Moment: Co-optimizing Physical Communication Topology and Data Placement for Multi-GPU Out-of-core GNN Training

Zuocheng Shi*
Zhejiang University
Hangzhou, China
shizuocheng@zju.edu.cn

Jie Sun*
Zhejiang University
Hangzhou, China
jiesun@zju.edu.cn

Ziyu Song
Zhejiang University
Hangzhou, China
songziyu@zju.edu.cn

Mo Sun
Zhejiang University
Hangzhou, China
sunmo@zju.edu.cn

Yang Xiao
Zhejiang University
Hangzhou, China
12221061@zju.edu.cn

Fei Wu
Zhejiang University
Hangzhou, China
wufei@zju.edu.cn

Zeke Wang
Zhejiang University
Hangzhou, China
wangzeke@zju.edu.cn

Abstract

Graph Neural Networks (GNNs) are widely employed in applications like recommendation systems, social network analysis, and fraud detection, but training large-scale GNNs is challenging due to its memory limitations. Existing systems face a trade-off between throughput and monetary cost: Distributed systems require expensive memory scaling, while single-machine out-of-core systems are limited by GPU/PCIe throughput. To this end, we propose Moment, a physical communication topology and data placement *co-optimizer* to enable high-throughput and low-cost GNN training in a single multi-GPU machine. Moment addresses communication contention and GPU load imbalance issues by modeling the physical topology as capacity-constrained directed graphs and formulating communication scheduling as a max-flow problem. It also introduces a data-distribution-aware knapsack algorithm for optimized data placement. Experimental results show that Moment outperforms out-of-core systems by up to 6.51× and distributed systems by up to 3.02×, with only 50% monetary cost.

CCS Concepts

• **Computing methodologies** → **Machine learning**; **Planning and scheduling**; **Modeling and simulation**.

Keywords

Graph Neural Network, High-performance Machine Learning, Data Placement, Scheduling and Resource Management

*Equal contribution.

SC '25, St Louis, MO, USA

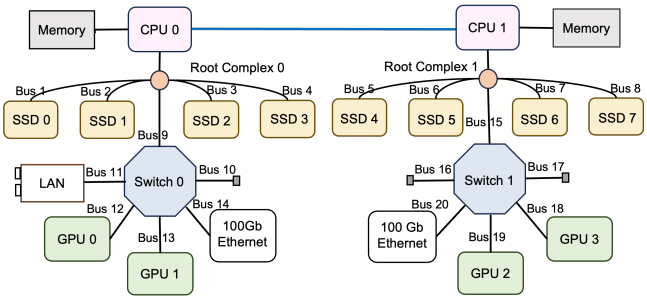
ACM ISBN 979-8-4007-1466-5/25/11
https://doi.org/10.1145/3712285.3759788

ACM Reference Format:
Zuocheng Shi, Jie Sun, Ziyu Song, Mo Sun, Yang Xiao, Fei Wu, and Zeke Wang. 2025. Moment: Co-optimizing Physical Communication Topology and Data Placement for Multi-GPU Out-of-core GNN Training. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '25), November 16–21, 2025, St Louis, MO, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3712285.3759788

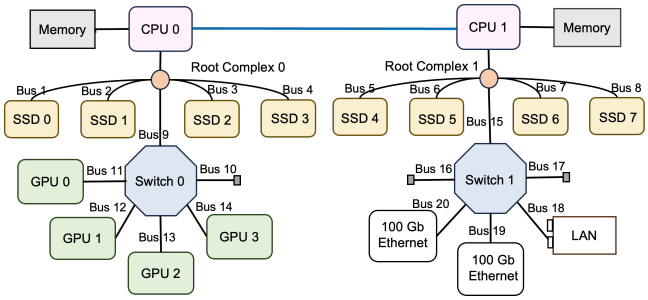
1 Introduction

Graph Neural Networks (GNNs) [10, 11, 23, 30, 58, 79] are a powerful approach for processing graph-structured data, making them particularly effective in applications such as recommendation systems [17, 25, 57, 66, 71, 77], social network analysis [2, 8, 40], and financial fraud detection [26, 37, 61, 67, 78, 82]. Applying GNNs on large-scale graphs in industrial settings is quite common [8, 12, 14, 24, 38, 52, 62, 70, 76, 77, 81, 87]. For example, in Alibaba's Taobao recommendation system [87], the user-item graph contains more than one billion vertices and tens of billions of edges, which requires several terabytes of storage space and can now easily exceed the upper limit of CPU memory capacity in a single machine. Therefore, training large-scale GNNs remains challenging.

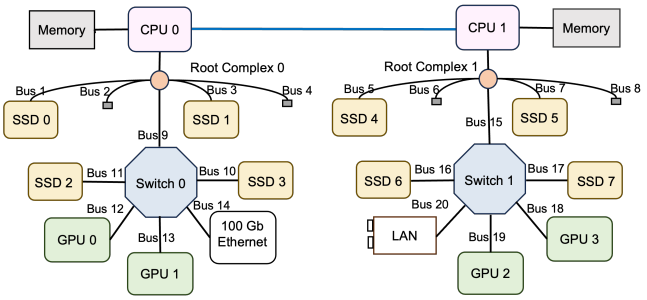
Existing solutions for training large-scale GNNs often compromise between throughput and monetary cost. One widely used approach is distributed clusters with multiple GPUs [16, 18, 21, 32, 36, 43, 56, 60, 64, 80, 83–87], but this requires significant memory across multiple machines, increasing costs linearly with graph size. Additionally, distributed systems require CPUs for graph sampling and extensive network communication, which struggle to match GPU training speeds [20, 75]. In contrast, single-machine solutions, such as in-memory and out-of-core GNN systems, provide more cost-effective alternatives [41, 42, 54, 59]. However, in-memory systems [34, 39, 44, 53, 74, 75] are limited by host memory size, making them unsuitable for training terabyte-scale graphs. Out-of-core systems [35, 41, 42, 48, 54, 59] expand storage to disks, such as NVMe SSDs, but are constrained by single GPU computation power and PCIe throughput (e.g., up to 20 GiB/s with PCIe 4.0 x16), limiting their training throughput.



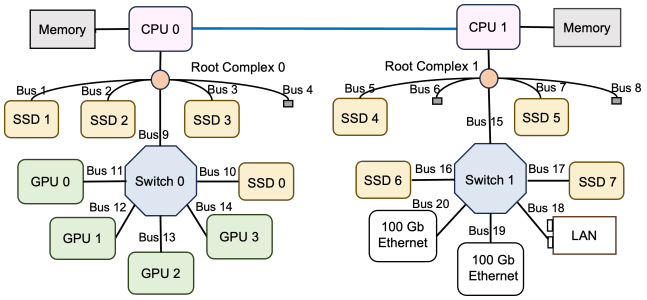
(a) SSD prioritizes the front board, GPUs are evenly distributed via PLX. Epoch Time: 15.9s



(b) SSD prioritizes the front board, GPUs are on the same PLX. Epoch Time: 26.7s



(c) SSDs are evenly distributed across PLX, GPUs are evenly distributed across PLX. Epoch Time: 14.9s



(d) SSDs are evenly distributed across PLX, GPUs are on the same PLX. Epoch Time: 24.1s

Figure 1: Comparison of Different Hardware Placement Strategies on Machine A. We report the epoch time of training the GraphSAGE [23] model on the IGB [29] dataset. We observe that hardware placement is critical to overall GNN performance.

Recent advances in GPU sampling [20, 75] and GPU-initiated direct SSD access [41, 54] make it possible for a single machine multi-GPU out-of-core GNN training system to achieve higher throughput at a lower cost than distributed systems. However, scaling an out-of-core system to multiple GPUs is challenging due to communication contention and load imbalance issues. The key issue lies in the impact of physical communication topology on data locality and interconnect bandwidth, which are overlooked by prior systems. Figure 1 shows that multi-GPU servers connect GPUs and SSDs via PCIe interfaces, with each GPU accessing SSDs through local or remote PCIe switches. Frequent access to remote SSDs by all GPUs can cause severe contention on CPU interconnects (e.g., QuickPath Interconnect, QPI), creating a bottleneck. Furthermore, Figure 6 illustrates that scaling from 2 to 4 GPUs can worsen performance due to IO bottlenecks.

To this end, we argue for co-optimizing the physical communication topology, i.e., hardware placement, with graph data placement for multi-GPU out-of-core GNN training. However, we have to address the challenge that the search space is huge, because different PCIe architectures introduce diverse ways of connecting CPUs, GPUs, and SSDs through switches and QPI, resulting in a multitude of complex communication combinations.

To address these challenges, we propose Moment, a physical topology and data placement *co-optimizer*, which simultaneously achieves high throughput and low monetary cost with a cheap customized single machine utilizing multiple GPUs and SSDs. The system automatically determines hardware and data placement during initialization. Our key idea is to formulate candidate combinations of physical communication links into a set of single-source single-sink capacity-constrained directed graphs, and then transform the communication planning into a *max flow* problem to maximize GPU PCIe throughput. First, it reduces the search space by removing symmetry-invariant and rotation-invariant structures. Then, it optimizes the max flow for each candidate to find the configuration with the highest throughput. Finally, the data-distribution-aware knapsack (DDAK) algorithm places graph embeddings in GPU/CPU memory and SSDs. At runtime, Moment introduces a multi-GPU-initiated disk I/O stack, allowing direct GPU-SSD access without CPU involvement. Experimental results show that Moment outperforms out-of-core systems by up to 6.51x and distributed systems by up to 3.02x at only about 50% monetary cost.

In summary, we make the following contributions:

- We highlight the impact of physical communication topology on multi-GPU out-of-core GNN training, including communication contention and GPU load imbalance, which previous works overlooked.
- We formulate the actual physical communication topology into a single-source single-sink weighted communication topology graph and minimize the optimization space by removing redundant structures, allowing solving the communication problem by a max flow modeling.
- We propose a data-distribution-aware knapsack algorithm that accounts for graph data skewness to optimally place embeddings across the entire memory hierarchy, including GPU/CPU memory and SSDs.

- We introduce Moment, which can train GNNs on terabyte-scale graphs within a single server, achieving up to 6.51× speedup over out-of-core systems and 3.02× over distributed systems at about half the cost.

2 Background and Motivation

2.1 Preliminaries

Graph Neural Networks (GNNs). Given a graph $G = (V, E)$, Graph Neural Networks (GNNs) are utilized to compute compact representations for each target vertex by applying neural networks across L layers. At each layer $l \in 1, 2, ..., L$, the activation h_v^l of a vertex $v \in V$ is updated by aggregating the features or hidden activations of its neighboring vertices, denoted as $N(v)$. This process is defined by:

$$\begin{aligned} a_v^l &= AGGREGATE^l(h_u^{l-1} | u \in N(v)) \\ h_v^l &= UPDATE^l(a_v^l, h_v^{l-1}) \end{aligned} \tag{1}$$

Here, the *AGGREGATE* function gathers information from the neighbors of vertex v, and the *UPDATE* function combines this aggregated information with the previous activation h_v^{l-1} to produce the new activation h_v^l.

Mini-batch GNN Training Workflow. This paper focuses on mini-batch GNN training using graph sampling to scale training for large graphs [8, 77, 87]. The overall workflow of mini-batch GNN training consists of three key steps: 1) graph sampling, 2) feature extraction, and 3) model training. Graph Sampling begins by selecting a batch of vertices and iteratively selecting their neighbors using strategies like random sampling [23, 79], which are then organized into a subgraph [63]. Feature Extraction involves collecting vertex embeddings from the sampled subgraph. These embeddings can vary in size depending on the vertices and their neighbors, capturing the necessary information for further processing. In model training, the *AGGREGATE* and *UPDATE* operations are performed on the sampled subgraph. During this process, model parameters are updated through forward and backward propagation. Previous studies [34, 36, 59, 75] have identified graph sampling and feature extraction as key bottlenecks in GNN training, particularly in out-of-core settings, unlike other DNNs where model training is the primary bottleneck.

2.2 Issues of Existing Distributed and Out-of-core Systems

Existing systems, including distributed systems and single-machine out-of-core systems for training large-scale GNNs, often compromise between throughput and monetary cost.

Distributed Systems. Distributed GNN systems, such as DistDGL [85], partition graph data across multiple machines to enable distributed data-parallel training. While this approach handles larger graphs by aggregating memory, it introduces several critical issues: *1) High Monetary Costs*: As graph sizes grow, the cost for more GPUs and machines increases linearly. *2) Limited Memory Capacity*: Even with scaling, the GPU/CPU memory in distributed systems may still fall short for large graphs. For instance, the CL dataset in Table 2 requires over 4 TB of memory. Storing the dataset in distributed CPU memory would require at least eight 500-GB machines, even before accounting for the memory expansion introduced by training frameworks such as DistDGL, which can demand up to 5× the dataset size. *3) Large Graph Sampling and Network Communication Overhead*: Distributed GNN training relies on data-dependent graph sampling, which often requires fetching data from remote machines and is typically handled by CPUs in existing systems [18, 36, 56, 83, 85, 87]. However, prior studies have shown that CPU-based sampling falls short of keeping up with GPU-based model training [20, 75].

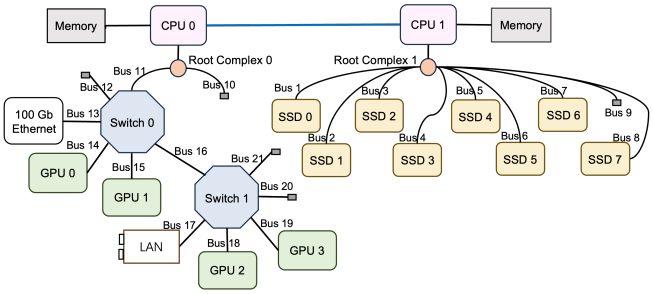
Single-Machine Out-of-core Systems. Existing out-of-core GNN systems typically rely on a single GPU, which limits their throughput. Specifically, these systems encounter two main issues:*1) Limited Computational Power of a Single GPU*: Training GNNs—especially complex models such as GAT [58]—is computationally demanding, yet the achievable throughput remains bounded by the computation power of a single GPU. *2) PCIe Bandwidth Bottlenecks*: System performance is further limited by the bandwidth of a single PCIe interface. For instance, a PCIe 4.0 NVMe SSD can deliver over one million random IOPS and approximately 6 GiB/s of bandwidth [22], which already approaches the maximum throughput of a PCIe 4.0×16 link (around 20 GiB/s). Meanwhile, SSD bandwidth continues to scale rapidly [3, 13, 19, 31, 47, 49, 51, 73]. On Machine A (Table 3), for example, 8 NVMe SSDs together sustain a peak throughput of 48 GiB/s—2.4× higher than that of a single PCIe link. Consequently, current out-of-core systems are unable to fully exploit the performance potential of modern SSDs.

2.3 Impact of Physical Communication Topology

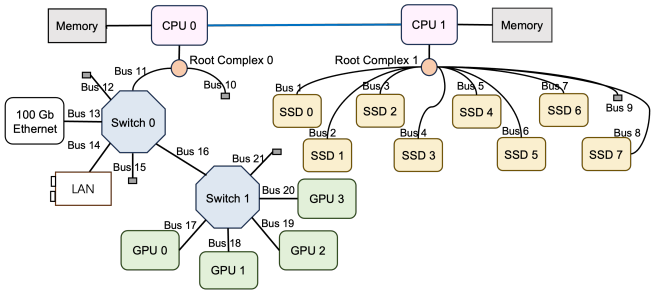
A promising direction to overcome the above limitations is scaling up GNN training with multiple GPUs and SSDs. However, this is non-trival due to the challenges of communication contention and load imbalance, both of which are highly dependent on the underlying physical communication topology.

Figures 1 and 2 illustrate two Machines A and B (Table 3) with different PCIe topologies, each equipped with 4 A100 GPUs, 8 P5510 NVMe SSDs, and additional devices. Machine A (Figure 1) adopts a balanced PCIe topology, whereas Machine B (Figure 2) follows a cascaded PCIe topology. To highlight the impact of physical topology, we construct four representative layouts based on a combined design space of SSDs and GPUs. For SSDs, we consider two placement principles: (i) balanced placement across front- and back-end boards, which is widely used to achieve load balance, and (ii) front-board–prioritized placement, which reflects practical constraints in modern servers (e.g., facilitating SSD hot-swapping). For GPUs, we also examine two strategies: (i) P2P-prioritized placement, which enables direct GPU communication without crossing PCIe switches or the CPU, thereby achieving higher P2P throughput, and (ii) even distribution across multiple PCIe switches. Combining these SSD and GPU strategies yields four hardware layouts, as summarized in Figures 1 and 2.

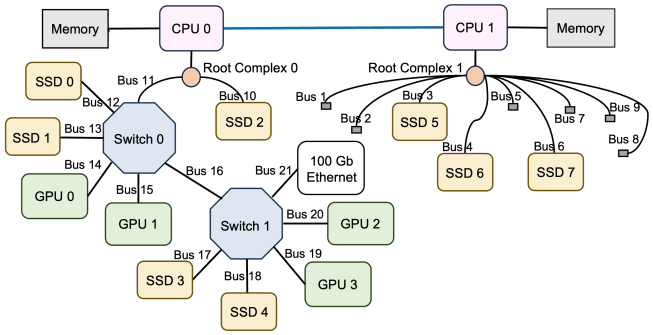
We extended Hyperion [54] and GIDS [41] from single-GPU to multi-GPU execution, creating M-Hyperion and M-GIDS. We evaluated M-Hyperion's training throughput on the IGB and UK datasets (see Table 2). As shown in Figure 3, Placement (c) achieved the best throughput, improving 1.86× over Placement (b). Similarly,



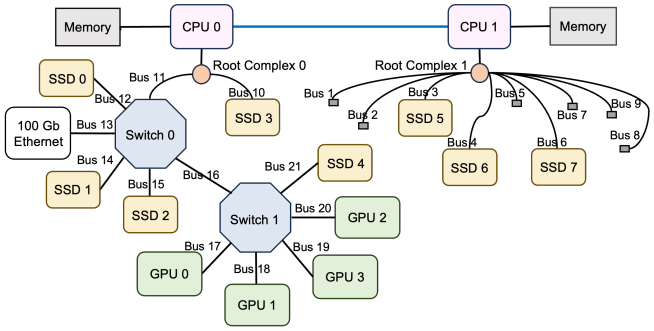
(a) SSD prioritizes the front board, GPUs are evenly distributed via PLX. Epoch Time: 28.4s



(b) SSD prioritizes the front board, GPUs are on the same PLX. Epoch Time: 29.7s



(c) SSDs are evenly distributed across PLX, GPUs are evenly distributed across PLX. Epoch Time: 18.6s



(d) SSDs are evenly distributed across PLX, GPUs are on the same PLX. Epoch Time: 24.0s

Figure 2: Comparison of Different Hardware Placement Strategies on Machine B. We report the epoch time of training the GraphSAGE [23] model on the IGB [29] dataset.

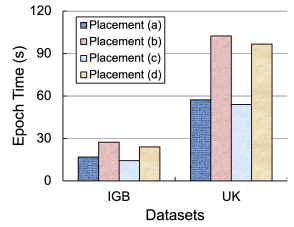


Figure 3: Comparison of Different Placements under Machine A

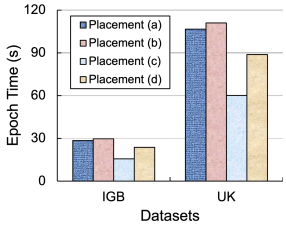


Figure 4: Comparison of Different Placements under Machine B

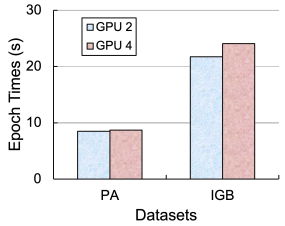


Figure 5: Throughput w.r.t. GPU expansion from 2 to 4 of M-Hyperion

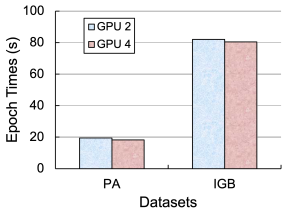


Figure 6: Throughput w.r.t. GPU expansion from 2 to 4 of M-GIDS

Figure 4 shows that Placement (c) was 1.96× better than Placement (b). Figures 5 and 6 demonstrate that increasing GPUs from 2 to 4 under Placement (d) resulted in little or decreased throughput. In summary, bad placement can degrade throughput by nearly 2× and reduce scalability. We explain the underlying reasons below.

Reason 1: Communication Contention. During GNN training, multiple GPUs will simultaneously access large volumes of graph vertex embeddings from storage devices, such as SSDs or CPU memory. The shared data paths, such as PCIe links and QPI, can become system bottlenecks due to communication contention. In machine A's setup, as illustrated in Figure 1b, the communication topology consists of two CPUs connected via QPI, with each CPU integrating a root complex. On the left, this root complex connects four NVMe SSDs via Bus 1-4 and links a PCIe switch hosting four GPUs via Bus 9. The PCIe topology on the right is symmetrical to the left but connects to other devices like network interface cards (NICs) and LAN adapters. The primary contention lies in the shared PCIe links between the PCIe switch 0 and root complex 0, i.e., Bus 9. The problems can similarly arise in machine B shown in Figure 2, such as the QPI links and Bus 11 in Figures 2a and 2b, as well as the Bus 11 and Bus 16 links in Figures 2c and 2d.

Reason 2: GPU Load Imbalance. Asymmetric GPU placements[1] can inherently lead to GPU load imbalance. Figure 2 shows that machine B's cascaded PCIe topology. The root complex 0 is connected to the PCIe switch 0 via Bus 11, and PCIe switch 1 is attached to PCIe switch 0 via Bus 16. According to Figure 4, *Placement (c) achieves the highest throughput among the four placements. However, it still suffers from load imbalance issues.* Specifically, GPU 0 and GPU 1 in this placement, directly connected to the PCIe switch 0, can access graph vertex embeddings through Bus 11-13 and Bus 16,

[1]There are asymmetric topology cases. E.g., PCIe expansion like H3 Falcon 4016 presented in a recent work [46] adopts a cascaded PCIe switch design, which is similar to the communication topology of Machine B in our paper.

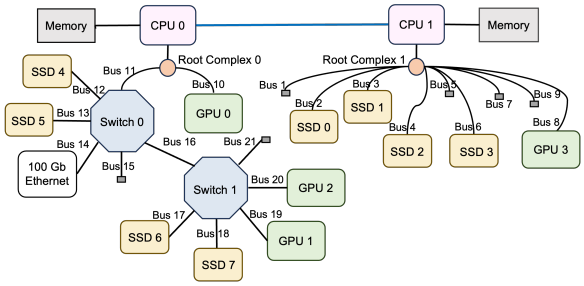


Figure 7: Moment's Placement on Machine B. Epoch Time: 13.2s

which can achieve approximately 40 GiB/s of overall IO throughput. In contrast, GPU 2 and GPU 3 connected to the PCIe switch 1 can only retrieve embedding data through the severely contended Bus 16 and Bus 17-18, achieving about 30 GiB/s maximal IO throughput. As such, the imbalanced IO throughput leads to imbalanced workloads for the GPU. *In contrast, Moment's optimization (Figure 7) selects an unusual configuration that reduces contention on Bus 16 and mitigates GPU load imbalance.*

Opportunities and Challenges: As highlighted, physical communication topology greatly affects GNN training throughput. As shown in our experiments, certain placements (e.g., placement (c)) can significantly outperform others, but the optimal choice varies with the machine configuration. Key influencing factors include the number and throughput of GPUs and SSDs, PCIe/QPI/UPI bandwidth, PCIe topology, and the distribution of graph data. With server vendors offering customized machines [15, 55] and large-scale GNN training being common in applications like e-commerce recommendations [8, 87], there is an opportunity to optimize hardware placement. Customized machines with multiple GPUs and SSDs can achieve high throughput at low costs, as shown in Figure 7. However, determining the optimal placement is non-trivial. The above factors form a vast and complex optimization space, making it difficult to design the most efficient communication solution. Our key insight is that throughput can be *predictable* by modeling the communication topology as a single-source, single-sink capacity-constrained directed graph. The formulation in Section 3.2 will guide us to find the optimal topology to maximize throughput.

3 Moment Design

We introduce Moment, a *co-optimizer* for physical communication topology and data placement to scale-up out-of-core GNN training, delivering high throughput and low cost on a customized machine with multiple GPUs and SSDs. Figure 8 outlines the Moment workflow, featuring an automatic module that models communication topology as a max-flow problem to optimize hardware placement and traffic distribution (see Section 3.2). Additionally, Moment employs a data-distribution-aware knapsack (DDAK) algorithm to optimize graph embedding placement (see Section 3.3).

3.1 System Workflow

Figure 8 illustrates the overall workflow of Moment. The inputs of Moment consist of (1) communication topology, which includes the physical connections between GPUs, SSDs, and CPUs, as well as the bandwidth constraints of interconnects like PCIe and QPI;

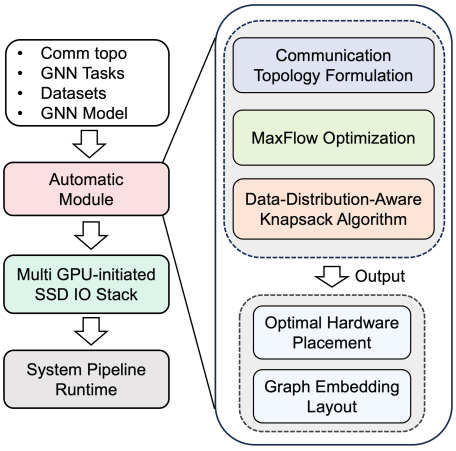


Figure 8: Moment Overview

(2) GNN Model (e.g., GCN [30], GraphSAGE [23]), including the graph sampling methods, determining the data access patterns; (3) Datasets including vertex embeddings and edge information.

Automatic Module. During offline initialization, Moment introduces an automatic module for hardware and data placement (i.e., graph embedding layout) using communication topology modeling, max flow optimization, and the data-distribution-aware knapsack algorithm (DDAK). The module first extracts the server's communication topology via Linux commands and libraries like libpci [69] and dmidecode [68], which provide PCI configuration and SMBIOS data. This topology is modeled as a directed communication graph for a *max flow* problem (see Section 3.2). It also profiles bandwidths of hardware components like SSDs, PCIe, and NVLinks, to establish throughput constraints. The module then eliminates redundant structures and compares feasible hardware placement schemes to identify the optimal placement and its predicted throughput. Note that with a given set of hardware, the optimal hardware placement can be reused across different GNN models and training loops. Finally, the DDAK algorithm determines the vertex embedding layout based on the flow through storage nodes (see Section 3.3).

Multi-GPU Disk IO Stack. Moment extends Hyperion's single-GPU IO stack [54] to allow multiple GPUs to initiate direct SSD access requests, enabling each GPU to fully utilize SSD IO throughput with minimal GPU core usage (e.g., 1%). Each GPU maintains submission queues (SQs), completion queues (CQs), and application buffers. During GNN training, each GPU retrieves the SSD ID and vertex offset, then initiates parallel IO requests. SSDs return the requested data to the application buffers, with each GPU independently managing its own IO stack.

System Runtime. Moment performs data-parallel training on multiple GPUs by evenly partitioning training vertices; each GPU pipelines sampling, feature extraction, and model training, like existing works [41, 54].

3.2 Communication Topology and Max Flow Formulation

To find the optimal hardware placement and communication planning, Moment first formulates physical communication topology into a directed communication graph and augments the physical

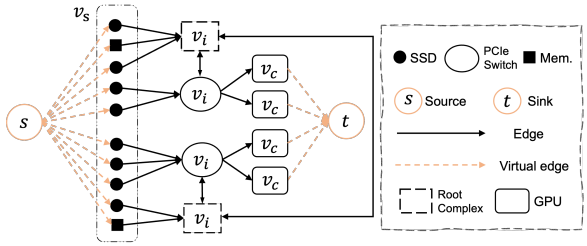


Figure 9: Example of converting a communication topology to a maximum flow problem. Vertex s is the abstract source, and t is the GPU sink. Edges represent physical link bandwidths; virtual edges have infinite bandwidth.

graph with virtual source/sink nodes and edges, as shown in Figure 9. Then Moment computes max flow based on the graph to find the optimal hardware placement.

Graph Representation. We represent the augmented communication topology as a directed graph $G = (V, E)$, where V is the set of nodes, including:

- Storage Nodes (V_s): Graph embeddings are stored in storage nodes, including GPU/CPU memory, and SSDs.
- Computation Nodes (V_c): Graph embeddings are consumed by computation nodes, i.e., GPUs.
- Interconnect Nodes (V_i): Interconnect nodes, such as PCIe switches, root complexes, and other interconnect components, do not store graph embeddings but function as interchange hubs for data transmission.
- Virtual Nodes: Source node s and sink node t. We add a single virtual source node that functions as a data source connecting to all storage nodes. We add a single sink node to aggregate the data flow from all computation nodes.

Therefore, the total set of nodes is:

$$V = V_s \cup V_c \cup V_i \cup \{s, t\}$$

The communication graph edges comprise physical edges (storage-interconnect, interconnect-interconnect, and interconnect-computation links) and virtual edges (source-to-storage input links and computation-to-sink output links), formally defined as $E = E_p \cup E_v$, where E_p and E_v denote physical and virtual edges, respectively. The physical edges E_p in the communication graph represent the complex hardware links facilitating data transfer between system components. These links can include various interconnect technologies, such as NVLink, PCIe, QPI, and others, depending on the system configuration. Importantly, E_p encompasses high-bandwidth connections, including those supporting NVLink, which is essential for efficient GPU-to-GPU communication in multi-GPU setups. During GNN training, GPU-driven embedding accesses across GPU memory, CPU memory, and SSDs enforce directional data flow: source → storage → interconnect → computation → sink.

Capacity Constraint. Each edge $e \in E$ has a capacity $c(e)$ representing the maximum data transfer rate, i.e., throughput constraint, over that link. Moment measures edge capacities by hardware profiling (See Section 3.1). For edge $(u, v) \in E_p$:

$$c(u, v) = ThroughputUpperLimit_{u \to v}$$

$$c(s, v_s) = c(v_s, v_i), c(v_c, t) = \infty$$

Definition of Flow. The flow f represents the actual amount of data transferred over each edge during GNN training. For the edge (s, v_c), $f(s, v_c)$ corresponds to the flow into the computation node v_c. For each edge, the flow cannot exceed its capacity constraint. Except for the source and sink nodes, the total flow input into a node must equal the total flow output from the node.

Optimization Objective. The objective is to **maximize the total flow** from the source s to the sink t:

$$\text{Maximize } F = \sum_{v_c \in V_c} f(s, v_c)$$

$$0 \le f(e) \le c(e), \quad \forall e \in E$$

$$\sum_{u:(u,v) \in E} f(u, v) = \sum_{v:(v,u) \in E} f(v, u)$$

Problem Solving. While the above discussion assumes a fixed hardware placement, there are numerous possible configurations given a set of hardware components. Since the machine's communication topology is accessible at the software level, we can simulate all feasible placement combinations. By systematically: 1) Eliminating Equivalent Variants (e.g., symmetrical-, rotation-invariant, or physically equivalent structures); 2) Considering Physical Slot Constraints (e.g., PCIe switch bandwidth or slot widths).

Specifically, placing GPUs and SSDs in PCIe slots leads to a large number of possible communication configurations. However, many of these are functionally equivalent due to symmetry-invariant properties of the topology. Symmetry arises in two main ways: (1) Topological symmetry—in most servers, PCIe topologies are balanced, meaning devices have identical access paths; swapping device positions or connections does not change traffic distribution or performance. (2) PCIe switch symmetry—devices connected to the same switch share identical bandwidth and connection characteristics, so their arrangement has no performance impact. In addition, rotation-invariant symmetry occurs when rotating or reordering components yields the same performance. By applying isomorphic graph reduction, we can eliminate these redundant configurations and simplify the search for optimal placements.

Furthermore, we use the dmidecode utility to check the actual lane widths of each PCIe slot, ensuring that only physically compatible combinations are considered, accounting for hardware constraints like the requirement for dual PCIe slots for high-end GPUs (e.g., NVIDIA A100) and single slots for NVMe SSDs.

After narrowing the search space to a limited set of distinct hardware placement candidates, we build a maximum flow model and perform simulations to estimate the total flow of each candidate. Specifically, Moment applies a time-bisection Ford-Fulkerson method procedure to estimate the minimum time required to reach the sink's total access demand. The optimal candidate is then selected based on the maximum predicted flow. This method effectively identifies the hardware placement that maximizes communication throughput.

3.3 Data-distribution-aware Knapsack Algorithm

In this section, we introduce a systematic method for optimizing graph data placement across different storage tiers, including CPU/GPU memory and SSDs. The goal is to match real I/O traffic

with the theoretically optimal distribution derived in Section 3.2, so that overall throughput can be maximized. However, naive uniform distribution methods (such as hash-based partitioning) are not effective for graph workloads because data access is highly skewed[2]. To address the challenge, we propose a data-distribution-aware knapsack algorithm that takes graph skewness and hotness into account for data placement.

Data-distribution-aware Knapsack Algorithm. We propose a data-distribution-aware knapsack algorithm (DDAK) to map storage node access traffic to graph vertex embeddings, ensuring precise allocation of each vertex's embedding information. This approach translates communication traffic patterns into physical data placement, achieving balanced load distribution for GPU-SSD access.

Specifically, we model GPU/CPU memory and NVMe SSDs as multiple bins. Using maximum-flow modeling, we estimate the access flow to each bin while respecting storage capacity constraints, determined by GPU/CPU cache rates and SSD sizes. We first collect vertex hotness information through pre-sampling, then sort the vertices in descending order of hotness for sequential allocation. The filling priority of each bin is defined as follows:

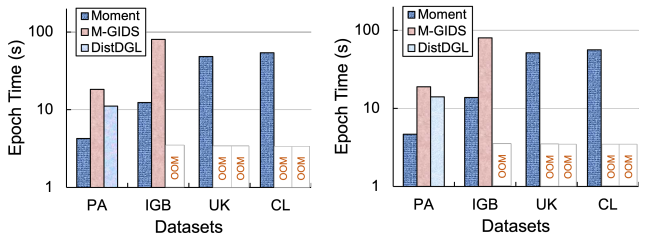
$$Bin_{priority} = \frac{Bin_{access}}{Bin_{traffic}} \cdot \frac{Bin_{currentCapacity}}{Bin_{Capacicity}} \tag{2}$$

In this formula, Bin_{access} represents the cumulative hotness of the current Bin, defined as the sum of the hotness values of all vertices stored in it. Bin_{traffic} denotes the expected traffic to the Bin, modeled by maximum flow. $Bin_{\text{currentCapacity}}$ is the current storage size of the bin holding embeddings, while Bin_{Capacity} is the desired capacity of the Bin. For GPU/CPU memory, Bin_{Capacity} is typically set to the cache size allocated for graph data, whereas for SSDs, it corresponds to the maximum available storage capacity.

Since GPUs provide High Bandwidth Memory (HBM) and CPU memory delivers higher I/O throughput than SSDs, we adopt the following storage hierarchy: GPU > CPU > SSD. Once a vertex embedding is placed into a bin according to this hierarchy, the bin's access and capacity usage are updated. After each placement, the priorities of all bins are re-evaluated to ensure that subsequent embeddings are allocated efficiently across the hierarchy.

The effectiveness of the DDAK algorithm lies in its dynamic priority mechanism, which optimizes access frequency, storage utilization, and hierarchy storage performance. 1) The dynamic priority adjustment ensures that high-throughput bins (e.g., GPU HBM) prioritize hot vertices, while excess data is redirected to underutilized bins as capacities approach limits, preventing overload; 2) DDAK aligns IO traffic with physical constraints, allocating skewed graph workloads and preventing traffic congestion by enforcing traffic limits for each bin (GPU, CPU, SSD); 3) The algorithm enforces a performance hierarchy (GPU>CPU>SSD) with adaptive migration of hot data and offloading of cold data.

Wide Applicability to Various Server Topologies. Our experiments demonstrate significant performance gains (30.6% improvement over hash-based allocation, as detailed in Section 4.5) in common balanced PCIe topologies through DDAK's intelligent data placement. However, real-world servers often exhibit asymmetric communication topologies due to heterogeneous hardware configurations or vendor-specific design choices.[3] Moment effectively addresses both scenarios: 1) In balanced topologies, DDAK optimally matches skewed data access patterns with uniform hardware resources; 2) In unbalanced topologies, the max flow model explicitly captures PCIe lane asymmetries through edge capacity constraints, while DDAK dynamically adapts data placement to match these hardware-imposed traffic limits. This dual approach ensures Moment automatically optimizes both hardware placement and data distribution to maximize throughput across arbitrary physical topologies. Moreover, the comprehensive evaluation of NVLink's communication enhancement effects is presented in Section 4.7.



(a) GraphSAGE (b) GAT

Figure 10: End-to-end Throughput of Moment, M-GIDS, and DistDGL

Pooling and Pre-processing Cost. While DDAK's vertex-wise allocation yields precise placement, it introduces pre-processing overheads. We therefore pool decisions over n vertices per step. Larger n accelerates planning but can increase cross-bin communication; in practice, we fix n=100 as a balanced default. With n=100, offline preprocessing (Max-Flow + DDAK) takes ~14 s on UK, versus ~90 s per epoch on a 2-GPU server. Because it runs once per model/hardware configuration and is reused across runs, the cost amortizes over many epochs (e.g., 48 for PA,), contributing <1% of total training time.

4 Evaluation

4.1 Experimental Setting

Experimental Platform. Table 3 shows the evaluated platforms: two single machines and one four-machine cluster.

GNN Models. We employ two sampling-based GNN models: GAT [58] and GraphSAGE [23]. Both models utilize a 2-hop random neighbor sampling strategy by default, with fan-out sizes of 25 and 10. For GAT, the hidden dimension is set to 64, and each layer consists of 8 attention heads. For GraphSAGE, the hidden dimension is configured to 256. Consistent with prior studies [53, 75], the batch size is set to 8000. Node classification serves as the primary task for these GNN models.

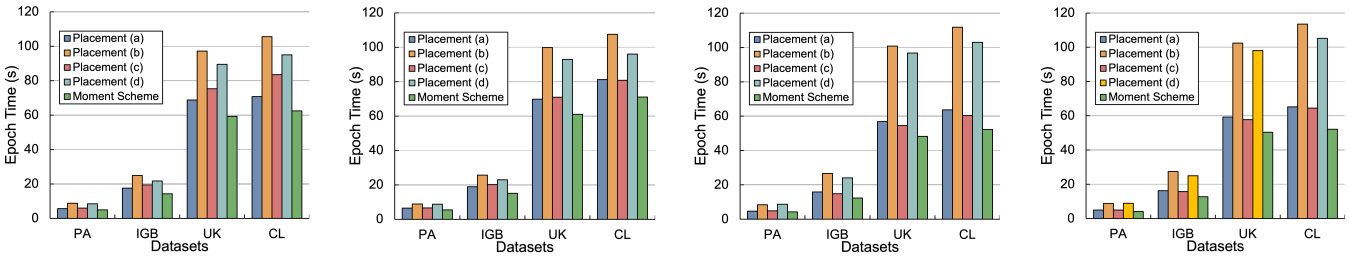
Datasets. We conduct our experiments on several real-world graph datasets of varying scales. Table 2 summarizes the characteristics of these datasets. The Paper100M (PA) dataset is sourced from the Open Graph Benchmark [28], while the IGB-HOM (IG) dataset is part of the IGB dataset [29]. The UK-2014 (UK) and ClueWeb (CL) datasets are obtained from WebGraph [4–7]. Since the UK

[2]A small set of vertices is accessed far more frequently than others.

[3]For example, to enable all-to-all GPU P2P, systems may employ a cascaded (nested) PCIe-switch topology, as shown in Fig. 2.

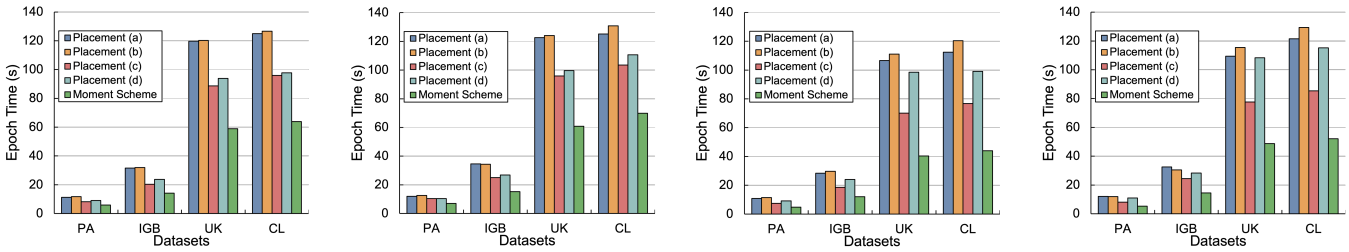
Table 1: Detailed evaluation platforms

Machine/Cluster	A	B	C (Cluster, 4× Machines)
GPU	40GB-PCIe-A100	40GB-PCIe-A100	40GB-PCIe-A100
SSD	8 × 3.84TB Intel P5510	8 × 3.84TB Intel P5510	/
PCIe/Network	4.0x16	4.0x16	3.0x16, 100Gbps
CPU	Intel(R) Xeon(R) Gold 5320 CPU (2 × 52 threads) @ 2.20GHz	Intel(R) Xeon(R) Gold 6426Y CPU (2 × 32 threads) @ 2.50GHZ	Intel(R) Xeon(R) Silver 4214 CPU (2 × 24 threads) @ 2.20GHz
CPU Mem.	768GB	512GB	256GB



(a) GPU2-GraphSAGE **(b) GPU2-GAT** **(c) GPU4-GraphSAGE** **(d) GPU4-GAT**

Figure 11: Comparison of Throughput of Four Classic Placements and Moment in Machine A



(a) GPU2-GraphSAGE **(b) GPU2-GAT** **(c) GPU4-GraphSAGE** **(d) GPU4-GAT**

Figure 12: Comparison of Throughput of Four Classic Placements and Moment in Machine B

Table 2: Dataset Statistics

Dataset	PA	IG	UK	CL
Vertices Num	111M	269M	0.79B	1B
Edges Num	1.6B	4B	47.2B	42.5B
Topology Storage	14GB	34GB	384GB	348GB
Feature Size	1024	1024	1024	1024
Feature Storage	56GB	1.1TB	3.2TB	4.1TB

and CL datasets lack features, we manually generate node features with a dimensionality of 1024, consistent with the settings for IG. Following the experimental setup in [75], we randomly select 1% of the vertices from each graph as training data.

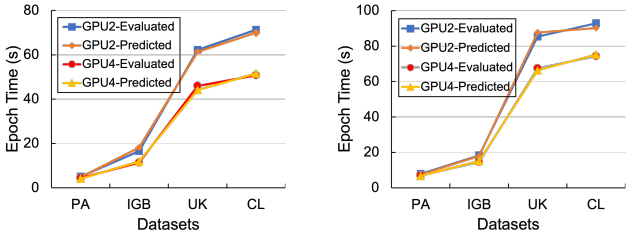
Baselines. We utilize the state-of-the-art out-of-core GNN system, GIDS [41], and the distributed system, DistDGL [85], as our baseline systems. DistDGL is deployed on Cluster C using 4 machines. Since DistDGL relies on CPUs for distributed sampling, we maximize the number of CPU threads per machine to 48. To measure DistDGL's network utilization, we employ Intel PCM and observe that its peak network throughput only reaches 20Gbps. This indicates that the PCIe 3.0 bandwidth of the cluster does not impose a bottleneck on DistDGL's performance. As GIDS supports only a single GPU, we extend its capability to multi-GPU execution by integrating PyTorch DDP and modifying its disk I/O stack, creating an enhanced version called M-GIDS. Specifically, since GIDS does not support shared access to a single SSD by multiple GPUs, we allocated a fixed number of SSDs to each GPU. For example, with a total of 8 SSDs, during 2 GPUs training, each GPU performed I/O access to 4 SSDs, and during 4 GPUs training, each GPU performed I/O access to 2 SSDs. For both M-GIDS and Moment, we store the entire graph topology in CPU memory, leveraging CPU memory as a cache for 1% of the vertices from each dataset.

4.2 End-to-End Throughput

We compare the throughput of Moment with state-of-the-art out-of-core baseline GIDS [41] and distributed baseline DistDGL [85] on all datasets in Table 2 and two GNN models, as shown in Figure 10.

Compared to Out-of-core Baseline. Moment outperforms M-GIDS by up to 6.51×, because Moment fully utilizes SSD throughput and overcomes the communication contention and load balance issues. M-GIDS runs out of GPU memory on UK and CL due to the requirement of its page cache (Based on BaM [45]) metadata.

Compared to Distributed Baseline. Moment outperforms DistDGL by up to 3.02× due to more efficient GPU-based graph sampling



(a) Machine A (b) Machine B

Figure 13: Prediction Accuracy of the Automated Prediction Module on Two Different Communication Architectures

and no network communication overhead. DistDGL runs out of CPU memory on IGB, UK, and CL datasets, as it allocates about 5× memory of the original dataset size. On the other hand, Moment only uses a single machine while DistDGL runs on 4× machines. The monetary costs of NVMe SSDs are negligible compared to distributed machines [55]. We refer to AWS [1] on-demand p5.16xlarge instances to evaluate the cost of a single machine with 4 GPUs and 4 machines each with 1 GPU, Moment achieves only about 50% monetary cost of DistDGL. Similarly, using TCO estimation in existing work [54], our configurations yield a 5-year TCO of $90,270 for Machine A/B versus $181,100 for the 4-node Cluster C.

4.3 Impact of Hardware Placements

We evaluate the impact of hardware placement on machines A and B, as shown in Figures 11 and 12. Moment is compared against four classical placements from Section 2.3, using 2 to 4 GPUs and two GNN models. Moment achieves up to 1.54× speedup on machine A, and up to 1.63× on machine B. *Figure 7 shows the optimal placement predicted by Moment on machine B with 4 GPUs and 8 SSDs.* In this setup, GPU 0 attaches to root complex (RC) 0; GPU 3 and four SSDs to RC 1; two SSDs to PCIe switch 0; and two SSDs plus two GPUs to PCIe switch 1. This layout maximizes locality to SSDs and CPU memory while keeping the critical paths (QPI, buses 9–11, and 16) uncongested, enabling near-full I/O utilization. With 4 GPUs, Moment delivers 15.61 GB/s average per-GPU inlet bandwidth, versus 10.92 GB/s for the best common layout (Placement (c)).

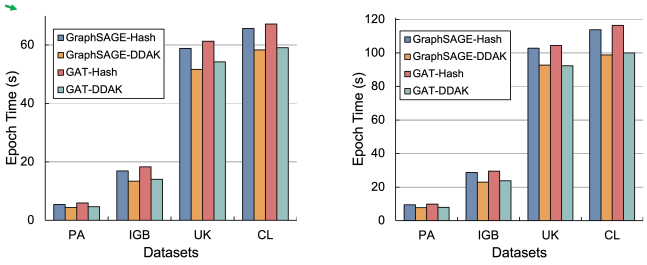
4.4 Automatic Prediction

To assess the automatic model's prediction accuracy, we compare Moment's predicted placements with the actual measured placements on Machines A and B from.

Figure 13 shows that Moment's prediction can generalize well across hardware. Specifically, across four datasets with 2- and 4-GPU settings, predicted and measured throughput closely match (max error 8.61%), indicating high predictive accuracy and robust generalization across configurations. Moment can leverage this to simulate different hardware combinations and select an optimal hardware placement.

4.5 Impact of Data-distribution-aware Knapsack Algorithm

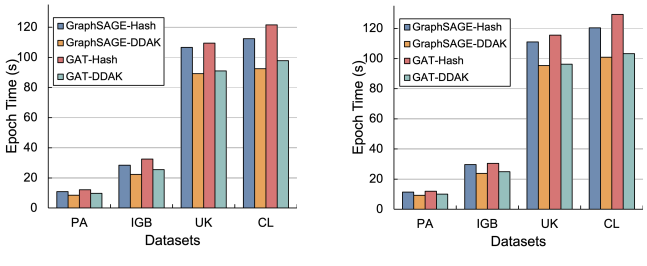
We evaluated the performance improvement of DDAK over the Hash data placements on Machine A and Machine B. With a fixed hardware placement of 4 A100 GPUs and 8 NVMe SSDs, DDAK



(a) Placement (a) (b) Placement (b)

(c) Placement (c) (d) Placement (d)

Figure 14: Comparison using DDAK in four classic placements under the Machine A communication architecture



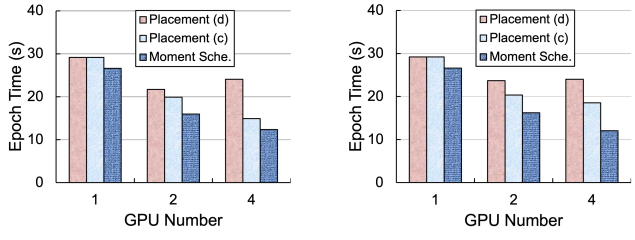
(a) Placement (a) (b) Placement (b)

(c) Placement (c) (d) Placement (d)

Figure 15: Comparison using DDAK in Four Classic Placements under the Machine B Communication Architecture

achieved a maximum improvement of 30.6% on Machine A and 34.0% on Machine B.

Moreover, we identify that DDAK alleviate QPI bottlenecks. We analyzed cross-QPI traffic on Machine A under four placement schemes, with results shown in Figure17. Across schemes (a)–(d), DDAK reduced cross-QPI traffic by 14.2%, 8.7%, 18.1%, and 9.5%, respectively. These results indicate that DDAK alleviates QPI-induced bottlenecks under diverse placement configurations.



(a) Machine A (b) Machine B

Figure 16: Scalability on Machine A and Machine B

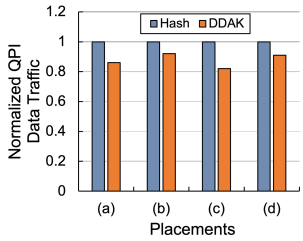


Figure 17: Comparison of QPI Data Traffic between Hash and DDAK Placement on Machine A

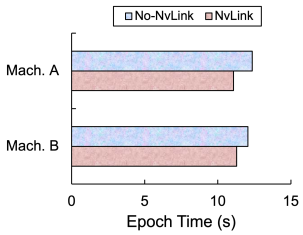


Figure 18: Comparison between No-NVLink and NVLink on Machine A and B

4.6 Scalability of Moment

To evaluate Moment's scalability, we vary the number of GPUs from 1 to 4 on Machine A and Machine B. We compare the throughput of Moment to the best placement (c) and the weaker placement (d) among four classic placements on IGB, as shown in Figure 16.

Scaling from 1 to 4 GPUs, under Machine A's topology the speedups are 1.92× (placement d), 1.21× (placement c), and 2.26× (Moment); under Machine B they are 1.57×, 1.21×, and 2.21×, respectively. With 4 GPUs, Moment nearly saturates the aggregate bandwidth of 8× SSDs, so further gains are limited unless more SSDs are added.

Notably, placement (d) shows negative scaling. After Bus 9 saturates (Fig. 1d), two GPUs can draw from two SSDs each, but with four GPUs the slot limits on PCIe Switch 0 restrict each GPU to one SSD—raising contention and cutting throughput.

4.7 NVLink Support

To evaluate Moment's support for general server architectures featuring NVLink, we conducted simulations under the Placement (c) scheme for both Machine A and Machine B using the IGB dataset. In the NVLink configuration, we added NVLink connections between GPU1 and GPU2, as well as between GPU3 and GPU4.

Figure 18 shows that enabling NVLink improves performance by 11.7% on Machine A and 6.8% on Machine B over non-NVLink configurations, due to the additional GPU–GPU communication paths through NVLink. Figure 1c illustrates that, without NVLink, GPU 0 and 1 could only exchange data through Bus9–11. With NVLink enabled (Bus12–13), alternative paths are available, allowing peer-to-peer transfers when PCIe channels become congested. These results indicate that Moment adapts well to mainstream server interconnect topologies while maintaining compatibility with heterogeneous hardware configurations.

5 Discussion

Generalization to Multi-node. We can extend Moment's optimization to multi-node environments. To do so, we can model the cluster-level communication topology by treating NICs, GPUs, and SSDs as hardware units connected via PCIe. As such, network communication links between NICs on different machines form the edges of the topology graph. By profiling the bandwidth with GNN training data transfers, Moment can obtain both theoretical and profiling-based bandwidths to construct the communication topology graph with edge capacity constraints. Then Moment determines the data traffic distribution and data placement based on the graphs. While network latency and congestion pose challenges, Moment's Max-Flow optimization and DDAK algorithm mitigate them by prioritizing local SSD/memory access. We leave this generalization as our future works.

SSD Wear Consideration. NVMe SSD wear only occurs from write operations during the initial dataset reorganization for graph data partitioning, while modern NVMe SSDs offer PB-level write endurance, far surpassing the demands of large-scale graph datasets. The GNN training process involves only read operations for embedding retrieval, eliminating additional wear. Thus, SSD wear is a negligible concern for Moment, due to this hardware robustness and our limited write operations.

Limitations. Moment assumes direct SSD access and high-bandwidth GPU interconnects; on commodity servers lacking these, performance may drop. It also targets mostly static workloads—using offline graph pre-sampling to estimate vertex popularity and DDAK for placement—whereas dynamic settings (e.g., online inference or streaming) require runtime monitoring and frequent embedding reallocation and index updates across SSDs/GPUs. We plan to add lightweight online profiling and adaptive placement to support real-time workloads.

Hardware-aware Optimization. Some recent works utilize the hardware topology and configurations for various applications such as optimizing collective communication [65], large language model training and serving [9, 27, 33], and database applications [50, 72]. In contrast, Moment targets GNN training that has different application characteristics and proposes the GNN-training-specific communication scheduling and data distribution technologies.

6 Conclusion

In this work, we introduce Moment, a physical communication topology and data placement co-optimizer, which achieves both high throughput and low monetary cost using an affordable, customized single machine equipped with multiple GPUs and SSDs. Moment reduces communication contention and GPU imbalance via max-flow scheduling on a capacity-constrained topology graph and optimizes data placement with a distribution-aware knapsack.

Acknowledgments

The work is supported by the following grants: the National Key R&D Program of China (Grant No. 2022ZD0119301), the National Natural Science Foundation of China under the grant numbers (62472384, 62441605, U24A20326), Starry Night Science Fund of Zhejiang University Shanghai Institute for Advanced Study (SN-ZJU-SIAS-0010). Zeke Wang is the corresponding author.

References

[1] AWS. 2024. Amazon EC2 G5 Instance. https://aws.amazon.com/cn/ec2/instance-types/g5/.

[2] Tian Bian, Xi Xiao, Tingyang Xu, Peilin Zhao, Wenbing Huang, Yu Rong, and Junzhou Huang. 2020. Rumor detection on social media with bi-directional graph convolutional networks. In *AAAI*.

[3] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien Le Moal, Gregory R Ganger, and George Amvrosiadis. 2021. ZNS: Avoiding the block interface tax for flash-based SSDs. In *USENIX ATC*.

[4] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. 2004. UbiCrawler: A Scalable Fully Distributed Web Crawler. *Software: Practice & Experience* (2004).

[5] Paolo Boldi, Andrea Marino, Massimo Santini, and Sebastiano Vigna. 2014. BUbiNG: Massive Crawling for the Masses. In *WWW*.

[6] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *WWW*.

[7] Paolo Boldi and Sebastiano Vigna. 2004. The Web Graph Framework: Compression Techniques. In *WWW*.

[8] Fedor Borisyuk, Shihai He, Yunbo Ouyang, Morteza Ramezani, Peng Du, Xiaochen Hou, Chengming Jiang, Nitin Pasumarthy, Priya Bannur, Birjodh Tiwana, et al. 2025. LiGNN: Graph Neural Networks at LinkedIn. *KDD* (2025).

[9] Chang Chen, Xiuhong Li, Qianchao Zhu, Jiangfei Duan, Peng Sun, Xingcheng Zhang, and Chao Yang. 2024. Centauri: Enabling efficient scheduling for communication-computation overlap in large model training via communication partitioning. In *ASPLOS*.

[10] Jie Chen, Tengfei Ma, and Cao Xiao. 2018. FastGCN: Fast Learning with Graph Convolutional Networks via Importance Sampling. In *ICLR*.

[11] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. 2019. Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks. In *KDD*.

[12] Ha Na Cho, Imjin Ahn, Hansle Gwon, Hee Jun Kang, Yunha Kim, Hyeram Seo, Heejung Choi, Minkyoung Kim, Jiye Han, Gaeun Kee, et al. 2022. Heterogeneous graph construction and HinSAGE learning from electronic medical records. *In Scientific Reports* (2022).

[13] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard Spillane, Amy Tai, and Rob Johnson. 2020. SplinterDB: closing the bandwidth gap for NVMe Key-Value stores. In *USENIX ATC*.

[14] Paul Covington, Jay Adams, and Emre Sargin. 2016. Deep neural networks for youtube recommendations. In *Recsys*.

[15] Dell. 2024. Custom Servers. https://www.dell.com/en-us/lp/custom-servers.

[16] Gunduz Vehbi Demirci, Aparajita Haldar, and Hakan Ferhatosmanoglu. 2022. Scalable Graph Convolutional Network Training on Distributed-Memory Systems. *In VLDB* (2022).

[17] Wenqi Fan, Yao Ma, Qing Li, Yuan He, Eric Zhao, Jiliang Tang, and Dawei Yin. 2019. Graph neural networks for social recommendation. In *WWW*.

[18] Swapnil Gandhi and Anand Padmanabha Iyer. 2021. P3: Distributed deep graph learning at scale. In *OSDI*.

[19] Aishwarya Ganesan, Ramnatthan Alagappan, Anthony Rebello, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2022. Exploiting Nil-external Interfaces for Fast Replicated Storage. *In TOS* (2022).

[20] Ping Gong, Renjie Liu, Zunyao Mao, Zhenkun Cai, Xiao Yan, Cheng Li, Minjie Wang, and Zhuozhao Li. 2023. gsampler: General and efficient gpu-based graph sampling for graph learning. In *SOSP*.

[21] Mingyu Guan, Anand Padmanabha Iyer, and Taesoo Kim. 2022. DynaGraph: dynamic graph neural networks at scale. In *GRADES & NDA*.

[22] Gabriel Haas and Viktor Leis. 2023. What Modern NVMe Storage Can Do, and How to Exploit It: High-Performance I/O for High-Performance Storage Engines. *In VLDB* (2023).

[23] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. *In NeurIPS* (2017).

[24] Junheng Hao, Tong Zhao, Jin Li, Xin Luna Dong, Christos Faloutsos, Yizhou Sun, and Wei Wang. 2020. P-companion: A principled framework for diversified complementary product recommendation. In *CIKM*.

[25] Xiangnan He, Kuan Deng, Xiang Wang, Yan Li, Yongdong Zhang, and Meng Wang. 2020. Lightgcn: Simplifying and powering graph convolution network for recommendation. In *SIGIR*.

[26] Binsheng Hong, Ping Lu, Runze Chen, Kaibiao Lin, and Fan Yang. 2024. Health Insurance Fraud Detection via Multiview Heterogeneous Information Networks With Augmented Graph Structure Learning. *In TCSS* (2024).

[27] Ke Hong, Xiuhong Li, Minxu Liu, Qiuli Mao, Tianqi Wu, Zixiao Huang, Lufang Chen, Zhong Wang, Yichong Zhang, Zhenhua Zhu, et al. 2025. FlashOverlap: A Lightweight Design for Efficiently Overlapping Communication and Computation. *arXiv preprint* (2025).

[28] 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 NeurIPS* (2020).

[29] Arpandeep Khatua, Vikram Sharma Mailthody, Bhagyashree Taleka, Tengfei Ma, Xiang Song, and Wen-mei Hwu. 2023. IGB: Addressing The Gaps In Labeling, Features, Heterogeneity, and Size of Public Graph Datasets for Deep Learning Research. In *KDD*.

[30] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *ICLR*.

[31] Nanqinqin Li, Mingzhe Hao, Huaicheng Li, Xing Lin, Tim Emami, and Haryadi S Gunawi. 2022. Fantastic SSD internals and how to learn and use them. In *SYSTOR*.

[32] Shuangchen Li, Dimin Niu, Yuhao Wang, Wei Han, Zhe Zhang, Tianchan Guan, Yijin Guan, Heng Liu, Linyong Huang, Zhaoyang Du, Fei Xue, Yuanwei Fang, Hongzhong Zheng, and Yuan Xie. 2022. Hyperscale fpga-as-a-service architecture for large-scale distributed graph neural network. In *ISCA*.

[33] Changyue Liao, Mo Sun, Zihan Yang, Jun Xie, Kaiqi Chen, Binhang Yuan, Fei Wu, and Zeke Wang. 2025. Ratel: Optimizing Holistic Data Movement to Fine-tune 100B Model on a Consumer GPU. In *ICDE*.

[34] 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*.

[35] Renjie Liu, Yichuan Wang, Xiao Yan, Haitian Jiang, Zhenkun Cai, Minjie Wang, Bo Tang, and Jinyang Li. 2025. DiskGNN: Bridging I/O Efficiency and Model Accuracy for Out-of-Core GNN Training. *Proc. ACM Manag. Data* (2025).

[36] Tianfeng Liu, Yangrui Chen, Dan Li, Chuan Wu, Yibo Zhu, Jun He, Yanghua Peng, Hongzheng Chen, Hongzhi Chen, and Chuanxiong Guo. 2023. BGL:GPU-Efficient GNN Training by Optimizing Graph Data I/O and Preprocessing. In *NSDI*.

[37] Yang Liu, Xiang Ao, Zidi Qin, Jianfeng Chi, Jinghua Feng, Hao Yang, and Qing He. 2021. Pick and choose: a GNN-based imbalanced learning approach for fraud detection. In *WWW*.

[38] Ziqi Liu, Chaochao Chen, Xinxing Yang, Jun Zhou, Xiaolong Li, and Le Song. 2018. Heterogeneous graph neural networks for malicious account detection. In *CIKM*.

[39] Seung Won Min, Kun Wu, Mert Hidayetoglu, Jinjun Xiong, Xiang Song, and Wen-mei Hwu. 2022. Graph Neural Network Training and Data Tiering. In *KDD*.

[40] Federico Monti, Fabrizio Frasca, and Eynard. 2019. Fake News Detection on Social Media using Geometric Deep Learning. *ICLR* (2019).

[41] Jeongmin Brian Park, Vikram Sharma Mailthody, Zaid Qureshi, and Wen-mei Hwu. 2024. Accelerating Sampling and Aggregation Operations in GNN Frameworks with GPU Initiated Direct Storage Accesses. *In VLDB* (2024).

[42] Yeonhong Park, Sunhong Min, and Jae W Lee. 2022. Ginex: SSD-enabled billion-scale graph neural network training on a single machine via provably optimal in-memory caching. *In VLDB* (2022).

[43] Jingshu Peng, Zhao Chen, Yingxia Shao, Yanyan Shen, Lei Chen, and Jiannong Cao. 2022. Sancus: staleness-aware communication-avoiding full-graph decentralized training in large-scale graph neural networks. *In VLDB* (2022).

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

[45] Zaid Qureshi, Vikram Sharma Mailthody, Isaac Gelado, Seungwon Min, Amna Masood, Jeongmin Park, Jinjun Xiong, CJ Newburn, Dmitri Vainbrand, I-Hsin Chung, Michael Garland, William Dally, and Wen-mei Hwu. 2023. GPU-Initiated On-Demand High-Throughput Storage Access in the BaM System Architecture. In *ASPLOS*.

[46] Zaid Qureshi, Vikram Sharma Mailthody, Isaac Gelado, Seung Won Min, Amna Masood, Jeongmin Park, Jinjun Xiong, C. J. Newburn, Dmitri Vainbrand, I-Hsin Chung, Michael Garland, William Dally, and Wen mei Hwu. 2022. *BaM: A Case for Enabling Fine-grain High Throughput GPU-Orchestrated Access to Storage*. Technical Report. https://arxiv.org/abs/2203.04910v2

[47] Zhaoyan Shen, Feng Chen, Gala Yadgar, Zhiping Jia, and Zili Shao. 2021. Prism-SSD: a flexible storage interface for SSDs. *In TCAD* (2021).

[48] Zeang Sheng, Wentao Zhang, Yangyu Tao, and Bin Cui. 2024. OUTRE: An OUT-of-Core De-REdundancy GNN Training Framework for Massive Graphs within A Single Machine. *In VLDB* (2024).

[49] Yongju Song, Wook-Hee Kim, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. 2023. Prism: Optimizing Key-Value Store for Modern Heterogeneous Storage Devices. In *ASPLOS*.

[50] Ziyu Song, Jie Zhang, Jie Sun, Mo Sun, Zihan Yang, Zheng Zhang, Xuzheng Chen, Fei Wu, Huajin Tang, and Zeke Wang. 2025. CAM: Asynchronous GPU-Initiated, CPU-Managed SSD Management for Batching Storage Access. In *ICDE*.

[51] Theano Stavrinos, Daniel S Berger, Ethan Katz-Bassett, and Wyatt Lloyd. 2021. Don't be a blockhead: zoned namespaces make work on conventional SSDs obsolete. In *HotOS*.

[52] Jie Sun, Zuocheng Shi, Li Su, Wenting Shen, Zeke Wang, Yong Li, Wenyuan Yu, Wei Lin, Fei Wu, Jingren Zhou, and Bingsheng He. 2025. Helios: Efficient Distributed Dynamic Graph Sampling for Online GNN Inference. In *PPoPP*.

[53] Jie Sun, Li Su, Zuocheng Shi, Wenting Shen, Zeke Wang, Lei Wang, Jie Zhang, Yong Li, Wenyuan Yu, Jingren Zhou, and Fei Wu. 2023. Legion: Automatically Pushing the Envelope of Multi-GPU System for Billion-Scale GNN Training. In *USENIX ATC*.

[54] Jie Sun, Mo Sun, Zheng Zhang, Jun Xie, Zuocheng Shi, Zihan Yang, Jie Zhang, Fei Wu, and Zeke Wang. 2025. Hyperion: Co-Optimizing SSD Access and GPU Computation for Cost-Efficient GNN Training. In *ICDE*.

[55] Supermicro. 2024. Supermicro SYS-420GP-TNR Dual Xeon Scalable 4U GPU SuperServer. https://store.supermicro.com/us_en/4u-gpu-superserver-sys-420gp-tnr.html.

[56] John Thorpe, Yifan QGraph attention networksiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Keval Vora, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 2021. Dorylus: Affordable, Scalable, and Accurate NN Training with Distributed CPU Servers and Serverless Threads. In *OSDI*.

[57] Rianne van den Berg, Thomas N Kipf, and Max Welling. 2017. Graph Convolutional Matrix Completion. *arXiv preprint arXiv:1706.02263* (2017).

[58] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In *ICLR*.

[59] Roger Waleffe, Jason Mohoney, Theodoros Rekatsinas, and Shivaram Venkataraman. 2023. MariusGNN: Resource-Efficient Out-of-Core Training of Graph Neural Networks. In *Eurosys*.

[60] Xinchen Wan, Kaiqiang Xu, Xudong Liao, Yilun Jin, Kai Chen, and Xin Jin. 2023. Scalable and Efficient Full-Graph GNN Training for Large Graphs. *In SIGMOD* (2023).

[61] Daixin Wang, Jianbin Lin, Peng Cui, Quanhui Jia, Zhen Wang, Yanming Fang, Quan Yu, Jun Zhou, Shuang Yang, and Yuan Qi. 2019. A semi-supervised graph attentive network for financial fraud detection. In *ICDM*.

[62] Jizhe Wang, Pipei Huang, Huan Zhao, Zhibo Zhang, Binqiang Zhao, and Dik Lun Lee. 2018. Billion-scale commodity embedding for e-commerce recommendation in alibaba. In *KDD*.

[63] Minjie Yu Wang. 2019. Deep Graph Library: Towards Efficient and Scalable Deep Learning on Graphs. In *ICLRW*.

[64] Qiange Wang, Yanfeng Zhang, Hao Wang, Chaoyi Chen, Xiaodong Zhang, and Ge Yu. 2022. Neutronstar: distributed GNN training with hybrid dependency management. In *SIGMOD*.

[65] Weiyang Wang, Moein Khazraee, Zhizhen Zhong, Manya Ghobadi, Zhihao Jia, Dheevatsa Mudigere, Ying Zhang, and Anthony Kewitsch. 2023. TopoOpt: Co-optimizing network topology and parallelization strategy for distributed training jobs. In *NSDI 23*.

[66] Xiang Wang, Xiangnan He, Meng Wang, Fuli Feng, and Tat-Seng Chua. 2019. Neural Graph Collaborative Filtering. In *SIGIR*.

[67] Xiao Wang, Houye Ji, Chuan Shi, Bai Wang, Yanfang Ye, Peng Cui, and Philip S Yu. 2019. Heterogeneous graph attention network. In *WWW*.

[68] Wikipedia-dmidecode. 2024. dmidecode. https://en.wikipedia.org/wiki/Dmidecode.

[69] Wikipedia-lspci. 2024. lspci. https://en.wikipedia.org/wiki/Lspci.

[70] Shiwen Wu, Fei Sun, Wentao Zhang, Xu Xie, and Bin Cui. 2022. Graph neural networks in recommender systems: a survey. *In ACM Computing Surveys* (2022).

[71] Shu Wu, Yuyuan Tang, Yanqiao Zhu, Liang Wang, Xing Xie, and Tieniu Tan. 2019. Session-based recommendation with graph neural networks. In *AAAI*.

[72] Yang Xiao, Mo Sun, Ziyu Song, Bing Tian, Jie Zhang, Jie Sun, and Zeke Wang. 2025. Breaking the Storage-Compute Bottleneck in Billion-Scale ANNS: A GPU-Driven Asynchronous I/O Framework. *arXiv preprint* (2025).

[73] Yi Xu, Henry Zhu, Prashant Pandey, Alex Conway, Rob Johnson, Aishwarya Ganesan, and Ramnatthan Alagappan. 2024. IONIA: High-Performance Replication for Modern Disk-based KV Stores. In *FAST*.

[74] Dongxu Yang, Junhong Liu, Jiaxing Qi, and Junjie Lai. 2022. WholeGraph: a fast graph neural network training framework with multi-GPU distributed shared memory architecture. In *SC*.

[75] 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 *Eurosys*.

[76] Xiaoyong Yang, Yadong Zhu, Yi Zhang, Xiaobo Wang, and Quan Yuan. 2020. Large scale product graph construction for recommendation in e-commerce. *arXiv preprint arXiv:2010.05525* (2020).

[77] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. 2018. Graph convolutional neural networks for web-scale recommender systems. In *KDD*.

[78] Zhongbao Yu, Jiaqi Zhang, Xin Qi, and Chao Chen. 2022. Application Research of Graph Neural Networks in the Financial Risk Control. *Mathematics and Computer Science* (2022).

[79] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. 2020. Graphsaint: Graph sampling based inductive learning method. *In ICLR* (2020).

[80] Dalong Zhang, Xin Huang, Ziqi Liu, Jun Zhou, Zhiyang Hu, Xianzheng Song, Zhibang Ge, Lin Wang, Zhiqiang Zhang, and Yuan Qi. 2020. AGL: A Scalable System for Industrial-purpose Graph Machine Learning. *In VLDB* (2020).

[81] Meng Zhang, Jie Sun, Qinghao Hu, Peng Sun, Zeke Wang, Yonggang Wen, and Tianwei Zhang. 2024. TorchGT: A Holistic System for Large-Scale Graph Transformer Training. In *SC*.

[82] Yiming Zhang, Yujie Fan, Yanfang Ye, Liang Zhao, and Chuan Shi. 2019. Key player identification in underground forums over attributed heterogeneous information network embedding framework. In *CIKM*.

[83] 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. *In VLDB* (2022).

[84] 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 *IA3*.

[85] Da Zheng, Xiang Song, Chengru Yang, Dominique LaSalle, and George Karypis. 2022. Distributed hybrid CPU and GPU training for graph neural networks on billion-scale heterogeneous graphs. In *KDD*.

[86] Da Zheng, Minjie Wang, Quan Gan, Xiang Song, Zheng Zhang, and George Karypis. 2021. Scalable graph neural networks with deep graph library. In *WSDM*.

[87] 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. *In VLDB* (2019).

Appendix: Artifact Description/Artifact Evaluation

Artifact Description (AD)

A Overview of Contributions and Artifacts

A.1 Paper's Main Contributions

The artifacts contain resources needed for our SC-25 submission.

- **C_1** We propose Moment, a physical communication topology and data placement *co-optimizer* to enable high-throughput and low-cost GNN training in a single multi-GPU machine.
- **C_2** Moment addresses communication contention and GPU load imbalance issues by modeling the physical topology as capacity-constrained directed graphs and formulating communication scheduling as a max-flow problem.
- **C_3** It also introduces a data-distribution-aware knapsack algorithm (DDAK) for optimized data placement
- **C_4** Finally, we propose a multi-GPU asynchronous direct disk IO stack. We integrate all these designs into a GNN training system.

A.2 Computational Artifacts

A_1 https://doi.org/10.5281/zenodo.16780272

Artifact ID	Contributions Supported	Related Paper Elements
A_1	$C_1 - C_4$	Figure 10,12,16
..		

B Artifact Identification

B.1 Computational Artifact A_1

Relation To Contributions

We implement max-flow and DDAK in an automatic module, which will suggest the best hardware placement and determine the corresponding data placement for users. Users can adjust their hardware according to the results. The DDAk module could also be executed independently of max-flow to generalize to more datasets and models with specific hardware placement. These steps should be executed only once for each training job. After that, users can run the main training loops.

We place the main scripts used in the root directory. We also place the main modules in the automatic module in the root directory, e.g., ddak.py, maxflow.py, profiler.py,... The sampling server directory includes all CUDA/C++ codes to implement the GNN systems' sampling module. Our multi-GPU disk IO stack is in the storage directory. The training backend includes different kinds of models' training scripts.

Expected Results

End-to-end results for Moment:

Compared to Out-of-core Baseline. Moment outperforms M-GIDS by up to 6.51×, because Moment fully utilizes SSD throughput and overcomes the communication contention and load balance issues. M-GIDS runs out of GPU memory on UK and CL due to the requirement of its page cache (Based on BaM) metadata.

Table 3: Detailed evaluation platforms

Machine/Cluster	A	B	C (Cluster, 4× Machines)
GPU	40GB-A100	40GB-A100	40GB-A100
SSD	8×3.84TB Intel P5510	8×3.84TB Intel P5510	/
PCIe/Network	4.0x16	4.0x16	3.0x16, 100Gbps
CPU	Intel Xeon Gold 5320 (2×52 threads) @ 2.20GHz	Intel Xeon Gold 6426Y (2×32 threads) @ 2.50GHz	Intel Xeon Silver 4214 (2×24 threads) @ 2.20GHz
CPU Mem.	768GB	512GB	256GB

Compared to Distributed Basline. Moment outperforms DistDGL by up to 3.02× due to more efficient GPU-based graph sampling and no network communication overhead. DistDGL runs out of CPU memory on IGB, UK, and CL datasets, as it allocates about 5× memory of the original dataset size. On the other hand, Moment only uses a single machine while DistDGL runs on 4× machines.

Expected Reproduction Time (in Minutes)

Automodule's reproduction time is approximately 10 minutes, while the GNN training process takes about 1 minute per epoch for IGB datasets.

Artifact Setup (incl. Inputs)

B.1.1 Hardware.
We evaluate on Machines in Table 3.

B.1.2 Software Dependencies. The software stack includes Ubuntu 22.04 (kernel 5.15.72), GCC/G++ 11.4.0, NVIDIA driver 515.43.04, CUDA 11.7–12.4, and Python 3.8+. **Python Packages.** Install Python packages with conda:

```
conda install pytorch==2.4.0 pytorch-cuda=12.4 \
-c pytorch -c nvidia
conda install torchmetrics
conda install -c dglteam/label/th24_cu124 dgl
```

Other required tools are `git`, `cmake`, and GNU Make.

GPU Direct Storage (BaM). We use the BaM kernel module for GPUDirect Storage.

```
cat /proc/cmdline | grep iommu
```

If either iommu=on or intel_iommu=on is found by grep, the IOMMU is enabled. To disable IOMMU, remove `iommu=on` and `intel_iommu=on` from `/etc/default/grub`, then the next time you reboot, the IOMMU will be disabled:

Build NVIDIA driver kernel symbols:

```
cd /usr/src/nvidia-515.43.04/
sudo make
```

Clone and build BaM:

```
git clone https://github.com/ZaidQureshi/bam.git
cd bam
git submodule update --init --recursive
mkdir build && cd build
cmake ..
make libnvm       # library
make benchmarks   # benchmarks
cd module && make # kernel module
```

Unload SSD's NVMe driver and load the BaM module (creates `/dev/libnvm*`):

```
sudo python Moment/unload_ssd.py
cd bam/build/module
sudo make load
```

To unload:

```
sudo make unload
```

B.1.3 Dataset Preparation. We conduct experiments on several real-world graph datasets of varying scales. The datasets used include Paper100M (PA), IGB-HOM (IG), UK-2014 (UK), and ClueWeb (CL). The UK and CL datasets do not have feature information, so we generate node features with a dimensionality of 1024, aligning with the features used in the IGB-HOM dataset. We provide scripts to get these datasets in our repo. Prepare graph datasets from OGB, Stanford-SNAP, and WebGraph by running:

```
cd Moment/dataset
bash prepare_datasets.sh
```

Refer to the README in the dataset directory for more instructions to customize datasets

B.1.4 Build and Installation. Build the project with:

```
cd Moment
bash build.sh
```

Artifact Execution

B.1.5 Automatic Module. Run the automatic module:

```
sudo python3 automatic_module.py
```

The module can be configured to adjust to different hardware. User can customize the file path, data access granularity (feature dimension, affecting the IO throughput), and number of GPUs/SSDs/CPUs in the script. For example:

```
file_path = "/share/gnn_data/igb260m/IGB-Datasets/data/"
feature_dim = 1024
num_gpu = 2
num_ssd = 6
```

The module will automatically get the hardware topology, profile the hardware bandwidth, collect data hotness, and run the maxflow and DDAk. See github repo to find the output details of each step.

B.1.6 Training. **GNN Models.** We use two sampling-based GNN models: GAT and GraphSAGE. Both models use a 2-hop random neighbor sampling strategy with fan-out sizes of 25 and 10. GAT has a hidden dimension of 64 and 8 attention heads per layer, while GraphSAGE has a hidden dimension of 256. Following prior studies, the batch size is set to 8000. Both models focus on node classification.
After running the automatic module, we can start training a GNN model. Open two different sessions to initiate the sampling server and training backend, respectively.
Start the sampling server:

```
sudo python3 moment_server.py --dataset_name igb \
--train_batch_size 8000 --fanout [25,10] --epoch 2
```

Model training with MPS enabled:

```
export CUDA_VISIBLE_DEVICES=0
sudo nvidia-smi -i 0 -c EXCLUSIVE_PROCESS
sudo nvidia-cuda-mps-control -d
export CUDA_MPS_ACTIVE_THREAD_PERCENTAGE=80
sudo python3 training_backend/moment_graphsage.py \
--class_num 2 --features_num 1024 \
--hidden_dim 256 --hops_num 2 --epoch 2
```

To stop MPS:

```
sudo nvidia-smi -i 0 -c DEFAULT
echo quit | nvidia-cuda-mps-control
```

Artifact Evaluation (AE)

C.1 Computational Artifact A_1

Artifact Setup (incl. Inputs)

C.1.1 Hardware. For the AE process, we'll create guest accounts for reviewers to access our machines, as shown in SC25's discussion session. During AE, we can provide experiments running on a Machine A in Table 3 with two Intel Xeon Gold 5320 CPUs (104 cores, two NUMA nodes, 768 GB RAM), PCIe 4.0,×16 slots, NVIDIA A100 GPUs (80 GB HBM2), and NVMe SSDs (e.g., Intel P5510).

C.1.2 Software Dependencies. Similar to hardware, we will provide a software environment for reviewers. All commands involving kernel modules or SSD unbinding require root privileges. As there might be concurrent usage of our environment, we prepared the environment in advance to avoid conflicts. (**Reviewers needn't prepare their environments**.)

The software stack includes Ubuntu 22.04 (kernel 5.15.72), GCC/G++ 11.4.0, NVIDIA driver 515.43.04, CUDA 11.7–12.4, and Python 3.8+.
Python Packages. Install Python packages with `conda`:

```
conda install pytorch==2.4.0 pytorch-cuda=12.4 \
-c pytorch -c nvidia
conda install torchmetrics
conda install -c dglteam/label/th24_cu124 dgl
```

Other required tools are `git`, `cmake`, and GNU Make.
GPU Direct Storage (BaM). We use the BaM kernel module for GPUDirect Storage.

```
cat /proc/cmdline | grep iommu
```

If either iommu=on or intel_iommu=on is found by grep, the IOMMU is enabled. To disable IOMMU, remove `iommu=on` and `intel_iommu=on` from `/etc/default/grub`, then the next time you reboot, the IOMMU will be disabled:
Build NVIDIA driver kernel symbols:

```
cd /usr/src/nvidia-515.43.04/
sudo make
```

Clone and build BaM:

```
git clone https://github.com/ZaidQureshi/bam.git
cd bam
git submodule update --init --recursive
mkdir build && cd build
cmake ..
make libnvm       # library
make benchmarks   # benchmarks
```

```
cd module && make # kernel module
```

Unload SSD's NVMe driver and load the BaM module (creates `/dev/libnvm*`):

```
sudo python Moment/unload_ssd.py
cd bam/build/module
sudo make load
```

To unload:

```
sudo make unload
```

C.1.3 Dataset Preparation. Prepare graph datasets from OGB, Stanford-SNAP, and WebGraph by running:

```
cd Moment/dataset
bash prepare_datasets.sh
```

Refer to the README in the dataset directory for more instructions to customize datasets

C.1.4 Build and Installation. Build the project with:

```
cd Moment
bash build.sh
```

Artifact Execution

In this section, we will show how to run the automatic module to find the best hardware placement and data placement. Then we will show how to run GNN training based on the placement.

C.1.5 Automatic Module. Run the automatic module:

```
sudo python3 automatic_module.py
```

The module can be configured to adjust to different hardware. User can customize the file path, data access granularity (feature dimension, affecting the IO throughput), and number of GPUs/SSDs/CPUs in the script. For example:

```
file_path = "/share/gnn_data/igb260m/IGB-Datasets/data/"
feature_dim = 1024
num_gpu = 2
num_ssd = 6
```

The module will automatically get the hardware topology, profile the hardware bandwidth, collect data hotness, and run the maxflow and DDAk. See github repo to find the output details of each step.

C.1.6 Training. After running the automatic module, we can start training a GNN model. Open two different sessions to initiate the sampling server and training backend, respectively. **GNN Models.** We use two sampling-based GNN models: GAT and GraphSAGE. Both models use a 2-hop random neighbor sampling strategy with fan-out sizes of 25 and 10. GAT has a hidden dimension of 64 and 8 attention heads per layer, while GraphSAGE has a hidden dimension of 256. Following prior studies, the batch size is set to 8000. Both models focus on node classification.

Start the sampling server:

```
sudo python3 moment_server.py --dataset_name igb \
--train_batch_size 8000 --fanout [25,10] --epoch 2
```

Model training with MPS enabled:

```
export CUDA_VISIBLE_DEVICES=0
sudo nvidia-smi -i 0 -c EXCLUSIVE_PROCESS
sudo nvidia-cuda-mps-control -d
export CUDA_MPS_ACTIVE_THREAD_PERCENTAGE=80
sudo python3 training_backend/moment_graphsage.py \
--class_num 2 --features_num 1024 \
--hidden_dim 256 --hops_num 2 --epoch 2
```

To stop MPS:

```
sudo nvidia-smi -i 0 -c DEFAULT
echo quit | nvidia-cuda-mps-control
```

Artifact Analysis (incl. Outputs)

See the detailed output in the repo.