Degree-Based Scheduling and Memory Management for Large-Scale Exact Online GNN Inference

Alireza Namazi*, Haiying Shen*, Tanmoy Sen*, Minjia Zhang†
*University of Virginia
†University of Illinois Urbana-Champaign

***Abstract*—Graph Neural Networks (GNNs) have demonstrated remarkable performance on tasks operating on graph-structured data. Some applications require *exact online inference*, which processes all neighbors rather than sampling a random subset to avoid stochasticity and accuracy degradation, while still meeting tight latency constraints. In this paper, we analyze exact online inference and identify two key challenges: head-of-line blocking caused by heterogeneous request processing times, and highly variable memory usage that complicates cache size optimization for balancing computation and CPU–GPU communication. To reduce the average request latency, we propose GSM, an exact online GNN inference serving system with efficient scheduling and memory management. GSM integrates three main components. The *parameter tuner* optimizes the tradeoff between computation and CPU–GPU communication overhead by finding near-optimal parameters, such as batch in-degree sum and batch subgraph size, for the target workload. The *degree-based scheduler* groups requests with similar in-degrees into batches, ensuring each batch's in-degree sum approaches the optimal value and prioritizing batches with shorter processing times. The *batch divider* partitions a batch subgraph to achieve the optimal subgraph size and improve processing efficiency. Large-scale experiments on graphs with billions of edges (Papers100M and Friendster) demonstrate that GSM achieves up to 89% lower latency and 71% higher throughput compared with state-of-the-art systems. The implementation of GSM is publicly available.**

I. Introduction

Graph Neural Networks (GNNs) are a class of neural networks designed for processing graph-structured data. GNNs have shown strong performance in tasks such as vertex classification [1]–[4], link prediction [5], [6], and graph classification [7], [8], with real-world applications including fraud detection [9], [10], activity recognition [11], [12], and recommender systems [13], [14]. In this paper, we focus on the node property prediction task where clients send requests to a server for the properties of certain vertices, referred to as *target vertices*. The server computes each target vertex's properties by aggregating data from its L-hop neighborhood using an L-hop GNN model [15]. Many of these applications have stringent latency requirements—tens to hundreds of milliseconds for recommender systems [16] and hundreds of milliseconds for action recognition [17]. To meet these latency requirements, research has explored accelerating GNN inference by approximate inference methods. Feature pruning methods [18] reduce vertex feature dimensionality by discarding features that have little effect on the output, while random neighbor sampling [15], [19] lowers the computational load by sampling a subset of neighbors. However, these approaches may degrade accuracy and introduce stochasticity in the output [20], which is unacceptable in accuracy-sensitive applications such as finance [20]. In such cases, inference must process all neighbors to ensure deterministic and accurate predictions, referred to as *exact online GNN inference* [21].

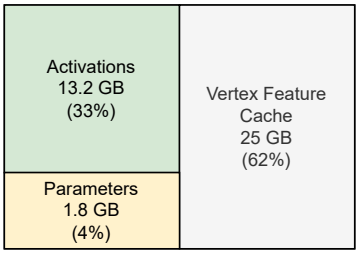


Fig. 1: The memory footprint of inference on the papers100M dataset using a three-layer GraphSAGE model.

Despite its importance, no prior work has addressed achieving low latency in *exact online GNN inference* without modifying the inference computation itself. We analyze exact online inference workloads and identify three major characteristics that degrade the latency. First, processing times for requests vary widely because the number of neighbors per vertex is highly skewed. This causes *head-of-line blocking*, where long-latency requests delay short ones (Observation 1). The in-degree sum of a batch can serve as a simple and effective predictor for batch processing time (Observation 2). Second, memory usage during inference is both high and volatile due to the *neighborhood explosion problem* [22], where the number of vertices within a few hops increases rapidly [23]. Because of this, GPU memory usage fluctuates significantly across batches (Observation 4), complicating cache management and increasing communication overhead between CPU and GPU. Finally, GNN workloads differ widely across datasets and models, requiring different system parameters such as batch size, cache size, and batch subgraph size for optimal performance (Observation 3).

The mentioned characteristics lead to long and unstable inference latency. Addressing these problems raises three key challenges.

Challenge 1: efficiently predict the processing time for an inference request. Head-of-the-line blocking can be mitigated by scheduling requests based on their processing time, but

predicting this in GNNs is difficult. Inference runs in batches for higher throughput and involves three steps: (i) extracting the L-hop neighborhood subgraph of the target vertices from the CPU-stored graph to form the batch subgraph, (ii) preprocessing it into a compact GPU-friendly format, and (iii) transferring it to the GPU for the forward pass. Intuitively, the L-hop neighborhood size strongly influences processing time. A naive approach is to extract each request's subgraph individually to estimate its cost, but this is impractical: extraction is prohibitively slow (Observation 2), and overlapping neighborhoods make it impossible to isolate each target vertex's contribution.

Challenge 2: efficiently utilizing GPU memory during inference. High memory usage restricts the space available for GPU caching and increases CPU–GPU data transfers. As shown in Fig. 1, larger batches reduce queuing delay but consume more activation memory, shrinking the cache and increasing communication overhead. Moreover, due to the skewed degree distribution in real-world graphs [23]–[25], memory usage is highly variable. To prevent out-of-memory errors, memory must be conservatively reserved to accommodate potential memory usage spikes, further limiting effective batch and cache sizes.

Challenge 3: efficiently optimize system parameters. Determining the optimal system parameters (e.g., batch in-degree sum, cache size, and batch subgraph size) requires balancing computation and communication costs. These parameters interact in complex ways, creating a large and interdependent search space that makes manual tuning difficult. For example, inference workloads using GAT models [3] are more compute-intensive than those using GraphSAGE [1]; larger batch subgraphs improve GPU efficiency in such cases, while communication-bound workloads benefit from smaller batches that allow larger cache sizes. As a result, a one-size-fits-all configuration is suboptimal, and systematic parameter tuning is necessary.

To address these challenges and reduce average request latency, we propose GSM—an exact online GNN inference serving system with efficient scheduling and memory management. GSM consists of three main components.

(i) Lightweight request scheduler. Based on our Observation 2 that the in-degree of a request's target vertex can predict the batch processing time, prioritizes requests with lower in-degrees and groups them to achieve an optimal in-degree sum per batch, balancing computation and communication.

(ii) Batch divider. Building on Observation 4 that memory usage surges stem from a few batches, we propose a batch divider that partitions oversized subgraphs when GPU memory is insufficient, reducing and stabilizing memory usage to enable larger caches and batch sizes.

(iii) Parameter tuner. The parameter tuner adapts the in-degree sum and batch subgraph size thresholds to each workload. To improve search efficiency, it performs two one-dimensional searches instead of an exhaustive two-dimensional one, efficiently finding effective thresholds.

Overall, our contributions are as follows:

- We analyze exact online GNN inference on large-scale graphs and identify its main performance challenges.
- We design GSM, a serving system with efficient scheduling and memory management to address these challenges.
- We evaluate GSM on billion-edge graphs and show up to 83% lower latency and 1.6× higher throughput compared with existing exact online GNN inference systems. The implementation of GSM is publicly available [26].

II. Background

A. Graph Neural Networks

Graph Neural Networks (GNNs) integrate neural computation with graph structure to learn vertex representations by propagating information along edges. Each layer updates a vertex embedding through three steps—message computation, aggregation, and activation [27]. For an attributed graph $G = (V, E)$, the l^{th}-layer update for vertex v is:

$$m_{e=(u,v)} = f_m^l(h_u^l, h_v^l, w_e), \tag{1}$$

$$a_v = f_{agg}^l(\{m_{e=(u,v)} | u \in \mathcal{N}_v\}), \tag{2}$$

$$h_v^{l+1} = f_{out}^l(a_v), \tag{3}$$

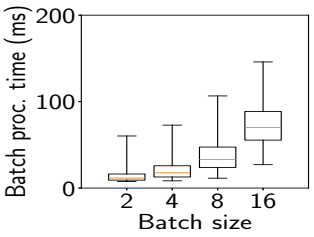
where h_u^l and h_v^l are layer l embeddings for vertices u and v, w_e is the edge feature, and f_m^l, f_{agg}^l, and f_{out}^l are the message, aggregation, and activation functions. The complete forward operation repeats these steps across L layers to obtain the final vertex representation h_v^L.

B. Exact Online Inference Pipeline

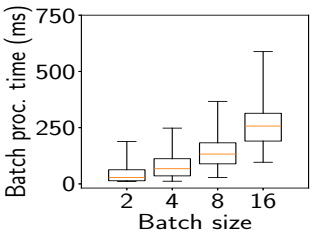
In *exact online inference*, a running service receives queries for individual vertices, called **requests**. Each request asks the system to compute the output of a target vertex using a pretrained L-layer GNN. To improve throughput, multiple requests are grouped into a **batch** that shares GPU resources during inference. All vertices in a batch are processed together, using their combined L-hop neighborhoods—referred to as the *batch subgraph*.

Since large graphs cannot fully reside in GPU memory, both the graph structure and features are stored in CPU memory. The GPU executes inference through three stages [28]:

- **Sampling and layout transformation:** For each batch, the system samples the L-hop neighborhoods of all target vertices to form a subgraph. A preprocessing step, called *layout transformation*, remaps vertex and edge indices from CPU memory addresses to contiguous GPU memory addresses, ensuring efficient access and kernel execution. Sampling proceeds layer by layer.
- **Fetching:** The preprocessed subgraph structure and its vertex features are transferred from CPU to GPU memory. To reduce communication, frequently accessed features—often those of high-degree vertices—are kept in an in-GPU cache [23], [29], [30].
- **Forward operation:** The GPU performs layer-wise message passing and activation over the subgraph to produce outputs for all target vertices.

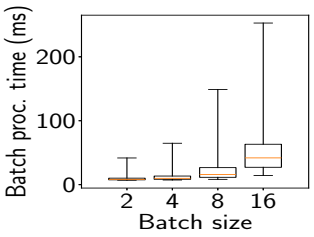


(a) GraphSAGE model.

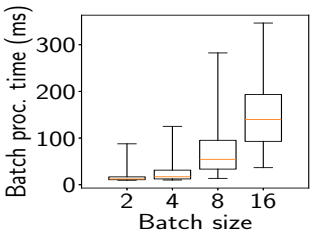


(b) GAT model.

Fig. 2: Batch processing time vs. batch size (papers100M).

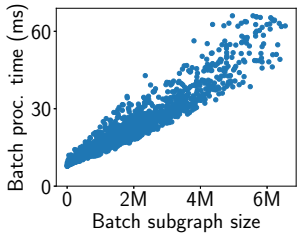


(a) GraphSAGE model.

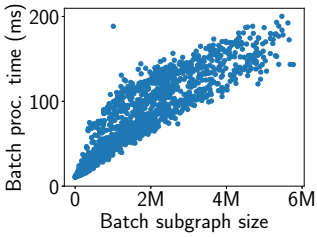


(b) GAT model.

Fig. 3: Batch processing time vs. batch size (Friendster).

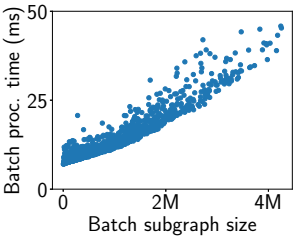


(a) GraphSAGE model.

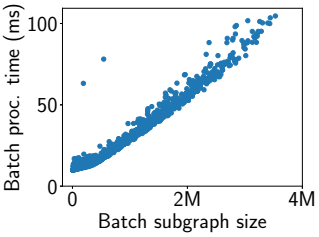


(b) GAT model.

Fig. 4: Batch processing time vs. batch subgraph size (Papers100M).



(a) GraphSAGE model.



(b) GAT model.

Fig. 5: Batch processing time vs. batch subgraph size (Friendster).

Overall, GPU-accelerated GNN inference consists of *preprocessing* (sampling and layout transformation), *fetching*, and *forward computation*. Sampling and fetching are communication-heavy, while layout transformation and forward computation dominate computation. Efficient scheduling and memory management across these stages are critical for achieving low-latency exact online inference.

III. Experimental Analysis

We profile exact online GNN inference to characterize latency variability, memory usage, and low-overhead predictors of processing time. We use two representative GNN models: a three-layer GraphSAGE model [1] with a hidden dimension of 256, and a three-layer GAT model with one attention head and a hidden dimension of 256 [19]. We evaluate on the Papers100M [31] and Friendster [32] datasets (Table I). Following prior work [33], [34], we generate a random 256-dimensional feature vector for each vertex in Friendster.

All experiments run on the University of Virginia HPC cluster using a single NVIDIA A100 GPU (40 GB) attached to an AMD EPYC 7742 CPU (8 cores enabled, 300 GB RAM). The GPU–CPU interconnect is PCIe with a bandwidth of approximately 22 GB/s. We use Python 3.10 with PyTorch 2.1.2, DGL 2.2, and CUDA 12.2.2.

As there is no publicly available exact online GNN inference system, we implement a baseline using DGL [21]. The request queue is FIFO. The in-GPU feature cache follows the PaGraph policy [23] and stores features for vertices with the highest out-degrees. The cache size is determined by running inference without caching, measuring memory usage, and allocating the remaining GPU memory to the cache. A total of 8,000 inference requests are sent with a Poisson arrival rate of 500 requests/s to observe the system under heavy load.

TABLE I: Dataset statistics.

	papers100M	Friendster
# of edges	3.2B	1.8B
# of vertices	111M	65M
Degree (min, avg, max)	(1, 29.0, 25K)	(0, 27.5, 4.2K)
Feature dimension	128	256

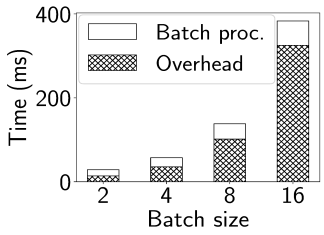
A. Batch Processing Time

Figs. 2 and 3 show the distribution of batch processing times for each dataset and model. The wide spread between the 1st and 99th percentiles indicates high variability: when a long-processing batch leads the queue, it blocks shorter ones, causing head-of-line blocking [35].

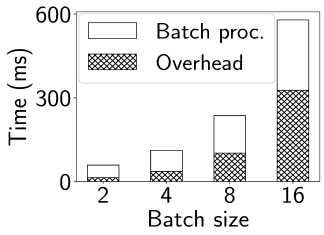
> Observation 1. *Batch processing time in exact online GNN inference is highly variable, making it prone to head-of-line blocking.*

Scheduling shorter batches ahead of longer ones can reduce latency, but doing so requires estimating batch processing times. Figs. 4 and 5 plot processing time versus batch subgraph size, showing an almost linear relationship. Thus, subgraph size can serve as a simple predictor.

A naive scheduling strategy would precompute each request's L-hop neighborhood size, sort requests by these sizes, and

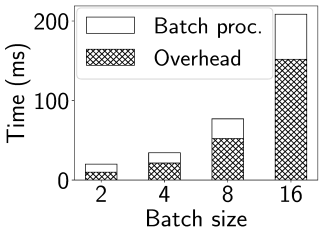


(a) GraphSAGE model.

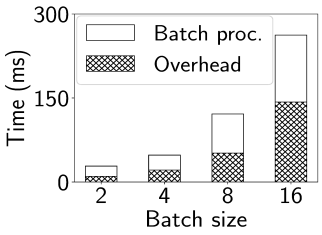


(b) GAT model.

Fig. 6: Time for request processing time estimation and batch processing (Papers100M).

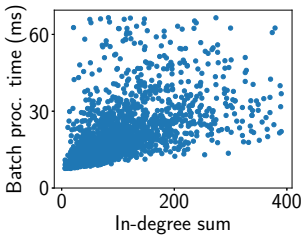


(a) GraphSAGE model.

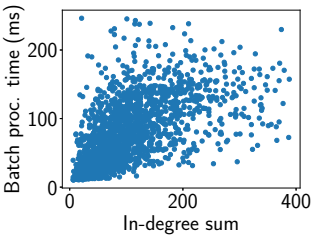


(b) GAT model.

Fig. 7: Time for request processing time estimation and batch processing (Friendster).

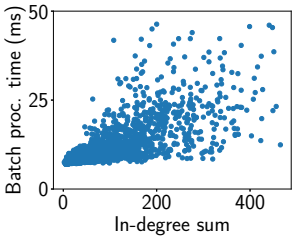


(a) GraphSAGE model.

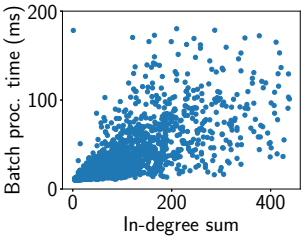


(b) GAT model.

Fig. 8: Batch processing time vs. in-degree sum (papers100M).

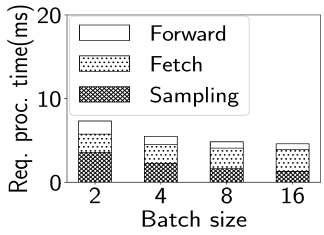


(a) GraphSAGE model.

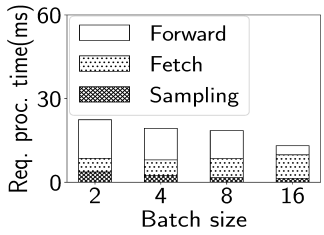


(b) GAT model.

Fig. 9: Batch processing time vs. in-degree sum (Friendster).

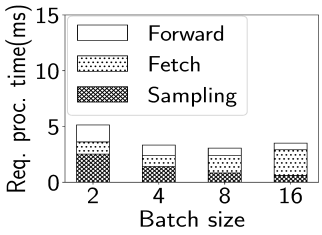


(a) GraphSAGE model.

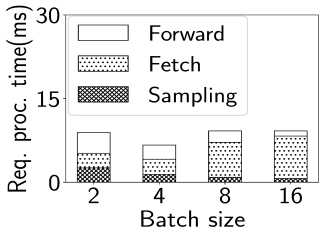


(b) GAT model.

Fig. 10: Request processing time breakdown (papers100M dataset).



(a) GraphSAGE model.



(b) GAT model.

Fig. 11: Request processing time breakdown (Friendster dataset).

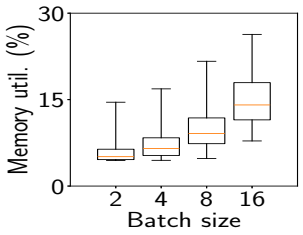
form batches sequentially. However, individually sampling each neighborhood is inefficient. Figs. 6 and 7 compare batch processing time with the overhead of this estimation, showing that the cost is substantial.

We therefore explore lighter-weight predictors. Figs. 8 and 9 show that batch processing time correlates strongly with the in-degree sum of target vertices, suggesting a more efficient proxy for estimating batch cost.

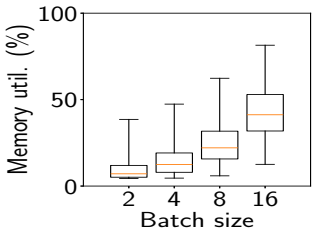
Observation 2. *Estimating batch cost using the in-degree sum is an efficient alternative, as individually sampling each target vertex's neighborhood is computationally infeasible.*

We next decompose batch processing time into sampling, fetch, and forward, defining request processing time as batch time divided by batch size. Figs. 10 and 11 show that as batch size grows, sampling and forward times decrease due to improved GPU utilization, while fetch time increases as larger batches reduce effective cache size. The highest throughput occurs at batch sizes of 16 on Papers100M (both models), 8 for GraphSAGE on Friendster, and 4 for GAT. GAT saturates the GPU with smaller batches because it is more compute-intensive, and Friendster requires smaller batches due to its higher feature dimensionality. These results highlight the need for an adequate cache size to mitigate fetch costs.

Observation 3. *Larger batch sizes reduce sampling and forward time per request but increase fetch time due to cache limitations, necessitating optimal parameter tuning (e.g., batch size, cache size) that varies across GNN workloads to minimize average latency.*

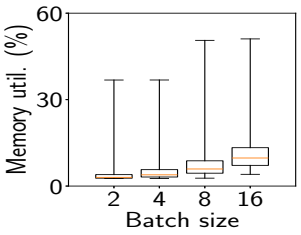


(a) GraphSAGE model.

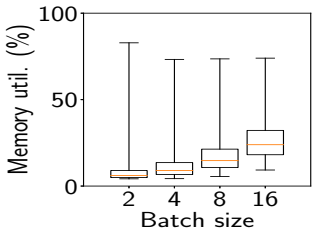


(b) GAT model.

Fig. 12: Memory utilization vs. batch size (papers100M).

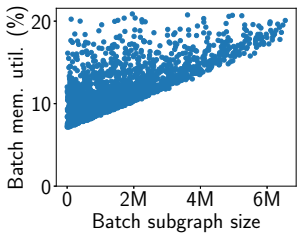


(a) GraphSAGE model.

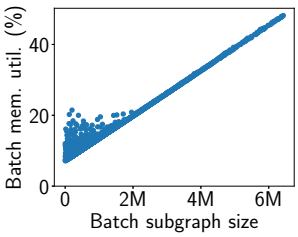


(b) GAT model.

Fig. 13: Memory utilization vs. batch size (Friendster).

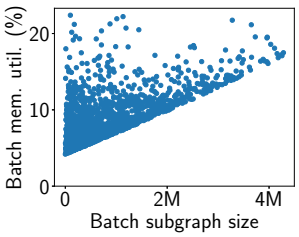


(a) GraphSAGE model.

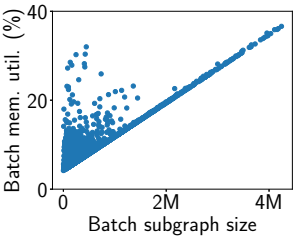


(b) GAT model.

Fig. 14: Memory utilization vs. batch subgraph size (papers100M).



(a) GraphSAGE model.



(b) GAT model.

Fig. 15: Memory utilization vs. batch subgraph size (Friendster).

B. Memory Footprint

GPU memory limits the optimal batch and cache sizes. To examine memory constraints, we conducted experiments without caching and measured memory usage across batch sizes. Figs. 12 and 13 show that memory usage is high, highly variable, and varies across models and datasets.

Figs. 14 and 15 plot batch memory utilization versus batch subgraph size from the same experiment. As batch subgraph size grows, memory usage increases accordingly, since matrices such as vertex features, attention scores, and subgraph representations scale with subgraph size.

> Observation 4. *Memory usage in exact online inference is high and volatile, driven by a few batches, and correlates with batch subgraph size.*

IV. System Design

Building on the observations in §III, we design GSM with three components addressing the main challenges:

- **Light-weight request scheduler** to prevent head-of-the-line blocking (Observation 1) by estimating batch processing time from in-degree (Observation 2).
- **Batch divider** to mitigate GPU memory overload (Observations 3, 4).
- **Parameter tuner** to select the optimal in-degree sum and batch subgraph size thresholds for each workload (Observation 3).

Fig. 16 shows GSM's architecture. The scheduler estimates request cost from the in-degree, sorts requests, and groups them into balanced batches. The batch divider keeps each batch subgraph under a memory threshold by splitting oversized ones into smaller subgraphs processed sequentially. The parameter tuner runs offline to determine the in-degree and subgraph-size thresholds and works with a degree-based caching policy [23] that stores features of high-degree vertices in GPU memory.

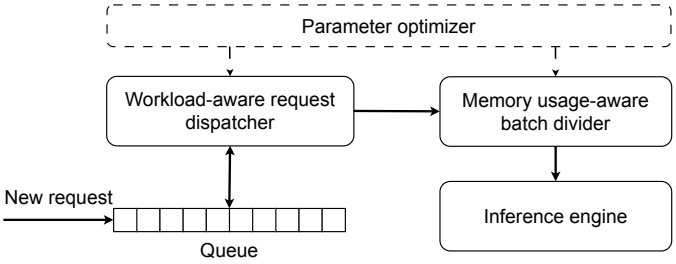


Fig. 16: GSM's architecture.

A. Light-weight Request Scheduler

GSM does not use a fixed batch size. To reduce latency, the scheduler prioritizes requests with shorter expected processing times, which are estimated using their in-degrees, and groups them into batches whose total in-degree sum reaches an optimal threshold. It follows two key principles:

1) **Preventing head-of-the-line blocking:** A first-come, first-served policy causes long-latency requests to block others (Observation 1).
2) **Efficiency:** Requests should be batched into appropriate sizes to minimize latency and avoid out-of-memory (OOM) errors (Observation 3).

Using Observation 2, the scheduler predicts each request's cost from its in-degree, sorts the queue in ascending order, and adds requests sequentially until the cumulative in-degree sum exceeds the threshold T_d. The parameter tuner (see §IV-C) determines T_d, which balances computation and communication: smaller values underutilize the GPU, while larger ones risk excessive memory use.

B. Batch Divider

As noted in Observation 3, feature fetching constitutes a significant portion of batch processing time. Increasing cache size can reduce this; however, high and volatile memory usage (Observation 4) limits feasible cache size and requires reserving extra space to prevent OOM errors. As a result, much of the GPU memory remains unused during inference. We flatten usage peaks with batch division, increasing available cache, and reducing communication load.

The batch divider reduces and stabilizes memory usage by dividing the batch subgraph into smaller subgraphs to reach the desired batch subgraph size. The batch subgraph size determines the batch's memory usage; therefore, controlling it is essential to prevent OOM errors. While memory can be partially controlled through the in-degree sum threshold T_d (Observation 2), it only accounts for the first hop. In deeper hops, vertex counts can grow unexpectedly and cause OOM errors, and some vertices may have very large l-hop neighborhoods that use too much memory alone. Reducing cache size is a naive fix, but it increases communication overhead and slows inference. Batch division instead acts as a safeguard to prevent OOM errors when memory spikes occur. Since memory usage correlates with the number of vertices, we control it by limiting the number of vertices in each layer of the batch subgraph. We set a threshold for the maximum number of vertices (batch subgraph size) and denote it by T_s. The inference proceeds layer by layer. If a layer's subgraph exceeds T_s, the sampler divides it into smaller subgraphs, each with at most T_s vertices. T_s determines the inference memory usage, per-request efficiency, and cache size, and is tuned automatically (see §IV-C). At each layer, the GPU fetches the neighborhood, partitions it according to T_s, performs layout transformation for each subbatch, and allocates the remaining memory for caching. The inference engine processes subbatches separately and concatenates their representations before continuing the forward pass as normal.

C. Parameter Tuner

Before the inference process starts, the tuner determines the following parameters in an offline manner: 1) the in-degree sum threshold (T_d), and 2) the batch subgraph size threshold (T_s). We first explain the tradeoffs between the parameters and how they affect the latency and throughput of the system. Then, we introduce our tuning algorithm.

1) Tradeoff Analysis: The latency for request i, denoted by t_l^i, can be expressed as,

$$\begin{aligned} t_l^i &= t_q^i + t_p^{B(i)} \\ &= t_q^i + (t_s^{B(i)} + t_c^{B(i)} + t_{\mathrm{f}}^{B(i)}). \end{aligned} \tag{4}$$

t_q^i is the queuing time for request i. $t_p^{B(i)}$ is the processing time for the batch that contains the request i, which consists of preprocessing time, $t_s^{B(i)}$, feature fetch time $t_c^{B(i)}$, and forward time $t_{\mathrm{f}}^{B(i)}$.

Increasing T_d makes the batch size larger and takes longer to process due to higher loads in preprocessing, communication, and forward operation. Additionally, larger batch sizes limit the cache size, further increasing the communication load. However, larger batch sizes are more efficient on a per-request basis, given a fixed cache size. Thus, increasing T_d reduces the queuing time, t_q^i; the positive effects of more efficient inference may outweigh the negative effects of a smaller cache size, thus resulting in a higher throughput system that clears the queue faster. The threshold T_s controls the batch division and cache size. A smaller T_s reduces the batch processing efficiency on a per-request basis and adds a division penalty to the sampling time. However, it enables a larger cache size, reducing communication time.

2) Parameter Selection: Considering the size of the search space, an exhaustive search in two dimensions is costly. Therefore, to improve the parameter search efficiency, we conduct two manageable one-dimensional searches over the batch subgraph size threshold (T_s) and the in-degree sum threshold (T_d), respectively.

Under high workloads with high request arrival rates, optimal thresholds are essential for balancing computation and communication. We first select several T_d values empirically. For each T_d, we run dry inference experiments to find the T_s yielding the lowest latency and form pairs (T_d, T_s). For each arrival rate, we then test these pairs and choose the one with the lowest latency. This process is repeated for each GNN model and dataset.

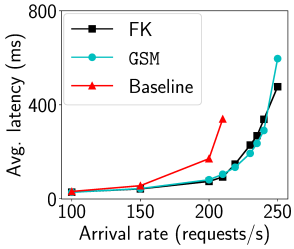
We intuitively limit the search space for the batch subgraph size threshold (T_s) to further speed up the parameter search process. The batch division has a penalty since it increases computation time. The batch divider divides the batch into subgraphs to be processed sequentially. Each step of sequential processing has a lower computational load, which fails to saturate the GPU and increases queuing time. Unlike reducing the in-degree sum, batch division does not shorten batch processing time. Its cost is higher in compute-heavy models, where the forward pass dominates runtime. Accordingly, we set the batch subgraph size threshold so that with GraphSAGE, 10–25% of batches are divided, and with GAT, only 1–4%. These values are chosen empirically, as compute-heavy models generally benefit from fewer divisions to maintain GPU utilization.

V. Performance Evaluation

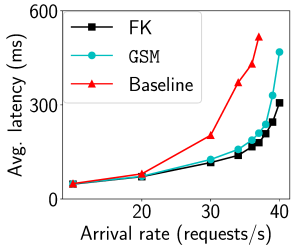
A. Methodology

We use the same hardware configuration, GNN models, and datasets as in §III unless stated otherwise.

Comparison baseline. Since no public systems exist for exact online GNN inference, we use the DGL-based system from §III as the baseline. It uses a fixed batch size; we search over 2, 4, 8, 16, and 32 and report the best result. Both

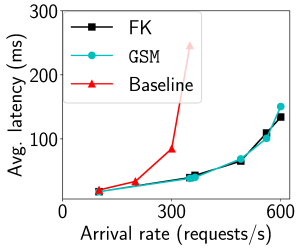


(a) GraphSAGE model.

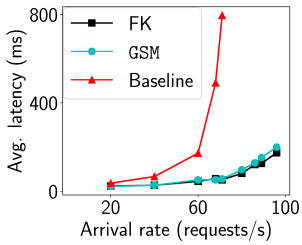


(b) GAT model.

Fig. 17: Average latency for papers100M dataset.

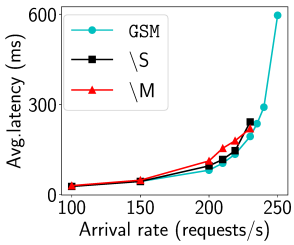


(a) GraphSAGE model.

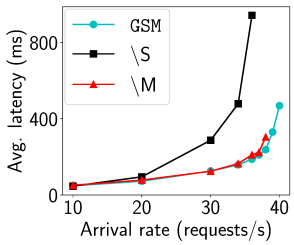


(b) GAT model.

Fig. 18: Average latency for Friendster dataset.

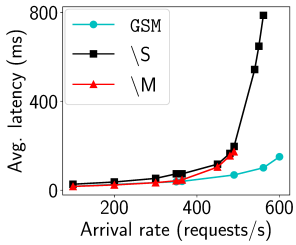


(a) GraphSAGE model.

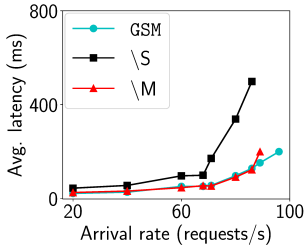


(b) GAT model.

Fig. 19: Average latency for papers100M dataset.



(a) GraphSAGE model.



(b) GAT model.

Fig. 20: Average latency for Friendster dataset.

the baseline and GSM adopt the degree-based caching policy. We also compare against a *full knowledge* (FK) variant that knows each request's l-hop neighborhood size and number of uncached vertex features, providing ideal processing time prediction.

Scenario. Each request is generated by randomly sampling a vertex from the graph, and requests arrive according to a Poisson process [36]. Unless otherwise specified, the arrival rates are 150 req/s (GraphSAGE) and 35 req/s (GAT) for Papers100M, and 350 and 80 req/s for Friendster. Each experiment processes 60,000 requests.

B. End-to-End Comparison

Average latency. Figs. 17 and 18 show average latency versus arrival rate. GSM reduces latency by up to 69% (GraphSAGE) and 60% (GAT) on Papers100M, and up to 83% and 89% on Friendster. It also performs competitively with the FK method—within 25%/52% (GraphSAGE/GAT) on Papers100M and 12%/20% on Friendster. As the arrival rate increases, queuing time grows and head-of-line blocking worsens for the FIFO scheduler, while GSM's scheduling mitigates this, yielding larger gains under heavy load.

Maximum throughput. We define maximum throughput as the highest request arrival rate at which the average latency remains below one second without triggering catastrophic out-of-memory (OOM) failures, where catastrophic failure is defined as more than 0.1% (one in a thousand) requests being dropped due to OOM errors. GSM improves throughput by 20% (GraphSAGE) and 8% (GAT) on Papers100M, and by 71% and 35% on Friendster.

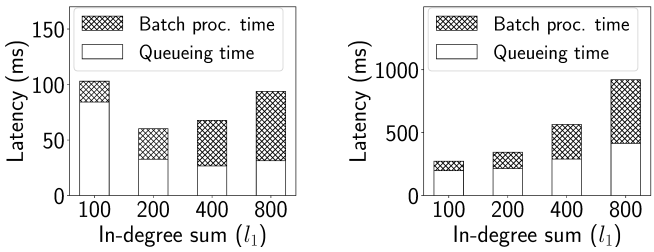
C. Ablation Study

In this section, we evaluate the effectiveness of the proposed scheduling method and the batch divider. The effectiveness of the parameter tuner is measured in §V-D by testing the effect of parameters on latency. We introduce two variants of GSM denoted by \S and \M. \S uses a first-come, first-served scheduler with a fixed batch size, rather than a degree-based scheduler. \M does not utilize the batch division. Fig. 19 and Fig. 20 show the average latency of GSM, \S, and \M on the papers100M and Friendster datasets versus various arrival rates.

a) Light-weight Request Scheduler: On the papers100M dataset, the proposed scheduler reduces latency by up to 21% with the GraphSAGE model and 88% with the GAT model. On the Friendster dataset, it achieves reductions of up to 60% with GraphSAGE and 74% with GAT. The results show the effectiveness of our proposed scheduler.

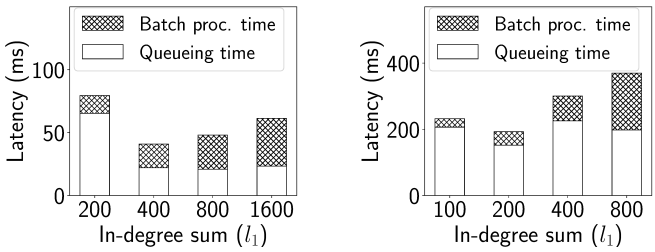
b) Batch Divider: On the papers100M dataset, the batch divider reduces the latency by up to 32% when using the GraphSAGE model and up to 10% when using the GAT model. On the Friendster dataset, the proposed scheduler reduces the latency by up to 65% when using the GraphSAGE model and up to 23% when using the GAT model. The results show the effectiveness of our proposed batch divider. \M's throughput is limited by OOM errors.

As the request arrival rate increases, the scheduler and batch divisor yield greater improvements, effectively reducing latency



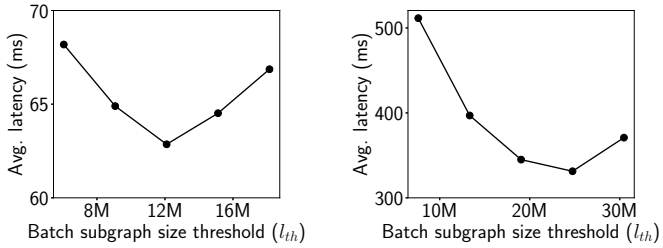
(a) GraphSAGE model. (b) GAT model.

Fig. 21: Average latency for papers100M dataset.



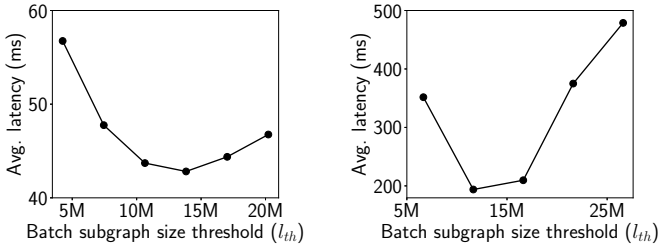
(a) GraphSAGE model. (b) GAT model.

Fig. 22: Average latency for Friendster dataset.



(a) GraphSAGE model. (b) GAT model.

Fig. 23: Average latency for papers100M dataset.



(a) GraphSAGE model. (b) GAT model.

Fig. 24: Average latency for Friendster dataset.

under high-workload conditions.

D. Sensitivity Test

In this section, we assess how parameters affect end-to-end latency to evaluate the tuner's effectiveness.

1) Effect of In-degree Sum: Figs. 21 and 22 show the impact of the in-degree sum T_d on latency, decomposed into queuing and processing times. Larger T_d increases batch processing due to bigger batches, while too small T_d raises queuing delays. For GraphSAGE, queuing first decreases with T_d before rising again as cache pressure grows, whereas for GAT both components increase with T_d. In all cases, `GSM` selects the in-degree sum near the minimum latency.

2) Effect of Batch Subgraph Size Threshold: Figs. 23 and 24 show the effect of the batch subgraph size threshold (T_s) on latency. For GraphSAGE, the tuner chose $T_s = 12$M on Papers100M and 10.6M on Friendster, both near the optimal values (60.2ms vs. 40.9ms). For GAT, the chosen thresholds (19M and 21.6M) led to latencies of 396ms and 209ms, within 15% and 8% of the minima (344ms and 193ms). Thus, `GSM` consistently identifies near-optimal T_s values without exhaustive search, delivering significant latency reductions over baselines.

E. Overhead

We measured the overhead of the scheduler and batch divider on both datasets and models at the maximum arrival rates supported by `GSM` (255/41 req/s on Papers100M and 592/95 req/s on Friendster for GraphSAGE/GAT). The tuner runs offline, so we excluded it. The scheduler's overhead is the average time to batch requests, and the divider's overhead is the extra computation from splitting batches.

As shown in Fig. 25, scheduling overhead is below 1% in all cases (0.31–0.76%) since it only relies on in-degrees readily available from the graph. Batch division overhead ranges from 0.8–3.4%, higher for GAT due to heavier computation and reduced GPU utilization when batches are split.

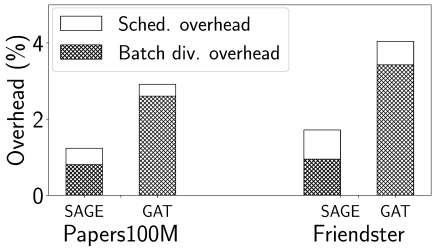


Fig. 25: `GSM`'s overhead

VI. Discussion and Future Work

Prefetching. Prefetching during inference can overlap communication and computation. In our experiments, enabling prefetching in `GSM` required holding two batches in GPU memory simultaneously, which reduced the effective batch size and offset its benefits. Prefetching may become practical on GPUs with larger memory or through adaptive strategies that prefetch only when sufficient memory is available, which we leave to future work.

Cache effects. `GSM` currently estimates batch processing time under the assumption that all feature data are loaded from CPU

memory. In practice, cache hits can significantly alter this cost. Integrating cache-hit prediction or cache-aware prioritization into the scheduler represents a promising extension.

Distributed setting. Although our work evaluates single-GPU systems, the identified problems (degree skew and volatile memory) persist in multi-node deployments. Distributed inference introduces additional challenges such as network overhead, graph partitioning, and load balancing. Adapting GSM to these settings while mitigating inter-node communication is an important direction for future work.

Resource starvation. In our setting, requests with high estimated processing times may stay in the queue for extended periods, causing long-tail latency and unfairness. This can be alleviated by promotion mechanisms, such as multi-level feedback queues (MLFQ), which gradually raise the priority of long-waiting requests.

VII. Related Work

GNN Inference. Several methods accelerate GNN inference. The authors in [20], [37], [38] exploit overlap between batches and eliminate redundancy through layerwise inference. Others [37], [38] identify data loading as a primary bottleneck and reorder target vertices to improve data locality. However, such reordering is infeasible in online inference, where requests must be served immediately, and waiting for locality-aligned requests would add delay. [39] focuses on distributed offline inference, reducing redundancy by sharing repeated computations across GPUs. Works such as [18], [40]–[42] sample subsets of vertex representations, while [15], [19], [38] sample neighborhoods. These sampling methods are unsuitable for exact inference, as users may not accept stochastic outputs [20]. [43] proposes a graph-structure cache and pre-sampling for dynamic graphs, and [44]–[46] design specialized hardware for GNN acceleration.

Scheduling for Machine Learning Inference. Scheduling is critical for model serving. The authors in [36], [47]–[49] rely on stable execution and memory profiles of DNN workloads to predict request characteristics and meet SLOs. Network-aware schedulers [50], [51] overlap communication phases across models to improve utilization. Large language models (LLMs) also exhibit variable execution times; prior work [52]–[54] shows that head-of-line blocking severely affects latency and proposes predictors and schedulers to mitigate it. Preemptive scheduling [52], [53] further improves latency. However, these techniques are not directly applicable to GNNs, which require different request time estimators and memory-aware scheduling.

VIII. Conclusion

Many GNN applications require full-graph (exact) online inference with low request latency, yet this setting remains largely underexplored. We identified its main challenges: head-of-line blocking, volatile memory usage, and the difficulty of tuning system parameters for diverse workloads. To address these, we proposed GSM, a scheduling and memory management framework for exact online GNN inference. Experiments on billion-edge graphs show that GSM reduces average request latency by up to 89% compared to state-of-the-art baselines. Future work will extend GSM to distributed and multi-model serving environments.

Acknowledgements

This research was supported in part by U.S. NSF grants NSF-2441601, NSF-2421782, NSF-2350425, NSF-2319988, NSF-2206522, Microsoft Research Faculty Fellowship 8300751, Amazon research award, AWS Cloud Credit for Research, and the Commonwealth Cyber Initiative (CCI), an investment in the advancement of cyber research, innovation and workforce development. For more information about CCI, visit cyberinitiative.org.

References

[1] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," *Advances in neural information processing systems*, vol. 30, 2017.

[2] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[3] P. Velickovic, G. Cucurull, A. Casanova, A. Romero, P. Lio, Y. Bengio, *et al.*, "Graph attention networks," *stat*, vol. 1050, no. 20, pp. 10–48550, 2017.

[4] F. Frasca, E. Rossi, D. Eynard, B. Chamberlain, M. Bronstein, and F. Monti, "Sign: Scalable inception graph neural networks," *arXiv preprint arXiv:2004.11198*, 2020.

[5] M. Zhang and Y. Chen, "Link prediction based on graph neural networks," *Advances in neural information processing systems*, vol. 31, 2018.

[6] Q. Tan, X. Zhang, N. Liu, D. Zha, L. Li, R. Chen, S.-H. Choi, and X. Hu, "Bring your own view: Graph neural networks for link prediction with personalized subgraph selection," in *Proceedings of the Sixteenth ACM International Conference on Web Search and Data Mining*, pp. 625–633, 2023.

[7] J. Wu, J. He, and J. Xu, "Net: Degree-specific graph neural networks for node and graph classification," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 406–415, 2019.

[8] Y. Wang, Y. Zhao, N. Shah, and T. Derr, "Imbalanced graph classification via graph-of-graph neural networks," in *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*, pp. 2067–2076, 2022.

[9] G. Zhang, Z. Li, J. Huang, J. Wu, C. Zhou, J. Yang, and J. Gao, "efraudcom: An e-commerce fraud detection system via competitive graph neural networks," *ACM Transactions on Information Systems (TOIS)*, vol. 40, no. 3, pp. 1–29, 2022.

[10] Z. Liu, Y. Dou, P. S. Yu, Y. Deng, and H. Peng, "Alleviating the inconsistency problem of applying graph neural network to fraud detection," in *Proceedings of the 43rd international ACM SIGIR conference on research and development in information retrieval*, pp. 1569–1572, 2020.

[11] T. Liu, R. Zhao, K.-M. Lam, and J. Kong, "Visual-semantic graph neural network with pose-position attentive learning for group activity recognition," *Neurocomputing*, vol. 491, pp. 217–231, 2022.

[12] T. Ahmad, L. Jin, X. Zhang, S. Lai, G. Tang, and L. Lin, "Graph convolutional neural network for human action recognition: A comprehensive survey," *IEEE Transactions on Artificial Intelligence*, vol. 2, no. 2, pp. 128–145, 2021.

[13] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, "Graph convolutional neural networks for web-scale recommender systems," in *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*, pp. 974–983, 2018.

[14] R. Yin, K. Li, G. Zhang, and J. Lu, "A deeper graph neural network for recommender systems," *Knowledge-Based Systems*, vol. 185, p. 105020, 2019.

[15] Z. Tan, X. Yuan, C. He, M.-K. Sit, G. Li, X. Liu, B. Ai, K. Zeng, P. Pietzuch, and L. Mai, "Quiver: Supporting gpus for low-latency, high-throughput gnn serving with workload awareness," *arXiv preprint arXiv:2305.10863*, 2023.

[16] U. Gupta, S. Hsia, V. Saraph, X. Wang, B. Reagen, G.-Y. Wei, H.-H. S. Lee, D. Brooks, and C.-J. Wu, "Deeprecsys: A system for optimizing end-to-end at-scale neural recommendation inference," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 982–995, IEEE, 2020.

[17] Y. Hu, R. Ghosh, and R. Govindan, "Scrooge: A cost-effective deep learning inference system," in *Proceedings of the ACM Symposium on Cloud Computing*, pp. 624–638, 2021.

[18] D. Gurevin, M. Shan, S. Huang, M. A. Hasan, C. Ding, and O. Khan, "Prunegnn: Algorithm-architecture pruning framework for graph neural network acceleration," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 108–123, IEEE, 2024.

[19] T. Kaler, N. Stathas, A. Ouyang, A.-S. Iliopoulos, T. Schardl, C. E. Leiserson, and J. Chen, "Accelerating training and inference of graph neural networks with fast sampling and pipelining," *Proceedings of Machine Learning and Systems*, vol. 4, pp. 172–189, 2022.

[20] D. Zhang, X. Song, Z. Hu, Y. Li, M. Tao, B. Hu, L. Wang, Z. Zhang, and J. Zhou, "Inferturbo: A scalable system for boosting full-graph inference of graph neural network over huge graphs," in *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, pp. 3235–3247, IEEE, 2023.

[21] M. Y. Wang, "Deep graph library: Towards efficient and scalable deep learning on graphs," in *ICLR workshop on representation learning on graphs and manifolds*, 2019.

[22] T. Chen, Q. Sun, C. Zhan, C. Liu, H. Yu, and B. Yu, "Deep h-gcn: Fast analog ic aging-induced degradation estimation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 7, pp. 1990–2003, 2021.

[23] Z. Lin, C. Li, Y. Miao, Y. Liu, and Y. Xu, "Pagraph: Scaling gnn training on large graphs via computation-aware caching," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, pp. 401–415, 2020.

[24] M. Faloutsos, P. Faloutsos, and C. Faloutsos, "On power-law relationships of the internet topology," *ACM SIGCOMM computer communication review*, vol. 29, no. 4, pp. 251–262, 1999.

[25] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: distributed graph-parallel computation on natural graphs," in *10th USENIX symposium on operating systems design and implementation (OSDI 12)*, pp. 17–30, 2012.

[26] Anon., "Gsm." https://anonymous.4open.science/r/Degree-Based-GNN-Inference-C2DA/README.md, 2025.

[27] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," in *International conference on machine learning*, pp. 1263–1272, Pmlr, 2017.

[28] D. Zheng, X. Song, C. Yang, D. LaSalle, and G. Karypis, "Distributed hybrid cpu and gpu training for graph neural networks on billion-scale heterogeneous graphs," in *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pp. 4582–4591, 2022.

[29] J. Yang, D. Tang, X. Song, L. Wang, Q. Yin, R. Chen, W. Yu, and J. Zhou, "Gnnlab: a factored system for sample-based gnn training over gpus," in *Proceedings of the Seventeenth European Conference on Computer Systems*, pp. 417–434, 2022.

[30] S. Gandhi and A. P. Iyer, "P3: Distributed deep graph learning at scale," in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pp. 551–568, 2021.

[31] W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, and J. Leskovec, "Open graph benchmark: Datasets for machine learning on graphs," *Advances in neural information processing systems*, vol. 33, pp. 22118–22133, 2020.

[32] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," in *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics*, pp. 1–8, 2012.

[33] Z. Cai, Q. Zhou, X. Yan, D. Zheng, X. Song, C. Zheng, J. Cheng, and G. Karypis, "Dsp: Efficient gnn training with multiple gpus," in *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pp. 392–404, 2023.

[34] Y. Park, S. Min, and J. W. Lee, "Ginex: Ssd-enabled billion-scale graph neural network training on a single machine via provably optimal in-memory caching," *Proceedings of the VLDB Endowment*, vol. 15, no. 11, pp. 2626–2639, 2022.

[35] A. Agrawal, N. Kedia, A. Panwar, J. Mohan, N. Kwatra, B. Gulavani, A. Tumanov, and R. Ramjee, "Taming throughput-latency tradeoff in llm inference with sarathi-serve," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 117–134, 2024.

[36] Z. Li, L. Zheng, Y. Zhong, V. Liu, Y. Sheng, X. Jin, Y. Huang, Z. Chen, H. Zhang, J. E. Gonzalez, *et al.*, "Alpaserve: Statistical multiplexing with model parallelism for deep learning serving," in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pp. 663–679, 2023.

[37] P. Yin, X. Yan, J. Zhou, Q. Fu, Z. Cai, J. Cheng, B. Tang, and M. Wang, "Dgi: An easy and efficient framework for gnn model evaluation," in *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pp. 5439–5450, 2023.

[38] T. Liu, P. Li, Z. Su, and M. Dong, "Efficient inference of graph neural networks using local sensitive hash," *IEEE Transactions on Sustainable Computing*, 2024.

[39] S. Chen, X. Song, V. Theodore, and H. Liu, "Deal: Distributed end-to-end gnn inference for all nodes," *arXiv preprint arXiv:2503.02960*, 2025.

[40] X. Gao, W. Zhang, J. Yu, Y. Shao, Q. V. H. Nguyen, B. Cui, and H. Yin, "Accelerating scalable graph neural network inference with node-adaptive propagation," *arXiv preprint arXiv:2310.10998*, 2023.

[41] J. Yik, S. R. Kuppannagari, H. Zeng, and V. K. Prasanna, "Input feature pruning for accelerating gnn inference on heterogeneous platforms," in *2022 IEEE 29th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pp. 282–291, IEEE, 2022.

[42] H. Zhou, A. Srivastava, H. Zeng, R. Kannan, and V. Prasanna, "Accelerating large scale real-time gnn inference using channel pruning," *arXiv preprint arXiv:2105.04528*, 2021.

[43] J. Sun, Z. Shi, L. Su, W. Shen, Z. Wang, Y. Li, W. Yu, W. Lin, F. Wu, B. He, *et al.*, "Helios: Efficient distributed dynamic graph sampling for online gnn inference," in *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pp. 2–15, 2025.

[44] Z. Wang, Y. Guan, G. Sun, D. Niu, Y. Wang, H. Zheng, and Y. Han, "Gnn-pim: A processing-in-memory architecture for graph neural networks," in *Advanced Computer Architecture: 13th Conference, ACA 2020, Kunming, China, August 13–15, 2020, Proceedings 13*, pp. 73–86, Springer, 2020.

[45] S. Mondal, S. D. Manasi, K. Kunal, R. S, and S. S. Sapatnekar, "Gnnie: Gnn inference engine with load-balancing and graph-specific caching," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pp. 565–570, 2022.

[46] R. Sarkar, S. Abi-Karam, Y. He, L. Sathidevi, and C. Hao, "Flowgnn: A dataflow architecture for real-time workload-agnostic graph neural network inference," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 1099–1112, IEEE, 2023.

[47] A. Gujarati, R. Karimi, S. Alzayat, W. Hao, A. Kaufmann, Y. Vigfusson, and J. Mace, "Serving dnns like clockwork: Performance predictability from the bottom up," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 443–462, 2020.

[48] M. Han, H. Zhang, R. Chen, and H. Chen, "Microsecond-scale preemption for concurrent gpu-accelerated dnn inferences," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 539–558, 2022.

[49] W. Cui, H. Zhao, Q. Chen, N. Zheng, J. Leng, J. Zhao, Z. Song, T. Ma, Y. Yang, C. Li, *et al.*, "Enable simultaneous dnn services based on deterministic operator overlap and precise latency prediction," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–15, 2021.

[50] S. Rajasekaran, M. Ghobadi, and A. Akella, "Cassini: Network-aware job scheduling in machine learning clusters," in *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pp. 1403–1420, 2024.

[51] Y. Zhao, Y. Liu, Y. Peng, Y. Zhu, X. Liu, and X. Jin, "Multi-resource interleaving for deep learning training," in *Proceedings of the ACM SIGCOMM 2022 Conference*, pp. 428–440, 2022.

[52] Y. Jin, C.-F. Wu, D. Brooks, and G.-Y. Wei, "s^3: Increasing gpu utilization during generative inference for higher throughput," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[53] B. Wu, Y. Zhong, Z. Zhang, G. Huang, X. Liu, and X. Jin, "Fast distributed inference serving for large language models," *arXiv preprint arXiv:2305.05920*, 2023.

[54] Z. Zheng, X. Ren, F. Xue, Y. Luo, X. Jiang, and Y. You, "Response length perception and sequence scheduling: An llm-empowered llm inference pipeline," *Advances in Neural Information Processing Systems*, vol. 36, 2024.