Legion: Automatically Pushing the Envelope of Multi-GPU System for Billion-Scale GNN Training

Jie Sun, *Collaborative Innovation Center of Artificial Intelligence, Zhejiang University, China;* Li Su, *Alibaba Group;* Zuocheng Shi, *Collaborative Innovation Center of Artificial Intelligence, Zhejiang University, China;* Wenting Shen, *Alibaba Group;* Zeke Wang, *Collaborative Innovation Center of Artificial Intelligence, Zhejiang University, China;* Lei Wang, *Alibaba Group;* Jie Zhang, *Collaborative Innovation Center of Artificial Intelligence, Zhejiang University, China;* Yong Li, Wenyuan Yu, and Jingren Zhou, *Alibaba Group;* Fei Wu, *Collaborative Innovation Center of Artificial Intelligence, Zhejiang University, China and Shanghai Institute for Advanced Study of Zhejiang University, China*

https://www.usenix.org/conference/atc23/presentation/sun

This paper is included in the Proceedings of the 2023 USENIX Annual Technical Conference.

July 10–12, 2023 • Boston, MA, USA

978-1-939133-35-9

Open access to the Proceedings of the 2023 USENIX Annual Technical Conference is sponsored by

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

Legion: Automatically Pushing the Envelope of Multi-GPU System for Billion-Scale GNN Training

Jie Sun[1], Li Su[2], Zuocheng Shi[1], Wenting Shen[2], Zeke Wang[1]
Lei Wang[2], Jie Zhang[1], Yong Li[2], Wenyuan Yu[2], Jingren Zhou[2], Fei Wu[1,3]

[1] *Collaborative Innovation Center of Artificial Intelligence, Zhejiang University, China*

[2] *Alibaba Group*

[3] *Shanghai Institute for Advanced Study of Zhejiang University, China*

Abstract

Graph neural network(GNN) has been widely applied in real-world applications, such as product recommendation in e-commerce platforms and risk control in financial management systems. Several cache-based GNN systems have been built to accelerate GNN training in a single machine with multiple GPUs. However, these systems fail to train billion-scale graphs efficiently, which is a common challenge in the industry. In this work, we propose Legion, a system that automatically pushes the envelope of multi-GPU systems for accelerating billion-scale GNN training. First, we design a hierarchical graph partitioning mechanism that significantly improves the multi-GPU cache performance. Second, we build a unified multi-GPU cache that helps to minimize the PCIe traffic incurred by caching both graph topology and features with the highest hotness. Third, we develop an automatic cache management mechanism that adapts the multi-GPU cache plan according to the hardware specifications and various graphs to maximize the overall training throughput. Evaluations on various GNN models and multiple datasets show that Legion supports training billion-scale GNNs in a single machine and significantly outperforms the state-of-the-art cache-based systems on small graphs.

1 Introduction

Graph neural networks (GNNs), such as [8, 10, 16, 22, 40, 50], are a class of deep learning algorithms that learn the low-dimensional embedding using the structure and attribute information of graphs. The learned embedding can be further used in machine-learning tasks including node classification and link prediction. GNNs have been successfully applied in many real-world applications, such as recommendation systems in e-commerce platforms, fraud detection and risk control in financial management, and molecular property prediction in drug development [13, 25, 37, 48, 49]. Systems such as DGL [42], PyG [31], and Graph-Learn [55] are proposed to ease the development and training of GNN models.

It is common to apply GNNs over large-scale graphs in industrial scenarios. For example, in Alibaba's Taobao recommendation system, the user behavior graph contains more than one billion vertices and tens of billions of edges [55]. In addition, as graphs are often skewed, it is infeasible to aggregate all neighboring vertices when training a specific vertex. Sampling-based mini-batch training, such as GraphSAGE [16], is proposed to extend GNN training to very large graphs. In the sampling-based GNN training, there are two key steps of data preparations before training a batch: (1) sampling the multi-hop sub-graph for each vertex in the batch, and (2) extracting the features of vertices in sampled sub-graphs. Systems such as DGL [42] and PyG [31] store the graph data in the CPU memory, prepare the training data of mini-batches using CPUs, and utilize GPUs for model training. As this approach requires transferring the sampled sub-graphs and high-dimension feature data to the GPU for every batch, the end-to-end training throughput is severely limited by the CPU-GPU data transferring bandwidth [23, 47]. In addition, the throughput of graph sampling using CPU is often insufficient to keep up with the throughput of GPU training, especially in multi-GPU machines.

Several cache-based approaches have been proposed to speed up GNN training [23, 29, 33, 47]. As it is the feature data that accounts for a majority of the CPU-GPU data transferring, caching the features of frequently accessed vertices in GPU can significantly reduce the amount of transferred data. To improve the throughput of graph sampling, GPU-based sampling has also been adopted in GNN systems [33, 42, 47].

We identify that existing approaches face severe limitations or performance issues in multi-GPU training, particularly when the graph is large. First, the multi-GPU cache scalability of existing cache-based systems is poor. Some cache-based GNN systems [33, 47] shuffle the training set across all GPUs and replicate an identical feature cache across all GPUs or NVLink cliques[1] to facilitate data parallel training. The cache capacity is constrained by the memory of a single GPU or

[1] NVLink clique denotes a group of GPUs where each pair of GPUs are connected with NVLink.

NVLink clique (an NVLink clique only consists of two GPUs in some multi-GPU architectures), resulting in poor cache performance when scaling up the number of GPUs (see the experiment in Figure 2). PaGraph [23] partitions the graph using a self-reliant algorithm and caches nodes with the highest in-degree for different partitions in different GPUs, trying to make use of data locality inside each partition. As partitions in PaGraph include the complete L-hop neighbors of their training vertices, there is a significant overlap between the caches of different partitions, resulting in the same duplication issue as the aforementioned cache-based GNN systems. Second, when adopting GPU-based graph sampling, existing systems manage the graph topology in a very coarse-grained manner: the topology has to be completely stored in a single GPU [33, 42, 47] or in the CPU memory [33, 42]. The former approach puts a hard limit on the graph scale, and further squeezes the cache capacity for features. The latter storing the topology in the CPU and accessing it from GPU would result in very low utilization of the PCIe bandwidth, as the data access of graph sampling is usually random and fine-grained.

This paper presents Legion, a GNN system that fully explores the hardware capabilities of modern multi-GPU servers for training large-scale graphs in a single machine. Legion proposes two key designs to fully exploit the memory space of multi-GPUs for feature and topology cache. First, to avoid cache replication, we propose **NVLink-aware hierarchical graph partitioning** technique that helps scale the cache on multi-GPU memory efficiently according to the specific hardware structure. Legion first partitions the graph with minimal edge-cut and assigns each partition exclusively to an NVLink clique, and then uses hash partition to further map the training vertices to GPUs inside each NVLink clique. Second, we propose a **hotness-aware unified cache** that manages both the feature and topology cache in a vertex-centric data structure. We enable an NVLink-enhanced cache space for the unified cache and prioritize the topology and features with the highest hotness to fill the cache, so as to improve the multi-GPU memory utilization.

The above designs pose a new challenge to Legion. Given a fixed size of GPU memory, it is hard to manually decide the optimal fractions of topology and feature cache such that the overall training throughput is maximized. To solve the challenge, we propose an **automatic cache management** mechanism. Specifically, we build a cost model in the mechanism to evaluate the key factor to the overall throughput, i.e., PCIe traffic, of both graph sampling and feature extraction in the training phase, which is used to guide the allocations of cache spaces for graph topology and feature. Overall, the three key designs in Legion enable automatic caching optimization and full utilization of hardware capability of various modern GPU servers. Experiments show that Legion can outperform state-of-the-art cache-based GNN systems up to 4.32×.

In summary, the contributions of this paper include:

1. We propose an NVLink-aware hierarchical graph partitioning technique that helps minimize cache replication between NVLink cliques and extends the threshold of cache capacity beyond the limit of an NVLink clique.
2. We propose a hotness-aware unified cache to store topology and features with the highest hotness in GPU memory, so as to improve the GPU memory utilization.
3. We present an automatic cache management mechanism that searches for the optimal cache plan without requiring extra knowledge of hardware specifications and GNN performance details from users.
4. We implement Legion that fully explores the hardware capabilities of multi-GPU systems targeting billion-scale GNN training, not supported by existing cache-based GNN systems, in a single server.

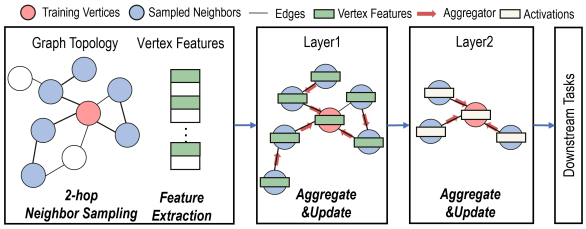


Figure 1: The workflow of 2-hop GraphSAGE training.

2 Preliminaries

In this section, we introduce the basic concept of GNN and the workflow of mini-batch GNN training.

2.1 Graph Neural Networks

Given a graph $G = (V, E)$, where each vertex is associated with a vector of data as its features $X_v, v \in V$, Graph Neural Networks(GNNs) learn a low-dimensional embedding for each target vertex by stacking multiple GNNs layers L. For each layer $l, l \in L$, vertex v updates its activation by aggregating features or hidden activations of its neighbors $N(v), v \in V$:

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

2.2 Mini-batch GNNs Training

Mini-batch training is a practical solution for scaling GNN training on very large graphs. Neighbor sampling is used to generate mini-batches, allowing sampling-based GNN models to handle unseen vertices. For example, GraphSAGE [16] samples multiple hops of neighbors for training as shown in Figure 1. The workflow of GraphSAGE training follows a vertex-centric computation paradigm including the following steps: 1, selecting a mini-batch of training vertices from the training set. 2, uniformly sampling the multiple hops of fixed-size neighbors for each training vertex. 3, extracting the

features of the sub-graph consisting of the training vertices and their neighbors to generate the mini-batch training data. Finally, performing *AGGREGATE* and *UPDATE* according to Equations 1, as well as executing the forward and backward propagation to update the model parameters.

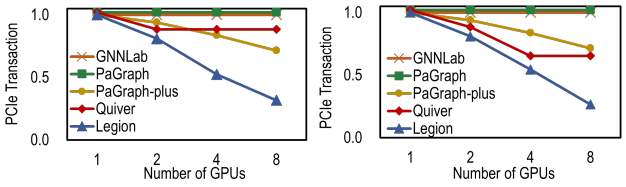
3 Observation and Motivation

When training large-scale graphs whose size exceeds the capacity of GPU memory on a multi-GPU server, the major performance bottleneck becomes the data movement from CPU to GPUs under the constraint of PCIe bandwidth. To this end, existing works [33, 42, 47] intend to relieve the PCIe bandwidth bottleneck by caching the hottest graph features on GPU memory. Though these cache-based approaches significantly reduce PCIe traffic, we still identify two issues of these existing cache-based GNN systems when training large-scale graphs: 1) poor multi-GPU cache scalability, and 2) coarse-grained GPU memory management for graph topology. In the following, we discuss each issue and the corresponding observation that motivates the design of Legion.

3.1 Multi-GPU Cache Scalability

As feature extraction occupies most of the data transferring from CPU to GPU, cache-based systems like GNNLab [47] maintain a global feature cache for vertices which are more frequently accessed via a pre-sampling phase. As training vertices are globally shuffled among all training GPUs, GNNLab replicates this cache across all GPUs involved in model training. Since a single GPU's memory space is quite limited, the fraction of cached features would inevitably become lower when the graph size grows, resulting in a lower cache hit ratio even on multi-GPU servers. To increase the cache capacity, the cache mechanism in Quiver [33] leverages high-speed NVLinks to support inter-GPU cache between NVLink-connected GPUs. Different from GNNLab, Quiver replicates feature cache between NVLink cliques and averagely hashes the features among GPUs in the same NVLink clique. However, this mechanism could still lead to poor cache scalability, especially when the NVLink clique is relatively small. E.g., the Siton server used in Table 1 has 4 NVLink cliques, each of which contains only 2 GPUs. Figure 2 illustrates that, in systems like Quiver, the PCIe transactions incurred by CPU-GPU data transferring stop decreasing when the number of GPUs is larger than the size of NVLink clique. This result shows that the cache performance in the above GNN systems cannot scale well with the increasing number of GPUs in modern servers.

To solve the scalability issue incurred by cache replication, PaGraph [23] partitions the graph in a self-reliance approach and maintains an independent cache for each partition using an in-degree-based metric on different GPUs. To train an L-layer GNN model, PaGraph extends every partition with redundant vertices and edges to include all the L-hop neighbor vertices for each train vertex in this partition. Each GPU only trains its own partition and synchronizes its local gradients periodically to update the model. However, the inclusion of the L-hop neighbor vertices leads to heavily duplicated cache contents on all GPUs. Figure 2 shows that the PaGraph exhibits a similar cache performance as GNNLab which adopts the cache replication mechanism. We further implement a PaGraph-plus design to alleviate the cache duplication issue in PaGraph. Specifically, we replace the graph partitioning algorithm in PaGraph with the XtraPulp [35] algorithm that minimizes edge-cuts between partitions and adopts a pre-sampling-based hotness metric to select the vertex features to be cached. Although PaGraph-plus achieves higher cache hit rates compared to PaGraph, the cache hit rates on different GPUs are very unbalanced as different partitions have various graph distributions. Figure 3 illustrates the load imbalance issue of PaGraph-plus by measuring the cache hit rates of eight GPUs. We observe that the hit rate varies by up to 17%.



(a) 2 GPUs per NVLink clique (b) 4 GPUs per NVLink clique

Figure 2: Comparing the cache scalability of cache-based GNN systems using the Products [17] dataset and 2-hop GraphSAGE [16] model in terms of normalized CPU-GPU PCIe transactions. The cache ratio is set to 5% $|V|$ on every GPU. The tested platforms are Siton (a) and DGX-V100 (b) servers, as shown in Table 1.

To sum up, for systems that globally shuffle the training vertices among GPUs in every iteration, such as GNNLab and Quiver, cache replication cannot be completely eliminated as each GPU may randomly access any vertex in the entire graph. Whereas the high-speed NVLinks between GPUs can be used to reduce the replication factor and expand the cache capacity. For systems that locally shuffle training vertices in each partition to produce mini-batches for different GPUs, such as PaGraph, the cache replication problem could be alleviated only when the model layer is small (e.g., less than 2). PaGraph-plus can further reduce cache duplication but faces another issue of unbalanced cache hit rates among GPUs.

Observation O1: Graph partitioning can be suitably guided by hardware structure. Different from Quiver, GNNLab, PaGraph, and PaGraph-plus do not take advantage of the NVLink between GPUs, which is a common capability in modern multi-GPU servers. As GPUs inside the same NVLink clique can access each other's memory via the low-latency high-throughput NVLink, an NVLink clique can hold the entire cache of a partition, which can be randomly sliced

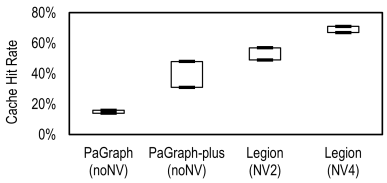
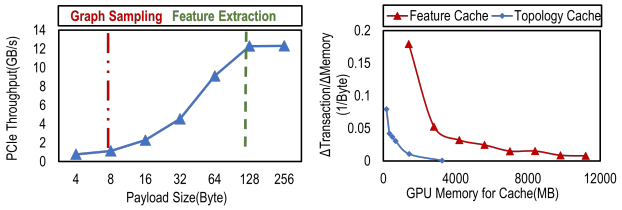


Figure 3: Cache hit rates of different systems in a server with 8 GPUs. The cache ratio is set to 5% $|V|$ on every GPU. The graph sampling follows the 2-hop GraphSAGE [16] model's setting using the Products [17] dataset. "NVx" means utilizing NVLink clique with x GPUs.



(a) PCIe throughput vs. payload size (b)

Figure 4: (a) The PCIe 3.0 throughput under different payload sizes of PCIe requests. (b) The PCIe traffic reduction rate for Paper100M with the growth of the cache capacity. The cache is on a single GPU and selected after pre-sampling.

and averagely allocated among GPUs inside a clique. This hardware-coherent design can balance the cache hit ratios between intra-clique GPUs. As the number of partitions is reduced to the number of NVLink cliques, it is more likely that the partitions follow a similar distribution (see the cache hit rate distribution of Legion in Figure 3). Inspired by **O1**, we propose an NVLink-aware hierarchical partitioning to preserve multi-GPU cache scalability in Legion (Section 4.1).

3.2 Coarse-grained GPU Memory Management for Graph Topology

In multi-GPU servers, the throughput of CPU-based graph sampling may not catch up with the throughput of GPU-based training. To improve the end-to-end training throughput, recent GNN systems [33,42,47] adopt GPUs to accelerate graph sampling. We observe that all these systems apply a very coarse-grained memory management mechanism for graph topology. In particular, they store the entire graph topology either in CPU memory or in a single GPU, depending on the size of graph topology: the graph topology is stored in CPU memory when it is too large or exceeds the capacity of a single GPU. The approach of storing the entire graph topology in a single GPU sets a hard limit on the scale of the graph. For example, a V100 GPU with 16GB memory can store at most 4 billion edges [16] without considering any other memory usage of feature cache and model training. When storing the graph topology in CPU memory, GPUs can directly access the graph topology via a unified virtual memory address (UVA [27]) technique. While the data access pattern of graph sampling is usually random and fine-grained. E.g., Figure 4a shows that the PCIe throughput of graph sampling is much lower than feature extraction. A large number of sampling PCIe transactions with small payload sizes will increase the CPU-GPU PCIe contention and lead to low bandwidth utilization.

Observation O2: The access of graph topology is skewed as graph features. Existing cache-based GNN systems [23, 33, 47] only maintain feature cache in GPU to reduce the CPU-GPU communication costs. However, we observe that the performance gain of the per-unit feature cache decreases once the cache capacity exceeds a threshold (see Figure 4b). We observe that the access of graph topology during graph sampling is also skewed as the access of features. Instead of allocating all the available GPU memory (except for the reservation for model training) for feature cache, it is reasonable to cache a subset of graph topology, i.e., edges of vertices that are frequently accessed during sampling, in the GPU memory to accelerate GPU sampling. Figure 4b shows that a relatively small topology cache can obviously reduce the number of PCIe transactions incurred by GPU sampling. Motivated by **O2**, we propose a hotness-aware unified cache in Legion. Specifically, Legion caches both graph topology and graph features with the goal of minimizing CPU-GPU communication overhead (see Section 4.2). Under the capacity limit of GPU memory, it is difficult to manually decide the optimal fractions of topology and feature cache. Legion solves this challenge with an automatic cache management mechanism, which can generate the optimal cache plan without requiring knowledge of hardware specifications from users.

4 Design of Legion

In order to address the aforementioned performance issues of existing cache-based GNN systems, we propose Legion, a cache-optimal GNN system that can push the envelope of the multi-GPU system automatically for billion-scale GNN training. The overall design of Legion is presented in Figure 5. We propose an NVLink-aware hierarchical partitioning technique (Section 4.1) in Legion that facilitates scaling up the cache capacity and reducing cache duplication in multi-GPU servers. To utilize GPU cache for both graph sampling and feature extraction, we present a hotness-aware unified cache (Section 4.2) that maintains both the topology and feature caches to optimize the overhead of PCIe traffic. We also develop an automatic cache management mechanism (Section 4.3) to automatically decide the memory allocations for both topology and feature caches.

4.1 NVLink-aware Hierarchical Partitioning

Motivated by **observation O1**, we propose a simple yet effective graph partitioning mechanism, referred to as **hierarchical**

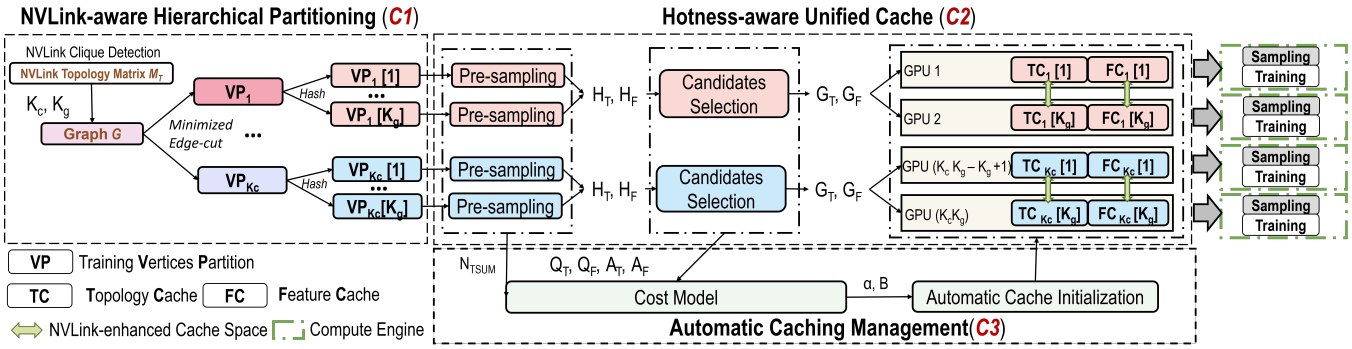


Figure 5: Design overview of Legion. Legion consists of three main contributions C1, C2, and C3.

partitioning, to facilitate cache scalability in Legion. Different from conventional graph partitioning algorithms which partition all edges/vertices of a graph into multiple tablets, hierarchical partitioning in Legion aims to divide the training vertices/edges into multiple disjoint tablets. The inputs of hierarchical partitioning are an NVLink topology matrix M_T of the underlying multi-GPU server and a graph G. The output is an assignment plan disseminating training vertices/edges among GPUs. Specifically, the process of hierarchical partitioning mainly consists of four steps:

S1: NVLink Clique Detection. With the topology matrix M_T of the server, Legion employs a MaxCliqueDyn algorithm [45] to identify the NVLink clique sets in M_T, and outputs the number of NVLink cliques K_c and the number of GPUs in each clique K_g.

S2: Inter-clique Graph Partitioning. To reduce the cache duplication between NVLink cliques, Legion uses an edge-cut minimizing partitioning algorithm, e.g., METIS [21] and XtraPulp [35], to split the input graph G into K_c partitions, i.e., P_1, P_2, ..., P_{Kc}, such that nodes are balanced among partitions and inter-partition edge-cuts are minimized. The training vertex set in P_i is denoted as VP_i. As the training vertices are randomly selected from G, the training vertex sets of different partitions are almost of the equal size. The number of partitions is equal to the number of detected NVLink cliques, and each NVLink clique hosts the cache for a dedicated partition. This way, Legion can reduce the cache duplication between NVLink cliques and take advantage of cache locality within an NVLink clique.

S3: Intra-clique Training Vertex Partitioning. As GPUs within an NVLink clique can access each other's memory via low-latency high-throughput NVLink interconnect, hierarchical partitioning further hashes the training vertex set of each partition into K_g tablets, where K_g is the GPU number in a clique. E.g., VP_i is split into $VP_i[1]$ and $VP_i[2]$ if K_g equals 2. Each tablet is exclusively mapped to a GPU in the corresponding NVLink clique. We explain how to generate the cache for each training vertex tablet in Section 4.2.

S4: Training Vertex Assignment. Finally, Legion assigns training vertices of each tablet to a corresponding GPU as the batch seeds, which will then be shuffled locally to generate mini-batches for graph sampling and training.

As such, Legion provides better cache scalability and load balancing compared to existing systems. Figure 2 shows the cache performance of Legion improves with the increase of GPUs almost linearly. Figure 3 illustrates that Legion has smaller fluctuations in the cache hit rates on multi-GPU servers with NVLink cliques of various sizes.

4.2 Hotness-aware Unified Cache

Motivated by the **observation O2**, we propose a hotness-aware unified cache to cache both graph topology and graph features. In this Section, we introduce the detailed mechanism of the unified cache.

4.2.1 Cache Structure

The unified cache consists of two parts: the topology cache and the feature cache. In particular, the topology cache maintains out-edge neighbor IDs for each selected hot vertex in the format of a compressed sparse row (CSR). As for the feature cache, Legion stores the feature vectors of selected hot vertices in the format of a 2D array, where each row is the feature vector of a selected hot vertex. Note that, the selected vertices in the topology and feature caches could be different.

4.2.2 Cache Construction

The construction of the unified cache is divided into three steps: (1) pre-sampling, (2) cache candidate selection, and (3) cache initialization. All the GPUs/NVLink cliques perform these steps concurrently to construct their own unified cache.

S1: Pre-sampling. Similar to GNNLab [47], Legion adopts a pre-sampling phase[2] to estimate the hotness metrics of graph topology and feature data during the training phase. Once the process of hierarchical partitioning is completed, the training vertex tablet assigned to each GPU is determined, which is used as the input for pre-sampling. The output of pre-sampling includes two hotness matrices: topology hotness matrix H_T and feature hotness matrix H_F. Each matrix's row represents the GPU IDs within an NVLink clique, the

[2]During pre-sampling, graph topology is stored in the CPU memory.

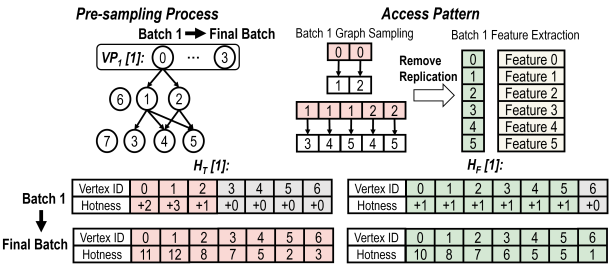


Figure 6: Update the hotness matrices of graph topology and features by pre-sampling. For simplicity, we only show the result for GPU 1.

column represents the vertex IDs, and the element H_{ij} of either matrix represents the hotness of the j-th vertex in the i-th GPU. During the pre-sampling, each GPU conducts a local shuffle on its own training vertex tablet to generate seeds for mini-batches, performs graph sampling for each mini-batch, and updates the corresponding row in H_T and H_F. Figure 6 shows a pre-sampling example. For H_T, whenever an edge is traversed during sampling, the hotness of its source vertex is incremented by 1. For H_F, the hotness for each vertex that appears in the sample results of the mini-batch is incremented by 1. Additionally, Legion uses Intel® Performance Counter Monitor (PCM) [18] to collect the summation of PCIe transactions number, N_{TSUM}, generated by all GPUs in an NVLink clique during pre-sampling.[3]

S2: Cache Candidate Selection. The objective of cache candidate selection is to select and disseminate the hot topology sub-structures and features among GPUs within the same NVLink clique based on pre-sampled hotness matrices. Thus this phase is conducted in the unit of NVLink clique, and each clique requires one GPU to perform the computation. The detailed process of cache candidate selection is presented in Algorithm 1. In brief, this algorithm computes the global topology/feature hotness of all vertices, i.e., A_T and A_F, in the NVLink clique by conducting a column-wise sum on H_T and H_F, respectively (Line 1). A_T and A_F are then sorted in descending order to generate Q_T and Q_F(Line 2). Next, We iterate Q_T and Q_F in order and assign every visited vertex to the GPU with the highest local hotness in H_T and H_F. For each GPU, we maintain two queues, i.e., G_T, G_F, whose order represents the priority of vertices to be included in this GPU cache. The outputs of Algorithm 1 are further used by the cost model (see Section 4.3) to generate the physical cache plan.

S3: Cache Initialization and Fill-up. Legion's cache management automatically decides the cache ratio for topology and feature so that the overall throughput is maximized (see Section 4.3). Guided by this mechanism, Legion allocates memory for both the topology and feature cache (TC and FC) of each GPU, and fetches the corresponding topology and feature data from CPU memory to fill up each GPU cache according to the corresponding cache orders in G_T and G_F.

[3] N_{TSUM} is further used by cost model's evaluation.

Algorithm 1 COMPLETE SHARING WITH LOCAL PREFERENCE (CSLP)

Input : K_g: number of GPUs per NVLink clique
H_F: feature hotness matrix
H_T: topology hotness matrix

Output : A_F: accumulated vertex-wise feature hotness vector
A_T: accumulated vertex-wise topology hotness vector
Q_T: vertex ID queue representing clique-level topology order,
Q_F: vertex ID queue representing clique-level feature order
G_T: vertex ID queue representing GPU-level topology order
G_F: vertex ID queue representing GPU-level feature order

/* Step 1: Accumulate each vertex's hotness from K_g GPUs. */
1 $A_F = H_F.columnWiseSum()$; $A_T = H_T.columnWiseSum()$;
/* Step 2: Sort vertices in A_F and A_T */
2 Q_F <- SortbyKeyDescend(A_F); Q_T <- SortbyKeyDescend(A_T);
/* Step 3: Assign each vertex to the GPU with the highest local hotness. */
3 **for** *v_id in* Q_T **do**
4 $gpu\_id = max(H_T[1:K_g][v\_id]).index$;
$G_T[gpu\_id].push(v\_id)$;
5 **end**
6 **for** *v_id in* Q_F **do**
7 $gpu\_id = max(H_F[1:K_g][v\_id]).index$;
$G_F[gpu\_id].push(v\_id)$;
8 **end**

4.3 Automatic Cache Management

The design of the unified cache poses a new challenge: how to properly specify the cache size for graph topology and features under the constraint of GPU memory such that the overall training throughput is maximized.

The general idea is to predict the overall throughput under different cache plans and search for the best cache plan that maximizes overall throughput. We define the cache plan as a cache memory management setting (B, α) at the NVLink clique granularity, where B is the multi-GPU cache memory size in an NVLink clique and α is the memory ratio for topology cache. B is identical among NVLink cliques and is by default set as the total multi-GPU memory minus the size of GPU memory reserved for GNN models and intermediate buffers in an NVLink clique. We need three steps to determine the optimal cache memory management setting (B, α), as discussed in Sections 4.3.1, 4.3.2, and 4.3.3.

4.3.1 Estimating Overall Throughput

The key goal of this Section is to build the relationship between the overall throughput and a cache plan. We build the relationship by estimating a key factor: the total PCIe traffic N_{total}, due to two reasons. First, the PCIe traffic is the major bottleneck of the overall system throughput, and lower PCIe traffic leads to higher overall system throughput. Second, varying cache plans major results in the variance of PCIe traffic.[4] Because each NVLink clique maintains caches for its own partition, we independently select the optimal cache plan for each NVLink clique so as to minimize the PCIe traffic of

[4] Though NVLink traffic is also influenced by the cache plan, we neglect it since NVLink has a much higher bandwidth than PCIe.

each NVLink clique. Thus, the overall system's PCIe traffic is minimized.

4.3.2 Cost Model to Estimate N_{total}

The key goal of this Section is to present a cost model to estimate N_{total} under a specific cache plan (B, α). First, given a specific cache plan (B, α), we can calculate the topology cache size m_T and the feature cache size m_F. Second, we find which vertices' topology/features should be stored in the topology/feature cache. Third, we estimate the PCIe traffic for graph sampling (N_T) and for feature extraction (N_F) with the current topology/feature cache utilization. At last, we estimate N_{total} by adding up N_T and N_F, as shown in Equation 2.

$$N_{total} = N_T + N_F \tag{2}$$

To estimate N_T and N_F, we need to collect other information apart from a given cache plan: the hotness vectors A_T and A_F, the summation of PCIe transaction number N_{TSUM} incurred by graph sampling, and the order queues of topology/feature cache candidates, Q_T and Q_F.

Estimating N_T. We estimate N_T when the memory size of a topology cache under one specific cache plan (B, α) is m_T, where $m_T = B \times \alpha$. The estimation consists of three steps.

First, with m_T, we decide which vertices' topology should be cached. We define V as the set of all vertices in the graph. And we define V_{Tcache} as the set of all vertices whose topology is cached under current topology cache size m_T. To get V_{Tcache}, we increase vertices and their topology into the cache with the growth of occupied topology cache memory by the order Q_T. Until the overall occupied topology cache memory size reaches m_T, we record V_{Tcache}. Equation 3 illustrates the relation between m_T and V_{Tcache}, where $nc(v)$ means the neighbor count of the vertex v. Here we assume the data types are Uint64 and Uint32 for the row and the column indices of the compressed sparse row format (CSR), respectively. We use s_{uint64} and s_{uint32} to denote the number of bytes to store a single Uint64 and Uint32 data accordingly.

$$\sum_{v \in V_{Tcache}} (nc(v) \times s_{uint32} + s_{uint64}) = m_T \tag{3}$$

Second, once we get V_{Tcache}, we can calculate the ratio of the PCIe transaction reduced by the topology cache by Equation 4. Let $a_T(v)$ mean the topology hotness of a specific vertex v ($a_T(v) \in A_T$).

$$R_T = \frac{\sum_{v \in V_{Tcache}} a_T(v)}{\sum_{v \in V} a_T(v)}) \tag{4}$$

Third, we get N_T by multiplying the entire PCIe transaction N_{TSUM} with the ratio of PCIe transactions that can **not** be reduced by the topology cache. We can get N_T by Equation 5.

$$N_T = N_{TSUM} \times (1 - R_T) \tag{5}$$

Estimating N_F. We explain how to calculate N_F when the feature cache memory size is m_F, where $m_F = B \times (1 - \alpha)$. There are also three steps in estimation.

First, given m_F, we decide which vertices' features should be cached. We define V_{Fcache} as the set of vertices whose feature data is cached. Then we increase the vertices with their feature into cache by the order Q_F until the occupied feature cache memory size reaches m_F, as defined in Equation 6. D represents the dimension of a feature vector and the feature data is the Float32 type each of which needs $s_{float32}$ bytes to store.

$$\sum_{v \in V_{Fcache}} D \times s_{float32} = m_F \tag{6}$$

Second, as shown in Equation 7, we calculate the total number of features U_F that still needs transferring through PCIe with a feature cache.

$$U_F = \sum_{v \in V} (a_F(v)) - \sum_{v \in V_{Fcache}} (a_F(v)) \tag{7}$$

Third, we get N_F by multiplying the transaction number needed by transferring one vertex's feature with the total number of features to be transferred, U_F, as shown in Equation 8. Here CLS means the transferred cache line size. CLS might be different for various CPUs and GPUs. We can get the CLS from PCM. E.g., CLS equals 64 in our machine settings. And $a_F(v)$ means the hotness of a specific vertex v ($a_F(v) \in A_F$).

$$N_F = (\lceil \frac{D \times s_{float32}}{CLS} \rceil) \times U_F \tag{8}$$

4.3.3 Searching for Optimal Cache Plan in Parallel

The key goal of this Section is to efficiently determine the optimal cache plan for each clique. As discussed in Section 4.3.1, we search for the optimal cache plan independently with one GPU for each NVLink clique. In each NVLink clique, we first need to traverse α from 0 to 1 by an interval $\Delta\alpha$ [5] to generate the candidate cache plans, and the calculate N_{total} accordingly. Then we need to search N_{total} sequences and find the smallest one with the dedicated α. To minimize overhead, the process is well parallelized, including four steps:

First, we generate all the candidate cache plans in parallel and get sequences of m_T and m_F in each setting.

Second, we get the boundaries of cached vertices set V_{Tcache} and V_{Fcache} using Equations 3 and 6, where the boundaries are the largest cached vertices' indexes in Q_T and Q_F. To do so, we get the topology and feature memory size of every single vertex in parallel and store them in two arrays, $S_{Tsingle}$ and $S_{Fsingle}$, following the vertices order, Q_T and Q_F. Next, we calculate the cumulative sum of $S_{Tsingle}$ and $S_{Fsingle}$ by a parallel inclusive scan and get S_{Tsum} and S_{Fsum}. Then for each cache plan with m_T and m_F, we use a parallel binary

[5] $\Delta\alpha$ is set to be 0.01 by default.

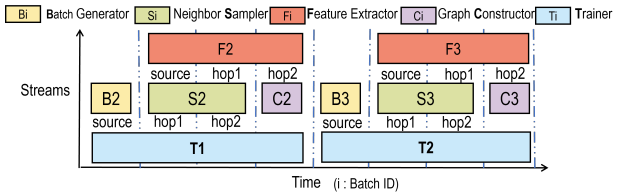


Figure 7: An example of fine-grained GNN training pipeline for 2-hop GraphSAGE model.

search towards S_{Tsum} and S_{Fsum} to get the boundary indexes of vertices, respectively.

Third, we get the R_T and U_F according to Equations 4 and 7. To do so, we calculate the cumulative sum of A_T and A_F by a parallel inclusive scan and get A_{Tsum} and A_{Fsum}. Then for each cache plan, we lookup A_{Tsum} and A_{Fsum} with the boundary indexes of vertices set V_{Tcache}, V_{Fcache}, and get $\sum_{v \in V_{Tcache}} a_T(v)$ and $\sum_{v \in V_{Fcache}} a_F(v)$, respectively. Similarly, after lookup the largest indexes in A_{Tsum} and A_{Fsum}, we get $\sum_{v \in V} a_F(v)$ and $\sum_{v \in V} a_F(v)$. As such, we can get the corresponding R_T and U_F.

At last, we calculate N_T and N_F for each cache plan according to Equation 5 and 8. Then we get N_{total} by Equation 2 and search in parallel for the smallest N_{total} with the corresponding α.

After getting the optimal cache plans (B, α), Legion can automatically allocate the cache space and fill up the cache.

5 Implementation of Legion

Legion mainly consists of two components, which are the sampling server and the training backend. The sampling server is implemented from scratch and the training backend is built on top of Pytorch [31]. The sampling server is responsible for generating sampled results, and the training backend takes the sampled results as input to train the GNN models.

In Legion, every GPU executes the graph sampling, feature extraction, and model training stages, and all these stages are scheduled in a fine-grained pipeline to fully utilize the GPU computation cycles. Figure 7 illustrates how the training process is pipelined for a 2-hop GraphSAGE [16] model. In order to improve the overall throughput, we design an inter-batch pipeline overlapping the tasks of the sampling server and the training backend for different batches. E.g., the training of batch B_i can be overlapped with the sampling and feature extraction of batch B_{i+1}. To further improve the throughput of sampling and feature extraction, we design an intra-batch pipeline inside the sampling server. Specifically, we break down the workloads of the sampling server into four types, each of which corresponds to a type of operator: (1) Batch generator shuffles the local training vertices to generate seeds for mini batches; (2) Neighbor sampler executes the L-hop neighbor sampling; (4) Feature extractor extracts the feature of the batch seeds and vertices in the sampled results; (4) Graph constructor is used to generating the subgraph based on the sampled results. For the same batch, graph sampling and graph construction can be overlapped with feature extraction.

Table 1: GPU Server Statistics.

Server	DGX-V100	Siton	DGX-A100
GPU Type	16GB-V100x8	40GB-A100x8	80GB-A100x8
NVLink Topo.	$K_c = 2, K_g = 4$	$K_c = 4, K_g = 2$	$K_c = 1, K_g = 8$
PCIe Gen.	3.0x16	4.0x16	4.0x16
PCIe Topo.	4 switches, 2 GPUs/switch	2 switches, 4 GPUs/switch	4 switches, 2 GPUs/switch
CPU Mem.	384GB	1TB	1TB
CPU Core Num.	96	104	128
Sockets, NUMA Num.	2, 1	2, 2	2, 1

Table 2: Dataset Statistics.

Dataset	PR	PA	CO	UKS	UKL	CL
Vertices	2.4M	111M	65M	133M	0.79B	1B
Edges	120M	1.6B	1.8B	5.5B	47.2B	42.5B
Topology Storage	640M	6.4GB	7.2GB	22GB	189GB	170GB
Feature Size	100	128	256	256	128	128
Feature Storage	960M	56GB	65GB	136GB	400GB	512GB

6 Evaluation

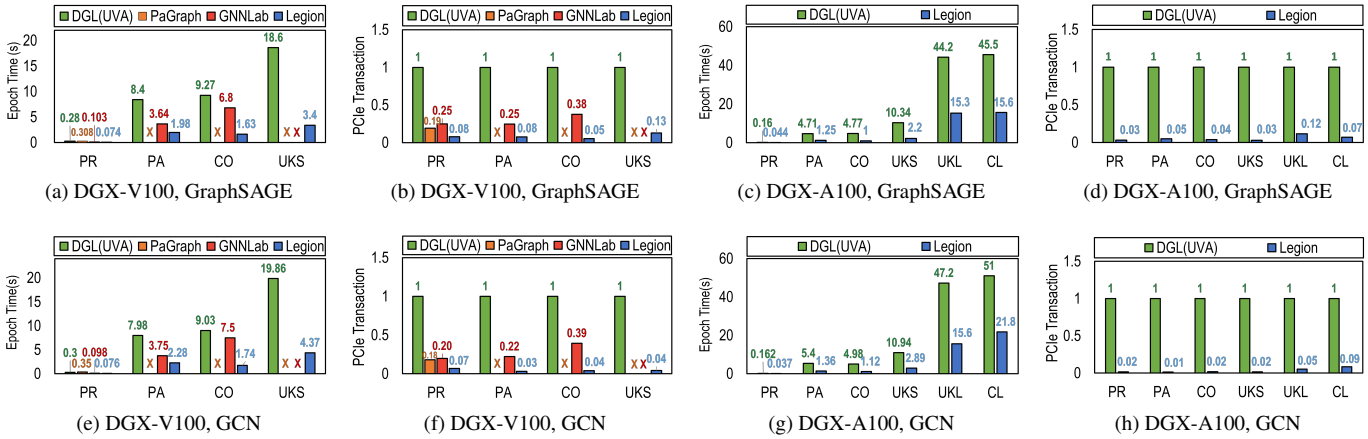
6.1 Experimental Setting

Experimental Platform. The experiments are conducted using three different GPU servers: DGX-V100, Siton, and DGX-A100, as shown in Table 1. For DGX-A100, we set the upper limit of GPU memory to 40 GB.

GNN Models. We use two sampling-based GNN models: GraphSAGE [16] and GCN [22], which both adopt a 2-hop random neighbor sampling. The sampling fan-outs are 25 and 10. The dimension of the hidden layers in both models is set to 256. Similar to existing work [47], the batch size is set to 8000. Unless explicitly explained, node classification is used as the GNN task.

Datasets. We conduct our experiments on multiple real-world graph datasets with various scales. Table 2 shows the dataset characteristics. The Products (PR) and Paper100M (PA) are available in Open Graph Benchmark [17]. The Com-Friendster (CO) graph is an online gaming network [46]. And the Uk-Union (UKS), UK-2014 (UKL), and Clue-web (CL) are from WebGraph [2–5]. As CO, UKS, UKL, and CL have no feature, we manually generate the features with the dimension specified as 128 or 256. Following PR's setting, we choose 10% of vertices from each graph as training vertices.

Baselines. We use DGL [42], PaGraph [23] and GNNLab [47] as the baseline systems. The DGL version is v0.9.1, which supports accessing graph topology and features via the UVA technique. We don't compare with Quiver [33] in the overall performance experiment as its open-sourced version cannot support training on servers with 8 GPUs. Instead,



(a) DGX-V100, GraphSAGE (b) DGX-V100, GraphSAGE (c) DGX-A100, GraphSAGE (d) DGX-A100, GraphSAGE

(e) DGX-V100, GCN (f) DGX-V100, GCN (g) DGX-A100, GCN (h) DGX-A100, GCN

Figure 8: Overall performance of Legion comparing with state-of-the-art systems. "×" denotes OOM (out of memory).

we implement a Quiver-like multi-GPU cache mechanism in Legion for comparison in Section 6.3.

6.2 End-to-end Performance

We compare the end-to-end performance of Legion with baseline systems on the DGX-V100 and DGX-A100 servers. On the DGX-V100 server, we evaluate PR, PA, CO, and UKS graphs whose graph topology and features can fit into 384 GB CPU memory. On the DGX-A100 server, we evaluate all six graphs. As PaGraph and GNNLab are implemented using CUDA 10 which cannot support A100 GPU, we exclude them from the experiments using DGX-A100.

Baseline Configuration. For all the baselines, we manually adjust their configurations to achieve optimal performance. DGL uses the UVA mode, where sampling is performed in GPU, and the topology and features are all stored in CPU memory. The number of worker threads in PaGraph is set to be 64 to maximize the CPU sampling throughput. For GNNLab, we adjust the numbers of sampling and training GPUs such that the overall throughput is maximized. In contrast, Legion relies on its automatic cache management mechanism to generate the unified cache plan.

Evaluation Metrics. We record the average epoch time for all systems. We also use PCM [18] to measure the maximum PCIe counter value across different sockets and report the normalized values based on the result of DGL for all systems.

Support training on large graphs. As shown in Figures 8a, 8e, 8c and 8g, Legion outperform all the baseline systems in every setting. Specifically, Legion achieves 3.78-5.69× speedup for GraphSAGE (3.5-5.19× for GCN) on DGX-V100 and 2.89-4.77× speedup for GraphSAGE (2.34-4.45× for GCN) on DGX-A100 over DGL(UVA). Figures 8b, 8f, 8d and 8h show that, compared with the baselines, Legion can sufficiently utilize the multi-GPU cache to minimize PCIe traffic incurred by CPU-GPU data transferring. GNNLab runs out of GPU memory for UKS on DGX-V100 as the size of graph topology exceeds the capacity of single GPU memory. PaGraph runs out of the CPU memory for most graphs except for PR on DGX-V100, as the memory management in PaGraph incurs extra memory overheads, including duplicated multi-hop neighbors in CPU memory and redundant intermediate buffers generated during computation.

Speedup over SOTA system on small graphs. Legion achieves 1.39-4.18× speedup for GraphSAGE (1.29-4.32× speedup for GCN) over GNNLab on the small graphs (PR, PA, CO). The performance gain mainly comes from two aspects. First, Figure 8b and 8f show that Legion significantly reduces the PCIe traffic for PA and CO, as it has a scalable multi-GPU cache design compared with GNNLab. The reduction of PCIe traffic relieves the CPU-GPU communication bottleneck such that the overall performance is improved. Second, Legion can use all GPUs for model training, while GNNLab needs to allocate several GPUs for sampling exclusively due to its factored design. In Legion, the graph sampling is overlapped by model training due to the fine-grained pipeline (see Section 5). E.g., when training GraphSAGE using the PR dataset, all the topology and feature data can be stored in GPU memory in both Legion and GNNLab. However, Legion can use 8 GPUs for training while GNNLab only uses 4 GPUs for training (see Figures 8a).

6.3 Effect of Hierarchical Partitioning

In this experiment, we examine the effect of hierarchical partitioning in Legion. We report the cache hit rates under different partition strategies in all three GPU servers: DGX-V100 (NV4: $K_c = 2$ and $K_g = 4$), Siton (NV2: $K_c = 4$ and $K_g = 2$) and DGX-A100 (NV8: $K_c = 1$ and $K_g = 8$).

6.3.1 Cache Performance

Baselines. For a fair comparison, we implement the cache designs of GNNLab, PaGraph-plus (described in Section 3.1), and Quiver-plus in Legion and compare their cache hit rates. Specifically, GNNLab maintains a globally replicated cache among all GPUs without using NVLinks (noPart+noNV).

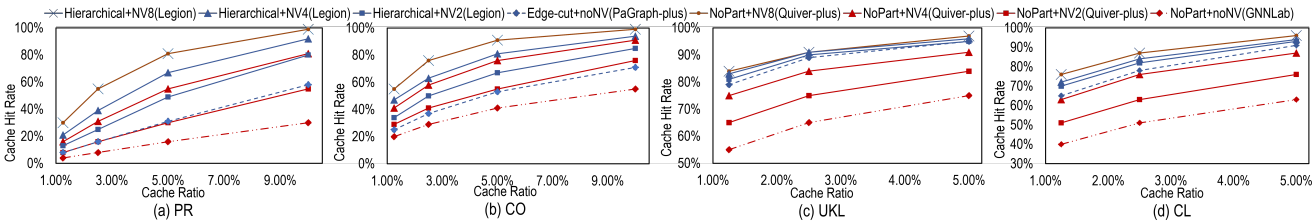


Figure 9: Effect of graph partition strategies (NoPart: no partitioning; Edge-cut: partitioning minimizing edge-cut; Hierarchical: hierarchical partitioning) to multi-GPU cache in terms of cache hit rate, with different NVLink infrastructures. (noNV: disable NVLinks; NV2: $K_c = 4$ and $K_g = 2$; NV4: $K_c = 2$ and $K_g = 4$; NV8: $K_c = 1$ and $K_g = 8$;).

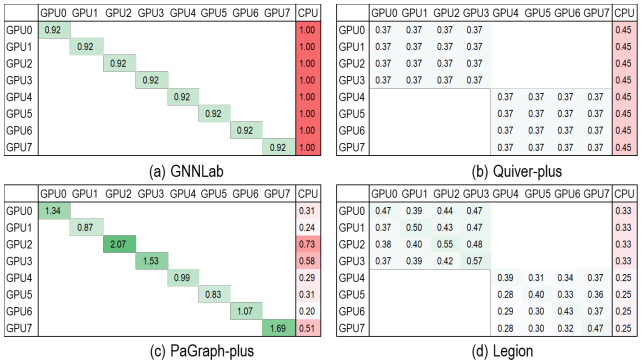


Figure 10: Data transferring in feature extraction of PA dataset on DGX-V100 (NV4). The rows and columns of each matrix denote the destination and source of data transferring. The right-most (red) column records the data transferring volume from CPU to GPU via PCIe. The middle (green) columns represent the GPU-GPU data transferring volume. We normalize the recorded values based on the CPU-GPU data transferring volumes in GNNLab.

Quiver-plus enables NVLink and maintains replicated cache among NVLink cliques (noPart+NV2 / noPart+NV4 / noPart+NV8). PaGraph-plus takes the XtraPulp [35] partitioning which minimizes across-partition edge-cuts and disables NVLinks (Edge-cut+noNV). Legion uses hierarchical partitioning (inter-NVLink-clique partitioning: XtraPulp) and enables NVLink (Hierarchical+NV2 / Hierarchical+NV4 / Hierarchical+NV8). We use the pre-sampling hotness metric for all these cache designs. The in-degree-based hotness metric in the original PaGraph and Quiver design are replaced with the pre-sampling hotness metric in Pagraph-plus and Quiver-plus, which has a better performance on cache hit rates [47].

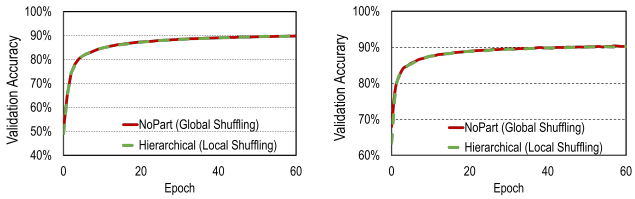
The datasets used in this experiment are PR, CO, UKL, and CL. We vary the cache ratio from 1.25% $|V|$ to 10% $|V|$ for PR and CO. For UKL and CL whose sizes are relatively large, the cache ratio varies from 1.25% $|V|$ to 5% $|V|$. Figure 9 shows that, for almost all the experiment settings, Legion has the highest cache hit rate. Specifically, Legion obviously outperforms Quiver-plus in the cases of NV2 and NV4, since Legion can reduce the inter-NVLink-clique cache duplication and achieves higher multi-GPU memory utilization compared with Quiver-plus. For the case of NV8, as all GPUs are in the same NVLink clique, the inter-clique graph partitioning in Legion can be skipped, and hierarchical partitioning turns into hash partitioning among all the GPUs, which is identical to Quiver-plus in the case of NV8. Legion outperforms PaGraph-plus because it has much less cache duplication. Specifically, PaGraph-plus's cache mechanism may replicate vertices with high global hotness on multiple GPUs. Compared with GNNLab, Legion has higher cache hit rates as it can scale up the cache capacity with the increase of GPUs, while GNNLab replicates the same feature cache across all GPUs. These results demonstrate that Legion can effectively adapt the cache plan to optimize the cache performance for multi-GPU servers with various NVLink topologies.

6.3.2 Data Transferring in Feature Extraction

In this experiment, we demonstrate the GPU-GPU and CPU-GPU data transferring volume during feature extraction using the PA dataset. Specifically, we perform the graph sampling and feature extraction stages using the PA graph on DGX-V100 (NV4) and record the data transferring volumes of feature extraction on each GPU in the format of a traffic matrix. We use GNNLab, PaGraph-plus, and Quiver-plus as the baselines, and set the feature cache ratio on each GPU to 2.5% $|V|$. The results are presented in Figure 10. We can see that Legion's data transferring volume from CPU to GPU is the smallest, indicating the best cache performance among the compared systems. As it is the GPU with the largest CPU-GPU data transferring volume that dominates the overall performance, although Legion's CPU-GPU volumes on some GPUs are higher than PaGraph-plus, Legion can still outperform PaGraph-plus because its largest CPU-GPU volume is lower than that of PaGraph-plus.

6.3.3 Model Convergence

Compared with global shuffling (randomly generating batch seeds from the vertex set of the entire graph), recent studies [23, 28] show that local shuffling (generating batch seeds within partitions) brings negligible impact on the rate of model convergence. Legion adopts local shuffling, and we conduct an experiment on the Siton server (NV2) to compare



(a) GraphSAGE (b) GCN

Figure 11: Comparing local shuffling and global shuffling on model convergence (NoPart: no partitioning; Hierarchical: hierarchical partitioning).

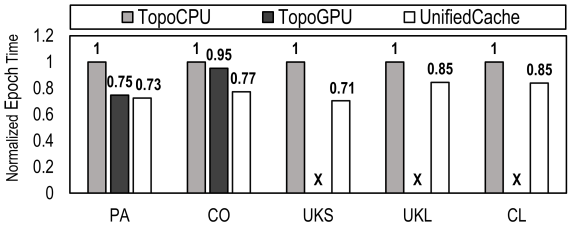


Figure 12: The impact of topology cache. "×" means OOM (out of memory).

its convergence speed with global shuffling on both GraphSAGE and GCN using the PR dataset. The results in Figure 11 show that the local shuffling of Legion could catch up with the convergence speed of global shuffling.

6.4 Effect of Unified Cache

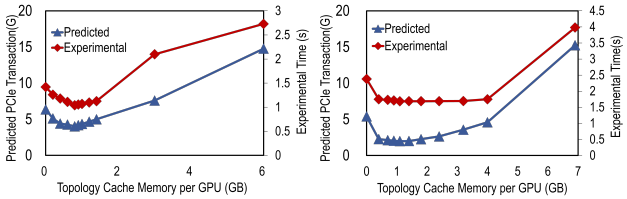
Different from existing cache-based systems, Legion's unified cache also takes graph topology into account. In this experiment, we demonstrate the benefits of topology cache.

We compare the training epoch time of unified cache in Legion with two baselines: (1) storing all topology in the CPU (denoted as TopoCPU) and (2) replicating the entire topology in every single GPU (denoted as TopoGPU). For a fair comparison, we implement both TopoCPU and TopoGPU in Legion and use the same GPU memory volume for the three settings. Among the three settings, TopoCPU has the most GPU memory available for the feature cache, and the TopoGPU has the least GPU memory for the feature cache or even runs out of GPU memory. We evaluate PA, CO, and UKS on DGX-V100 and evaluate UKL and CL on DGX-A100.

As shown in Figure 12, the unified cache outperforms the other two baselines for all graphs. This result demonstrates that, when the size of the feature cache exceeds a threshold, the increase of cache hit rate slows down. In this case, caching some hot topology data in GPU memory will save the system from severe PCIe contention incurred by graph sampling and benefit the overall GNN training throughput.

6.5 Evaluation of Cost Model

Legion proposes the cost model to guide allocating GPU memory for both graph topology and feature cache. In this experiment, we evaluate the effectiveness of this mechanism.



(a) PA, Single GPU (b) UKS, DGX-V100 (NV4)

Figure 13: Evaluation of cost model. The left y-axis means the PCIe transaction number predicted by the cost model. The right y-axis represents the experimental per-epoch graph sampling and feature extraction time.

Table 3: Evaluation of Partitioning Cost.

Dataset	PA (DGX-V100)	UKL (Siton)
Graph Partition(min)	7.2	75
Data Loading From Disk To Memory(min)	0.32	3.5
Node Classification Epoch(s)	1.98	15.6
Link Prediction Epoch(min)	49.8	402

Specifically, we compare the predicted PCIe traffic with the experimental per-epoch execution time of graph sampling and feature extraction. In the experiment using the PA dataset, the GPU memory allocated for the cache is 10 GB. And in the experiment using the UKS dataset, the GPU memory allocated for the cache is 8 GB. When varying the size of the topology cache, the size of the feature cache is adjusted accordingly. Figure 13 shows that our cost model can precisely predict the trend of per-epoch execution time without manual interference.

6.6 Partitioning Cost

In this experiment, we study the partitioning cost in Legion. We run our experiment on the UKL dataset that has the largest number of edges among all the datasets, resulting in the highest cost of edge-cut partitioning. We also present the results of the PA data (medium size) to show the partitioning costs of different graph scales. We partition PA on DGX-V100 and UKL on Siton using the XtraPulp algorithm. For node classification, we set the training set to be 10% of the total edges for both graphs. For link prediction, we set the training set to be 80% of total edges. When the graph is too large to be partitioned in memory, like UKL, we randomly sample a fraction of edges (25% for UKL) and keep all vertices in the graph such that the subgraph can be partitioned in memory. This technique can obviously speedup graph partitioning and preserves a low edge-cut ratio.

Table 3 shows the preprocessing cost of Legion's hierarchical partitioning. We observe that the partitioning cost is tolerable, because 1) we only partition the graph once but can use the partitioning results for multiple GNN training jobs, and 2) the GNN task like link prediction needs multiple epochs to converge while a single epoch often costs a long time to finish.

7 Related Work

To our knowledge, Legion is the first work that automatically pushes the envelope of multi-GPU systems for billion-scale GNN training. In the following, we contrast Legion and existing works in the following aspects.

GNN Frameworks. Several GNN systems [11, 12, 20, 23, 26, 33, 38, 42, 43, 47, 51, 53, 55] have emerged in recent years. Most of these GNN systems are built on top of deep learning frameworks like Pytorch [31], TensorFlow [1] and MXNet [9].

GPU Sampling. NextDoor [19] and C-SAW [30] focus on accelerating GPU sampling kernel. DGL [42] also supports GPU sampling in its recent release. Quiver [33] can support GPU sampling with the entire topology either stored in the single GPU or in the CPU memory. GNNLab [47] adopts a factored design where each GPU is dedicated to graph sampling or model training exclusively. In contrast, Legion uses all GPUs for end-to-end GNN acceleration.

Graph Partitioning. Graph partitioning such as [6, 14, 15, 21, 32, 35, 36, 39], has been widely adopted in GNN systems. DGL [42] adopts METIS [21] to partition the graph. PaGraph [23] adopts a self-reliant partitioning strategy with the goal of achieving balanced training vertex allocation across GPUs and improving data locality on every GPU. DGCL [7] adopts a partitioning algorithm to partition the graph's physical edges and features and store them among distributed machines. In contrast, Legion adopts hierarchical partitioning to automatically partition graphs to each GPU in a single multi-GPU server accordingly to GPU interconnections.

GPU Feature Cache. PaGraph [23], BGL [24], GNNLab [47], Quiver [33] and [29] explore feature caching on GPU to accelerate GNN training. PaGraph [23] and Quiver [33] use the in-degree of vertexes as the hotness metric. BGL [24] applies a FIFO dynamic cache policy and selects training vertices in a BFS order for a higher cache hit rate, but hinders model convergence and incurs cache replacement overheads. [29] uses a weighted reverse PageRank algorithm as a hotness metric. GNNLab [47] uses vertices' access frequencies in the pre-sampling epoch as a hotness metric. In contrast, Legion automatically caches both features and topology with the highest hotness. And Legion statically partitions the graph with minimal edge-cut to preserve intra-partition data locality. Figures 9 and 11 show that Legion can achieve a high cache hit rate even with small cache ratios without compromising the model convergence rate.

Large Graph Systems. SSD-based GNN systems [41] and distributed GNN systems [12, 24, 52, 54] also aim at large-graph training and propose distinct approaches to solve I/O problems at various levels. MariusGNN [41] minimizes I/O between SSD and CPU by including valid graph data in a single swap as much as possible. Systems like BGL [24], DistDGLv2 [54], and P3 [12] optimize network I/O between distributed machines, whose network performance can be improved when introducing GPU-centric SmartNIC [44]. In contrast, Legion focuses on utilizing GPU caches to minimize PCIe traffic from CPU memory to multiple GPUs, which is orthogonal to the above systems.

8 Conclusion

We present Legion, a system that automatically pushes the envelope of multi-GPU systems for billion-scale GNN training. Legion has three key innovations. First, we propose an NVLink-aware hierarchical partitioning technique that helps minimize cache replication and extends the threshold of cache capacity beyond the limit of a single GPU or NVLink clique. Second, we propose a novel hotness-aware unified cache mechanism that helps accelerate both graph sampling and feature extraction. Third, we present an automatic cache management mechanism enabling optimal cache planning without requiring extra knowledge of hardware specifications and GNN performance details from users. Experiments show Legion outperforms SOTA cache-based GNN systems up to 4.32× and supports training on billion-scale graphs. And Legion is open-sourced at https://github.com/RC4ML/Legion.

Acknowledgements. We thank our shepherd Anand Iyer and anonymous reviewers for their detailed feedback. The work is supported by the following grants: the Program of Zhejiang Province Science and Technology (2022C01044), a research grant from Alibaba Group through the Alibaba Innovative Research (AIR) Program, the Fundamental Research Funds for the Central Universities 226-2022-00151, Key Laboratory for Corneal Diseases Research of Zhejiang Province, Starry Night Science Fund of Zhejiang University Shanghai Institute for Advanced Study (SN-ZJU-SIAS-0010). Zeke Wang and Fei Wu are the corresponding authors.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberga, Sherry Moore Rajat Monga, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Martin Wicke Pete Warden, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: a system for large-scale machine learning. In *OSDI*, 2016.

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

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

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

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

[6] Erik G Boman, Karen D Devine, and Sivasankaran Rajamanickam. Scalable matrix computations on large scale-free graphs using 2d graph partitioning. In *SC*, 2013.

[7] Zhenkun Cai, Xiao Yan, Yidi Wu, Kaihao Ma, James Cheng, and Fan Yu. DGCL: An efficient communication library for distributed GNN training. In *Eurosys*, 2021.

[8] Jie Chen, Tengfei Ma, and Cao Xiao. Fastgcn: fast learning with graph convolutional networks via importance sampling. *arXiv preprint arXiv:1801.10247*, 2018.

[9] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[10] 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.

[11] Matthias Fey and Jan Eric Lenssen. Fast graph representation learning with pytorch geometric. *arXiv preprint arXiv:1903.02428*, 2019.

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

[13] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *ICML*, 2017.

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

[15] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *OSDI*, 2014.

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

[17] 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. *NIPS*, 2020.

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

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

[20] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. Improving the accuracy, scalability, and performance of graph neural networks with roc. *MLSys*, 2020.

[21] George Karypis and Vipin Kumar. Metis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. 1997.

[22] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[23] 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.

[24] 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. *arXiv preprint arXiv:2112.08541*, 2021.

[25] 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.

[26] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. Neugraph: Parallel deep neural network computation on large graphs. In *USENIX ATC*, 2019.

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

[28] Qi Meng, Wei Chen, Yue Wang, Zhi-Ming Ma, and Tie-Yan Liu. Convergence analysis of distributed stochastic gradient descent with shuffling. *Neurocomputing*, 2019.

[29] 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.

[30] 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.

[31] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary

DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. *NeurIPS*, 2019.

[32] Fabio Petroni, Leonardo Querzoni, Khuzaima Daudjee, Shahin Kamali, and Giorgio Iacoboni. Hdrf: Stream-based partitioning for power-law graphs. In *CIKM*, 2015.

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

[34] Zaid Qureshi, Vikram Sharma Mailthody, Isaac Gelado, Seungwon Min, Amna Masood, Jeongmin Park, Jinjun Xiong, CJ Newburn, Dmitri Vainbrand, I-Hsin Chung, et al. Gpu-initiated on-demand high-throughput storage access in the bam system architecture. In *ASPLOS*, 2023.

[35] George M Slota, Sivasankaran Rajamanickam, Karen Devine, and Kamesh Madduri. Partitioning trillion-edge graphs in minutes. In *IPDPS*, 2017.

[36] Isabelle Stanton and Gabriel Kliot. Streaming graph partitioning for large distributed graphs. In *SIGKDD*, 2012.

[37] Chang Su, Yu Hou, and Fei Wang. Gnn-based biomedical knowledge graph mining in drug development. In *Graph Neural Networks: Foundations, Frontiers, and Applications*. 2022.

[38] 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.

[39] Charalampos Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. Fennel: Streaming graph partitioning for massive scale graphs. In *WSDM*, 2014.

[40] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

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

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

[43] Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding. Gnnadvisor: An adaptive and efficient runtime system for gnnacceleration on gpus. In *OSDI*, 2021.

[44] Zeke Wang, Hongjing Huang, Jie Zhang, Fei Wu, and Gustavo Alonso. FpgaNIC: An FPGA-based versatile 100gb SmartNIC for GPUs. In *ATC*, 2022.

[45] Wikipedia. MaxCliqueDyn. https://en.wikipedia.org/wiki/MaxCliqueDyn_maximum_clique_algorithm, 2022.

[46] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. In *SIGKDD*, 2012.

[47] 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.

[48] 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.

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

[50] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. Graphsaint: Graph sampling based inductive learning method. *arXiv preprint arXiv:1907.04931*, 2019.

[51] Dalong Zhang, Xin Huang, Ziqi Liu, Zhiyang Hu, Xianzheng Song, Zhibang Ge, Zhiqiang Zhang, Lin Wang, Jun Zhou, Yang Shuang, and Yuan Qi. Agl: a scalable system for industrial-purpose graph machine learning. *arXiv preprint arXiv:2003.02454*, 2020.

[52] 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 *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, 2020.

[53] 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.

[54] Da Zheng, Xiang Song, Chengru Yang, Dominique LaSalle, Qidong Su, Minjie Wang, Chao Ma, and George Karypis. Distributed hybrid cpu and gpu training for graph neural networks on billion-scale graphs. *arXiv preprint arXiv:2112.15345*, 2021.

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

A Appendices

A.1 Generalization of Legion

Generalizing Legion to SSDs. Legion is primarily designed for in-memory graph training, but it can also be extended to SSD-based systems. First, Legion can still use GPU to execute end-to-end GNN training while storing graph topology and features in SSDs. The data path between GPU and SSDs could be enabled by BaM [34], which is a GPU-initiated on-demand high-throughput storage access technique. Second, the throughput of reading data from SSDs could be much lower than that in memory, leading to more severe I/O problems. Legion's hierarchical partitioning and unified cache design could still help reduce I/O and benefit overall throughput in this situation. Finally, due to the limited GPU memory, there still exists a trade-off between topology cache and feature cache in SSD-based systems. Thus automatic cache management could still be important and should be extended with more considerations of the specific hardware characteristics. We leave Legion's generalization to SSDs as our future work.
Generalizing Legion to none-NVLink systems. Legion can still bring performance benefits in multi-GPU systems without NVLink. To achieve this, Legion splits the graph into N partitions and each GPU maintains a cache for a partition. This approach can have a higher cache hit rate compared to replicating a global cache, as shown in Figure 9. We can also apply Legion on other GPU platforms, e.g., on AMD GPUs by leveraging the AMD Infinity inter-GPU bus.