that conduct different tasks, which we detail as follows.

Sampler. In a mini-batch, each GPU loads a batch of seed nodes from the seed nodes assigned to it, and the sampler on each GPU constructs graph samples for its seed nodes by cooperating with samplers on other GPUs. During data preparation, we assign a seed node v to the GPU whose graph patch contains v to reduce communication. When a sampler needs to access graph topology stored on other GPUs, it notifies samplers on these GPUs rather than pulling the data to reduce communication. This is achieved by our collective sampling primitive (CSP), which will be introduce in Section 4. For the example in Figure 3, given seed node A, a sampler will construct the graph sample in Figure 3(b).

Loader. For a mini-batch, the loader on each GPU fetches the node feature vectors for the graph samples handled by the GPU. For instance, the graph sample in Figure 3(b) requires the feature vectors of node A, B, C, E, G, H and K. Before feature loading, the loader collects the requested nodes of all graph samples to avoid repetitive loading. For hot feature vectors cached in GPU memory, the loaders use a collective all-to-all communication over NVLink. For cold feature vectors stored in CPU memory, the loaders fetch them using the UVA technique. We parallelize the loading for hot and cold feature vectors as they use different communication links.

Trainer. For a mini-batch, the trainer on each GPU takes the graph samples (along with the features) as input, computes output embeddings for the seed nodes in the forward pass, and calculates gradients of the model parameters in the backward pass. All trainers hold a copy of the model parameters and use collective allreduce communication to aggregate gradients for model update in a synchronous manner. As GNN models are small, gradient communication is usually much cheaper than graph sampling and feature loading.

The sampler, loader and trainer tasks for the same mini-batch are executed sequentially due to their data dependencies. However, we observe that the GPU utilization is low when running one task each time. To improve GPU utilization, a pipeline is designed to parallelize tasks from different mini-batches, which we detail in Section 5.

To utilize GPUs on multiple machines, DSP replicates the graph topology and hot features across the machines and partitions the cold features among the machines. Thus, the machines only communicate for cold features and model synchronization. When running on a single GPU, the cross-GPU communications (e.g., for sampling and feature loading) become local memory access. DSP conducts inter-GPU communication with NCCL while the NVSHMEM library [3] may be more efficient. We make this choice because NVSHMEM can only handle GPUs with direct NVLink connections while some GPU servers do not have a NVLink mesh. Moreover, the key designs of DSP (e.g., the data layout, CSP and training pipeline) are orthogonal to the communication library and can also be integrated with NVSHMEM.

4 Collective Sampling Primitive

In this part, we present our CSP, which coordinates multiple GPUs to construct graph samples on a graph topology partitioned over the GPUs. We first assume that node-wise sampling is used in the description and then show how CSP can express other sampling methods.

4.1 Working Procedure

CSP conducts graph sampling layer by layer. A layer takes as input some *frontier nodes* on each GPU, whose neighbors need to be sampled; the output are the sampled neighbors for the frontier nodes, which in turn become frontier nodes for the next layer. Take the graph sample in Figure 3(b) for example, in the first layer, the frontier node is the seed node

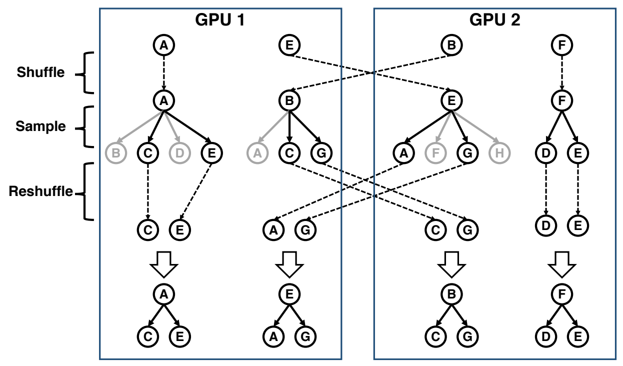


Figure 5. An example of conducting one layer of CSP in three stages, i.e., shuffle, sample, and reshuffle. GPU 1 has frontier nodes *A* and *E*, and holds adjacency list for node *A* and *B*; GPU 2 has frontier nodes *B* and *E*, and holds adjacency list for node *E* and *F*; each frontier node samples 2 neighbors.

A, the number of neighbors to sample is 2, and the outputs are *C* and *E*; for the second layer, the frontier nodes are *C* and *E*, both nodes need to sample 2 neighbors, and the outputs are *B* and *K* for node *C*, *G* and *H* for node *E*.

For each layer, CSP is conducted by the samplers on all GPUs jointly via three stages, i.e., *shuffle*, *sample* and *reshuffle*. For shuffle, each frontier node is transferred to the GPU holding its adjacency list; for sample, each GPU locally samples the required number of neighbors for the frontier nodes it receives; for reshuffle, the sampled neighbors of each frontier node v is transferred back to the GPU requesting neighbor samples for node v. Figure 5 shows a working example of the three stages with 2 GPUs and 4 frontier nodes. In the shuffle stage, node *E* and *B* are transferred to GPU 2 and GPU 1, respectively, where their adjacency lists are stored. In the sampling stage, each GPU samples 2 neighbors for every frontier node it manages, for example, GPU 2 samples node *A* and *G* for frontier node *E*. In the reshuffle stage, the sampled neighbors are transferred to construct the graph samples, for example, *A* and *G* are transferred from GPU 2 to GPU 1 as neighbor samples for node *E*. For frontier nodes whose adjacency lists are stored locally (e.g., node *A* and *F*), shuffle and reshuffle become local memory access.

CSP is implemented as a synchronous operation by imposing a synchronization barrier at the end of each stage. In particular, each GPU first prepares the data to be sent, then notifies the other GPUs the amount of data they will receive, and finally send the data via NCCL all-to-all [27]. The loaders also uses this procedure to fetch hot node features cached in remote GPU memory. In the sample stage, each GPU executes all sampling tasks it receives in a layer using one single kernel. An alternative is to implement CSP as an asynchronous operation, where each GPU communicates with other GPUs once it finishes a stage and executes each received task individually. This design removes synchronization but is observed to have poor efficiency as the communication and sampling tasks of a single GPU are small.

Table 2. Some of the configurable parameters in CSP

	Type	Description
Seed	Integer array	Set of seed nodes for a GPU
Scheme	String	Neighbor-wise or layer-wise sampling
Layer	Integer	Number of layers to sample
IsBiased	Boolean	Biased or unbiased sampling
FanOut	List of integers	Number of neighbor nodes to sample

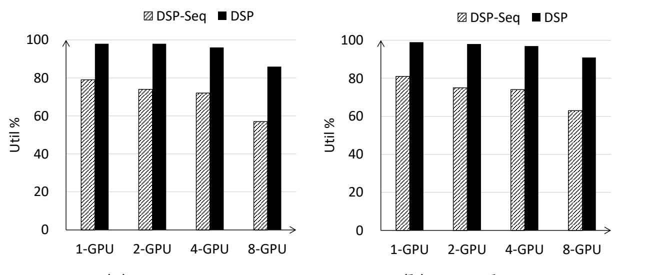
Discussions. Compared with the UVA sampling adopted by Quiver [2] and DGL-UVA [39], CSP has three advantages. First, CSP utilizes the fast NVLink connections among the GPUs to conduct communication rather than the slow PCIe connection between GPU and CPU. Second, UVA suffers from read amplification while NVLink does not. Finally, we assign each seed node to the GPU holding its adjacency list and manage a well-connected graph patch with one GPU, which makes many accesses to the adjacency lists local on each GPU. In contrast, UVA sampling needs CPU-GPU communication for every access to adjacency lists.

CSP uses a *task push* paradigm, where the sampling tasks for a frontier node are forwarded to the GPU holding its adjacency list. An alternative is the *data pull* paradigm, which fetches adjacency lists of frontier nodes from remote GPUs to conducted sampling. The task push paradigm has lower communication costs because usually a small number (e.g., 10) of neighbors are sampled for each frontier node but the adjacency list can contain many neighbors (e.g., 1000).

4.2 Expressiveness of CSP

CSP is general in expressing different graph sampling algorithms, and we list some configurable parameters for CSP in Table 2. For unbiased sampling [14], each neighbor of a frontier node is sampled uniformly at random. For biased sampling [31], each node u in the graph is associated with a non-negative weight w_u, and for a frontier node v with neighbor set $\mathcal{N}(v)$, its neighbor u is sampled with probability $w_u/\sum_{u\in\mathcal{N}(v)} w_u$. DSP stores node weight w_u along with edge $e_{v,u}$ during data preparation such that the GPUs can access node weights locally during sampling. To conduct biased sampling, UVA sampling needs to read the entire adjacency list of a node from CPU memory while CSP only transfers the frontier and sampled nodes by pushing tasks to data.

For node-wise sampling, the fan-out vector directly specifies the number of neighbors CSP should sample for each frontier node. For layer-wise sampling, the fan-out vector only specifies the total number of nodes to sample in each layer, and we determine the number of neighbors to sample for each frontier node as follows. For a graph sample s, denote the set of current frontier nodes as $\mathcal{F}_s$ and assume that



(a) Papers **(b)** Friendster

Figure 6. GPU utilization for sequential execution (DSP-Seq) and the pipeline, experiment settings detailed in Section 7

n nodes need to be sampled, we sample each frontier node u in $\mathcal{F}_s$ with probability

$$p_u = \frac{W_u}{\sum_{u \in \mathcal{F}_s} W_u}, \tag{2}$$

where W_u is the total weight associated with u's neighbors, which is the number of u's neighbors for unbiased sampling. The sampling for current frontier nodes is conducted with replacement for n times, and we use the number of times that u has been sampled as the number of neighbors to sample for u in CSP. The above procedure is equivalent to fetching the adjacency lists of the frontier nodes and then conducting sampling but has much smaller communication costs. We show that DSP can also conduct layer-wise sampling without replacement [4].

We note that CSP can also conduct *graph random walk*, which starts with a node and randomly moves to one of its neighbors in a recursive manner [13, 31, 43]. Specifically, random walk can be implemented as a special case of node-wise sampling, whose fan-out is 1 for all layers. Termination conditions can be introduced to determine whether a random walk should continue in the shuffle stage, and the reshuffle stage can be removed.

5 Pipeline for GNN Training

In this part, we first show the motivation for overlapping the sampler, loader and trainer tasks, and then introduce the designs of our training pipeline.

Motivation. Initially, we implemented DSP-Seq, which executes different mini-batches sequentially. As reported in Figure 6, the GPU utilization of DSP-Seq is low when multiple GPUs are utilized. The low utilization is caused by two reasons. First, as observed for Quiver and DGL-UVA in Figure 2, some GPU kernels are small and cannot fully utilize GPU resources, especially when many GPUs are used and the workload of each GPU is lightweight. For example, the communication kernels of the sampler only need a small number of threads to fully utilize the NVLink bandwidth. The computation kernels of the trainer are lightweight for the last GNN layers as a small number of nodes are involved. Second, the GPU kernels need to wait for and communicate with each other, causing stalls that leave the GPU idle. For example, both sampler and trainer need to wait for their peers to finish on other GPUs while loader needs to communicate with peers for data.

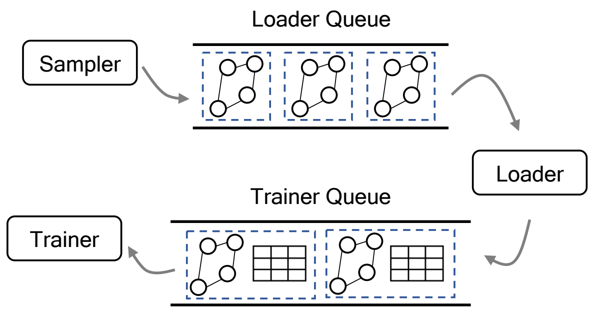


Figure 7. The workflow of the GNN training pipeline.

Queue-based pipeline. The sampler, loader and trainer tasks of each mini-batch must be executed sequentially due to their data dependencies. Thus, to improve GPU utilization, the opportunity is to overlap the tasks of different mini-batches. We observe that the samplers and loaders of different mini-batches have no data dependencies as they work on different graph samples. Thus, we use the producer-consumer-based training pipeline in Figure 7, where the sampler, loader and trainer put data into the queues and/or consume data from the queues asynchronously in the granularity of a mini-batch. Specifically, sampler puts the graph samples into the loader queue; loader consumes graph samples in the loader queue and puts the graph samples along with their required feature vectors into the trainer queue; trainer consumes the graph samples in the trainer queue one mini-batch at a time to compute gradient.

With the pipeline, the sampler, loader and trainer tasks can run concurrently by working for different mini-batches. For example, at a given time, the sampler can construct graph samples for mini-batch $t + 2$, the loader can load features for mini-batch $t + 1$, while the trainer is conducting computation for mini-batch t. We control the progress of the tasks by configuring a capacity limit for each queue, and tasks that run too fast are blocked when they cannot put results into the queues. Empirically, we find that setting the queue capacity limit to 2 is sufficient for overlapping the tasks. Figure 6 shows that our pipeline effectively improves GPU utilization, especially when the number of GPU is large.

Our pipeline uses a single worker instance for each task. An alternative is to use multiple workers for both sampler and loader and assign each worker to work on graph samples for different mini-batches. Note that we cannot use multiple workers for trainer as this violates the semantics of BSP training. The multi-instance design is not used for two reasons. First, it consumes more memory for in-flight works and thus leaves less GPU memory to cache graph topology and node features. Second, with more workers on each GPU, the resource contention for both CPU and GPU is more severe.

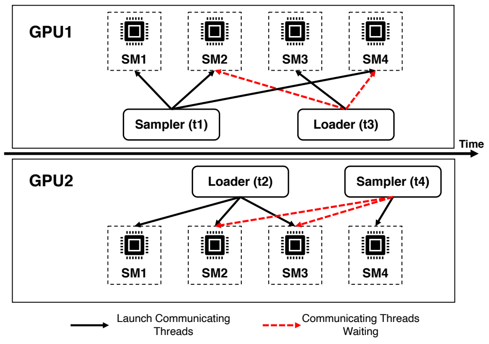


Figure 8. An illustration of communication deadlock.

Empirically, we find that using multiple samplers and loaders degrades overall performance.

Centralized communication coordination. The workers for sampler, loader and trainer on different GPUs communicate with their peers via NVLink all-to-all communication. However, deadlocks can happen when we run the workers concurrently because of two properties. (i) The resource allocation to GPU kernels is irrevocable, and thus a kernel can only release resource when it finishes. (ii) All-to-all communication can only proceed when the communication kernels of the peer workers are started on all GPUs.

We show such an example in Figure 8, where GPU 1 starts communication kernel for sampler first but GPU 2 starts communication for loader first. The kernels can proceed if GPU 1 also starts communication for loader or GPU 2 also starts communication for sampler. However, this may not be possible due to resource contention. For example, on GPU 1, the communication kernel for loader may request a streaming multiprocessor (SM) that is already acquired by the sampler, but the sampler cannot release the SM as it is waiting for the sampler on GPU 2. For similar reasons, the device synchronization functions in CUDA (e.g., cudaMemcpy, cudaMalloc and cudaFree) can also cause deadlocks because they block subsequent communications and can only start when all their preceding communications finish.

The communication deadlock boils down to the fact that the communication kernels can have different launching order on different GPUs. Thus, we use a centralized communication coordination (CCC) scheme, which assigns one GPU (call leader) to decide a unified launching order of the communication kernels for all GPUs. Specifically, we assign an id to each worker, and ensure that peer workers (e.g., samplers for the same mini-batch) have the same id on different GPUs. On each GPU, a worker registers its id in a pending set when it is ready for communication. The leader GPU uses a queue to manage its ready communication kernels and starts the kernels in their submission order. Once the leader starts the communication kernel for a worker, it broadcasts the id of the worker to all GPUs. On receiving the worker id, a follower GPU starts communication kernel for the worker if the worker is in the pending set and waits for the worker to become ready otherwise. Note that CCC only controls the launching order of the kernels and the communication kernels for different workers can still run in parallel.

6 Implementation

DSP's implementation is based on DGL [39] and Pytorch [30]. Specifically, we use the GPU graph sampling kernels of DGL to conduct sampling locally for each graph patch in the sample stage of CSP. The data flow graph and auto-gradient functionalities of Pytorch are used in the trainer for gradient computation. We adopt the NCCL [27] library for communication among the GPUs via NVLink.

The graph patch on each GPU is stored using the compressed sparse row (CSR) format, where a node records its in-neighbors in the adjacency list to facilitate graph sampling. DSP assigns a global id and a local id for each node. The global of a node is used to lookup the GPU holding its adjacency list. We renumber the nodes to ensure that nodes in each graph patch have consecutive global ids, which makes the lookup a simple range check. The local id is used to find a node's adjacency list in the CSR on each GPU, and global id is converted to local id when a GPU receives frontier nodes for sampling tasks. We store global ids in the adjacency list as this avoids the conversion from local id to global id for the sampled nodes (because they will be used as frontier nodes or to conduct feature fetching).

Each GPU holds an adjacency position list and a feature position list, which indicates where the adjacency list and feature vector are stored (i.e., in GPU or CPU) for each node in its graph patch. This allows DSP to handle large graphs whose topology does not fit in GPU memory and enables flexible adjustment of the GPU memory used for caching graph topology and feature vectors. As DSP uses CSP for sampling, accesses to the adjacency list of a node v are always conducted by the GPU managing v's graph patch. Thus, the GPUs find the position of the required adjacency list via a local lookup to the adjacency position list. For feature loading, DSP first uses an all-to-all operation over NVLink to fetch the positions of the feature vectors managed by remote GPUs. If a feature vector is cached in GPU, it is obtained via an all-to-all operation over NVLink; otherwise, the feature vector is loaded from CPU memory by the requesting GPU.

7 Experimental Evaluation

In this part, we conduct extensive experiments to evaluate the performance of DSP. We first introduce the experiment settings in Section 7.1, then compare DSP with state-of-the-art GNN training systems in Section 7.2, and finally test the our designs (including data layout, CSP and the training pipeline) in Section 7.3.

Table 3. Statistics of the graph datasets in the experiments

	Products	Papers	Friendster
Vertices	2M	111M	66M
Edges	123M	3.2B	3.6B
Average degree	50.5	28.8	54.5
Feature dimension	100	128	256
Topology size (GB)	0.984	25.6	28.8
Feature size (GB)	0.8	56.8	67.6

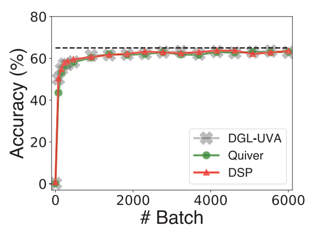
7.1 Experiment Settings

We conduct experiments on three graph datasets that are widely used for GNN evaluation, and their statistics are summarized in Table 3. *Products* [5] is a product co-purchasing network from Amazon, where nodes are products and an edge indicates that two products are purchased together. *Papers* [38] is a citation graph, where each node is an paper and an edge means that one paper cites another paper. *Friendster* [41] is a gaming network, where each node is a user and there is an edge between two users that are friends.

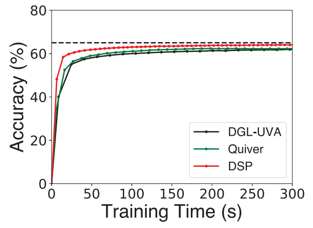
We compare DSP with four state-of-the-art GNN training systems, i.e., PyG (v2.0), DGL-CPU (v0.8), DGL-UVA (v0.8) and Quiver (v0.1). PyG and DGL-CPU store both the graph structure and node features in CPU memory, conduct graph sampling using CPU, and send graph samples to the GPUs to conduct training. DGL-UVA and Quiver store the graph structure in CPU memory and use the UVA technique to access the graph topology during sampling. The difference is that DGL-UVA stores all node features on CPU while Quiver caches hot node features in GPU. We note that DGL-UVA allows feature caching but requires all node features to fit in the memory of a single GPU. As both the node features of Papers and Friendster cannot fit in our experiment GPU, we disable feature caching for DGL-UVA in the experiments. We do not compare with GNNLab [42] as it requires the graph topology to fit in one GPU and conducts asynchronous training for efficiency.

Following existing works [25, 46], we train a 3-layer GraphSAGE model with a hidden size of 256 for all layers and a batch size of 1024, and conduct neighbor-wise sampling with a fan-out of [15, 10, 5] by default. The experiments are conducted on a server equipped with 8 V100 GPUs each having 16 GB memory and a Intel Xeon E5-2686 CPU having 64 cores and 480 GB main memory. The inter-device communication topology of the server is similar to DGX-1 [26], where the GPUs are connected via NVLink while the CPU and the GPUs are connected via PCIe. We keep three significant figures when reporting the results.

Correctness. DSP executes the same BSP training logic as the baseline systems, and thus when the same number of training epochs (or mini-batches) are executed, DSP has the same training progress (i.e., training loss or test accuracy) as other systems. For a sanity check, we plot the training progress of DSP, DGL-UVA, and Quiver in Figure 9. We omit PyG in this experiment because Quiver adopts the same training backend as PyG. Figure 9a shows that the accuracy v.s. mini-batch count curve of DSP overlaps with DGL-UVA and Quiver, which indicates that our implementations are correct. However, because DSP has shorter mini-batch time than DGL-UVA and Quiver, DSP achieves higher accuracy when running for the same amount of time in Figure 9b. Specifically, DSP, DGL-UVA, and Quiver take 183s, 698s and 629s, respectively, to train the model to convergence. In the subsequent experiments, we use the *epoch time* (i.e., the time to go over one pass of all seed nodes) as the main performance metric as short epoch time directly translates to short end-to-end training time.



(a) Accuracy v.s. batch count



(b) Accuracy v.s. training time

Figure 9. Training quality for GraphSAGE on the Papers graph with 8 GPUs. Dashed line denotes converged accuracy.

7.2 Main Results

We report the epoch time of the systems in Table 4, which lead to several observations. First, DSP consistently outperforms the baselines for different datasets and GPU counts. Compared with PyG and DGL-CPU, the speedup of DSP can be more than 10x; considering the fastest baseline under each configuration, DSP achieves more than 2x speedup in most cases. Second, DSP usually has larger performance advantage over the baselines when using more GPUs, indicating that DSP achieves good scalability, which is crucial when running on multi-GPU servers. In fact, DSP even achieves super-liner speedup in some cases. For example, on Papers, the speedup is 3.31x when increasing from 4 GPUs to 8 GPUs. This is because DSP stores graph topology and caches node features on GPU, and thus when the number of GPUs increases, DSP benefits not only from more computation power but also cheaper communication.

It can also be observed that PyG and DGL-CPU, have poor scalability. This is because they use the CPU to conduct graph sampling, and the GPUs contend for limited CPU threads, which makes CPU the bottleneck for training. Although Quiver caches node features on GPU and DGL-UVA does not, Quiver is slower than DGL-UVA in most cases. We find that this is because Quiver uses the GPU memory

Table 4. The training time of each epoch (in seconds), best in bold and second best in italic

Systems	Products				Papers				Friendster			
	1-GPU	2-GPU	4-GPU	8-GPU	1-GPU	2-GPU	4-GPU	8-GPU	1-GPU	2-GPU	4-GPU	8-GPU
PyG	28.8	20.4	17.1	16.1	131	89.0	68.3	49.2	1110	828	575	477
DGL-CPU	14.7	9.29	6.43	5.45	111	76.0	62.3	45.1	1080	781	537	470
Quiver	*5.71*	*4.06*	*2.82*	2.51	70.9	42.3	*23.8*	*17.2*	449	*249*	*145*	118
DGL-UVA	6.87	6.03	3.17	*1.61*	*47.5*	*39.6*	30.2	18.3	*432*	410	207	*107*
DSP	**3.11**	**1.75**	**0.992**	**0.613**	**39.1**	**24.5**	**15.3**	**4.62**	**270**	**116**	**64.6**	**44.8**

Table 5. Epoch time for training GCN (in seconds) using 8 GPUs, best in bold and second best in italic

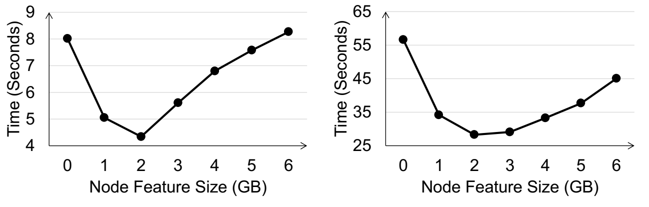
Dataset	Products	Papers	Friendster
PyG	15.5	41.4	501
DGL-CPU	8.32	48.7	478
Quiver	*3.97*	23.7	172
DGL-UVA	4.91	*13.6*	*137*
DSP	**0.552**	**5.97**	**29.9**

allocation interfaces in CUDA for sampling, e.g., cudaMalloc and cudaFree, which have large overheads. In contrast, DGL-UVA adopts more efficient memory management interfaces in Pytorch, which are also used by DSP. However, the scalability of DGL-UVA is very poor when increasing from 1 GPU to 2 GPUs. This because the 2 GPUs are on the same PCIe switch and contend for bandwidth when accessing CPU memory. Similar issues also exist for Quiver and DSP but DGL-UVA suffers the most as it accesses all data via PCIe. This phenomenon shows the importance of using NVLink as it is not only faster than PCIe and but also helps to alleviate contention by offloading communication.

To show the generality of DSP, we experiment with another popular GNN model, GCN [19]. Table 5 reports the epoch time of the systems. The results show that DSP consistently outperforms the baselines by a large margin. Compared with the case for GraphSAGE, the speedup of DSP is generally larger for GCN. For example, on the Frindster graph, the speedup of DSP over DGL-UVA is 2.39x for GraphSAGE but 4.58x for GCN. This is because computation is more lightweight for GCN than GraphSAGE, and thus the communication efficient designs of DSP yield larger benefits.

7.3 Design Choices and Insights

Caching strategy. Quiver caches only node features in GPU memory while our DSP caches both graph topology and node features. To compare the two strategies, we conduct training



(a) Papers **(b)** Friendster

Figure 10. Epoch time when varying the cache size for node feature, experimented conducting using 8 GPUs

on the two larger graphs with 8 GPUs and limit the total cache size to 6 GB on each GPU. Figure 10 reports the epoch time of DSP under different cache allocations among the graph topology and node features. The results show that the epoch time first decreases but then increases when increasing the memory used to cache node features. This can be explained by two reasons. First, as observed by related works [2, 21, 25], accesses to node features are dominated by a small portion of popular nodes, and thus the gain of increasing node cache diminishes with cache size. Second, by caching the graph topology, sampling avoids read amplification and enjoys faster communication over NVLink. Moreover, the shortest epoch time is achieved when using 2GB feature cache for both graphs, which means that the entire graph topology is stored in GPU memory (4GB/GPU×8GPU=32GB exceeds the graph sizes in Table 3). This suggests that it is reasonable to prioritize caching graph topology as in DSP.

Graph sampling. DSP uses CSP to conduct graph sampling while Quiver and DGL-GPU stores graph structure on CPU and use the UVA technique to conduct sampling. Table 6 compares the sampling time of the baselines systems in each epoch. For a fair comparison, the results of DSP are obtained when running the sampler individually without interference from other workers. The results show that DSP consistently outperforms the baselines and the speedup can be more than 20x. Similar to the case of epoch time, the CPU-based solutions (i.e., PyG and DGL-CPU) are slow and have poor scalability to the limited computation power of CPU. Moreover, DSP achieves super-linear speedup when increasing

Table 6. Sampling time of each epoch (in seconds), best in bold and second best in italic

Systems	Products				Papers				Friendster			
	1-GPU	2-GPU	4-GPU	8-GPU	1-GPU	2-GPU	4-GPU	8-GPU	1-GPU	2-GPU	4-GPU	8-GPU
PyG	5.03	4.41	4.26	4.21	30.0	31.0	35.0	29.1	134	140	145	152
DGL-CPU	4.96	3.89	2.86	2.57	30.3	21.8	19.4	16.1	189	176	141	137
Quiver	3.72	2.94	2.19	1.98	24.1	18.1	15.1	11.3	108	78.9	54.4	41.2
DGL-UVA	*2.39*	*1.97*	*1.12*	*0.613*	*14.2*	*11.5*	*4.91*	*2.61*	*95.3*	*71.2*	*30.0*	*15.2*
DSP	**1.60**	**0.834**	**0.461**	**0.323**	**12.1**	**6.91**	**2.47**	**1.40**	**61.3**	**33.2**	**13.4**	**7.09**

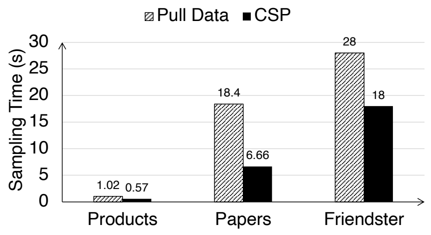


Figure 11. Comparison between CSP and Pull Data

from 2 GPUs to 4 GPUs for Papers and Friendster. This is because the two graphs can be stored entirely in GPU memory with 4 GPUs and thus graph sampling does not need to access CPU memory via PCIe.

CSP uses a task push paradigm that pushes sampling tasks to graph data. To conduct sampling on a graph partitioned over multiple GPUs, an alternative is to pull adjacency list from remote GPUs when necessary (i.e., *Pull Data*). Note that pulling the entire adjacency list is necessary for cases such as biased sampling and temporal graph sampling. In Figure 11, we compare the sampling time of CSP and *Pull Data* for biased sampling with 4 GPUs. Note that Table 6 reports the results for *unbiased sampling* and thus is different from Figure 11. The results show that CSP consistently outperforms *Pull Data* and reduces its sampling time by at most 64%. This is because *Pull Data* pulls adjacency and weight lists for each frontier node while CSP only needs to transfer frontier nodes and sampled nodes. As adjacency and weight lists are usually significantly larger than the sampled nodes, CSP transfers much less data than *Pull Data*, which we have also shown in Figure 1.

We report the performance of DSP for layer-wise sampling in Table 7. As existing systems do not support layer-wise sampling on GPU, we compare with the TensorFlow implementation of FastGCN [6] on CPU. The batch size is 1024, the fan-out is 1000 for both layers, and DSP uses all 8 GPUs on a machine. Although the comparison is not apple-to-apple, the results show that DSP is also efficient for layer-wise sampling and runs much faster than CPU implementation.

Training pipeline. We measure the epoch time of DSP and DSP-Seq, which executes the sampler, loader and trainer

Table 7. Sampling time (in seconds) in an epoch for lay-wise sampling without replacement

Dataset	Products	Papers	Friendster
FastGCN	37.5	489	252000
DSP	0.12	8.96	52.8

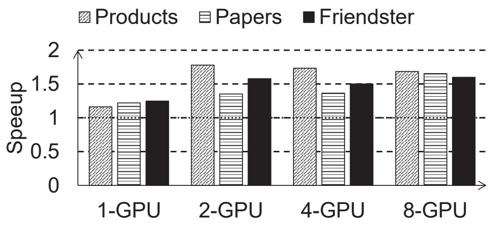


Figure 12. The speedup of DSP over DSP-Seq in epoch time

sequentially, and report the speedup of DSP over DSP-Seq in Figure 12. The results show that DSP consistently outperforms DSP-Seq for all datasets and GPU counts, which is in line with the GPU utilization results in Figure 6. Figure 12 also shows that the performance gain of our pipeline is modest with 1 GPU but becomes more significant when using more GPUs. For example, with 8 GPUs, the speedup of DSP over DSP-Seq is over 1.5x for all three datasets. This is because when using more GPUs, the kernels become lighter and communication costs increase, and thus there is more opportunity to parallelize the tasks.

8 Conclusions

In this paper, we present DSP, a system that tackles the high communication costs and low GPU utilization problems of existing systems when using multiple GPUs for GNN training. For high communication efficiency, DSP adopts a tailored data layout and proposes a collective primitive for graph sampling, which are designed to utilize the fast NVLink connections among the GPUs. To improve GPU utilization, DSP overlaps the tasks of different mini-batches with a producer-consumer style pipeline. Extensive experiments show that DSP significantly outperforms state-of-the-art GNN training systems in efficiency and our designs are effective.

References

[1] 2020. Euler. https://github.com/alibaba/euler.

[2] 2021. Quiver. https://github.com/quiver-team/torch-quiver.

[3] 2022. NVSHMEM. https://developer.nvidia.com/nvshmem.

[4] anonymous. 2022. Procedure for sampling without replacement. https://anonymous.4open.science/r/Anonymous-593E/CSP-layerwise-proof.pdf. [Online; accessed January-2023].

[5] K. Bhatia, K. Dahiya, H. Jain, P. Kar, A. Mittal, Y. Prabhu, and M. Varma. 2016. The extreme classification repository: Multi-label datasets and code. http://manikvarma.org/downloads/XC/XMLRepository.html

[6] Jie Chen, Tengfei Ma, and Cao Xiao. 2018. FastGCN: Fast Learning with Graph Convolutional Networks via Importance Sampling. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net.

[7] Jianfei Chen, Jun Zhu, and Le Song. 2018. Stochastic Training of Graph Convolutional Networks with Variance Reduction. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*. 941–949.

[8] Mark Cheung and José M. F. Moura. 2020. Graph Neural Networks for COVID-19 Drug Discovery. In *2020 IEEE International Conference on Big Data (IEEE BigData 2020), Atlanta, GA, USA, December 10-13, 2020*. 5646–5648.

[9] Matthias Fey and Jan E. Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*.

[10] Hongyang Gao, Zhengyang Wang, and Shuiwang Ji. 2018. Large-Scale Learnable Graph Convolutional Networks. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2018, London, UK, August 19-23, 2018*. ACM, 1416–1424.

[11] Alberto García-Durán and Mathias Niepert. 2017. Learning Graph Representations with Embedding Propagation. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*. 5119–5130.

[12] Thomas Gaudelet, Ben Day, Arian R. Jamasb, Jyothish Soman, Cristian Regep, Gertrude Liu, Jeremy B. R. Hayter, Richard Vickers, Charles Roberts, Jian Tang, David Roblin, Tom L. Blundell, Michael M. Bronstein, and Jake P. Taylor-King. 2020. Utilising Graph Machine Learning within Drug Discovery and Development. *CoRR* abs/2012.05716 (2020).

[13] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable Feature Learning for Networks. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*. ACM, 855–864.

[14] William L. Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*. 1024–1034.

[15] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open Graph Benchmark: Datasets for Machine Learning on Graphs. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

[16] Abhinav Jangda, Sandeep Polisetty, Arjun Guha, and Marco Serafini. 2021. Accelerating graph sampling for graph machine learning using GPUs. In *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*. 311–326.

[17] George Karypis and Vipin Kumar. 1998. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.* 20, 1 (1998), 359–392.

[18] Taehyun Kim, Changho Hwang, KyoungSoo Park, Zhiqi Lin, Peng Cheng, Youshan Miao, Lingxiao Ma, and Yongqiang Xiong. 2021. Accelerating GNN training with locality-aware partial execution. In *APSys '21: 12th ACM SIGOPS Asia-Pacific Workshop on Systems, Hong Kong, China, August 24-25, 2021*. 34–41.

[19] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net.

[20] Jérôme Kunegis and Andreas Lommatzsch. 2009. Learning spectral graph transformations for link prediction. In *Proceedings of the 26th Annual International Conference on Machine Learning, ICML 2009, Montreal, Quebec, Canada, June 14-18, 2009*. 561–568.

[21] Zhiqi Lin, Cheng Li, Youshan Miao, Yunxin Liu, and Yinlong Xu. 2020. PaGraph: Scaling GNN training on large graphs via computation-aware caching. In *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020*. 401–415.

[22] Xin Liu, Mingyu Yan, Lei Deng, Guoqi Li, Xiaochun Ye, and Dongrui Fan. 2022. Sampling Methods for Efficient Training of Graph Convolutional Networks: A Survey. *IEEE CAA J. Autom. Sinica* 9, 2 (2022), 205–234.

[23] Supriyo Mandal and Abyayananda Maiti. 2021. Graph Neural Networks for Heterogeneous Trust based Social Recommendation. In *International Joint Conference on Neural Networks, IJCNN 2021, Shenzhen, China, July 18-22, 2021*. IEEE, 1–8.

[24] Seungwon Min, Vikram Sharma Mailthody, Zaid Qureshi, Jinjun Xiong, Eiman Ebrahimi, and Wen-Mei Hwu. 2020. EMOGI: Efficient Memory-access for Out-of-memory Graph-traversal In GPUs. *Proc. VLDB Endow.* 14, 2 (2020), 114–127.

[25] Seungwon Min, Kun Wu, Mert Hidayetoglu, Jinjun Xiong, Xiang Song, and Wen-mei W. Hwu. 2021. Graph Neural Network Training with Data Tiering. *CoRR* abs/2111.05894 (2021). arXiv:2111.05894 https://arxiv.org/abs/2111.05894

[26] NVIDIA. 2022. DGX Systems. https://www.nvidia.com/en-sg/data-center/dgx-systems. [Online; accessed March-2022].

[27] NVIDIA. 2022. NVIDIA Collective communications library (NCCL). https://https://developer.nvidia.com/nccl. [Online; accessed March-2022].

[28] NVIDIA. 2022. NVLink and NVSwitch. https://www.nvidia.com/en-sg/data-center/nvlink. [Online; accessed March-2022].

[29] Santosh Pandey, Lingda Li, Adolfy Hoisie, Xiaoye S. Li, and Hang Liu. 2020. C-SAW: a framework for graph sampling and random walk on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*. IEEE/ACM, 56.

[30] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*. 8024–8035.

[31] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. DeepWalk: online learning of social representations. In *The 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '14, New York, NY, USA - August 24 - 27, 2014*. ACM, 701–710.

[32] Xiaoru Qu, Zhao Li, Jialin Wang, Zhipeng Zhang, Pengcheng Zou, Junxiao Jiang, Jiaming Huang, Rong Xiao, Ji Zhang, and Jun Gao. 2020. Category-aware Graph Neural Networks for Improving E-commerce Review Helpfulness Prediction. In *CIKM '20: The 29th ACM International Conference on Information and Knowledge Management, Virtual*

Event, Ireland, October 19-23, 2020. 2693–2700.

[33] Kaspar Riesen and Horst Bunke. 2010. *Graph Classification and Clustering Based on Vector Space Embedding*. Series in Machine Perception and Artificial Intelligence, Vol. 77. WorldScientific.

[34] Kaspar Riesen and Horst Bunke. 2010. *Graph Classification and Clustering Based on Vector Space Embedding*. Series in Machine Perception and Artificial Intelligence, Vol. 77. WorldScientific.

[35] Tim C Schroeder. 2011. Peer-to-peer & unified virtual addressing. In *GPU Technology Conference, NVIDIA*.

[36] Marco Serafini. 2021. Scalable Graph Neural Network Training: The Case for Sampling. *ACM SIGOPS Oper. Syst. Rev.* 55, 1 (2021), 68–76.

[37] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net.

[38] Kuansan Wang, Zhihong Shen, Chiyuan Huang, Chieh-Han Wu, Yuxiao Dong, and Anshul Kanakia. 2020. Microsoft Academic Graph: When experts are not enough. *Quant. Sci. Stud.* 1, 1 (2020), 396–413.

[39] Minjie Wang, Lingfan Yu, Da Zheng, Quan Gan, Yu Gai, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Chao Ma, Ziyue Huang, Qipeng Guo, Hao Zhang, Haibin Lin, Junbo Zhao, Jinyang Li, Alexander J. Smola, and Zheng Zhang. 2019. Deep Graph Library: Towards Efficient and Scalable Deep Learning on Graphs. *CoRR* abs/1909.01315 (2019). arXiv:1909.01315 http://arxiv.org/abs/1909.01315

[40] Zhouxia Wang, Tianshui Chen, Jimmy S. J. Ren, Weihao Yu, Hui Cheng, and Liang Lin. 2018. Deep Reasoning with Knowledge Graph for Social Relationship Understanding. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*. 1021–1028.

[41] Jaewon Yang and Jure Leskovec. 2012. 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*. 745–754.

[42] Jianbang Yang, Dahai Tang, Xiaoniu Song, Lei Wang, Qiang Yin, Rong Chen, Wenyuan Yu, and Jingren Zhou. 2022. GNNLab: a factored system for sample-based GNN training over GPUs. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 417–434.

[43] Ke Yang, Mingxing Zhang, Kang Chen, Xiaosong Ma, Yang Bai, and Yong Jiang. 2019. KnightKing: a fast distributed graph random walk engine. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*. ACM, 524–537.

[44] Muhan Zhang and Yixin Chen. 2018. Link Prediction Based on Graph Neural Networks. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*. 5171–5181.

[45] Chenguang Zheng, Hongzhi Chen, Yuxuan Cheng, Zhezheng Song, Yifan Wu, Changji Li, James Cheng, Hao Yang, and Shuai Zhang. 2022. ByteGNN: efficient graph neural network training at large scale. *Proceedings of the VLDB Endowment* 15, 6 (2022), 1228–1242.

[46] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. 2020. DistDGL: Distributed Graph Neural Network Training for Billion-Scale Graphs. In *10th IEEE/ACM Workshop on Irregular Applications: Architectures and Algorithms, IA3 2020, Atlanta, GA, USA, November 11, 2020*. IEEE, 36–44.

[47] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. 2019. AliGraph: A Comprehensive Graph Neural Network Platform. *Proc. VLDB Endow.* 12, 12 (2019), 2094–2105.

A Artifact information

A.1 Artifact and experiment requirements

Our artifact is available at https://doi.org/10.5281/zenodo.7463498. There are two types of experiments in this paper. First, we compare the epoch time of GNN training and sampling of DSP with three baseline systems, i.e., PyG, DGL and Quiver. Second, we use micro benchmarks to demonstrate the insights behind our design choices, which include flexible data caching, multi-GPU sampling, and pipeline training.

Hardware. We run all the experiments on an AWS EC2 p3.16xlarge machine, which is equipped with a 64-core dual-socket Intel Xeon E5-2686 v4 CPU, 480 GB host memory, and 8 V100 GPUs with 16 GB memory.

Software. We build DSP on DGL(v0.7). In particular, we modify the sampler and add a loader to DGL, which provides distributed graph sampling and data loading on multiple GPUs. We use the original code base of DGL for model training. Other required software and their version information are listed in Table 8.

Table 8. The required software and their versions

Name	Version
Ubuntu	18.04
CUDA-Toolkit	11.1
NCCL	2.8
Cub	1.10
Metis	5.1
Pytorch	1.9.0
DGL	0.8
PyG	2.0
Quiver	0.1

Dateset. We conduct all our experiments on three publicly available datasets, i.e., *Products*, *Papers* and *Friendster*. Products and Papers can be downloaded from https://ogb.stanford.edu/, and Friendster can be downloaded from https://snap.stanford.edu/data/com-Friendster.html.

A.2 Guidelines to reproduce the experiments

We provide a docker image to reproduce all the experiments, which contains running environments for DSP and the baseline systems. We also provide scripts for preparing the datasets and running the experiments. To reproduce the experiments, we first pre-process the datasets and then run each experiment individually. All scripts are stored under */root/experiments/scripts/*.

Data preprocessing. The dataset preprocessing step uses different scripts for DSP and the baselines.

1. **DSP.** As DSP works on partitioned graph, we partition the graph datasets and store them on disk. We use the script *partition.sh* to download and partition a graph. Users only need to provide the graph name and the number of partitions. For example, *./partition.sh products 4* partitions the Products graph into 4 partitions. The partitioned graph is stored under */data/ds/*, which is used as the default data directory in subsequent experiments.
2. **Baselines.** We use a script *preprocess.sh* to download all datasets and convert them into the formats required by DGL, PyG and Quiver. The processed datasets for the three baseline systems are stored under */data/dgl/*, */data/pyg/*, and */data/quiver/*, respectively.

Running the experiments. We use a unified script to run DSP and the baselines for each experiment. For example, *scripts/table_V.sh* runs the experiments in Table V by enumerating DSP and the baselines, the datasets and the number of GPUs. The profiled time are averaged over 10 epochs after 5 warm-up epochs. For each experiment, a script with documentation is included in the docker image.