Hyperion: Optimizing SSD Access is All You Need to Enable Cost-efficient Out-of-core GNN Training

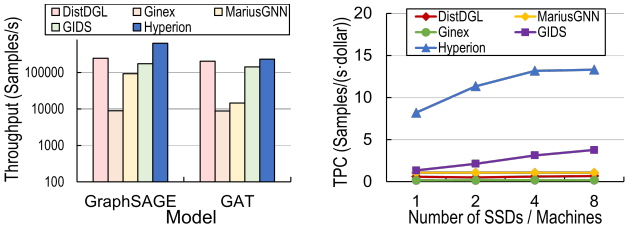
[1]Jie Sun, [1]Mo Sun, [2]Zheng Zhang, [1]Zuocheng Shi, [1]Jun Xie, [1]Zihan Yang, [1]Jie Zhang, [1]Fei Wu, [1]Zeke Wang
[1]Collaborative Innovation Center of Artificial Intelligence, College of Computer Science and Technology, Zhejiang University
[2]Purdue University

***Abstract*—SSDs are traditionally regarded as a cheap but slow way to scale up GNN training. Several GNN systems explore cheap single-machine single-GPU out-of-core training but fall short in terms of TPC (throughput per monetary cost). The underlying reason is that the existing systems 1) overly focus on minimizing the number of SSD accesses, which results in substantial unnecessary overhead on the CPU side, or 2) exhaust all GPU parallelism to saturate SSD but fail to overlap SSD accesses with GNN computation. In this work, we present Hyperion, a cost-efficient system for terabyte-scale GNN training. We argue that *co-optimizing GPU-initiated asynchronous SSD access and GNN computation pipeline enables us to only add cheap NVMe SSDs, rather than expensive GPU servers, to achieve in-memory-like throughput and thus maximal TPC of GNN training*. However, this is non-trivial due to imbalanced workloads and interference among IO submission, IO completion, and cache lookup. To tackle the challenges, Hyperion proposes three key designs. First, Hyperion proposes the first *GPU-initiated pipeline-friendly* asynchronous disk IO stack, which only requires about 1% GPU cores to saturate SSD throughput and wastes no GPU cores between IO submission and completion to fully overlap disk IO and computation. Second, we propose a new GPU-managed, *disaggregated, and unified* cache that disaggregates cache lookup from disk IO and fully utilizes CPU/GPU memory hierarchy by a unified static cache policy. Third, we propose a GNN-aware general TPC-analytical model that precisely predicts TPC under diverse hardware settings and GNN models and provide a *hint* to guide users to select hardware, e.g., number of SSDs, under a limited budget to maximize TPC. Experiments demonstrate that Hyperion can improve the TPC by over 3.1× on terabyte-scale graphs compared to SOTA out-of-core baselines and improve 60× TPC compared to distributed in-memory baselines.**

I. Introduction

Graph neural network (GNNs) [1], [16], [17], [29], [36], [37], [75], [80], [83], [92] are deep learning models designed to train on both structural and attribute data of graphs to generate low-dimensional embeddings. These embeddings are important in executing machine learning tasks such as node classification and link prediction. GNNs have been successfully applied to various domains such as recommendation systems [87] and financial risk control [45], [91] in many companies like LinkedIn [14] and Alibaba [103]. The graph size can now easily exceed the upper limit of CPU memory capacity in a single machine. For example, in Alibaba's Taobao recommendation system, the user behavior graph contains more than one billion vertices and tens of billions of edges [103], which need several terabytes of storage space.

Cost-efficient GNN training on large-scale graphs remains a great challenge with the continuously growing graph data size. The widely adopted approach to train large-scale GNNs is to use distributed in-memory solutions [21], [44], [99], [100], which utilize multiple machines' host memory to store graphs. However, these systems bring an extremely high monetary cost. For instance, training an IG [35] dataset with 1.1TB feature data requires at least eight commodity machines, each with 256 GB host memory. Eight commodity GPU machines cost 362K dollars for 5-year TCO[1] or 32.8 dollars per hour on AWS [7] for on-demand g5.16xlarge instance. To reduce monetary cost, several works [56], [57], [78] have explored out-of-core GNN training in a single machine with a single GPU, only requiring about 13% price of distributed counterparts[2]. However, existing out-of-core systems have low training throughput compared to distributed systems, as shown in Figure 1(a). To comprehensively compare the cost-efficiency of large-scale GNN training, we introduce a new metric named **TPC** (throughput per monetary cost), which is defined as the number of training samples per second for every dollar spent. Figure 1(b) demonstrates that existing out-of-core systems [56], [57], [78] have low TPC because these out-of-core systems do not make full potentials of SSDs, leading to very low GPU utilization and thus low TPC. In the following, we categorize these systems into two types, according to where the out-of-core mini-batch preparation is executed, as shown in Figure 2:



(a) Throughput (b) TPC

Figure 1. Throughput and TPC Comparison of Distributed In-memory System (DistDGL [100]) and Out-of-core Systems (MariusGNN [78], Ginex [57], GIDS [56], and Hyperion).

CPU-managed Out-of-core Systems. Ginex [57] and MariusGNN [78] leverage the CPU to prefetch a super-batch (SBG) of essential data from SSDs to CPU memory, thereby

[1]Total cost of ownership. TCO is computed in the § VI.

[2]Existing out-of-core systems identify that adding more GPUs in a single machine can not increase the overall throughput [56], [78] due to IO bound of GNN training, because adding GPUs can nearly not increase the storage capacity for terabyte-scale graphs. Therefore, in this paper, we focus on a single machine that has only one GPU for out-of-core training.

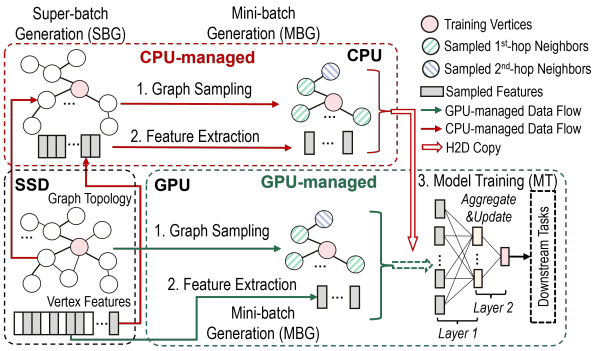


Figure 2. Training Workflow of Out-of-core GNN Systems. GPU-managed systems can also access topology and features from the CPU/GPU cache, not presented here for simplicity.

minimizing subsequent SSD accesses when generating mini-batch data (MBG) from the corresponding super-batch. In each mini-batch generation, these systems use the CPU to prepare sampled vertices with their features and copy the mini-batch to the GPU for model training (MT). However, these two systems overly focus on minimizing SSD accesses, resulting in substantial data preparation overhead on the CPU side: SBG and MBG dominate the majority of overall time, over 5× more than model training time on GPU (See Figure 3(a)).

GPU-managed Out-of-core System. To avoid CPU bottlenecks, GIDS [56] utilizes the GPU-initiated direct SSD access system BaM [60] to prepare mini-batch data without CPU involvement. But GIDS still has low training throughput due to two severe issues: 1) failing to maximize the disk IO throughput (down to 60% as shown in Figure 3(b)) due to insufficient parallelism, even using all GPU cores (See Figure 4(a)); 2) almost serial execution of graph sampling, feature extraction, and model training due to severe GPU core contention issues.

In essence, existing out-of-core systems underestimate the throughput of SSDs and the parallel power of GPUs in the GNN training. These systems either suffer from substantial data preparation overhead on the CPU side to overly minimize SSD accesses, or exhaust all GPU parallelism to hide SSD access latency and fail to fully overlap SSD access with computation. Therefore, they achieve significantly low TPC.

In this paper, we propose Hyperion, a cost-efficient out-of-core GNN training system on terabyte graphs. We observe that: 1) SSDs' throughput is increasingly higher and 2) a few GPU parallelism, e.g., 1% cores, is sufficient to saturate SSDs while enabling IO/computation overlapping (See § II-C). Therefore, we argue that **co-optimizing GPU-initiated SSD access and computation enables us to only add cheap NVMe SSDs, rather than on expensive GPU servers, to achieve in-memory-like throughput thus maximal TPC.** However, it is non-trivial to achieve. We propose three key designs to solve the corresponding challenges:

First, we propose a new GPU-initiated pipeline-friendly asynchronous disk IO stack to address the challenge of underutilized GPU parallelism due to imbalanced workloads between disk IO submission and completion (**Challenge 1**). The latency of IO submission is primarily determined by writing NVMe commands into submission queues (SQs) on GPU memory, whereas the latency for IO completion is dominated by long SSD access times. Collocating IO submission and completion within a single GPU thread provides essential thread-grained programming flexibility for many graph workloads [24], [48] but leads to interference between the processes of IO submission and completion, as discussed in § II-B. Fortunately, we observe that existing GNN training computation operations like graph sampling and feature aggregation could be efficiently executed *in a batched manner* through careful pipeline scheduling. Therefore, we disaggregate the IO process into two separate kernels, and thus only require very few (e.g. 1%) GPU cores to submit sufficient IO requests to saturate disk IO throughput and waste no GPU cores between IO submission and IO completion. Additionally, this disaggregation enhances the scalability of both IO stages to accommodate varying hardware configurations.

Second, we propose a new GPU-managed *disaggregated* cache with a *unified* static cache placement strategy to tackle the cache/disk IO interference challenge arising from asynchronous IO design (**Challenge 2**). Leveraging the full memory hierarchy, including CPU and GPU memory, as graph cache is crucial for accelerating GNN training as evidenced by various studies [43], [44], [57], [70], [86]. However, an efficient cache lookup kernel [50], [70] requires GPU threads within a warp concurrently reading coalesced memory addresses in a synchronous manner. Collocating cache lookup with disk IO kernels, e.g., like BaM [60], disrupts the asynchrony of disk IO designs. Otherwise, naively disaggregating cache with disk IO faces the challenges of data dependency, i.e., cache replacement, which causes serial execution of cache and disk IO tasks and results in low GPU PCIe utilization. Inspired by previous works [43], [70], [86], we observe that due to the inherent skewness of real-world graphs, a static cache can effectively accelerate GNN training while avoiding cache replacement. To maximize the utilization of CPU-GPU memory hierarchy in the static cache setting, we design a new *unified* static cache placement strategy to identify cache specialization for both graph topology and features. We then disaggregate cache management from disk IO into parallel GPU streams. Similar to the disk IO stack, we carefully allocate GPU parallelism for cache management by assessing the efficiency of PCIe burst requests.

Third, we propose the GNN-aware TPC-analytical model to provide a *hint* to users on how to further improve the TPC under a limited budget. However, the combination numbers of hardware and GNN model is huge (**Challenge 3**). To solve the challenge, the model uses a lightweight TPC Profiler to precisely measure GNN's data distribution and hardware throughput. This model guides users to take single-component hardware adjustment, e.g., only adding cheap NVMe SSDs instead of the entire server, as a new optimization dimension to maximize TPC.

We implement Hyperion on the top of the SOTA in-memory system Legion [70]. We evaluate Hyperion on three popular GNN models [29], [37], [75] and real-world graphs [6], [10]–

[13], [30], [35] whose sizes are up to 23TB. Experiments show that Hyperion outperforms the state-of-the-art GPU-managed baselines by up to a factor of 3.1× and exceeds CPU-managed counterparts by over 167× on terabyte-scale graphs (See Figure 8). And Hyperion achieves up to 60× TPC compared to the distributed system DistDGL (See Figure 10).

The contributions of this paper are:

- ✓ **A new insight about NVMe SSDs.** Co-optimizing IO and computation enables us to only add cheap NVMe SSDs, rather than expensive GPU servers, to achieve in-memory-like throughput, and thus maximal TPC of GNN training.
- ✓ **New disaggregation design of GPU-initiated disk IO stack.** To the best of our knowledge, Hyperion proposes the first GPU-initiated **pipeline-friendly** asynchronous disk IO stack with IO submission/completion disaggregation.
- ✓ **Novel disaggregated cache with unified policy.** Hyperion is also the first to propose the GPU-managed **disaggregated**, and **unified** static cache for out-of-core GNN training.
- ✓ **A GNN-aware TPC-analytical model to guide hardware adjustment.** We propose the TPC-analytical model within Hyperion. The model can precisely predict TPC under diverse GNN models as well as hardware combinations and further provide a hint that guides users to add an individual component, e.g., SSD, rather than the entire server, to maximize TPC.

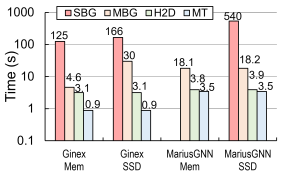
II. Background and Motivation

A. Preliminaries

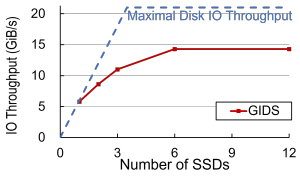
Graph Neural Networks (GNNs). For a graph $G = (V, E)$, GNNs are utilized to derive a compact representation for each target vertex by applying L layers of neural networks. During the iteration at layer $l, l \in L$, the activation h_v^l of vertex $v, v \in V$ is updated by aggregating the features or the hidden activation of its adjacent vertices, denoted as $N(v)$:

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

Mini-batch GNN Training Workflow. This paper focuses on mini-batch GNN training, which is a practical solution for scaling GNN training to very large graphs [14], [87], [103]. There are three main steps in mini-batch GNN training: 1) graph sampling, 2) feature extraction, and 3) model training. Neighbor sampling [29] is a widely adopted graph sampling approach, which starts from a subset of training vertices, iteratively samples multi-hop neighbor vertices according to a specific sampling strategy [29], [92], and organizes them into a subgraph [100]. The second step is to extract the features, i.e., vertex embeddings with hundreds to thousands of bytes, depending on the training vertices and their sampled neighbors. The third step is performing *AGGREGATE* and *UPDATE* according to Equations 1 based on the sampled subgraph, as well as updating the model parameters. We define the first two steps as mini-batch generation (MBG) and the third step as model training (MT). The inherent multi-hop neighbor expansion in MBG requires *iterative irregular* access to the graph storage and retrieves *fine-grained* graph data. And



(a) Execution Time Breakdown of Existing CPU-managed Systems.



(b) GPU PCIe Throughput Achieved by GIDS. The dimension is 1024.

Figure 3. Issues of existing CPU- and GPU-managed systems motivate the design of Hyperion. SBG, MBG, H2D, and MT represent the super-batch generation, the mini-batch generation, copying mini-batch to GPU, and model training on GPU.

the accessed data volume is large due to the neighbor explosion problem [92]. Therefore, previous studies [43], [44], [86] have identified MBG as the primary bottleneck in GNN training, distinguishing it from other DNNs, where model training (MT) is typically the main bottleneck [42], [102]. This challenge is further exacerbated in out-of-core GNN training settings due to lower bandwidth and the lack of byte-addressability in storage devices, e.g., NVMe SSDs[3], compared to traditional in-memory systems.

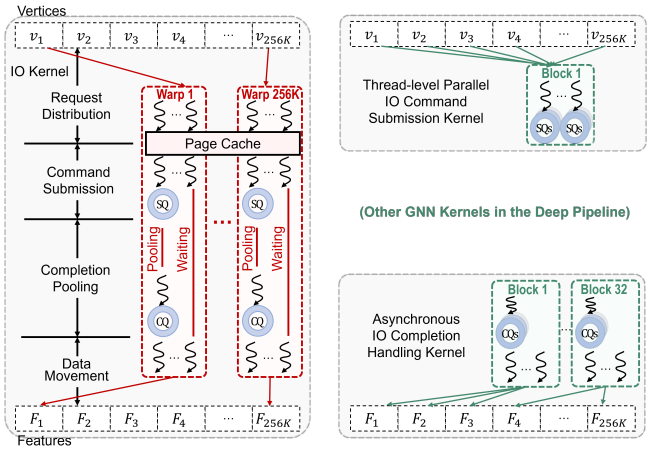
B. Issues of Existing Out-of-core Systems

Figure 2 illustrates the workflow of mini-batch GNN training in out-of-core systems. Depending on where MBG is executed, we classify out-of-core systems as two types: CPU-managed out-of-core systems (Ginex [57] and MariusGNN [78]) and GPU-managed out-of-core systems (GIDS [56] and Hyperion). Regarding the GNN training challenge in out-of-core settings, prior works have primarily focused on either reducing SSD access [56], [57], [78] or hiding SSD access latency through extensive GPU parallelism [56]. However, these approaches fall short of fully leveraging the capabilities of NVMe SSDs and GPUs, resulting in significant performance penalties such as CPU side bottleneck, GPU core contention, and thus poor TPC.

1) CPU-managed Out-of-core Systems:

Issue 1: Low GPU utilization due to the CPU bottleneck. Ginex [57] and MariusGNN [78] leverage the CPU to prefetch a super-batch (SBG) of essential data from SSDs to CPU memory, thereby minimizing subsequent SSD accesses when generating mini-batch data (MBG) from the corresponding super-batch. Specifically, Ginex adopts an inspector-executor model in each GNN training epoch. During the inspector stage, Ginex uses the CPU to process a super-batch including sampled vertices of all corresponding mini-batches, and execute feature cache management including loading features from disk to CPU memory. In MariusGNN, the SBG mainly uses the CPU to load graph partitions from disk to CPU memory. Previous in-memory works [59], [86] have proven that CPU suffers from irregular memory access and imbalanced workload during graph sampling. The additional overheads

[3]NVMe (Non-Volatile Memory Express) SSDs are high-performance storage devices that offer faster data transfer speeds and lower latency compared to traditional SATA (Serial Advanced Technology Attachment) SSDs. They enable parallel processing through multiple IO queues. In this work, we explore the potential of NVMe SSDs for out-of-core GNN training



(a) In GIDS, Pipeline-blocking (b) In Hyperion, Pipeline-friendly

Figure 4. Comparison of GIDS and Hyperion: IO process example of extracting $256K$ vertices' features in GPU-initiated systems. GIDS occupies $8K$ thread blocks (100% GPU cores) during IO access, while Hyperion only needs one thread block (about 1% GPU cores) to submit sufficient IO commands and 32 thread blocks (about 30% cores) to handle the completion of IO completion, and needs no cores for IO stack between IO submission and IO completion. v_i represents vertex i and F_i represents its feature.

from SBG and disk IO management in MBG make the CPU bottleneck even worse.

Figure 3(a) shows the execution time breakdown of Ginex and MariusGNN[4] on the PA [30] dataset by running each stage serially (We turn on the pipeline support of Ginex and MariusGNN in the end-to-end comparison, § IV-B). We also configure them in memory mode, where Ginex-Mem keeps running SBG including sampling a large batch of neighbors while MariusGNN-Mem does not execute the SBG. All systems store topology in memory. Ginex-Mem and MariusGNN-Mem store all features in CPU memory while Ginex-SSD and MariusGNN-SSD store all features in SSD. We configure 96 CPU threads and 12× P5510 NVMe SSDs for them. However, the CPU-managed SBG and MBG take 5-184× more time than model training (MT) on GPU, leading to low GPU utilization. These CPU-managed designs overly minimize SSD accesses, which brings substantial CPU-side data preparation overheads.

2) GPU-managed Out-of-core Systems:

To solve Issue 1, GIDS [56] enables GPU-initiated direct SSD access based on BaM [60] and utilizes GPU parallelism without CPU involvement to hide SSD/CPU memory access latency for MBG. As BaM focuses on array-like thread-grained programming abstraction for general purposes, it adopts a synchronous IO design and integrates cache management inside the IO kernel. We identify that it causes interference between IO submission and completion, so GIDS has low training throughput. The concrete issues are twofold:

Issue 2: Failing to maximize disk IO throughput. Figure 3(b) shows that GIDS achieves low throughput (down to 60%) compared to the maximal disk IO throughput, because of insufficient parallelism exploited by its design.

[4]MariusGNN provides an official breakdown mode.

Figure 4(a) shows how GIDS extracts vertices' features. GIDS utilizes one GPU IO kernel to handle the whole feature extraction procedure, in which each warp extracts one vertex's feature. The detailed process of the IO kernel consists of four steps. First, GIDS assigns each warp to look up BaM [60]'s page cache for one feature reading request (❶). Second, if getting a cache miss, a leader thread in the warp converts one request to one NVMe command and submits the command to a submission queue (SQ) of SSDs (❷). Subsequently, the leader thread keeps polling completion queues (CQs) of SSDs until it gets a completion entry of a command (❸). On receipt of the NVMe commands, SSDs will prepare the feature data and send them to the designated temporary IO buffer in GPU memory. After getting the completion entry, all threads in the warp move features from the temporary IO buffer to the output feature buffer (❹).

According to the nature of SSDs, the key to maximizing disk IO throughput is to send enough concurrent NVMe commands. However, GIDS fails to do so due to two concrete reasons. First, most of the threads in a warp are not utilized. During the step (❶-❸), only one leader thread executes command submission and completion polling, while all other threads are in a waiting state. This leads to most threads occupying GPU cores but doing nothing. Second, due to BaM [60]'s synchronous IO stack design, each warp can not execute other operations between the IO submission (❷) and the IO completion polling (❸). As such, each GPU core needs to execute/wait for a large number of long-latency disk IO processes consisting of serial IO submission and completion polling. In conclusion, GIDS's design allows a limited number of active working GPU cores and does not fully exploit the computing power of each GPU core. As illustrated in Figure 4(a), though GIDS launches all GPU cores for the feature extraction kernel ($256K$ warps, i.e., $8K$ thread blocks, 1024 threads in each block, and 100% GPU core utilization[5]), it can still hardly reach the maximal disk IO throughput (See Figure 3(b)).

Issue 3: Almost serial execution of graph sampling, feature extraction, and model training. Unlike traditional out-of-core workloads, such as graph BFS, which primarily focus on optimizing disk IO [2], out-of-core GNN training requires heavy computational workloads, e.g. graph sampling and model training, on the GPU. However, GIDS relies on all GPU cores for initiating disk IO commands, executing graph sampling, and model training. For example, GIDS launches kernels with $49K$, $10K$, and up to $24K$ thread blocks for graph sampling, feature extraction, and model training on the IG [35] dataset, respectively. As a result, each stage leads to severe GPU core contention issues, resulting in an almost serial execution of all three stages and thus low training throughput.

C. Observations and Opportunities

Issues 1-3 indicate that existing systems underestimate the throughput of SSDs and the parallel power of GPUs for GNN

[5]Measured with the SM (streaming multiprocessor) Active metric via NVIDIA NSight Systems [53].

training. Conversely, we have the following observations that shed light on new optimization opportunities:

Observation 1: Higher NVMe SSD throughput and easier to add more SSDs. Nowadays, a single PCIe 4.0 NVMe SSD can achieve more than one million random IOPS and a bandwidth of 7 GB/s [28], which is not far from the throughput of accessing CPU memory by GPU (e.g., 24GB/s by PCIe 4.0). The SSD throughput is continuously increasing [9], [18], [22], [40], [62], [66], [68], [85]. Furthermore, cloud vendors like Google Cloud and Amazon Web Service allow adding local NVMe SSDs [5], [25] for higher IOPS and larger capacity. In a customized local machine, PCIe expansions like H3 Platform Falcon-4016 [27] can accommodate PCIe devices. A single PCIe expansion supports adding more than 16 SSDs. Figure 1(b) shows that adding NVMe SSDs can remarkably increase TPC in single-machine out-of-core GNN training and can achieve up to 22× TPC compared to distributed counterparts.

- **Opportunity 1**: Adding cheap NVMe SSDs can be a new optimization dimension to maxmize TPC.

Observation 2: A small proportion of GPU parallelism is sufficient for saturating SSDs. We examine how much parallelism on GPU is needed to saturate SSDs (*without IO completion handling*). Assume that we have Q NVMe SSDs, each requiring M IOPS to maximize its IO throughput under the accessing granularity of K bytes. The GPU PCIe bandwidth is B bytes/s. Note that the aggregated disk IO bandwidth is up-bounded by PCIe bandwidth. To fulfill this requirement, we evaluate the number of parallel threads in a IO submission kernel. Assume that each thread needs T seconds to write an NVMe command to the SQs. Equation 2 estimates the required thread number P_s:

$$P_s = \begin{cases} Q \times M \times T & Q \times M \times K < B \\ B/K \times T & Q \times M \times K >= B \end{cases} \quad (2)$$

We implement an IO command submission kernel on an A100 GPU with PCIe 4.0x16 (B: 24GB/s) and 12 × Intel P5510 NVMe SSDs each with maximum theoretical 7.5×10^5 IOPS. We allocate the SQs with 4096 depth in GPU global memory and batch the doorbell for each SQ so that the doorbell time can be negligible. As such, the T can be approximatively regarded as the time that writing $warpsize$ × 64-byte NVMe commands into GPU memory by a warp. We measure the time on A100 and get 6.5 us for it. We show the case under two disk accessing granularities K: 512 bytes (aggregated SSD bandwidth less than PCIe bandwidth) and 4096 bytes (aggregated SSD bandwidth higher than PCIe bandwidth). As a result, only 59 threads under 512 bytes and 32 threads under 4096 bytes are needed to saturate 12 × Intel P5510 NVMe SSDs by A100. In practice, we can allocate more threads, e.g., 1 thread block with 512 threads (1% GPU cores on A100) to saturate SSD throughput.

- **Opportunity 2**: Fully utilizing parallelism of a few, e.g., 1%, GPU cores makes it possible to saturate SSD access and overlap IO with computation at the same time without CPU involvement (Solving Issues 1,2,3).

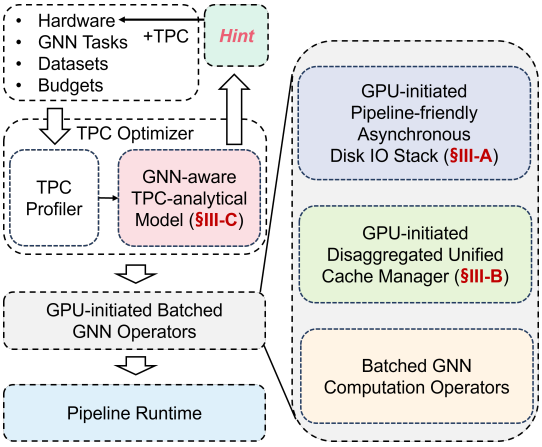


Figure 5. Hyperion Overview.

Motivated by these opportunities, we argue that **co-optimizing GPU-initiated SSD access and computation enables us to only add cheap NVMe SSDs for GNN training, rather than expensive GPU servers, to achieve in-memory-like throughput and maximal TPC.** However, this is non-trivial due to Challenge 1-3, as discussed in § I.

III. Hyperion Design

We build Hyperion, a cost-efficient general out-of-core GNN training system on terabyte graphs. We propose three key novel designs in Hyperion to address the associated challenges: 1) GPU-initiated pipeline-friendly asynchronous disk IO stack (Solve Challenge 1, see §III-A), 2) GPU-managed disaggregated unified cache (Solve Challenge 2, see §III-B), and 3) GNN-aware TPC-analytical model (Solve Challenge 3, see §III-C). Figure 5 shows an overview of Hyperion. In the following, we describe Hyperion's components and training workflow:

TPC Optimizer. Hyperion automatically takes GNN tasks, hardware settings, and graph datasets as inputs and allows users to configure their budgets to constrain hardware selection. Hyperion runs a lightweight TPC Profiler during the first training epoch to collect: 1) vertex hotness[6] by pre-sampling; 2) SSD/PCIe throughput and GNN model computation throughput to analyze the system TPC by a few, e.g., 10, training iterations[7]. Next, Hyperion uses a GNN-aware TPC-analytical model (See §III-C for more details) to give users a hint on how to adjust cheap NVMe numbers to achieve in-memory-like throughput and thus maximize TPC (Solving Challenge 3).

GPU-initiated Batched GNN Operators. Hyperion performs GNN computations on the GPU using batched processing instead of handling individual vertices one at a time. This approach allows for batched execution of IO operations, helping to disaggregate asynchronous disk IO kernels, which resolves Challenge 1. Specifically, graph sampling starts from a batch of sampling seeds, e.g., training vertices, and samples neighbor

[6]Used by cache initialization, see §III-B

[7]Since the batch seeds are randomly shuffled, several iterations exhibit similar statistical characteristics to an entire epoch.

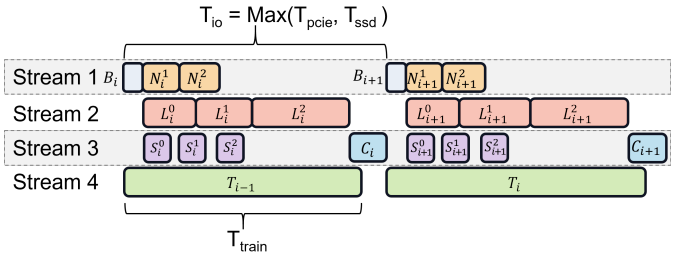


Figure 6. Hyperion's Pipeline Runtime for a k-hop, e.g., Two-hop, GraphSAGE [29] Training. S, C, B, T, N, and L represent the IO submission, IO Completion, batch seed generation, model training, neighbor sampling, and feature lookup, respectively. i represents the mini-batch ID. The superscripts stand for the hop index for the operator.

vertices by parallel GPU-based sampling [23], [33], [55]. During model training, particularly for feature aggregation and updates, Hyperion leverages batched message-passing operations supported by frameworks such as DGL [80] and PyG [20]. Hyperion implements disk IO kernels (See § III-A) and cache lookup kernels (See § III-B) in a batched manner. While batch processing improves throughput, it can lead to synchronization overhead due to uneven workload distribution among vertices and PCIe bottlenecks. To address this, Hyperion breaks down multi-hop/multi-layer GNN computations into a fine-grained pipeline, as discussed in the following.

Pipeline Runtime. Figure 6 illustrates the pipeline runtime of a k-hop GraphSAGE model [29], where i represents the mini-batch ID. Hyperion overlaps the mini-batch preparation of i with the mini-batch model training of i-1 (Can be disabled for models like [88]). To avoid GPU core contention and head-of-line blocking, Hyperion allocates a minimum GPU SMs for all MBG operations to achieve maximal throughput, e.g., 1% SM for IO submission and 11.1% cores for cache lookup by adjusting grid sizes, and configures rest GPU SMs for model training kernels by CUDA-MPS [54]. When preparing the mini-batch i, Hyperion first generates a batch of sampling seeds B_i on GPU. Then Hyperion executes k-hop neighbor sampling N_i^1,..., N_i^k. Hyperion initiates cache lookup L_i^k (See §III-B) and IO submission S_i^k (See §III-A) of k-th hop concurrently when initiating neighbor sampling of $(k+1)$-th hop (Solve Challenge 2). After finishing the cache lookup of k-th hop, Hyperion initiates IO completion C_i (See §III-A) corresponding to all IO submissions' results.

A. GPU-initiated Pipeline-friendly Async. Disk IO Stack

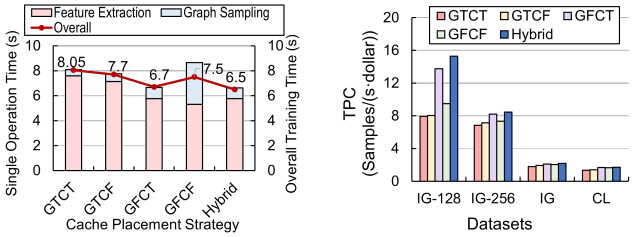
We propose a GPU-initiated *pipeline-friendly* asynchronous disk IO stack to solve Challenge 1, as shown in Figure 4(b). Hyperion disaggregates the IO process into two separate kernels, namely thread-level parallel IO command submission kernel and asynchronous IO completion handling kernel. Such a design has two benefits: 1) only requiring very few (e.g. 1%) GPU cores to submit sufficient IO requests to reach the maximal disk IO throughput; 2) wasting no GPU cores between IO submission and IO completion and leaving the majority of GPU cores for other GNN kernels.

1) Thread-level Parallel IO Command Submission: Hyperion proposes a thread-level parallel IO command submission kernel that fully utilizes threads in each warp to submit sufficient NVMe commands to SSDs. Hyperion configures the submission kernel with P_s threads and inputs the SSD logic block IDs of N_s sampled vertex features as well as their addresses in the output feature buffer. Each thread in the kernel processes a batch of IO command submissions corresponding to $\frac{N_s}{P_s}$ vertices' feature extraction. Each thread generates one NVMe command for each vertex and writes $\frac{N_s}{P_s}$ NVMe commands into $\frac{N_s}{P_s}$ entries in the SQs. During this process, each thread submits multiple NVMe commands without waiting for their completion. SSDs are informed to read the submitted NVMe commands in parallel by batched SQ doorbells. This IO submission kernel design enables Hyperion to parallelize the command submission at thread level, and thus maximize the utilization of all threads in each warp. Moreover, binding multiple vertex feature extraction requests to one thread reduces the total thread number for NVMe commands, improving GPU core utilization.

2) Asynchronous IO Completion Handling: To avoid the long completion handling time and waste of GPU cores, Hyperion proposes an asynchronous IO completion handling kernel, as shown in Figure 4(b). Hyperion allows users to determine the time to initiate the IO completion handling kernel. By default, Hyperion launches the IO completion handling kernel with P_c threads to process N_c IO completions at the time when all feature lookup operations are finished. In the kernel, each warp handles $\frac{N_c \times warpsize}{P_c}$ IO completions. During one handling operation, one leader thread in each warp first polls for the completion entry in the CQs. After the leader thread gets a completion entry, all threads in the warp move data from the IO stack's internal temporary buffer to the output feature buffer in a coalesced manner.

Disaggregating the IO completion handling and the IO submission can minimize GPU parallelism of polling and thus leave the majority of GPU cores for other GNN kernels like graph sampling and feature aggregation. Specifically, the IO submission kernel is initiated after graph sampling while the IO completion kernel is only initiated after finishing feature lookup and before the current batch's model training starts. The latency of polling a completion entry and data movement inside GPU memory is determined by GPU global memory access latency, which is orders of magnitudes less than disk IO. As a result, the completion handling kernel can bring negligible overhead to the overall throughput.

3) Scalability and Generalization of Hyperion's IO Stack: Hyperion's IO stack is general to various hardware. We evaluate Hyperion's IO stack on different hardware, e.g., H800 GPU, A100 GPU, Intel P5510 NVMe SSDs, and Samsung 980 pro NVMe SSDs (see §IV-D and §IV-F). In these platforms, allocating 1% GPU cores is sufficient to saturate SSDs. Due to the IO disaggregation design, Hyperion can allocate more GPU cores with less powerful GPU and higher-throughput SSDs. Hyperion can be extended to a multi-GPU platform by allocating SQs/CQs on multiple GPUs and maintaining an individual asynchronous disk IO stack for each GPU.



(a) Breakdown of IG-256 (b) Overall TPC

Figure 7. Impact of the Cache Placement Strategies on GraphSAGE Training. The GPU cache and CPU cache are configured to 20GB and 40GB while the SSD number is set to 12. Hybrid placement configures 15GB GPU cache and 35GB CPU cache for features.

B. GPU-managed Disaggregated Unified Cache

Hyperion adopts a GPU-managed *disaggregated* cache design to cooperate with the asynchronous IO stack (Solve Challenge 2). Hyperion utilizes GPU to initialize the cache placement and issues cache lookups (§III-B1). Hyperion determines the cache contents with an efficient *unified* static pre-sampling-based cache policy (§III-B2) to fully utilize CPU-GPU memory hierarchy.

1) Disaggregated Cache Management by GPU:

Cache Initialization. During system initialization, Hyperion's TPC-Profiler runs an epoch of pre-sampling by GPU[8], similar to previous work [70], [86], and collects all vertices' hotness, i.e., access frequency. Subsequently, Hyperion uses GPU to sort all vertices by their hotness in descending order. According to our cache placement strategy (§III-B2), Hyperion fills up the available CPU and GPU memory with the hottest topology or feature data.

Cache Lookup. During neighbor sampling, Hyperion identifies the cache hit/miss of vertices' features and next-hop neighbors. Subsequently, Hyperion launches next-hop neighbor sampling and feature cache lookup GPU kernels (For cache hit) that execute in parallel with disk IO kernels (For cache miss), as shown in Figure 6. GPU kernels directly access the cached data in CPU or GPU memory by Unified Virtual Addressing (UVA [46]) in warp-granularity. Multiple threads in each warp concurrently access coalesced columns of feature tables or topology tables (in CSR format). Hyperion's TPC-Optimizer automatically utilizes a minimum number of GPU threads in each cache lookup kernel that maximizes cache lookup throughput. For instance, Hyperion configures 11% cores on A100 to generate sufficient concurrent outstanding read requests to saturate PCIe 4.0x16, if assuming access features only from the CPU cache. During the cache lookup process, Hyperion does not run cache replacement.

2) Unified Cache Placement Policy:

We propose a unified static cache placement policy to allow efficient disaggregation of cache and disk IO while fully utilizing memory hierarchy.

Challenges of Static Cache Policy. Previous work [86] has shown that pre-sampling-based static caching policy achieves about 90% – 99% of the optimal cache hit rate in GNN training scenario, making it possible to remove cache replacement. This is because many real-world graphs are highly skewed [43], [70], [86]. For example, only 5% hottest vertices dominate up to 70% total access frequency among all vertices [70]. However, existing static policies [43], [70], [86] only regard the GPU memory as a cache without consideration of the entire memory hierarchy or only maintain feature cache. Fully utilizing the CPU-GPU memory hierarchy and comprehensively considering topology and features is non-trivial due to different dataset distributions and CPU/GPU memory bandwidth/latency. Specifically, we compare five strategies for memory allocation: (1-4) filling GPU (G) or CPU (C) memory with the hottest topology (T) or feature (F) first, then using the remaining space for the second hottest data type. For example, GTCF means filling the GPU memory with the hottest topology, then the hottest feature, followed by the second hottest feature in the CPU memory. In (1-4), at least one type of memory is filled by only one type of data. (5) A hybrid strategy splits both GPU and CPU memory to store both the hottest and second-hottest topology and features. Figure 7 illustrates the breakdown of the operation execution time[9] on dataset IG-256 and the overall TPC of four datasets (see § IV-A). As Figure 7(a) shows, there are trade-offs between caching topology and feature data, affecting both feature extraction and graph sampling processes. Figure 7(b) illustrates that the worst cache placement could degrade the overall training TPC to down to 52%. Mixing the GPU/CPU cache both with topology and feature is often the best case but splitting suitable cache space is hard to manually decide.

Hyperion's Unified Cache Principle. Hyperion proposes a unified optimization principle, *minimum PCIe transaction*, to comprehensively decide the cache specialization on CPU/GPU memory. Hyperion splits available GPU and CPU memory into many small chunks of size Z, e.g., 100 MB. After pre-sampling, Hyperion sorts vertices according to their topology/feature access hotness (similar to previous work [70]) and gets two order queues Q_T and Q_F. Hyperion logically fills all topology (feature) into a chunk set S_T (S_F) in the order of Q_T (Q_F) by a GPU parallel scan. Note that Hyperion only records vertices' IDs in each chunk instead of filling physical topology/feature into physical memory chunks in this step. Next, Hyperion decides how to fill physical memory chunks in the GPU cache, followed by the CPU cache.

GPU Cache Policy. For each logical topology chunk in S_T, Hyperion evaluates PCIe transactions that it might reduce if cached in GPU. Specifically, Hyperion calculates transaction number by $\lceil \frac{H_i^T \times N_i}{CLS} \rceil$ for vertex i and sums up all vertices' transaction in this chunk, where H_i^T, N_i, and CLS represent topology hotness and neighbor size (in bytes) of vertex i and the transferred cache line size by PCIe. Similarly, Hyperion calculates feature transaction by $\lceil \frac{H_j^F \times D_j}{CLS} \rceil$, where H_j^F and D_j represents the feature hotness and the feature vector size (in bytes) of vertex j. Next, Hyperion uses dynamic programming

[8]In the process, GPU directly accesses the topology data stored in SSDs.

[9]For simplicity, we group all kernels related to graph sampling or feature extraction into a single operation.

Table I
DETAILED EVALUATION PLATFORMS.

Machine/Cluster	A	B	C (Cluster, Eight Machines)
GPU	80GB-PCIe-A100	80GB-PCIe-H800	80GB-PCIe-A100
SSD	12 × 3.84TB Intel P5510, 6 × 1TB Samsung 980pro	12 × 3.84TB Intel P5510	/
PCIe/Network	4.0x16	5.0x16	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
TCO	48,971 (with 12 × Intel P5510)	78,794 (with 12 × Intel P5510)	45,275 × 8

(DP) to select the chunks with their physical topology or feature to fill the physical GPU cache chunks so that the overall PCIe transactions are minimized.

CPU Cache Policy. After filling up the GPU cache, Hyperion masks out the GPU chunks and fills up the CPU cache physically. Similarly, Hyperion calculates PCIe transactions for all residual chunks if accessed from SSDs. For topology chunks, Hyperion calculates transaction number by $\lceil \frac{H_i^T \times N_i}{SSDS} \rceil \times \frac{SSDS}{CLS}$, where $SSDS$ represents the minimum SSD access granularity, e.g. 512 Bytes. For feature chunks, Hyperion calculates transaction number by $\lceil \frac{H_j^F \times D_j}{SSDS} \rceil \times \frac{SSDS}{CLS}$, where $SSDS$ represents the minimum SSD access granular, e.g. 512 Bytes. Hyperion adds PCIe transaction numbers from accessing topology/feature in both CPU memory/SSDs and minimizes overall PCIe transaction by dynamic programming.

C. GNN-aware TPC-analytical Model

To identify optimal TPC, we propose a GNN-aware TPC-analytical model that can predict GNN's TPC under various combinations of GNN models and hardware (Solve Challenge 3) and provide a *hint* on how to maximize TPC.

User-configurable Parameters. Hyperion captures the users' GNN-specific parameters of their training job such as batch size Bs, GNN layer numbers, and sampling fan out by default. Hyperion also allows users to limit their overall budget $C_{overall}$ and will search for optimal hardware combinations for them. Additionally, users need to input the prices C_{ssd} of one SSD and $C_{machine}$ of all other hardware except SSDs. The entire monetary cost of a given system including Q SSDs is $C_{ssd} \times Q + C_{machine}$.

Overall TPC Estimation. We define the average mini-batch execution time as T_{batch}. TPC can be calculated by Equation 3:

$$TPC = \frac{Bs}{T_{batch} \times (C_{ssd} \times Q + C_{machine})} \quad (3)$$

As Figure 6 shows, preparing mini-batch i can be overlapped with the training of mini-batch $i-1$ while the SSD access is overlapped with cache lookup. As such, we calculate T_{batch} by $Max(T_{ssd}, T_{pcie}, T_{train})$, which represents the SSD access, PCIe access, and model computation time, respectively.

$$T_{batch} = Max(T_{ssd}, T_{pcie}, T_{train}) \quad (4)$$

TPC Profiling. Next, we estimate T_{ssd}, T_{pcie}, and T_{train}. First, TPC Profiler records the average per-batch model computation time T_{train}. Second, TPC Profiler collects: 1) the overall PCIe transaction number N_{ssd}/N_{mem} of reading SSD/CPU memory (as discussed in § III-B2), 2) per SSD throughput Th_{ssd}, 3) PCIe throughput Th_{pcie}, and 4) cache lookup overhead α. Hyperion calculates $T_{ssd} = \frac{N_{ssd}}{Th_{ssd} \times Q} + \alpha$ and $T_{pcie} = \frac{N_{ssd}+N_{mem}}{Th_{pcie}}$. T_{pcie} is the maximal PCIe time to access all data.

Problem Solving. To address the optimization of system TPC, Hyperion employs a nonlinear programming (NLP) approach. The objective is to maximize the system's TPC. This problem formulation includes a constraint (*optional*) that the total cost does not exceed a predefined budget and the SSD number does not exceed the maximal available PCIe slots detected by Hyperion. Hyperion iteratively adjusts Q to find the maximum of the TPC while respecting the defined constraints.

Hint to Maximize TPC. Hyperion provides users a hint on how to adjust a single component, e.g., SSD, to maximize TPC and provides the predicted *TPC-pred*[10]. Hyperion allows users to decide how to adjust hardware if running multiple GNN models and datasets. Hyperion provides the predicted value of three metrics T_{ssd}, T_{pcie}, and T_{train} for a given task to identify system bottlenecks: 1) SSD bound, 2) PCIe bound, and 3) GPU bound, which are *positively correlated* to the optimal TPC. Thus users can decide their hardware based on these metrics for more GNN models and datasets.

IV. EVALUATION

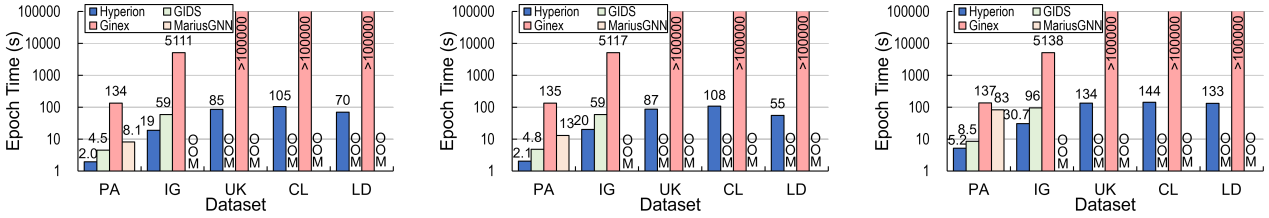
A. Experimental Setting

Experimental Platform. Table I illustrates the evaluation platforms including two single machines and one eight-machine cluster.

GNN Models. We use three sampling-based GNN models: GAT [75], GraphSAGE [29], and GCN [37]. All models adopt a 2-hop random neighbor sampling by default. The sampling fan-outs are 25 and 10. The hidden dimension of GAT is set to 64 and the head number of each layer is set to 8. The hidden dimensions of GraphSAGE and GCN are set to 256. Similar to existing work [70], [86], the batch size is set to 8000 by default. Node classification is used as the GNN task.

Datasets. We conduct our experiments on multiple real-world graph datasets with various scales. Table II shows the dataset characteristics. The Paper100M (PA) is available in Open Graph Benchmark [30]. The IGB-HOM (IG) is from the IGB dataset [35]. The UK-2014 (UK), and Clue-web (CL)

[10]If users provide multiple types of SSDs or GPUs, Hyperion can also give a hint on which combinations of hardware are more cost-efficient.



(a) GraphSAGE (b) GCN (c) GAT

Figure 8. End-to-end Throughput of Hyperion, GIDS, MariusGNN, and Ginex.

Table II
DETAILED INFORMATION OF DATASETS

Dataset	PA	IG	UK	CL	LD
Vertices Num.	111M	269M	0.79B	1B	5.6B
Edges Num.	1.6B	4B	47.2B	42.5B	10B
Topo. Size	14GB	34GB	384GB	348GB	125GB
Feat. Dim.	128	1024	1024	1024	1024
Feat. Size	56GB	1.1TB	3.2TB	4.1TB	23TB

are from WebGraph [10]–[13]. The LDBC-SNB-Bi-SF3000 (LD) is available at the LDBC social network benchmarks [6]. Because UK, CL, and LD have no feature, we manually generate the features with the dimension specified as 1024, following IG's setting. Similar to [86]'s setting, we randomly choose 1% of vertices from each graph as training vertices. We also adapt the feature dimension of PA, IG, and UK for experiments in section III-B2 IV-B and IV-C. We mark these variants as Name-Dimension. For instance, PA-1024 refers to a variant of PA whose feature dimension is 1024.

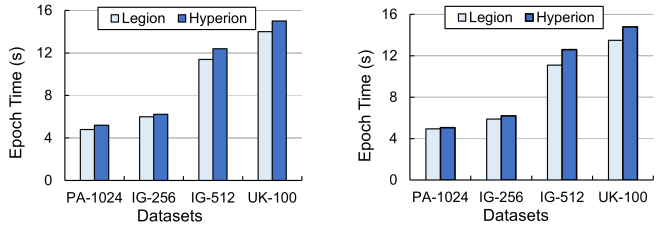
Baselines. We use the state-of-the-art out-of-core GNN systems, Ginex [57], MariusGNN [78] and GIDS [56] and distributed system DistDGL [100] as the baseline systems.

B. Comparison of End-to-end Throughput

Comparing to Out-of-core Baselines. We compare the throughput of Hyperion with all out-of-core three baselines [56], [57], [78] in machine A. We report the average training epoch time of all compared systems on all datasets in Table II and three GNN models, as illustrated by Figure 8. We use 12 P5510 SSDs [65] to store the datasets. We set the number of CPU threads to 96 for CPU-managed baselines Ginex and MariusGNN.

We first examine the average epoch time of each system on terabyte-scale datasets (IG, UK, CL, LD). We observe that Hyperion outperforms GIDS by up to 3.1×. Moreover, GIDS runs out of GPU memory on larger UK, CL, and LD datasets, because BaM [60] requires over 80GB metadata in the page cache design. MariusGNN runs out of memory due to large memory consumption during pre-processing on these datasets. Compared to CPU-managed baseline Ginex, Hyperion achieves over 167× speedup.

On the smallest dataset PA, all systems can store all topology and features in CPU memory. In this case, Hyperion outperforms GIDS, MariusGNN, and Ginex by up to 2.3×, 15.9×, and 68×, respectively, indicating Hyperion still has superior throughput on small graphs.



(a) GraphSAGE (b) GCN

Figure 9. Epoch Time: Hyperion vs. In-memory System Legion.

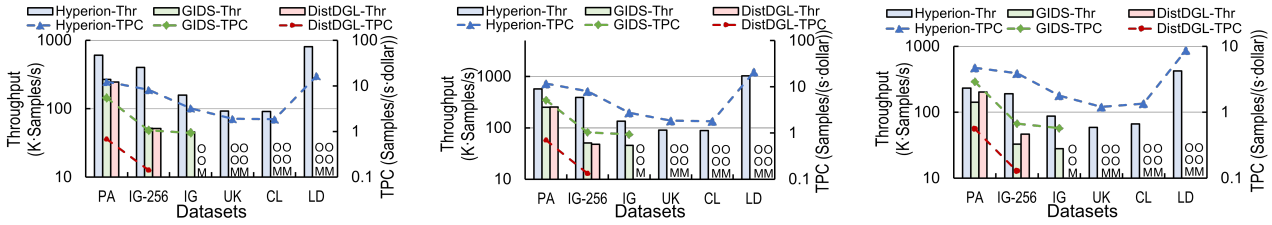
Comparing to In-memory System. We compare Hyperion with SOTA in-memory system Legion [70] to show that Hyperion can achieve in-memory-like training throughput by adding cheap NVMe SSDs. We evaluate in machine A with 12× P5510 SSDs. We report two GNN models due to space limitations. We evaluate on four large-scale datasets in Table II of up to 700 GB by adapting the feature dimension under the constraint of CPU memory (768 GB). For example, PA-1024 represents adapts the feature dimension of original PA datasets to 1024. For both systems, we fill all the available GPU memory as the cache. Hyperion only utilizes 128 GB CPU memory as the cache while Legion stores all the feature/topology data in CPU memory.

Figure 9 illustrates the average epoch time of Hyperion compared to Legion. We observe that Hyperion can achieve 88%~97% throughput of Legion with both GNN models, indicating that out-of-core training with Hyperion can achieve close training throughput to in-memory systems.

C. Comparison of End-to-end TPC

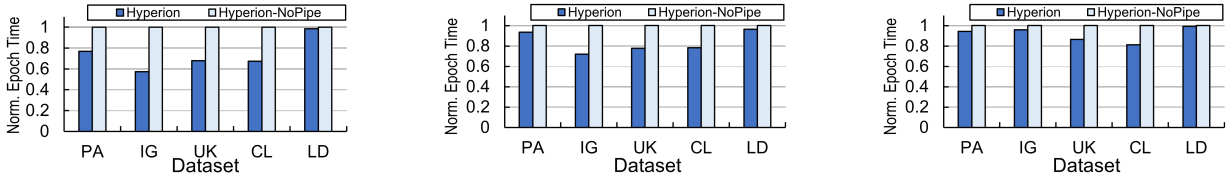
We compare the end-to-end throughput and TPC of Hyperion with out-of-core and distributed baselines[11]. We calculate the TPC based on five-year TCO (See § VI and Table I). Hyperion and GIDS run on machine A with 12 P5510 SSDs and DistDGL runs on cluster C using 8 machines. As DistDGL requires CPU to execute distributed sampling, we maximize the CPU thread number per machine to 48. We measure the network utilization of DistDGL by Intel PCM [32] and find that DistDGL only reaches 20Gbps peak network throughput so DistDGL would not be bottlenecked by the PCIe 3.0 bandwidth of cluster C. Because DistDGL runs out of memory on IG due to memory overhead (we find DistDGL allocates

[11]For out-of-core baselines, we only the report GIDS because Figure 8 proves Ginex and MariusGNN are significantly slower than GIDS and Hyperion in the same machine.



(a) GraphSAGE (b) GCN (c) GAT

Figure 10. End-to-end Throughput and TPC of Hyperion, GIDS, and DistDGL. X-Thr and X-TPC present the system throughput and TPC, respectively.



(a) GraphSAGE (b) GCN (c) GAT

Figure 11. Impact of Deep GNN-aware Pipeline.

about 5× memory compared to the original dataset size), we also evaluate IG-256 whose feature dimension is 256.

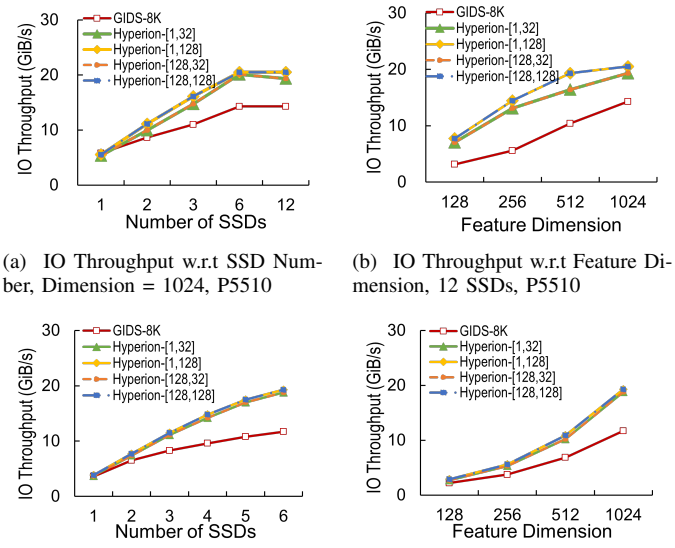
Figure 10 demonstrates that Hyperion outperforms the TPC of GIDS by up to 7.7× and outperforms DistDGL by up to 60×, indicating that Hyperion can maximize TPC for GNN training. Hyperion achieves higher throughput than DistDGL because DistDGL's CPU-based distributed graph sampling is less efficient than GPU-based sampling in Hyperion [23], [59], [86]. Though DistDGL can achieve higher throughput than Hyperion with more machines, the TPC of DistDGL can hardly increase because the monetary cost grows linearly with machine number. We evaluate on the PA dataset using the three GNN models and observe that the throughput of DistDGL scales nearly linearly with 1 to 8 machines, reaching 87% of Hyperion's throughput when using 8 machines. Though DistDGL would achieve higher throughput than Hyperion with over 8 machines, Hyperion can consistently outperform the TPC of DistDGL by over 18×.

D. Impact of Hyperion's Disk IO Stack

In this experiment, we evaluate the effect of GPU-initiated pipeline-friendly asynchronous disk IO stack. There are three evaluation metrics: 1) the impact of our IO stack on the GNN pipeline, 2) the achieved disk IO throughput, and 3) the used ratio of GPU computation resources of our design.

1) Impact on the GNN Pipeline:
To examine the impact of Hyperion's pipeline-friendly asynchronous IO stack on the GNN pipeline, our comparison introduces Hyperion-NoPipe that launches all GPU cores for all stages in the GNN training process and executes all GPU-initiated operators in a serial order. We conduct the experiments on machine A with 12 P5510 SSDs.

Figure 11 illustrates the comparison results under three GNN models and all datasets. We observe that Hyperion outperforms Hyperion-NoPipe by up to 1.74× on GraphSAGE, up to 1.39× on GCN, and up to 1.23× on GAT, because Hyperion can overlap GNN computation and disk IO accesses.

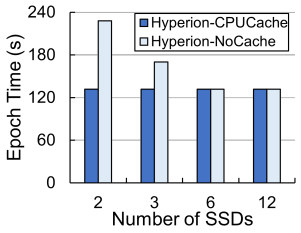


(a) IO Throughput w.r.t SSD Number, Dimension = 1024, P5510 (b) IO Throughput w.r.t Feature Dimension, 12 SSDs, P5510

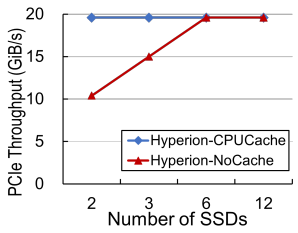
(c) IO Throughput w.r.t SSD Number, Dimension = 1024, 980pro (d) IO Throughput w.r.t Feature Dimension, 6 SSDs, 980pro

Figure 12. Comparison of Overall IO Throughput and GPU Core Utilization of Hyperion and GIDS. Hyperion-[x,y] represents utilizing x and y thread blocks for the IO submission and completion kernels, respectively. GIDS-8K means that GIDS utilizes 8K thread blocks for the IO kernel.

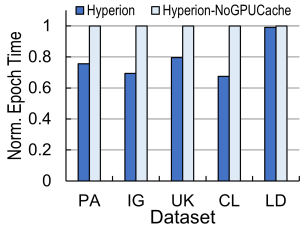
2) Achieved IO Throughput and GPU Utilization:
Experimental Setting. We examine the achieved disk IO throughput and the used ratio of GPU computation resources on machine A with two different kinds of SSDs, i.e., Intel P5510 and Samsung 980pro. GIDS exhausts all GPU cores for IO kernels (8K thread blocks, 1024 threads per block, 100% GPU core utilization). For a fair comparison, Hyperion starts the IO completion kernel immediately after the IO submission kernel and records the IO throughput for the overall process. For the IO submission kernel, we set the GPU thread block number to 1 and 128, which is about 1% and 100% GPU core utilization. For the IO completion kernel, we set the thread block number to 32 and 128, which is about 30% and 100% GPU core utilization, respectively. On P5510 (or 980pro), we fix the feature dimension to 1024 and vary the SSD numbers



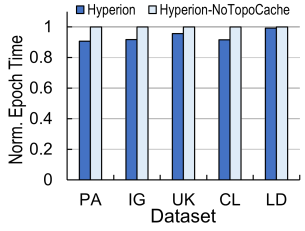
(a) Impact of CPU Cache on Epoch Time



(b) Impact of CPU Cache on GPU PCIe Throughput



(c) Impact of GPU Cache.



(d) Impact of Topology Cache.

Figure 13. Impact of Unified Cache.

from 1 to 12 (or 6), as shown in Figure 12(a) (or Figure 12(c)). Besides, we fix the SSD number to 12 (6) and vary the feature dimension from 128 to 1024, as illustrated by Figure 12(b) (or Figure 12(d)).

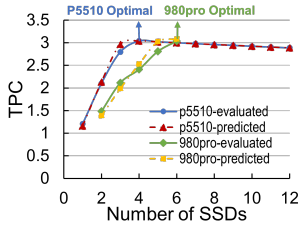
Overall IO Throughput. Figure 12 shows that Hyperion's IO stack outperforms GIDS's IO stack under different numbers of SSDs and feature dimensions, because the Hyperion's IO stack design can submit sufficient parallel IO requests to maximize the disk IO throughput, as discussed in § III-A. As shown in Figure 12(a), the IO throughput of Hyperion from 6 to 12 SSDs is almost the same due to the saturation of PCIe 4.0.

GPU Core Utilization. Figure 12 also proves that Hyperion only needs 1 thread block (1% GPU cores) for IO submission kernels to achieve a comparable IO throughput with 128 thread blocks (all GPU cores). Meanwhile, the IO completion kernel only needs 32 thread blocks (30% GPU cores) to reach an almost maximal throughput. The slight overhead is the data movement from the IO stack's internal buffer to the output feature buffer. In conclusion, the IO stack design enables Hyperion to maximize the disk IO throughput while leaving the majority of GPU cores for other useful GNN kernels, rather than waiting for the completion of IO commands.

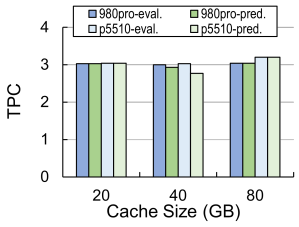
E. Impact of Hyperion's Cache

Hyperion proposes a unified cache that takes topology/feature and CPU/GPU memory into account. We evaluate the impact of components in the unified cache: CPU cache, GPU cache, and topology cache.

Impact of CPU Cache. To illustrate the impact of CPU cache, we propose two implementations: Hyperion-CPUCache and Hyperion-NoCache. Hyperion-CPUCache reads features from both the CPU cache and SSDs, while Hyperion-NoCache reads features only from SSDs. We disable GPU caches and maintain CPU cache for all graph topology data in both implementations. We use all the available CPU memory for



(a) TPC w.r.t. SSD number



(b) TPC w.r.t. Cache Size

Figure 14. Case Study of IO bound.

the CPU feature. Figures 13 show the evaluation results on a terabyte-scale CL dataset (4.1TB). We vary the number of P5510 SSDs from 2 to 12. Figure 13(a) illustrates that Hyperion-CPUCache outperforms Hyperion-NoCache by up to 1.73× and maintains a stable throughput even with only two SSDs. The underlying reason is that Hyperion-CPUCache reaches the maximal PCIe throughput under different numbers of SSDs, as shown in Figure 13(b).

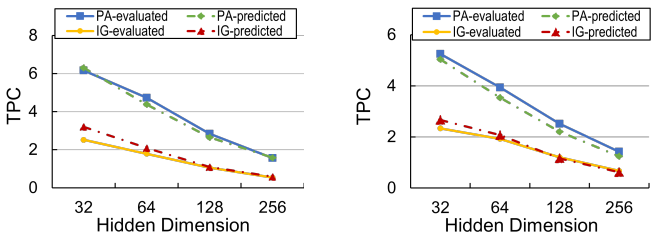
Impact of GPU Cache. We examine the impact of GPU cache on different datasets with 12× P5510 SSDs. The GPU cache sizes are set to all the GPU memory except other GPU buffers like the GNN model and mini batch buffers while the CPU cache is set to all the available CPU memory. Figure 13(c) illustrates that Hyperion with GPU cache (Hyperion) can achieve speedup up to 1.48×, compared to Hyperion without GPU cache.

Impact of Topology Cache. We examine the impact of the topology cache. We use 12× P5510 SSDs. We compare Hyperion to Hyperion-NoTopoCache. For both systems, we use all the available GPU/CPU memory for cache. For Hyperion-NoTopoCache, we fill the CPU cache with feature data and access topology data from SSDs. Figure 13(d) illustrates that Hyperion outperforms Hyperion-NoTopoCache by up to 1.1× because accessing topology data from SSDs incurs more IO amplification compared to features.

F. Validation of TPC-analytical Model

We validate the predicting accuracy of the TPC-analytical model on various hardware combinations and different GNN models. We comprehensively examine the TPC-prediction under IO bound cases and computation bound cases.

Case study: IO Bound. We examine the TPC of GraphSAGE training on machine A with 1 to 12 P5510 SSDs and 2 to 6 980pro SSDs. In this series of experiments, GraphSAGE model training can be overlapped with PCIe IO, representing IO-bound cases of GNN training. First, we fix the GPU/CPU cache sizes to 40GB/20GB and report the predicted/evaluated TPC with different numbers of SSDs. Figure 14(a) demonstrates that Hyperion can precisely predict the TPC with different SSD numbers and can find the optimal numbers of SSDs that maximize the TPC. Second, we fix the GPU cache size and vary the CPU cache from 20 to 80 GB. Under each cache setting, Hyperion will predict an optimal number of SSDs that maximizes TPC and we also manually evaluate the optimal number of SSDs. Then we compare the predicted optimal TPC under Hyperion's suggestion to the manually



(a) Machine A, PCIe 4.0, A100 (b) Machine B, PCIe 5.0, H800

Figure 15. Case Study of Computation Bound. TPC of GAT model with Various Model Sizes on Machines A and B.

evaluated optimal TPC in each cache setting. Figure 14(b) shows that Hyperion can select a suitable number of SSDs that achieves over 90% TPC to the evaluated ones.

Case Study: Computation Bound. We examine the TPC of GAT training on machines A and B. We use 12× P5510 SSDs in both machines. We vary the model sizes of GAT, i.e., hidden dimensions, from 32 to 256. From hidden dimensions 64 to 256, the GNN training is bounded by the model computation. We report two datasets: PA and IG. Figure 15 shows that Hyperion can precisely predict the TPC in different machines with different GPUs and PCIe throughput.

V. Related Work

Large-scale GNN Systems. Distributed systems [19], [21], [26], [41], [44], [58], [67], [69], [73], [79], [81], [82], [84], [89], [90], [93], [94], [96], [98]–[101], [103] leverage expensive multiple machines' host memory and GPU memory to store large-scale graphs. Existing out-of-core systems [56], [57], [78] use SSD to achieve cheap training but have low training throughput due to CPU overhead or GPU core contention. OUTRE [63] proposes batch construction and historical embedding cache to reduce data redundancy. In contrast, Hyperion fully overlaps the GNN pipeline and SSD access, achieving the highest TPC and accurate training.

GPU Direct SSD Access with CPU Involvement. Existing works [8], [15], [34], [38], [47], [61], [74], [95] rely on CPU to initiate or trigger SSD access and enable direct GPU-SSD data transfer using the GPUDirect [52] technology. Systems [64], [76] enable GPU to send SSD access requests on demand and rely on CPU to initiate/trigger SSD access but do not support GPUDirect [52]. However, the involvement of the CPU in the control path leads to high CPU-GPU synchronization overheads, I/O traffic amplification, and long CPU processing latencies [60]. These overheads especially make it hard to saturate SSD throughput for data-dependent, irregular, and fine-grained access patterns of graph workloads.

GPU Direct SSD Access without CPU Involvement. BaM [60] proposes GPU-initiated on-demand direct SSD access without CPU involvement. However, BaM's design introduces new GPU core contention issues, failing to overlap IO with computation. In contrast, Hyperion's asynchronous IO stack only requires 1% GPU cores to saturate SSDs and allow IO/computation overlap.

Cache-enabled GNN Systems. Many prior systems [31], [43], [44], [49], [56], [57], [59], [70], [72], [86], [97] have explored cache design to accelerate GNN training. Some of them [44], [56], [57] propose dynamic cache design, which faces challenges for asynchronous IO design due to cache/IO interference. Others [43], [49], [59], [70], [86], [97] adopt static cache design but do not take the entire CPU-GPU memory hierarchy to cache both topology and features. Some other out-of-core systems [39], [51], [77] utilize the entire memory hierarchy to minimize PCIe transactions but do not focus on GNN training. In contrast, Hyperion disaggregates cache from disk IO and fully utilizes the entire memory hierarchy by considering the GNN access pattern to minimize PCIe transactions for out-of-core GNN training.

VI. Calculation of Monetary Cost

Table I shows the estimation of the 5-year total cost of ownership (TCO) of a local machine/cluster. The overall TCO is estimated as Equation 5, where C_{mac}, C_{gpu}, C_{ssd} and C_{ele} represent the cost of a single host machine, GPU, and SSD, and the electricity per machine. N_{gpu}, N_{ssd}, and N_{mac} represent the number of GPUs and SSDs per host machine and the number of host machines. We provide the cost of each individual component. C_{mac} is estimated as $14,098 using the price of machine A collected from its provider [71]. For the electricity cost (C_{ele}), we estimate the electricity price to be 10 cents per kWh and the power consumption of a single machine to be 4,000 Watts, assuming the machine runs GPU workloads continuously in its lifecycle ($17,000 in total). The prices of GPUs and SSDs are gathered from Amazon [3], [4]. The prices (C_{gpu}) of an 80GB A100 and H800 PCIe GPU are $14,177 and $44,000. The prices ($C_{ssd}$) of a P5510 and 980 pro SSD are $308 and $100. **Though the absolute number of costs could vary over time and location, a single component, e.g., SSDs, can be much cheaper than an entire machine.**

$$TCO = (C_{mac} + C_{gpu} \times N_{gpu} + C_{ssd} \times N_{ssd} + C_{ele}) \times N_{mac} \quad (5)$$

VII. Conclusion

In this work, we argue that co-optimizing GPU-initiated asynchronous SSD access and GNN computation pipeline enables us to only add cheap NVMe SSDs, rather than expensive GPU servers, to achieve in-memory-like throughput and thus maximal TPC of GNN training. We propose Hyperion, the first to propose a GPU-initiated asynchronous disk IO stack with *pipeline-friendliness* without CPU involvement and the first to propose the GPU-managed, *disaggregated*, *unified* cache for out-of-core GNN training. We propose the TPC-analytical model to guide users on how to select their hardware with a limited budget. Hyperion is open sourced at https://github.com/RC4ML/Hyperion.

Acknowledgement. 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 number (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] Anshul Ahluwalia, Rohit Das, Payman Behnam, Alind Khare, Pan Li, and Alexey Tumanov. Abkd: Graph neural network compression with attention-based knowledge distillation. *In arxiv*, 2023.

[2] Zhiyuan Ai, Mingxing Zhang, Yongwei Wu, Xuehai Qian, Kang Chen, and Weimin Zheng. Squeezing out all the value of loaded data: An out-of-core graph processing system with reduced disk {I/O}. In *ATC*, 2017.

[3] Amazon. Intel D7-P5510 3.84 TB Solid State Drive from Amazon. https://www.amazon.sg/dp/B08R3XKK5H?ref_=mr_referred_us_sg_sg, 2024.

[4] Amazon. NVIDIA Tesla PCIe A100 from Amazon. https://www.amazon.sg/NVIDIA-Tesla-A100-Ampere-Graphics/dp/B08X13X6HF, 2024.

[5] Amazon Web Service. Amazon EC2 R6id Instances with NVMe Local Instance Storage of up to 7.6 TB. https://aws.amazon.com/cn/blogs/aws/new-amazon-ec2-r6id-instances/, 2024.

[6] Renzo Angles, János Benjamin Antal, Alex Averbuch, Altan Birler, Peter Boncz, Márton Búr, Orri Erling, Andrey Gubichev, Vlad Haprian, Moritz Kaufmann, Josep Lluís Larriba Pey, Norbert Martínez, József Marton, Marcus Paradies, Minh-Duc Pham, Arnau Prat-Pérez, David Püroja, Mirko Spasić, Benjamin A. Steer, Dávid Szakállas, Gábor Szárnyas, Jack Waudby, Mingxi Wu, and Yuchen Zhang. The ldbc social network benchmark. *In arxiv*, 2020.

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

[8] Shai Bergman, Tanya Brokhman, Tzachi Cohen, and Mark Silberstein. Spin: Seamless operating system integration of peer-to-peer dma between ssds and gpus. *In TOCS*, 2019.

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

[10] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. Ubicrawler: A scalable fully distributed web crawler. *In Software: Practice & Experience*, 2004.

[11] Paolo Boldi, Andrea Marino, Massimo Santini, and Sebastiano Vigna. Bubing: Massive crawling for the masses. In *WWW*, 2014.

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

[13] Paolo Boldi and Sebastiano Vigna. The web graph framework: Compression techniques. In *WWW*, 2004.

[14] Fedor Borisyuk, Shihai He, Yunbo Ouyang, Morteza Ramezani, Peng Du, Xiaochen Hou, Chengming Jiang, Nitin Pasumarthy, Priya Bannur, Birjodh Tiwana, et al. Lignn: Graph neural networks at linkedin. In *SIGKDD*, 2024.

[15] Tanya Brokhman, Pavel Lifshits, and Mark Silberstein. Gaia: An os page cache for heterogeneous systems. In *ATC*, 2019.

[16] Jie Chen, Tengfei Ma, and Cao Xiao. Fastgcn: Fast learning with graph convolu-tional networks via importance sampling. In *ICLR*, 2018.

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

[18] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard Spillane, Amy Tai, and Rob Johnson. Splinterdb: closing the bandwidth gap for nvme key-value stores. In *ATC*, 2020.

[19] Gunduz Vehbi Demirci, Aparajita Haldar, and Hakan Ferhatosmanoglu. Scalable graph convolutional network training on distributed-memory systems. *In VLDB*, 2022.

[20] Matthias Fey and Jan Eric Lenssen. Fast graph representation learning with pytorch geometric. *In arxiv*, 2019.

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

[22] Aishwarya Ganesan, Ramnatthan Alagappan, Anthony Rebello, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Exploiting nil-external interfaces for fast replicated storage. *In TOCS*, 2022.

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

[24] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, 2012.

[25] Google Cloud. About Local SSD Disks. https://cloud.google.com/compute/docs/disks/local-ssd, 2024.

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

[27] H3Platform. H3Platform. https://www.h3platform.com, 2024.

[28] Gabriel Haas and Viktor Leis. What modern nvme storage can do, and how to exploit it: High-performance i/o for high-performance storage engines. *In VLDB*, 2023.

[29] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. *In NeurIPS*, 2017.

[30] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open graph benchmark: Datasets for machine learning on graphs. *In NeurIPS*, 2020.

[31] Kezhao Huang, Haitian Jiang, Minjie Wang, Guangxuan Xiao, David Wipf, Xiang Song, Quan Gan, Zengfeng Huang, Jidong Zhai, and Zheng Zhang. Freshgnn: Reducing memory access via stable historical embeddings for graph neural network training. *In VLDB*, 2024.

[32] Intel. PCM. https://github.com/intel/pcm, 2022.

[33] Abhinav Jangda, Sandeep Polisetty, Arjun Guha, and Marco Serafini. Accelerating graph sampling for graph machine learning using gpus. In *Eurosys*, 2021.

[34] Min-Gyo Jung, Chang-Gyu Lee, Donggyu Park, Sungyong Park, Jungki Noh, Woosuk Chung, Kyoung Park, and Youngjae Kim. Gpukv: an integrated framework with kvssd and gpu through p2p communication support. In *SAC*, 2021.

[35] Arpandeep Khatua, Vikram Sharma Mailthody, Bhagyashree Taleka, Tengfei Ma, Xiang Song, and Wen-mei Hwu. Igb: Addressing the gaps in labeling, features, heterogeneity, and size of public graph datasets for deep learning research. In *SIGKDD*, 2023.

[36] Kevin Kiningham, Philip Levis, and Christopher Ré. Greta: Hardware optimized graph processing for gnns. In *ReCoML 2020*, 2020.

[37] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *ICLR*, 2017.

[38] Jing Li, Hung-Wei Tseng, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. Hippogriffdb: Balancing i/o and gpu bandwidth in big data analytics. *In VLDB*, 2016.

[39] Jing Li, Hung-Wei Tseng, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. Hippogriffdb: Balancing i/o and gpu bandwidth in big data analytics. *In VLDB*, 2016.

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

[41] 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. Hyperscale fpga-as-a-service architecture for large-scale distributed graph neural network. In *ISCA*, 2022.

[42] Changyue Liao, Mo Sun, Zihan Yang, Jun Xie, Kaiqi Chen, Binhang Yuan, Fei Wu, and Zeke Wang. Lohan: Low-cost high-performance framework to fine-tune 100b model on a consumer gpu. In *ICDE*, 2025.

[43] Zhiqi Lin, Cheng Li, Youshan Miao, Yunxin Liu, and Yinlong Xu. Pagraph: Scaling gnn training on large graphs via computation-aware caching. In *SoCC*, 2020.

[44] Tianfeng Liu, Yangrui Chen, Dan Li, Chuan Wu, Yibo Zhu, Jun He, Yanghua Peng, Hongzheng Chen, Hongzhi Chen, and Chuanxiong Guo. Bgl:gpu-efficient gnn training by optimizing graph data i/o and preprocessing. In *NSDI*, 2023.

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

[46] Mark Harris. Unified Memory for CUDA Beginners. https://developer.nvidia.com/blog/unified-memory-cuda-beginners/, 2017.

[47] Pak Markthub, Mehmet E Belviranli, Seyong Lee, Jeffrey S Vetter, and Satoshi Matsuoka. Dragon: breaking gpu memory capacity limits with direct nvm access. In *SC*, 2018.

[48] Seung Won Min, Vikram Sharma Mailthody, Zaid Qureshi, Jinjun Xiong, Eiman Ebrahimi, and Wen-mei Hwu. Emogi: Efficient memory-access for out-of-memory graph-traversal in gpus. *In VLDB*, 2020.

[49] Seung Won Min, Kun Wu, Mert Hidayetoglu, Jinjun Xiong, Xiang Song, and Wen-mei Hwu. Graph neural network training and data tiering. In *SIGKDD*, 2022.

[50] Seung Won Min, Kun Wu, Sitao Huang, Mert Hidayetoğlu, Jinjun Xiong, Eiman Ebrahimi, Deming Chen, and Wen Mei Hwu. Large graph convolutional network training with gpu-oriented data communication architecture. *In VLDB*, 2021.

[51] Hamish Nicholson, Aunn Raza, Periklis Chrysogelos, and Anastasia Ailamaki. Hetcache: Synergising nvme storage and gpu acceleration for memory-efficient analytics. In *CIDR*, 2023.

[52] NVIDIA. NVIDIA GPUDirect. https://developer.nvidia.com/gpudirect, 2011.

[53] NVIDIA. Nvidia nsight systems, 2018.

[54] NVIDIA. MULTI-PROCESS SERVICE. https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf, 2020.

[55] Santosh Pandey, Lingda Li, Adolfy Hoisie, Xiaoye S Li, and Hang Liu. C-saw: A framework for graph sampling and random walk on gpus. In *SC*, 2020.

[56] Jeongmin Brian Park, Vikram Sharma Mailthody, Zaid Qureshi, and Wen-mei Hwu. Accelerating sampling and aggregation operations in gnn frameworks with gpu initiated direct storage accesses. *In VLDB*, 2024.

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

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

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

[60] 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. Gpu-initiated on-demand high-throughput storage access in the bam system architecture. In *ASPLOS*, 2023.

[61] Sagi Shahar, Shai Bergman, and Mark Silberstein. Activepointers: a case for software address translation on gpus. *Comput. Archit. News*, 2016.

[62] Zhaoyan Shen, Feng Chen, Gala Yadgar, Zhiping Jia, and Zili Shao. Prism-ssd: a flexible storage interface for ssds. *In TCAD*, 2021.

[63] Zeang Sheng, Wentao Zhang, Yangyu Tao, and Bin Cui. Outre: An out-of-core de-redundancy gnn training framework for massive graphs within a single machine. *In VLDB*, 2024.

[64] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. Gpufs: Integrating a file system with gpus. In *ASPLOS*, 2013.

[65] Solidigm. D7-P5510 high-performing, standard-endurance PCIe 4.0 NVMe SSD drive. https://www.solidigm.com/products/data-center/d7/p5510.html, 2021.

[66] Yongju Song, Wook-Hee Kim, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. Prism: Optimizing key-value store for modern heterogeneous storage devices. In *ASPLOS*, 2023.

[67] Zhen Song, Yu Gu, Jianzhong Qi, Zhigang Wang, and Ge Yu. Ec-graph: A distributed graph neural network system with error-compensated compression. In *ICDE*, 2022.

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

[69] Jie Sun, Zuocheng Shi, Li Su, Wenting Shen, Zeke Wang, Yong Li, Wenyuan Yu, Wei Lin, Fei Wu, Jingren Zhou, and Bingsheng He. Helios: Efficient distributed dynamic graph sampling for online gnn inference. In *PPoPP*, 2025.

[70] Jie Sun, Li Su, Zuocheng Shi, Wenting Shen, Zeke Wang, Lei Wang, Jie Zhang, Yong Li, Wenyuan Yu, Jingren Zhou, and Fei Wu. Legion: Automatically pushing the envelope of multi-gpu system for billion-scale gnn training. In *ATC*, 2023.

[71] Supermicro. Supermicro 4U GPU SuperServer. https://store.supermicro.com/us_en/4u-gpu-superserver-as-4125gs-tnrt2.html, 2024.

[72] Zeyuan Tan, Xiulong Yuan, Congjie He, Man-Kit Sit, Guo Li, Xiaoze Liu, Baole Ai, Kai Zeng, Peter Pietzuch, and Luo Mai. Quiver: Supporting gpus for low-latency, high-throughput gnn serving with workload awareness. *In arxiv*, 2023.

[73] John Thorpe, Yifan Qiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Keval Vora, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Dorylus: Affordable, scalable, and accurate nn training with distributed cpu servers and serverless threads. In *OSDI*, 2021.

[74] Hung-Wei Tseng, Yang Liu, Mark Gahagan, Jing Li, Yanqin Jing, and Steven Swanson. Gullfoss: Accelerating and simplifying data movement among heterogeneous computing and storage resources. 2015.

[75] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *ICLR*, 2017.

[76] Ján Veselỳ, Arkaprava Basu, Abhishek Bhattacharjee, Gabriel H Loh, Mark Oskin, and Steven K Reinhardt. Generic system calls for gpus. In *ISCA*, 2018.

[77] Lukas Vogel, Daniel Ritter, Danica Porobic, Pinar Tözün, Tianzheng Wang, and Alberto Lerner. Data pipes: Declarative control over data movement. In *CIDR*, 2023.

[78] Roger Waleffe, Jason Mohoney, Theodoros Rekatsinas, and Shivaram Venkataraman. Mariusgnn: Resource-efficient out-of-core training of graph neural networks. In *Eurosys*, 2023.

[79] Xinchen Wan, Kaiqiang Xu, Xudong Liao, Yilun Jin, Kai Chen, and Xin Jin. Scalable and efficient full-graph gnn training for large graphs. *In SIGMOD*, 2023.

[80] Minjie Yu Wang. Deep graph library: Towards efficient and scalable deep learning on graphs. In *ICLRW*, 2019.

[81] Qiange Wang, Yao Chen, Weng-Fai Wong, and Bingsheng He. Hongtu: Scalable full-graph gnn training on multiple gpus. *In SIGMOD*, 2023.

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

[83] Yuke Wang, Boyuan Feng, Zheng Wang, Guyue Huang, and Yufei Ding. Tc-gnn: Bridging sparse gnn computation and dense tensor cores on gpus. In *ATC*, 2023.

[84] Yaqi Xia, Donglin Yang, Xiaobo Zhou, and Dazhao Cheng. Scaling new heights: Transformative cross-gpu sampling for training billion-edge graphs. In *SC*, 2024.

[85] Yi Xu, Henry Zhu, Prashant Pandey, Alex Conway, Rob Johnson, Aishwarya Ganesan, and Ramnatthan Alagappan. Ionia: High-performance replication for modern disk-based kv stores. In *FAST*, 2024.

[86] Jianbang Yang, Dahai Tang, Xiaoniu Song, Lei Wang, Qiang Yin, Rong Chen, Wenyuan Yu, and Jingren Zhou. Gnnlab: A factored system for sample-based gnn training over gpus. In *Eurosys*, 2022.

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

[88] Minji Yoon, Théophile Gervet, Baoxu Shi, Sufeng Niu, Qi He, and Jaewon Yang. Performance-adaptive sampling strategy towards fast and accurate graph neural networks. In *SIGKDD*, 2021.

[89] Hui Yu, Yu Zhang, Ligang He, Yingqi Zhao, Xintao Li, Ruida Xin, Jin Zhao, Xiaofei Liao, Haikun Liu, Bingsheng He, et al. Rahp: A redundancy-aware accelerator for high-performance hypergraph neural network. In *MICRO*, 2024.

[90] Hui Yu, Yu Zhang, Jin Zhao, Yujian Liao, Zhiying Huang, Donghao He, Lin Gu, Hai Jin, Xiaofei Liao, Haikun Liu, et al. Race: An efficient redundancy-aware accelerator for dynamic graph neural network. *In TACO*, 2023.

[91] Zhongbao Yu, Jiaqi Zhang, Xin Qi, and Chao Chen. Application research of graph neural networks in the financial risk control. 2022.

[92] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. Graphsaint: Graph sampling based inductive learning method. In *ICLR*, 2019.

[93] Juxiang Zeng, Pinghui Wang, Lin Lan, Junzhou Zhao, Feiyang Sun, Jing Tao, Junlan Feng, Min Hu, and Xiaohong Guan. Accurate and scalable graph neural networks for billion-scale graphs. In *ICDE*, 2022.

[94] Dalong Zhang, Xin Huang, Ziqi Liu, Jun Zhou, Zhiyang Hu, Xianzheng Song, Zhibang Ge, Lin Wang, Zhiqiang Zhang, and Yuan Qi. Agl: A scalable system for industrial-purpose graph machine learning. *In VLDB*, 2020.

[95] Jie Zhang, David Donofrio, John Shalf, Mahmut T Kandemir, and Myoungsoo Jung. Nvmmu: A non-volatile memory management unit for heterogeneous gpu-ssd architectures. In *PACT*, 2015.

[96] Meng Zhang, Jie Sun, Qinghao Hu, Peng Sun, Zeke Wang, Yonggang Wen, and Tianwei Zhang. Torchgt: A holistic system for large-scale graph transformer training. In *SC*, 2024.

[97] Xin Zhang, Yanyan Shen, Yingxia Shao, and Lei Chen. Ducati: A dual-cache training system for graph neural networks on giant graphs with the gpu. *In SIGMOD*, 2023.

[98] Chenguang Zheng, Hongzhi Chen, Yuxuan Cheng, Zhezheng Song, Yifan Wu, Changji Li, James Cheng, Hao Yang, and Shuai Zhang. Bytegnn: efficient graph neural network training at large scale. *In VLDB*, 2022.

[99] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. Distdgl: distributed graph neural network training for billion-scale graphs. In *IA3*, 2020.

[100] Da Zheng, Xiang Song, Chengru Yang, Dominique LaSalle, and George Karypis. Distributed hybrid cpu and gpu training for graph neural networks on billion-scale heterogeneous graphs. In *SIGKDD*, 2022.

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

[102] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, et al. Alpa: Automating inter-and {Intra-Operator} parallelism for distributed deep learning. In *OSDI*, 2022.

[103] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. Aligraph: a comprehensive graph neural network platform. *In VLDB*, 2019.