UGache: A Unified GPU Cache for Embedding-based Deep Learning

Xiaoniu Song[1,2] Yiwen Zhang[1] Rong Chen[1,2] Haibo Chen[1]

[1]Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
[2]Shanghai Artificial Intelligence Laboratory

Abstract

This paper presents UGache, a unified multi-GPU cache system for embedding-based deep learning (EmbDL). UGache is primarily motivated by the unique characteristics of EmbDL applications, namely read-only, batched, skewed, and predictable embedding accesses. UGache introduces a novel factored extraction mechanism that avoids bandwidth congestion to fully exploit high-speed cross-GPU interconnects (e.g., NVLink and NVSwitch). Based on a new *hotness* metric, UGache also provides a near-optimal cache policy that balances local and remote access to minimize the extraction time. We have implemented UGache and integrated it into two representative frameworks, TensorFlow and PyTorch. Evaluation using two typical types of EmbDL applications, namely graph neural network training and deep learning recommendation inference, shows that UGache outperforms state-of-the-art replication and partition designs by an average of 1.93× and 1.63× (up to 5.25× and 3.45×), respectively.

Keywords: GPU cache, Embedding, GPU interconnect

ACM Reference Format:
Xiaoniu Song, Yiwen Zhang, Rong Chen, and Haibo Chen. 2023. UGache: A Unified GPU Cache for Embedding-based Deep Learning. In *ACM SIGOPS 29th Symposium on Operating Systems Principles (SOSP '23), October 23–26, 2023, Koblenz, Germany.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3600006.3613169

1 Introduction

Embedding-based deep learning (EmbDL), as exemplified by deep learning recommendation (DLR) and graph neural network (GNN), has garnered significant attention and found a widespread use in production environments. Unlike traditional deep learning (DL), EmbDL can efficiently handle sparse inputs such as user IDs and graph nodes by mapping them into embedding entries via embedding tables. However, the embedding table size associated with DLR and GNN applications can typically reach up to 400 GBs, surpassing the limited capacity of GPU memory. Consequently, fetching embedding entries from host memory via the PCIe bandwidth, which is an order of magnitude slower than the GPU's high bandwidth memory (HBM), has become a system bottleneck.

In real-world workloads, accessing embedding entries often reveals a skewed pattern where some entries are accessed more frequently than others. Therefore, many EmbDL systems cache a portion of the most frequently accessed (i.e., hottest) embeddings on GPU memory [43, 46]. However, the performance bottleneck still remains due to memory limitations on a single GPU (typically several dozens of GBs). For example, prior work [46] reports that 67% of the time (20.7 ms out of 31.3 ms) is spent on embedding extraction of GNN training, and fetching missing data from host memory accounts for 86% of the extraction time (17.9 ms).

Recently, multi-GPU platforms with high-speed interconnects (e.g., NVLink and NVSwitch) have become mainstream in modern datacenters. In such setups, each GPU can directly access the memory of other GPUs with an-order-of-magnitude higher bandwidth compared to accessing host memory. This presents an opportunity to build a larger aggregated embedding cache across multiple GPUs. However, blindly deploying single-GPU cache systems [43, 46] on multi-GPU platforms is inefficient, since each GPU will independently cache the same frequently accessed embeddings. Such a *replication* cache policy disregards the high bandwidth among GPUs and causes cache redundancy.

Several recent research efforts [5, 7, 8, 45] therefore attempt to develop a multi-GPU embedding cache that efficiently utilizes high-speed interconnects. However, prior work is often inefficient due to the variety and complexity of multi-GPU platforms and fails to address fundamental challenges in cache policy and extraction mechanism. In terms of cache policy, a *partition* approach is proposed to cache as many embedding entries as possible. However, this policy suffers from diminishing marginal utility, as it slightly improves the global hit rate while greatly reducing the local hit rate. Given that the inter-GPU bandwidth is still much slower than the local bandwidth (300 vs. 900 GB/s), the performance gain will be insignificant or even negative. In terms of

SOSP '23, October 23–26, 2023, Koblenz, Germany

ACM ISBN 979-8-4007-0229-7/23/10…$15.00
https://doi.org/10.1145/3600006.3613169

extraction mechanism, most existing EmbDL systems leverage message passing for cross-GPU embedding extraction, which involves additional cost of buffering and copying data. While recent work [45] eliminates this cost by exploiting the zero-copy semantics of modern GPUs, it still suffers from bandwidth congestion and consequent GPU core stall.

We propose UGache, an embedding cache system that addresses challenges related to extraction mechanism and cache policy, efficiently unifying the memory of multiple GPUs. In order to avoid GPU cores from excessively competing for limited link bandwidth, UGache introduces a novel factored extraction mechanism that statically dedicates different GPU cores to fetch embedding entries from different sources. This dedication turns previous random, unrestricted parallelism into an organized one, thereby avoiding bandwidth congestion and consequent core stall. To achieve an optimal cache policy, the key is to find a balance between caching more distinct entries to improve global hit rate and caching more replicas to improve local hit rate. UGache introduces a new metric called "*hotness*" to measure the access frequency of embedding entries and models the extraction time on multi-GPU platforms as a mixed-integer linear programming (MILP) problem. Given available GPU memory, the bandwidth hierarchy, and hotness statistics, UGache can find a near-optimal solution to minimize the extraction time.

UGache acts as an embedding layer that can be seamlessly integrated into EmbDL application workflows. To demonstrate its flexibility and support for rich EmbDL applications, we have implemented UGache and integrated it into two popular DL frameworks, TensorFlow and PyTorch. The source code of UGache is publicly available at https://github.com/SJTU-IPADS/ugache. Our evaluation included two typical types of EmbDL applications: GNN training and DLR inference. For GNN training, UGache outperforms state-of-the-art replication and partition designs by an average of 2.21× and 1.33× (up to 5.25× and 1.85×), respectively. Similarly, for DLR inference, UGache still provides an average of 1.51× and 2.07× performance improvements (up to 2.34× and 3.45×), respectively.

Contributions. We make the following contributions.

(1) A comprehensive analysis of performance issues and challenges for building an efficient embedding cache on multi-GPU platforms (§3).

(2) A factored extraction mechanism for implementing cross-GPU embedding extraction to avoid bandwidth congestion and improve GPU utilization (§5).

(3) A cache policy solver that handles the variety and complexity of multi-GPU platforms, ensuring minimized extraction time through mixed-integer linear programming (§6).

(4) A prototype implementation integrated into mainstream DL frameworks (§7) and an evaluation that shows the efficacy and efficiency of UGache over state-of-the-art (§8).

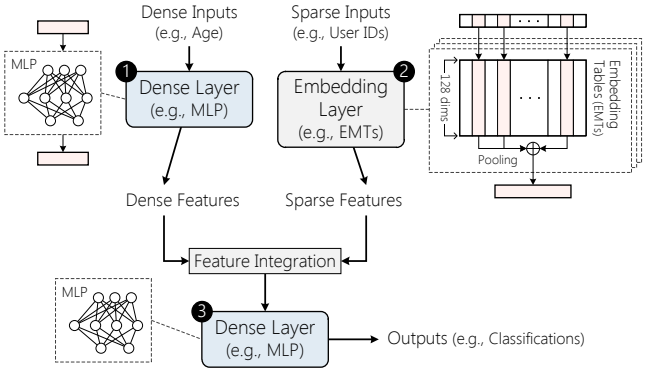

Figure 1. *The execution flow of embedding-based deep learning.*

2 Characterizing Embedding-based DL

The embedding technique is designed to handle sparse inputs in deep learning applications. Compared to traditional deep learning that only uses dense inputs such as user age or pixel color in images, embedding-based deep learning (EmbDL) involves sparse inputs that are one-hot and encoded as a list of IDs in a large value domain. For instance, in the deep learning recommendation (DLR) model, the sparse inputs may refer to the ID of a movie or a user involved in the recommendation. In GNN, the sparse inputs may refer to the ID of a paper or author in a citation graph.

The traditional DL model cannot process sparse inputs directly; therefore, an embedding layer is introduced to convert them into dense inputs (see Figure 1). Specifically, *embedding tables* (EMTs) represent a mapping from sparse inputs into dense values where keys are IDs and values are D-dimensional embedding vector entries. The embedding table is often organized as a matrix with shape $N \times D$, where N represents possible ID cases. The embedding layer extracts corresponding entries for each key from the embedding table into a continuous output region. These extracted embedding entries can be used in conjunction with dense layers (e.g., MLP layers) and dense inputs for further computation.

In EmbDL, the end-to-end time is often dominated by the embedding layer [21, 29, 43, 44, 46]. Compared to megabyte-sized dense layers, typical embedding layers can reach sizes up to hundreds of gigabytes, far exceeding the capacity of existing GPU's memory, and are often stored in host memory. Due to PCIe bandwidth limitations, time spent transferring the embedding from host to GPU becomes the system bottleneck. According to Table 1, during the training of unsupervised GNN, the embedding layer can take up to 113 ms, while dense layers only take around 10 ms.

Characteristics. We summarize the following features of embedding access.

Read-only access. In EmbDL, embedding access is typically restricted to read-only. This is due to the read-only nature of inference and the use of *pre-training* schemes, which ensure read-only access to embedding even during training.

In pre-training, the embedding table is trained beforehand and distributed across different downstream workloads. The model's dense portion is trained and updated in downstream workloads, while the embedding table remains static. For instance, the embedding table in GNN is incorporated into the dataset and remains unchanged during the model training.

Batched, subset access. Computation in DL applications is often performed in a batched, iterative manner. However, unlike traditional DL applications where all parameters participate in each iteration, EmbDL only accesses a subset of embedding entries using batched sparse inputs as keys. The final number of selected entries is often orders-of-magnitude larger than the original batch size of the input. For instance, in DLR, each input in a batch contains multiple keys for dozens, or even hundreds, of embedding tables. Similarly, in GNN, the embedding of k-hop neighbors of each input node is also required. Consequently, the final number of embedding entries to access can be at the million level.

Skewed access. In the subset access pattern, the keys of the embedding entries are selected with skewness rather than uniformity. For instance, in DLR [35], inputs closely relate to user preferences which follow a power-law distribution [38, 38, 43, 44]. In GNN, due to the power-law nature of the graph topology, the k-hop neighbors of input nodes are also skewed [15, 19, 22], leading to skewed access of embedding entries [29, 39, 46].

Stable, predictable access. The skewness of accessing embeddings is predictable and stable in EmbDL. Mature methods exist in current systems to predict the frequency of access for each embedding entry. For instance, during model training, profiling the access pattern of the initial few epochs can anticipate access in subsequent epochs [46]. Similarly, in DLR inference, hot entries in different daily traces are highly alike [17]. In GNN, leveraging the graph's degree can help since embedding entries associated with high-degree nodes are more likely to be accessed [29].

3 Challenges of Multi-GPU Embedding Cache

Single-GPU embedding cache. The characteristics of embedding access in EmbDL applications inspire the design of embedding cache [16, 18, 29, 43, 46], which caches frequently accessed entries into GPU memory to reduce the cost of fetching data from host memory. However, the performance issue remains due to memory limitations on a single GPU, and the time of embedding layer still dominates the end-to-end time. As shown in Table 1, in unsupervised GNN training, enabling embedding cache (w/ $) on a single GPU can reduce the extraction time from 113.3 ms to 20.7 ms, yet it still takes up about 66% of the end-to-end time.

Multi-GPU platforms with fast interconnects. In modern datacenters, multiple GPUs with fast interconnects (e.g., NVLink) on a single machine has been a common setting.

Table 1. *The breakdown of runtime and data amount for a typical EmbDL application. The MLP and EMT layers are the same as those shown in Figure 1. "w/ $" indicates that the embedding cache is enabled. "Access G_{mem} Ratio" means the ratio of GPU memory accesses to total memory accesses.* **EmbDL***: Unsupervised GraphSAGE [24] training.* **Dataset***: OGB-MAG240M [3].* **Testbed***: The server has one NVIDIA A100 GPU with 80 GB memory.*

EmbDL Application	❶+❸ MLP	❷ EMT (w/ $)	Total (w/ $)
Execution Time (ms)	10.6	113.3 (20.7)	123.9 (31.3)
Data Size (GB)	0.002	363 (69.6 in $)	363 (69.6 in $)
Access G_{mem} Ratio	100%	0% (84.6%)	0% (84.6%)

These interconnects unite the high bandwidth memory (HBM) of multiple GPUs, enabling a larger cache. However, creating an embedding cache across multiple GPUs poses two fundamental challenges in *cache policy* and *extraction mechanism*. The cache policy must reasonably place embedding entries across multiple GPUs, while the extraction mechanism must efficiently utilize fast connections to fetch embeddings across multiple GPUs. Existing systems offer suboptimal solutions from both perspectives.

3.1 Cache Policy

The cache policy of existing multi-GPU embedding cache can be categorized as replication or partition.

Replication cache. Systems such as HPS [43] and GNNLab [46] directly port single-GPU cache to multi-GPU platform, where each GPU independently caches the hottest entries. However, in data-parallel deployment, requests are randomly sent to GPUs, causing each GPU to observe highly similar access frequency. Consequently, the cache of each GPU covers similar requests, behaving like a replication cache, which wastes bandwidth across GPUs. On HPS, we observe that over 99% of cache-hit requests on each GPU are identical when running DLR inference workload.

Partition cache. To utilize the fast interconnects between GPUs, existing systems evenly partition the hottest entries among each GPU [5, 7, 8, 45]. This partition policy caches as many individual entries as possible and serves majority of accesses through fast GPU interconnects, resulting in improved performance.

However, the partition policy faces issues with marginal effects and low local hit rates. With a given total GPU memory budget, the partition policy blindly caches as many embeddings into GPU memory as possible. Since the power-law distribution of embedding access results in a long-tail effect, increasing cache capacities using a partition policy does not lead to proportionate increases in hit rates. On the other hand, because the partition policy extracts most entries from remote GPUs, it results in a much lower local hit rate compared to the replication policy. Since the bandwidth between GPUs is still not fast enough, partition policy does not significantly improve, and can even worsen performance compared

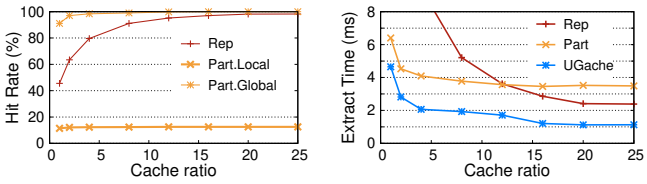

Figure 2. *Comparison of (a) hit rate and (b) extraction time between replication and partition cache with the increase of cache ratio. Evaluation using supervised GraphSAGE training with OGB-Papers100M on a 8×A100 platform.*

to replication policy, despite the fact that partition policy benefits from fast GPU interconnects.

Challenges. We train supervised GraphSAGE [24] with OGB-Papers100M [4] on a testbed with eight A100 GPUs.[1] Figure 2(a) shows the hit rates of using replication and partition cache. For partition cache, we present both the global and local hit rates to differentiate between cache hits on local memory and NVLink. The replication policy reaches a 95% local hit rate under a cache ratio of 12%, but the extraction is still bottlenecked by PCIe, which is nearly two-orders-of-magnitude slower than GPU's local bandwidth. The partition policy, however, only improves the global hit rate from replication's 95% to 99%, due to the long-tail nature of power-law distribution. Also, this improvement in global hit rate comes at the cost of the low local hit rate. The partition policy forces $\frac{7}{8}$ hit access on other GPUs, decreasing the local hit rate to 12%. As the bandwidth of NVLink is still much lower than GPU's local bandwidth, the performance improvement of partition is eliminated, as shown in Figure 2(b). Under a cache ratio of 12%, the partition policy achieves a performance identical to the replication policy. Meanwhile, the partition policy cannot leverage a cache capacity above 12.5%, while the replication policy continues to increase local hit rate and surpass the partition policy.

Several optimizations have been proposed recently to increase the local hit rate and reduce non-local accesses for partition cache, including locality-aware dispatching [31] and vertical partitioning [21]. However, these approaches are still inefficient in memory utilization due to the long-tail effect. Furthermore, they are limited to specific workloads (e.g., GNN) and require significant changes in applications, which hinders their use in more general scenarios.

3.2 Extraction Mechanism

The variety and limitation of existing multi-GPU platform's topology and bandwidth bring challenges to developing an efficient extraction mechanism.

GPU topology. The topology of GPU interconnects can be categorized into two types: ***hard-wired*** and ***switch-based***. In a hard-wired platform, the total outbound bandwidth of a single GPU is physically divided among all connected remote

[1]Detailed experimental setup can be found in §8.

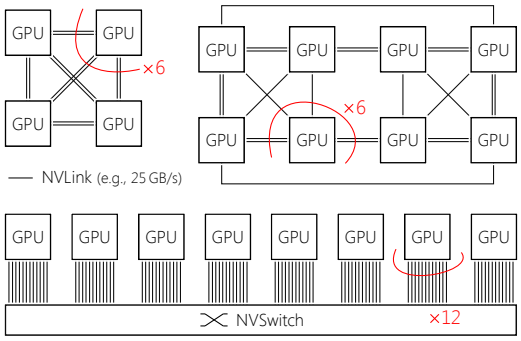

Figure 3. *The interconnect topology of multi-GPU servers: (a) 4 GPU with hard-wired interconnects, (b) 8 GPU with hard-wired interconnect like DGX-1 (V100), and (c) 8 GPU with switch-based interconnects like DGX A100.*

GPUs, creating a uniform, fully-connected graph as shown in Figure 3(a). This ensures that the connection between each pair can be concurrently utilized without collision. However, as the number of GPUs scales up, the connections between GPUs become irregular due to numeric limitations of the GPU's port. Figure 3(b) shows the non-uniform topology of DGX-1 system with 8×V100 GPUs. The bandwidth varies between GPU pairs, and some GPU pairs are even unconnected, forcing them to resort to the slower PCIe links.

In a switch-based platform shown in Figure 3(c), all GPUs are directly connected to NVSwitch, where the bandwidth between different GPU pairs is dynamically allocated. While the NVSwitch's bandwidth is able to support all GPUs' full outbound bandwidth, bandwidth collision can still occur when a GPU is accessed by multiple GPUs simultaneously.

Remote extraction arrangement. The procedure for reading embedding entries is accomplished through the `extract` function below:

```
__global__ void Extract(keys, output):
1  for i, key in enumerate(keys)
2    Elem* src = locate(key)
3    for j in range(dimension)
4      output[i][j] = src[j]
```

In each iteration, the application provides a batch of keys that identify the required entries and calls the `extract` function. These keys are then dispatched to a set of cores on the GPU, such as *stream processors* (SM) in NVIDIA GPUs, where each core concurrently extracts entries for different keys.

To perform the extraction procedure on a multi-GPU platform, existing systems can be classified into two categories: *message-based* and *peer-based* approaches. Message-based systems utilize message-passing interfaces [5, 7, 8, 20] to exchange embedding entries spread across the embedding table. These systems employ buffering to conduct one round of large message passing instead of multiple rounds of small messages. Specifically, each GPU first extracts globally required embedding entries into a contiguous buffer, then exchanges these buffers, and finally reorders the exchanged buffers back to the original key order of the input batch. This

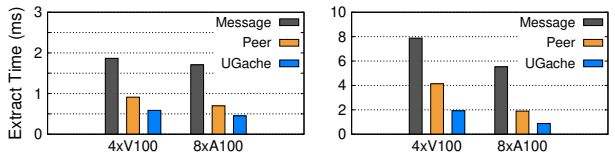

Figure 4. *The extraction time in DLR inference with (a) criteo-TB and (b) a Zipfian-generated synthetic dataset ($\alpha = 1.2$) using different extraction mechanism.*

approach leverages AllToAll primitives in collective communication libraries (e.g., NCCL [11]) to exchange buffers, achieving both high-efficiency embedding exchange and adaptability to various multi-GPU platforms. However, the inevitable buffering involves multiple rounds of data movement, and causes performance degradation [45].

A modern multi-GPU platform can unify host memory and every GPU memory into a single address space [1], allowing for peer-based access. This means that a GPU can directly load or store a *non-local* address, eliminating redundant data movement in the message-based approach and enabling zero-copy embedding extraction. Prior work [33, 45] has utilized peer-based access to optimize embedding extraction. A single kernel function can be launched to extract from multiple source locations, where each streaming multiprocessor (SM) locates the required embeddings on the fly through a hash table or hash function (refer to the `locate` function at Line 2 in the `extract` function above).

The peer-based approach requires manual handling of various topology and link usage. However, we have found that existing systems' naive use of peer access has flaws in both aspects. The existing system overlooks the mismatch between the high parallelism of GPUs and interconnects' capacity, resulting in bandwidth congestion that causes GPU cores to stall and reduces system performance by up to 50%. Additionally, existing systems assume a uniform, fully-connected graph for GPU interconnect topology and employ a partition policy, leading to decreased performance when deployed to platforms with non-uniform bandwidths and unconnected GPU pairs like 8×V100.

To demonstrate the performance issues in existing extraction mechanisms, we conducted DLR inference with Criteo-TB [6] and a Zipfian-generated synthetic dataset ($\alpha = 1.2$). As shown in Figure 4, the peer-based approach shows a significant improvement in performance compared to the message-based approach, which involves additional data movement. However, a considerable gap still remains between the peer-based approach and our proposed system.

4 Overview of UGache

We propose UGache, a unified multi-GPU embedding cache system for EmbDL applications. As shown in Figure 5, UGache seamlessly enhances existing embedding layers by caching embedding entries across multiple GPUs while hiding platform details. Internally, UGache comprises two major components: ***Extractor*** (§5) and ***Solver*** (§6), which address two

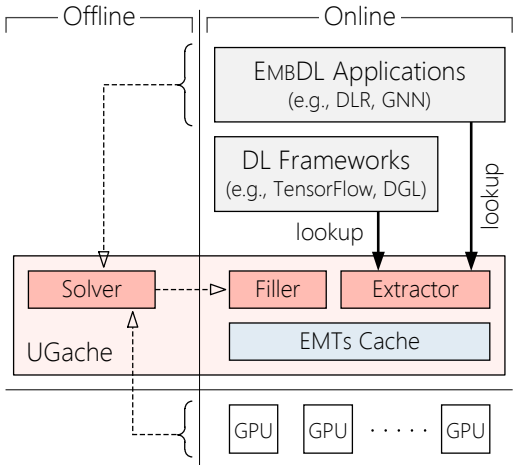

Figure 5. *The system architecture of UGache. As an embedding layer, UGache serves lookups from applications and DL frameworks.*

fundamental challenges in *extraction mechanism* and *cache policy*, respectively.

Extractor provides a factored extraction mechanism to extract embedding entries from multiple sources. The core idea is to statically dedicate GPU cores to access different sources. This static dedication limits concurrent GPU cores accessing same link within the link's capacity, thus avoiding bandwidth congestion and consequent core stall. For switch-based platforms, it also avoids cross-GPU bandwidth collision without involving explicit synchronization. Extractor further introduces local extraction padding to tolerate potential load imbalance caused by dedication. Consequently, UGache improves bandwidth and GPU core utilization and speeds up embedding extraction.

Solver determines the cache policy for placing embedding entries on multiple GPUs and provides guidance to Extractor on where to find the entries. To obtain an optimal cache policy, Solver finds a balance between caching more distinct entries to improve global hit rate and caching more replicas to improve local hit rate. To achieve this balance, Solver defines a hotness metric to measure the access frequency for each entry and profiles hardware platform's information to estimate embedding extraction time. Then, it utilizes MILP to solve a cache policy to minimize the extraction time. To simplify the solving procedure, Solver further batches similar entries to reduce the scale of MILP problem. Cooperating with UGache's Extractor, Solver further guarantees load balance, handles diverse platforms, and ultimately reduces global extraction time.

UGache adopts a straightforward design that takes advantage of EmbDL's batched, read-only access guarantee to coordinate Extractor and Solver without the complexity of maintaining cache consistency. In the foreground, Extractor processes lookup requests from DL frameworks and applications. In the background, Solver solves a new cache policy

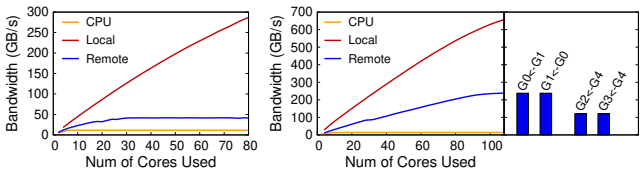

Figure 6. *The tolerance of concurrent cores for each source location. Tested on (a) 4×V100, a hard-wired platform and (b) 8×A100, a switch-based platform.*

and invokes Filler to (re)fill the cache content when necessary. Coordination between Extractor and Solver is achieved through a simple per-GPU hashtable. Each cached embedding entry is associated with its source location in the format of `<GPU_i, Offset>`, which guides Extractor to read the entry from `Offset` on `GPU_i`, while Solver and Filler update the hashtable and cache contents.

5 Extraction Mechanism

To achieve efficient embedding extraction across multiple GPUs, the major challenge is handling the conflict between high parallelism of GPUs and limited bandwidth to non-local memory. We conducted a microbenchmark and found that non-local memory can only tolerate a few concurrent GPU cores. Then we analyse how existing systems suffer from this limitation, and present how UGache's Extractor resolves it.

5.1 Characteristics of Extraction Procedure

Figure 6(a) illustrates link tolerance of concurrent GPU cores. While local memory allows for all cores to extract concurrently, non-local memory can only tolerate a portion of cores to exhaust its bandwidth. PCIe limits host extraction bandwidth, allowing fewer than 10% of cores. On hard-wired platform like 4×V100 (§3.2), the outbound bandwidth is physically partitioned to other three GPUs, causing the tolerance of $\frac{1}{3}$ cores to concurrently extract from each remote GPU.

On switch-based multi-GPU platform, NVSwitch dynamically allocates bandwidth instead of static partitioning. Figure 6(b) shows the same test on a switch-based 8×A100 platform. When only single GPU is extracting without interference from other GPU, the remote bandwidth tolerates almost all GPU cores. However, when multiple GPUs simultaneously extract from the same GPU, their bandwidth conflicts as Figure 6(b)'s right part demonstrates.

5.2 Performance Issues: Link Congestion

Existing systems (e.g., WholeGraph [45]) utilize peer-based approach to extract entries (§3.2). They leverage the high parallelism of GPUs by enabling GPU cores to extract from different locations concurrently, albeit in an unorganized manner. The input keys in the current batch are randomly dispatched to GPU cores. Each core identifies the source location of the dispatched key and fetches the corresponding embedding entry to local memory. However, the random dispatching manner can easily cause link congestion when concurrent cores are extracting from the same source, which exhausts the link bandwidth towards the source location.

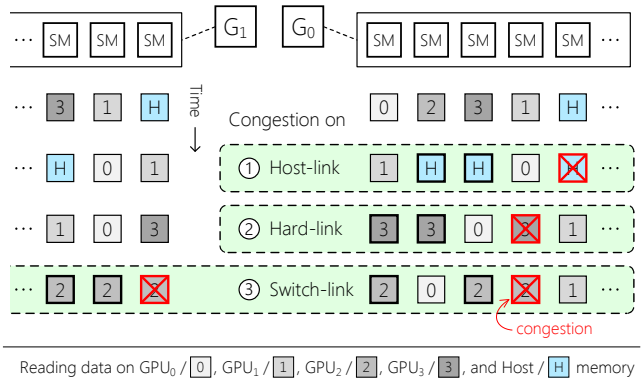

Figure 7. *An example of congestions on different interconnects when accessing data on GPU memory and host memory.*

Figure 7 provides an example of congestion. Each numbered block represents a key to be read and its source location. While the random partition of embeddings ensures statistical load balance of keys to read from each source at a global perspective, the random dispatch manner causes an instantaneous load imbalance from a local perspective. The first and second dashed boxes illustrate that too many cores are concurrently reading from a slow physical link (i.e., PCIe to host memory, partitioned link to pair GPU in a hard-wired platform). These slow links tolerate fewer cores, forcing over-allocated cores to stall, causing low core utilization. The third dashed box illustrates that in a switch-based platform, concurrent cores of different GPUs may also exhaust inbound link bandwidth and lead to GPU core stall.

5.3 Factorized Extraction Mechanism

UGache proposes a factored extraction mechanism (FEM) based on the above observation. The idea is to dedicate specialized cores to extract from each source location, limiting concurrency within link's tolerance, preventing link congestion and core stall.

Figure 8 shows the scheme of factored extraction mechanism. With batched incoming keys of embedding entries spread across multiple source locations, these keys are grouped according to their source location. Then, GPU cores are dedicated to handle non-local groups first. For each non-local group of keys, a subset of GPU cores are dedicated to extract their embedding entries. This limits the number of concurrent cores extracting from the same link within the link's tolerance, avoiding congestion and stalls.

Dedicating cores to non-local extraction avoids congestion but may result in ragged extraction time and cause load imbalance. UGache leverages the remaining local group to pad the ragged extraction time of the non-local group. Local extraction is arranged after non-local extraction and given low priority. Once the corresponding extraction of the non-local group finishes, dedicated cores handle the remaining

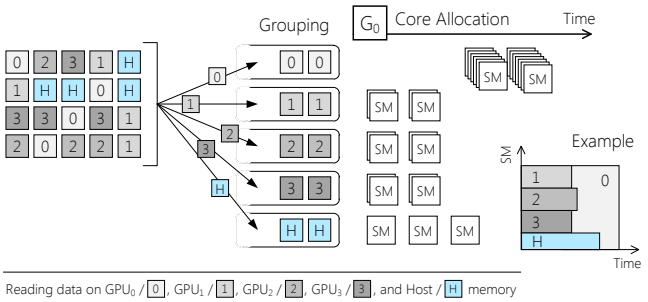

Figure 8. *The execution flow of factored extraction mechanism.*

keys in the local group. Since local extraction can tolerant all GPU cores, ragged extraction time can be sufficiently padded.

Core Dedication Strategy. The remaining question is how to determine the number of cores dedicated to each non-local group. Figure 6 indicates that only a few cores are needed for host extraction. Given the low host extraction bandwidth, UGache first dedicates a small number of cores for host to prevent extremely ragged time.

For remote GPUs connected via NVLink, the remaining cores are sliced based on the ratio of link bandwidth. This design is compatible with both switch-based and hard-wired platforms. On a hard-wired platform, given the small number of cores dedicated for host extraction, the remaining cores are in line with the tolerance of remaining cross-GPU links. Thus, the dedication ensures bandwidth and core utilization, and handles potential non-uniform bandwidth.

For switch-based platforms, the remaining cores are divided equally among each remote GPU. This abstracts the switch topology into a fully connected hard-wired platform. On a multi-GPU platform with N GPUs, the dedication only occupies $\frac{1}{N-1}$ outbound bandwidth of each remote GPU, due to the control of number of cores as Figure 6. Even when multiple GPUs are extracting simultaneously from the same remote GPU, their bandwidth does not overlap, resulting in implicit coordination to avoid bandwidth congestion.

The static dedication allows UGache to fully utilize all GPU cores for concurrent embedding extraction. By avoiding GPU cores competing for and being stalled by a congested link, UGache efficiently utilizes multiple links concurrently, and improves link utilization from a global perspective.

6 Cache Policy

UGache's Solver provides cache policy for guiding the placement of embedding entries on multiple GPUs and directs UGache's Extractor to locate embedding entries. Through selectively caching more copies of frequently accessed embedding entries, Solver can control the ratio of data extracted from different sources (i.e., local GPU memory, remote GPU memory with varying bandwidths, and host memory). The control decision takes into account the distribution of access frequency of each embedding entry and the topology of the multi-GPU platform, as well as its bandwidth hierarchy. To make a reasonable decision, Solver defines a hotness metric to model the access frequency. Based on hotness and the profiled information of the multi-GPU platform, Solver models the optimal cache policy as a MILP problem and provides a solution that minimizes the estimated extraction time.

Table 2. *Terminology*

Symbol	Type	Description
$\mathbb{G}$	Par	Available GPUs
$\mathbb{M}$	Par	Available source (all GPUs and host DRAM)
$\mathbb{E}$	Par	All cacheable entries
Cap_j	Par	Capacity of source location $j \in \mathbb{M}$
$T_{i \leftarrow j}$	Par	Time for GPU_i to extract an entry from $j \in \mathbb{M}$
h_e	Par	Hotness of entry e
$R_{i \leftarrow j}$	Par	Ratio of GPU_i's core to extract from $j \in \mathbb{M}$
$a^e_{i \leftarrow j}$	Var	Whether GPU_i reads entry e from $j \in \mathbb{M}$
t_i	Var	Total extraction time of GPU_i
t^j_i	Var	Extraction time of GPU_i spent on $j \in \mathbb{M}$
s^e_j	Var	Whether entry e is stored on $j \in \mathbb{M}$

6.1 Hotness Metric

The hotness metric measures the frequency of access for each embedding entry. UGache utilizes this metric to estimate the contribution of accessing different entries to the total extraction time.

One way to implement the hotness metric is to internally count the accesses of each individual entry in UGache. However, real-world applications often provide additional semantics to simplify measuring hotness. For example, in model training workloads, the same dataset is accessed repeatedly by each epoch. Recent work [46] observes that measuring hotness solely on the first epoch is sufficient to estimate the frequency of subsequent epochs. In many GNN workloads, the vertex degree in graph datasets can approximate the access frequency; vertices with higher degrees are more likely to be accessed [29]. To ensure flexibility for applications with these semantics, UGache's Solver allows applications to directly provide measured hotness.

6.2 Modelling of Extraction Time

Using hotness and platform information, UGache's Solver builds a model to estimate the time spent on embedding extraction with the arrangement variation of *embedding storage* and *embedding access*. The storage arrangement describes where to put an embedding entry, while the access arrangement describes which source location to read from when the same embedding entry is stored by multiple reachable GPUs. Solver models and minimizes the extraction time through access arrangement, which limits storage arrangement and is further constrained by each GPU's cache capacity.

The Solver first defines a set of binary variables $a^e_{i \leftarrow j}$, indicating whether destination GPU_i accesses embedding

entry e from source location j when $a^e_{i\leftarrow j} = 1$:

$$\sum_{j\in\mathbb{M}} a^e_{i\leftarrow j} = 1 \qquad \forall i \in \mathbb{G}, \forall e \in \mathbb{E}$$

To ensure accessibility, the sum of $a^e_{i\leftarrow j}$ must be 1, as shown in the equation above.

Based on $a^e_{i\leftarrow j}$, UGache is now able to ❶ construct the storage arrangement for each source GPU and constrain it by cache capacity, and ❷ estimate the total extraction time for each GPU as the target to be minimized.

Storage Arrangement. The connection between storage and access arrangement is straight forward: if any GPU accesses an embedding entry e from source location j, then j must store e. To model this behavior, UGache defines another set of binary variables s_{je} indicating whether source location j stores embedding entry e:

$$s_{je} \geq a^e_{i\leftarrow j} \qquad \forall i \in \mathbb{G}, \forall j \in \mathbb{M}, \forall e \in \mathbb{E}$$

$$\sum_{e\in\mathbb{E}} s_{je} \leq Cap_j \qquad \forall j \in \mathbb{M}$$

The first equation describes the aforementioned constraint between s_{je} and $a^e_{i\leftarrow j}$, while the second equation limits the total stored embedding entries on j within its capacity.

Time Estimation. Based on $a^e_{i\leftarrow j}$, hotness metric, and profiled platform's information, UGache can construct the total extraction time by summing $a^e_{i\leftarrow j}$ with hotness and access time cost as weight:

$$t^j_i = \sum_{e\in\mathbb{E}} T_{i\leftarrow j} * h_e * a^e_{i\leftarrow j} \qquad \forall i \in \mathbb{G}, \forall j \in \mathbb{M}$$

In the above equation, h_e is the hotness of embedding e, $T_{i\leftarrow j}$ is the time of reading one embedding from source j to GPU i. t^i_i,t^{CPU}_i,t^j_i describes the total time of GPU i reading from local cache, fallback location, and remote GPU j respectively. The platform's bandwidth and topology are introduced by $T_{i\leftarrow j}$, whose value is the reciprocal of bandwidth. For unconnected GPU pairs, its value can be set as infinity. In practice, UGache removes the corresponding t^j_i of unconnected GPU pairs to simplify the model.

Based on t^j_i, the total extraction time t_i of GPU i can be constructed according to the design of UGache's Extractor:

$$t_i \geq t^j_i \qquad \forall i \in \mathbb{G}, \forall j \in \mathbb{M}$$

$$t_i \geq \sum_{j\in\mathbb{M}} t^j_i * R_{i\leftarrow j} \qquad \forall i \in \mathbb{G}$$

The first equation considers that the absolute time extracting from each non-local location may be extremely imbalanced, e.g., host extraction dominates when cache capacity is extremely insufficient. The second equation considers the usual case that Extractor's local extraction is able to handle load imbalance of each source, as Figure 8 shows. In the second equation, $R_{i\leftarrow j}$ is the ratio of cores dedicated to extract from source j. Summing t^j_i with $R_{i\leftarrow j}$ as weight calculates the area of Figure 8 to estimate total extraction time.

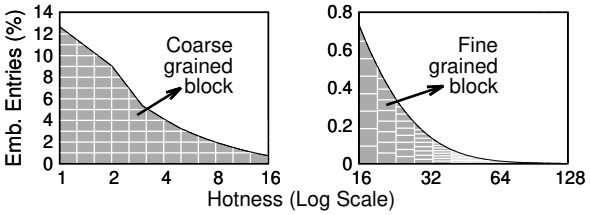

Figure 9. *Batch embedding entries with similar log-scale hotness into blocks to reduce model complexity.*

Finally, the target of the MILP can be constructed by the maximum of each GPU's extraction time:

$$\texttt{minimize } z, \qquad z \geq t_i, \forall i \in \mathbb{G}$$

The constructed model can be solved by mature optimization tools like Gurobi [9].

6.3 Complexity, Approximation and Optimizations

The MILP problem above provides an optimal solution that adapts various platforms and workloads. However, on large datasets, the problem is impossible to solve in an acceptable time. With E representing number of embedding entries and G representing number of GPUs, the MILP problem in UGache contains $EG^2 + 2EG + G^2 + 2G + 1$ variables and $EG^2 + EG + 2G^2 + 5G$ constraints. The total number of non-zero elements in the MILP problem is also at $O(EG^2)$. As the MILP problem is NP-complete, solving this problem has an exponential cost with respect to E and G. For a large E (e.g., millions of entries), the cost becomes unacceptable. A recent work [39] proposes a heuristic approach to finding an optimal policy in an acceptable time by replicating hot and partitioning warm embedding entries across every GPU. However, this approach is limited to a uniform fully-connected platform and cannot be generalized to non-uniform platforms.

To simplify problem-solving complexity while maintaining generality, UGache proposes an approximate solution that provides a near-optimal result in an acceptable time (e.g., 10 seconds). The idea is to batch similar entries, decide their placements and access policies together. Since entries with similar hotness tend to have similar placement decisions, UGache batches entries with similar hotness into blocks, and builds the MILP problem at the granularity of a block.[2]

To control the loss of precision, UGache deploys two optimizations. The first optimization classifies similar hotness levels on a log scale (see the X-axis of Figure 9). This is because a hotness difference at a larger base (e.g., 110 and 120) is less distinguishable than the same difference at a smaller base (e.g., 10 and 20). The second optimization controls the size of a single block by dividing it into smaller ones (see the Y-axis in Figure 9). The block size is controlled through a combination of coarse and fine granularity. Due to the large number of infrequent entries at a low hotness level, the

[2]The size each block needs to be added to the MILP problem.

maximum size of a single block is limited to a fixed portion to prevent excessively large blocks. On the contrary, at high hotness levels, the maximum size of a block is limited based on the number of entries in the same hotness level, allowing for a sophisticated policy when available cache ratio is low. In practice, UGache limits the maximum size of a single block at a low hotness level to 0.5% of total entries and adjusts block size at high hotness levels to ensure that each level is split into at least N (the number of GPUs) fine-grained blocks. We find that this strategy adapts well to most scenarios.

By reducing the granularity of placement from embedding entry to block, UGache decreases E from several billions to less than one thousand. As G is a small constant, the scale of the MILP problem is significantly reduced. UGache takes around 10 seconds to solve the MILP problem, which is acceptable compared to the data loading time [46]. Furthermore, the difference between our near-optimal results and the theoretical optimal results is less than 2% on average.

7 Implementation

7.1 System Integration

To integrate UGache into EmbDL's workflow, we follow prior works to expose it as an embedding layer for mainstream DL frameworks with compatible APIs. Applications can take advantage of UGache's high-performance embedding extraction by replacing the reference of embedding layer with UGache. The core logic of UGache was implemented with roughly 12K lines of C++ and CUDA code. Above the core logic, UGache is wrapped into an embedding layer in PyTorch [37] and TensorFlow [14].

7.2 Cache Refresh

As a cache system, UGache needs to handle potential variations in workload's hotness. Traditional systems often use evict policies such as LRU to track hot entries and perform evictions on the fly. While this design provides great feasibility for dynamic workloads, it incurs maintenance overhead and increases the complexity of cache system, especially for multi-GPU cache. Fortunately, the stable and predictable pattern of embedding workloads (§2) ensures that hotness remains stable for a long period (e.g., days). This allows UGache to build a static cache with infrequent refreshes in a background, periodical manner, reducing design complexity significantly.

In the foreground, UGache samples input requests and records hotness by CPU, hiding its performance impact. In the background, UGache's Refresher collects statistics and periodically re-evaluates Solver's model with the new hotness. When the estimated extraction time increases significantly, the refresh procedure is triggered. Refresher first invokes Solver to solve a new cache policy. According to the new cache policy, Refresher evicts old entries and inserts new entries into UGache's hashtable and cache content on GPU. To ensure that foreground embedding extraction always reads correct content, Refresher waits for a foreground batch between the update of hashtable and cache content.

To balance the trade-off between the impact on foreground embedding extraction and the duration of the refresh procedure, Refresher controls the resource usage and refresh granularity. Typically, the entire refresh procedure takes around 20 seconds, with less than 10% impact on foreground requests during the procedure.

7.3 Hardware Requirements

Core dedication. UGache's Extractor requires explicit control over GPU's cores to execute different code to implement GPU core dedication. While there isn't yet a method to implement such control, CUDA provides Multi-Process Service (MPS) as a workaround. UGache leverages MPS's API to create multiple GPU contexts with limitation of GPU core occupation, and launches kernels on these contexts to achieve GPU core dedication.

Kernel priority. To handle potential load imbalance, local extraction must be launched early but scheduled after the extraction kernel of non-local locations. UGache leverages stream priority in CUDA, letting local extraction be launched on a stream with lower priority to achieve this goal.

8 Evaluation

We evaluated UGache on two types of EmbDL applications: GNN training and DLR inference. For GNN training, we integrated UGache into PyTorch [37] and followed WholeGraph [45] to reuse its graph sampler. All systems leveraged DGL [40] as the backend engine for model training. For DLR inference, we integrated UGache into TensorFlow [14].

8.1 Experimental Setup

Testbeds. To demonstrate the generalization of UGache, we conducted experiments on three multi-GPU servers with different configurations:

- Server A: four V100 (16 GB, SXM2) GPUs, two Intel Xeon Gold 6138 CPUs (total 40 cores), and 384 GB of host memory.
- Server B: eight V100 (32 GB, SXM2) GPUs, two Intel Xeon Platinum 8163 CPUs (total 48 cores), and 724 GB of host memory.
- Server C: eight A100 (80 GB, SXM4) GPUs, two Intel Xeon Gold 6348 CPUs (total 56 cores), and 1 TB of host memory.

Figure 3 shows the GPU interconnect topology of these servers, where Server A and Server B are hard-wired platforms, and Server C is a switch-based platform. Each NVLink has a bandwidth of 25 GB/s, resulting in a total outbound bandwidth of 150 GB/s and 300 GB/s for V100 and A100, respectively. All experiments were conducted in Docker using NVIDIA's image with Python v3.8, PyTorch v1.13.0, CUDA v11.7, DGL v0.9.1, and TensorFlow v2.9.1.

Table 3. *GNN and DLR Datasets used in evaluation. $Volume_G$ (resp. $Volume_E$) is the data volume of graph topological (resp. embedding) data in host memory. Note that MAG ships with float16 embedding, while other datasets use float32.*

Dataset	#Vertex	#Edge	Dim.	$Volume_G$	$Volume_E$
PA [4]	111 M	3.2 B	128	12.8 GB	53 GB
CF [27]	65.6 M	3.6 B	256	14 GB	62 GB
MAG [3]	232 M	3.2 B	768	13.8 GB	349 GB
Dataset	**#Entry**	**#Table**	**Dim.**	**Skewness**	**$Volume_E$**
CR [6]	882 M	26	128	N/A	420.9 GB
SYN-A	800 M	100	128	1.2	381.5 GB
SYN-B	800 M	100	128	1.4	381.5 GB

Applications. For GNN, we trained two popular models, GCN [26] and GraphSAGE [24]. Both models were evaluated in supervised settings. Furthermore, we evaluated GraphSAGE in an unsupervised setting for link prediction applications. The training procedure of GCN (resp. GraphSAGE) adopted standard 3-hop (resp. 2-hop) random neighborhood sampling [49], following GNNLab [46]. We refer the readers to the original papers [24, 26] for more details.

For DLR, we ran inference workloads using two popular models, DLRM [36] and DCN [41]. DLRM consisted of six MLP layers and one embedding layer associated with the dataset, following the settings in [43]. DCN included an additional Cross layer, following the settings in the official TensorFlow example [14].

By default, we set the per-GPU batch size to 8K, similar to prior work [46]. However, training large datasets or unsupervised models in GNN can lead to out-of-memory (OOM) errors, even without caching, due to the high memory consumption of intermediate results. In such cases, we turned down the batch size on GPUs with smaller memory capacity.

Datasets. For GNN training, we used 3 datasets, namely a social graph, Com-Friendster [27] (CF), and two GNN datasets from Open Graph Benchmark (OGB) [2]: a citation network, OGB-Papers100M (PA), and an academic network, OGB-MAG240M (MAG). Since CF originally lacks embeddings and labels, and no training set was provided officially, we generated random embeddings and labels, and randomly selected a small portion of vertices as the training set, similar to prior works [21, 30]. We followed the standard preprocessing procedure in the OGB leaderboards [13] to convert PA and MAG into unidirectional homogeneous graphs.

For DLR inference, we first evaluated Criteo-TB [6] (CR), a commonly used dataset that contains 26 embedding tables. Each request contains a single key for each table. We further evaluated two synthetic datasets, SYN-A and SYN-B, to keep up with the growing size of datasets. Both SYN-A and SYN-B include 100 embedding tables, and their request keys were generated from a power-law distribution with α=1.2 and 1.4, respectively. Although the total size of cacheable entries in SYN-A and SYN-B is similar to CR, their requests are larger.

Among these datasets and models, the main factors affecting the workload of embedding extraction are the dataset and the sampling method used in GNN. while the model type (e.g., GCN and GraphSAGE) mainly affects the performance of the dense layer (MLP). Since the embedding layer is our main optimization target, we primarily vary the dataset and GNN sampling method to obtain a diverse range of embedding extraction workloads in order to demonstrate the performance of UGache.

Baselines. In GNN training, we compare UGache with two state-of-the-art systems: GNNLab [46] and WholeGraph [45]. GNNLab dedicates specialized GPUs for graph sampling, which saves GPU memory from graph storage to enable a larger replication cache. WholeGraph builds a partition cache and conducts straightforward peer-based embedding extraction. However, WholeGraph fails to launch when either ① the total GPU memory cannot hold all embeddings or ② unconnected GPU pairs exist. Therefore, we further implemented a new baseline called $Part_U$ that extends WholeGraph. $Part_U$ partitions only hot embeddings to GPU and stores cold embeddings on the CPU. To handle unconnected GPU pairs on the 8×V100 platform, $Part_U$ uses the clique approach in Quiver [7], which splits the 8 GPUs into two fully-connected cliques. Each clique has an individual partition cache, avoiding cross-clique access. We also offer Rep_U that uses an identical codebase as $Part_U$ but employs a replication cache.

In DLR inference, we compare UGache with HPS [43] and SOK [8], both of which are TensorFlow embedding plugins from the same codebase as HugeCTR [42]. HPS builds a replication cache with LRU-based online eviction, while SOK builds a partition cache and conducts message-based embedding extraction.

Finally, all systems, including UGache, use the same backend: DGL+PyTorch for GNN training and TensorFlow for DLR inference. Each system is warmed up using either the first epoch for training or the first 1,000 iterations for inference. We conduct each experiment three times and report the average results.

8.2 Overall Performance

We first compare UGache to state-of-the-art systems in GNN and DLR across a wide range of application settings. Figure 10 reports the end-to-end time for each system. On average, UGache outperforms GNNLab and HPS (replication cache systems) by 2.21× (up to 5.25×) and 1.51× (up to 2.34×), respectively. In GNN, UGache has improved performance over GNNLab due to the unification of multiple GPUs to build a larger cache. In unsupervised GNN, negative sampling reduces the skewness of embedding access, resulting in a even larger improvement over GNNLab's limited cache capacity

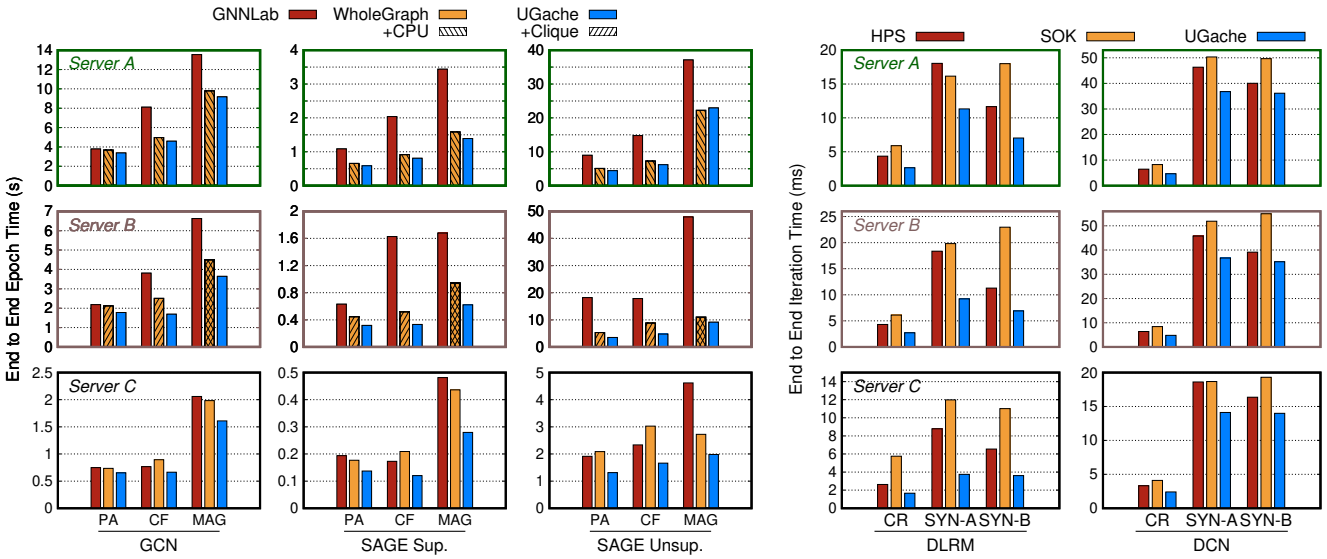

Figure 10. *The end-to-end time of (a) an epoch in GNN training and (b) an iteration in DLR inference on different systems and platforms.*

of single GPU. When compared to HPS in DLR, UGache further benefits from its static design with no online eviction cost. While the workload in DLR is much more skewed than in GNN, the improvement of UGache over the replication cache system is slightly lower than in GNN.

UGache also outperforms WholeGraph and SOK (partition cache systems) by 1.33× (up to 1.85×) and 2.07× (up to 3.45×) on average, respectively. Comparing to WholeGraph in GNN, UGache benefits from a more efficient cache policy and its factored extraction mechanism. When deployed on a 4×V100 platform or with the MAG dataset, host extraction still dominates due to limited cache capacity, explaining UGache's small improvement in these cases. The only case when UGache is slower is running unsupervised GraphSAGE with MAG on 4×V100 platform. In this case, the cache capacity is significantly inefficient, exposing UGache's cost of approximate cache policy solution. Comparing to SOK in DLR, UGache further benefits from peer-based embedding extraction and has a larger improvement. Note that in DLR, with higher skewness, SOK's performance is often worse than HPS's replication cache due to the low local hit rate of partition policy.

We break down the end-to-end time to focus on the improvement of UGache's techniques on embedding extraction in Figure 11. The improvement over GNN baselines comes from UGache's novel technique in cache policy and extraction mechanism, with UGache outperforming GNNLab and WholeGraph by 3.57× and 2.62× in embedding extraction, respectively. Note that despite GNNLab's extraction being faster than WholeGraph in some cases with higher skew, its end-to-end time is slower due to transferring samples across GPUs through queues in host memory.

In DLR, we introduce a comparison between Part_U and Rep_U to isolate UGache's performance improvement of novel techniques. Comparing to HPS and SOK, Rep_U and

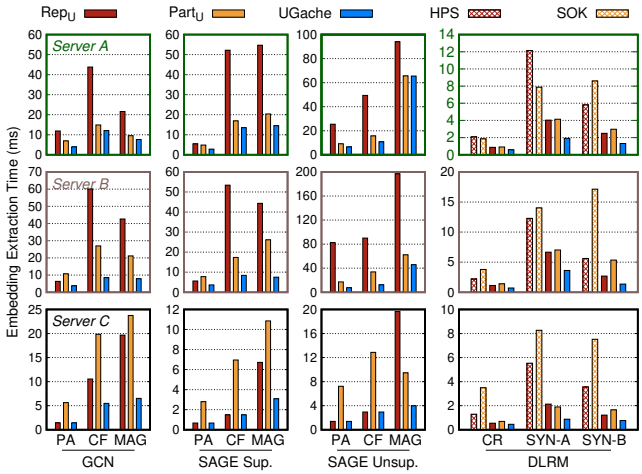

Figure 11. *The embedding extraction time in one iteration of GNN training and DLR inference on different systems and platforms.*

Part_U improve by 2.39× and 3.18×, respectively. The performance gap is brought by Rep_U's static cache design with no online eviction and Part_U's peer-based embedding extraction. UGache further improves embedding extraction from Rep_U and Part_U by 1.79× and 2.19×, respectively.

8.3 Performance Breakdown

We now break the performance improvement of UGache that comes from cache policy and extraction mechanism. Figure 12 demonstrates the time taken for embedding extraction when these two techniques are incrementally applied in supervised GraphSAGE with PA and CF on an 8×A100 platform. When the cache ratio is small (e.g., 2% in PA), UGache produces a cache policy similar to partition, and the 1.72× improvement over partition is mainly due to UGache' extraction mechanism.

As the cache ratio increases, UGache's cache policy diverges from partition by leveraging sufficient cache capacity

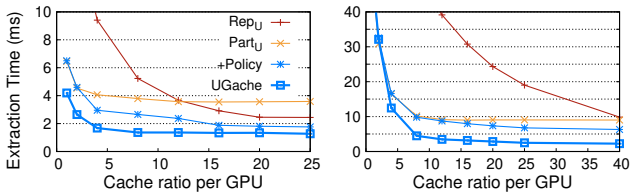

Figure 12. *The embedding extraction time by incrementally applying techniques in UGache on Server C.*

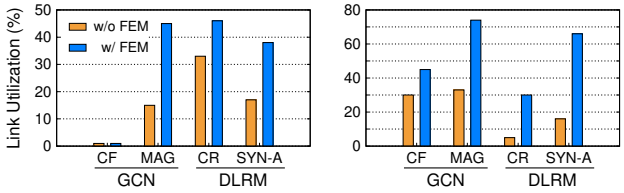

Figure 13. *The utilization of (a) PCIe and (b) NVLink during embedding extraction on Server C.*

to balance the local and global hit rate. The divergence point and the improvement brought by cache policy depend on the skewness of the datasets. Eventually, the cache policy always dominates UGache's performance improvement under high cache ratio.

8.4 Bandwidth Utilization

To study the improvement of UGache's factored extraction mechanism (FEM), we use NVIDIA Nsight Systems [12] to measure the bandwidth utilization of PCIe and NVLink. Measuring the utilization during embedding extraction requires sub-millisecond sampling, and it is currently only available on the A100 GPU. To ensure a fair comparison, we removed locally hit keys in each batch in advance, leaving only access to remote GPU and host memory. As shown in Figure 13, UGache's congestion-avoid factored extraction mechanism (w/ FEM) significantly improves the utilization of PCIe and NVLink by 1.91 × and 3.47× on average, respectively. However, for GCN with CF, the amount of non-local access is small due to the small dataset volume and high cache ratio, making performance improvement relatively trivial.

8.5 Cache Policy

UGache improves cache performance by balancing the local and global hit rate. We measured the portion of data accessed from each cache location and compared the results of Rep$_U$, Part$_U$, and UGache. The top of Figure 14 shows the results for supervised GraphSAGE training with PA, a highly skewed dataset, on the 8×A100 platform. At low cache ratio (2%), only a small portion of entries can be cached. Therefore, UGache produces an almost identical cache policy to partition cache, resulting in similar local and global hit rates to Part$_U$. In contrast, Rep$_U$ suffers from a low global hit rate (i.e., 63.5%).

As the cache ratio increases (8%), UGache can cache more copies of the hottest entries, significantly improving the local hit rate from 12.4% (partition) to 86.7%, while only slightly sacrificing the global hit rate from 99.1% to 98.1%. Figure 15

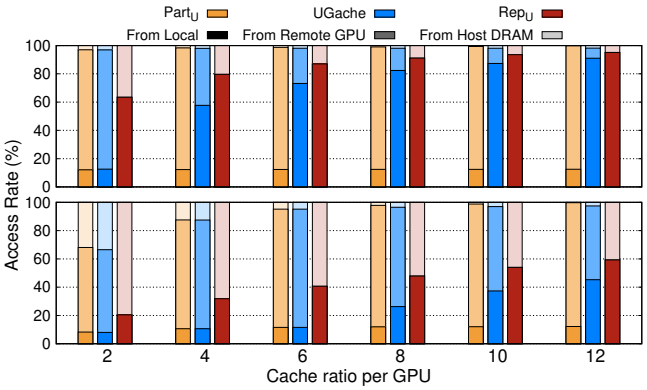

Figure 14. *The hit rate of local GPU cache, remote GPU cache, and host memory (from bottom to top) in one step on Server C, measured in GraphSAGE training with PA (high skew) and CF (low skew).*

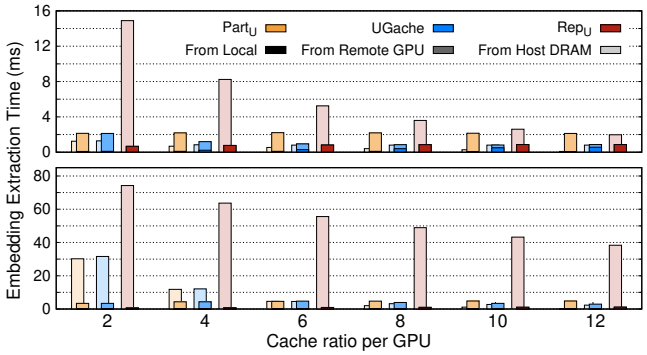

Figure 15. *The runtime for extracting entries in one step, following the same setup as Figure 14. All baselines enable the extraction technique in UGache.*

shows that this trade-off leads to better performance for UGache compared to Rep$_U$ and Part$_U$. The increased local extraction time in UGache reduces remote extraction time[3], resulting in a 2.0× improvement over partition cache.

The bottom of Figures 14 and 15 show the results of using CF. As CF is not highly skewed, the global hit rate is lower when the cache ratio is small. Therefore, sacrificing the global hit rate to improve local hit rate is unwise. UGache's cache policy values the global hit rate more, resulting in cache hit rate and performance similar to partition cache.

UGache's Solver introduces approximation when solving a cache policy. Therefore, we quantify the effect of UGache by comparing to theoretically optimal cache policy. For each platform, the optimal cache is constructed as described in §6.3. However, for Server B, the MILP problem at the granularity of embedding entry cannot be solved in a feasible time due to the large volume of datasets. Thus, we construct a special case to reduce the scale of MILP problem and obtain an optimal policy on Server B. In DLR with synthetic datasets, each embedding table shares the same size and skewness. We limit the scope of MILP to the first embedding table, and

[3]To measure the extraction time in Part$_U$ and Rep$_U$ for each source, both systems adopt UGache's extract mechanism. The local extraction time can only be estimated due to padding in UGache's extraction mechanism.

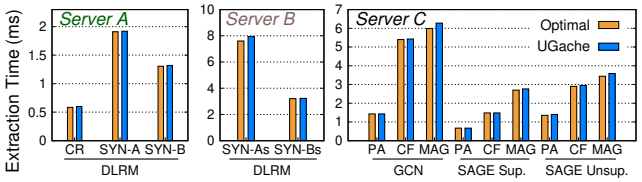

Figure 16. *Comparison of extraction time between UGache and the theoretically optimal cache policy.*

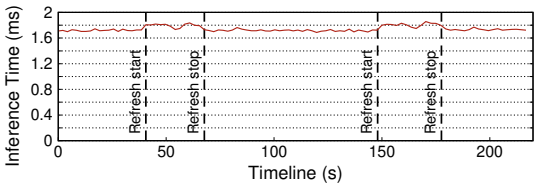

Figure 17. *The inference timeline of DLRM with cache refresh triggered in UGache.*

apply the same solved policy to the rest of tables. We further reduced the size of a single embedding table to 10,000 to achieve a yet feasible time for solving. The reduced dataset contains 1 million entries in total, and is named as SYN-As and SYN-Bs. Figure 16 reports the embedding extraction time of UGache and optimal cache. Both results adopt UGache's extraction mechanism and only differ in cache policy. The gap between UGache and optimal cache policy is 1.9% on average, proving the efficiency of UGache approximation.

8.6 Refreshment

We further evaluated UGache's refresh procedure. Since there is little variation in skewness across the evaluated workloads, refresh hardly improves extraction time. So we focused on the speed of the refresh procedure and its impact on foreground application. Figure 17 shows the time consumed by foreground DLRM inference with CR over time on an 8×A100 platform, where the refresh procedure is manually triggered at around 40 and 150 seconds.

Since the cache policy solving and background cache eviction compete for CPU and GPU resources, the refresh procedure inevitably impacts foreground application. To limit the impact, UGache restricts the number of CPU cores used and splits the cache updates into multiple small-batch updates. By controlling the intervals between cache updates, UGache manages to strike a balance between faster refresh and a smaller impact on application. In Figure 17, the entire refresh procedure takes 28.69 seconds on average and only impacts foreground application by 10%.

9 Related Work

There has been increasing attention paid to eliminating bottlenecks caused by embedding layers in EmbDL applications.

Single-GPU EmbDL systems. HPS [43] builds a single GPU embedding cache using LRU with multi-level hierarchical storage architecture and specialized support for online model updating for industrial deployment. Fleche [44] flattens multiple embedding tables to build a single cache with global perspective and leverages kernel fusion to reduce overhead of kernel maintenance. EVStore [28] further improves cache capacity and inference throughput by exploiting approximation from mixed-precision and embedding similarity. However, when deployed to multi-GPU platforms, these systems lead to a replication cache and ignore the fast connections between GPUs.

Multi-GPU EmbDL systems. Recently, EmbDL systems have proposed specialized designs for multi-GPU platforms. For GNN, PaGraph [29] estimates the access frequency of embeddings based on degree information of the local-deducted graph and caches embeddings associated with high-degree nodes. GNNLab [46] dedicates specialized GPUs to conduct graph sampling, reducing redundant copies of graph data to increase cache capacity. It also proposes pre-sampling to more accurately estimate access frequency for cache embeddings. To efficiently leverage cross-GPU interconnects, WholeGraph [45] partitions graph and embeddings onto multiple GPUs and adopts peer-based access to implement cross-GPU graph sampling and zero-copy embedding extraction. Quiver [7] moves graph data to host memory to increase the cache capacity for embeddings and proposes clique-based partition cache to handle unconnected GPUs. P3 [21] vertically partitions embeddings and postpones transferring embeddings to reduce communication costs. In [39], hot data is replicated and warm data is partitioned across every GPU for GNN training based on a heuristic search.

For DLR, prior work on multi-GPU embedding cache mainly focuses on model training [8, 23, 34, 42, 47, 48]. These systems partition embeddings across multiple GPUs and exchange them through message-based communication libraries (e.g., NCCL [11]). RecShard [38] builds a table-level partition cache for DLR training on multiple GPUs and also uses MILP to reduce communication costs.

10 Conclusion

This paper present UGache, a unified multi-GPU cache designed for EmbDL. UGache fully exploits high-speed interconnects to accelerate remote GPU memory accesses, and offers an near-optimal data placement with a unified abstraction for various GPU interconnects and bandwidths. Our evaluation confirms the advantage and efficacy of UGache.

Acknowledgments

We sincerely thank our shepherd Adam Belay and the anonymous reviewers. This work was supported in part by the National Key Research and Development Program of China (No. 2022YFB4500700), the National Natural Science Foundation of China (No. 62272291, 61925206, 62132014), the HighTech Support Program from STCSM (No. 22511106200), and a research grant from Shanghai Artificial Intelligence Laboratory. Corresponding author: Rong Chen (rongchen@sjtu.edu.cn).

References

[1] 2011. Peer-to-Peer and Unified Virtual Addressing. https://developer.download.nvidia.com/CUDA/training/cuda_webinars_GPUDirect_uva.pdf.

[2] 2021. Open Graph Benchmark: Benchmark datasets, data loaders and evaluators for graph machine learning. https://ogb.stanford.edu/.

[3] 2021. Open Graph Benchmark: The MAG240M dataset. https://ogb.stanford.edu/docs/lsc/mag240m/.

[4] 2021. Open Graph Benchmark: The ogbn-papers100M dataset. https://ogb.stanford.edu/docs/nodeprop/#ogbn-papers100M.

[5] 2022. Distributed Embeddings · NVIDIA-Merlin. https://github.com/NVIDIA-Merlin/distributed-embeddings

[6] 2022. Download Criteo 1TB Click Logs dataset - Criteo AI Lab. https://ailab.criteo.com/download-criteo-1tb-click-logs-dataset/.

[7] 2022. quiver-team/torch-quiver. https://github.com/quiver-team/torch-quiver

[8] 2022. Sparse Operation Kit · NVIDIA-Merlin. https://github.com/NVIDIA-Merlin/HugeCTR

[9] 2023. Gurobi Optimizer. https://www.gurobi.com/.

[10] 2023. Multi-Process Service. http://docs.nvidia.com/deploy/mps/index.html

[11] 2023. Nvidia Collective Communication Library (NCCL). https://developer.nvidia.com/nccl.

[12] 2023. NVIDIA Nsight Systems. https://developer.nvidia.com/nsight-systems.

[13] 2023. Open Graph Benchmark Leaderboards. https://ogb.stanford.edu/docs/leader_overview/.

[14] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. 265–283. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[15] Lada A Adamic and Bernardo A Huberman. 2000. Power-law distribution of the world wide web. *science* 287, 5461 (2000), 2115–2115.

[16] Muhammad Adnan, Yassaman Ebrahimzadeh Maboud, Divya Mahajan, and Prashant J. Nair. 2021. Accelerating recommendation system training by leveraging popular choices. *Proceedings of the VLDB Endowment* 15, 1 (Sept. 2021), 127–140.

[17] Saurabh Agarwal, Ziyi Zhang, and Shivaram Venkataraman. 2022. BagPipe: Accelerating Deep Recommendation Model Training. arXiv:2202.12429 [cs].

[18] Keshav Balasubramanian, Abdulla Alshabanah, Joshua D Choe, and Murali Annavaram. 2021. cDLRM: Look Ahead Caching for Scalable Training of Recommendation Models. In *Proceedings of the 15th ACM Conference on Recommender Systems (RecSys '21)*. Association for Computing Machinery, New York, NY, USA, 263–272.

[19] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. 2015. PowerLyra: differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems*. 1–15.

[20] Message P Forum. 1994. *MPI: A Message-Passing Interface Standard*. Technical Report. University of Tennessee, USA.

[21] Swapnil Gandhi and Anand Padmanabha Iyer. 2021. P3: Distributed Deep Graph Learning at Scale. In *Proceedings of the 15th USENIX Conference on Operating Systems Design and Implementation (OSDI'21)*.

[22] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. Powergraph: Distributed graph-parallel computation on natural graphs. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)*. 17–30.

[23] Huifeng Guo, Wei Guo, Yong Gao, Ruiming Tang, Xiuqiang He, and Wenzhi Liu. 2021. ScaleFreeCTR: MixCache-based Distributed Training System for CTR Models with Huge Embedding Table. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '21)*. Association for Computing Machinery, New York, NY, USA, 1269–1278.

[24] William L. Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems (NeurIPS'17)*. 1025–1035.

[25] Mohamed Assem Ibrahim, Onur Kayiran, and Shaizeen Aga. 2022. Efficient Cache Utilization via Model-aware Data Placement for Recommendation Models. In *The International Symposium on Memory Systems (MEMSYS 2021)*. Association for Computing Machinery, New York, NY, USA, 1–11.

[26] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *Proceedings of the 5th International Conference on Learning Representations (ICLR'17)*.

[27] Jérôme Kunegis. 2013. KONECT: the Koblenz network collection. In *Proceedings of the 22nd International Conference on World Wide Web (WWW '13 Companion)*. Association for Computing Machinery, New York, NY, USA, 1343–1350.

[28] Daniar H. Kurniawan, Ruipu Wang, Kahfi S. Zulkifli, Fandi A. Wiranata, John Bent, Ymir Vigfusson, and Haryadi S. Gunawi. 2023. EVStore: Storage and Caching Capabilities for Scaling Embedding Tables in Deep Recommendation Systems. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 281–294.

[29] Zhiqi Lin, Cheng Li, Youshan Miao, Yunxin Liu, and Yinlong Xu. 2020. PaGraph: Scaling GNN training on large graphs via computation-aware caching. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. ACM, Virtual Event USA, 401–415.

[30] Zhiqi Lin, Cheng Li, Youshan Miao, Yunxin Liu, and Yinlong Xu. 2020. Pagraph: Scaling GNN Training on Large Graphs via Computation-aware Caching. In *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC'20)*. 401–415.

[31] Xupeng Miao, Yining Shi, Hailin Zhang, Xin Zhang, Xiaonan Nie, Zhi Yang, and Bin Cui. 2022. HET-GMP: A Graph-based System Approach to Scaling Large Embedding Model Training. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 470–480.

[32] Xupeng Miao, Hailin Zhang, Yining Shi, Xiaonan Nie, Zhi Yang, Yangyu Tao, and Bin Cui. 2021. HET: scaling out huge embedding model training via cache-enabled distributed framework. *Proceedings of the VLDB Endowment* 15, 2 (Oct. 2021), 312–320.

[33] Seung Won Min, Kun Wu, Sitao Huang, Mert Hidayetoğlu, Jinjun Xiong, Eiman Ebrahimi, Deming Chen, and Wen-mei Hwu. 2021. Large graph convolutional network training with GPU-oriented data communication architecture. *Proceedings of the VLDB Endowment* 14, 11 (Oct. 2021), 2087–2100.

[34] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Zhihao Jia, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, Liang Luo, Jie (Amy) Yang, Leon Gao, Dmytro Ivchenko, Aarti Basant, Yuxi Hu, Jiyan Yang, Ehsan K. Ardestani, Xiaodong Wang, Rakesh Komuravelli, Ching-Hsiang Chu, Serhat Yilmaz, Huayu Li, Jiyuan Qian, Zhuobo Feng, Yinbin Ma, Junjie Yang, Ellie Wen, Hong Li, Lin Yang, Chonglin Sun, Whitney Zhao, Dimitry Melts, Krishna Dhulipala, KR Kishore, Tyler Graf, Assaf Eisenman, Kiran Kumar Matam, Adi Gangidi, Guoqiang Jerry Chen, Manoj Krishnan, Avinash Nayak, Krishnakumar Nair, Bharath Muthiah, Mahmoud khorashadi, Pallab Bhattacharya, Petr Lapukhov, Maxim Naumov, Ajit Mathews, Lin Qiao, Mikhail Smelyanskiy, Bill Jia, and Vijay Rao. 2022. Software-hardware co-design for fast and scalable training of deep learning recommendation models. In *Proceedings of the 49th Annual International Symposium*

on Computer Architecture (ISCA '22). Association for Computing Machinery, New York, NY, USA, 993–1011.

[35] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. 2019. Deep Learning Recommendation Model for Personalization and Recommendation Systems.

[36] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. 2019. Deep Learning Recommendation Model for Personalization and Recommendation Systems. arXiv:1906.00091 [cs].

[37] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[38] Geet Sethi, Bilge Acun, Niket Agarwal, Christos Kozyrakis, Caroline Trippel, and Carole-Jean Wu. 2022. RecShard: statistical feature-based memory optimization for industry-scale neural recommendation. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 344–358.

[39] Shihui Song and Peng Jiang. 2022. Rethinking graph data placement for graph neural network training on multiple GPUs. In *Proceedings of the 36th ACM International Conference on Supercomputing (ICS '22)*. Association for Computing Machinery, New York, NY, USA, 1–10.

[40] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. 2019. Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks. *arXiv preprint arXiv:1909.01315* (2019).

[41] Ruoxi Wang, Bin Fu, Gang Fu, and Mingliang Wang. 2017. Deep & Cross Network for Ad Click Predictions. In *Proceedings of the ADKDD'17 (ADKDD'17)*. Association for Computing Machinery, New York, NY, USA, 1–7.

[42] Zehuan Wang, Yingcan Wei, Minseok Lee, Matthias Langer, Fan Yu, Jie Liu, Shijie Liu, Daniel G. Abel, Xu Guo, Jianbing Dong, Ji Shi, and Kunlun Li. 2022. Merlin HugeCTR: GPU-accelerated Recommender System Training and Inference. In *Proceedings of the 16th ACM Conference on Recommender Systems (RecSys '22)*. Association for Computing Machinery, New York, NY, USA, 534–537.

[43] Yingcan Wei, Matthias Langer, Fan Yu, Minseok Lee, Jie Liu, Ji Shi, and Zehuan Wang. 2022. A GPU-specialized Inference Parameter Server for Large-Scale Deep Recommendation Models. In *Proceedings of the 16th ACM Conference on Recommender Systems (RecSys '22)*. Association for Computing Machinery, New York, NY, USA, 408–419.

[44] Minhui Xie, Youyou Lu, Jiazhen Lin, Qing Wang, Jian Gao, Kai Ren, and Jiwu Shu. 2022. Fleche: an efficient GPU embedding cache for personalized recommendations. In *Proceedings of the Seventeenth European Conference on Computer Systems*. ACM, Rennes France, 402–416.

[45] Dongxu Yang, Junhong Liu, Jiaxing Qi, and Junjie Lai. 2022. WholeGraph: A Fast Graph Neural Network Training Framework with Multi-GPU Distributed Shared Memory Architecture. IEEE Computer Society, 767–780. ISSN: 2167-4337.

[46] Jianbang Yang, Dahai Tang, Xiaoniu Song, Lei Wang, Qiang Yin, Rong Chen, Wenyuan Yu, and Jingren Zhou. 2022. GNNLab: a factored system for sample-based GNN training over GPUs. In *Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 417–434.

[47] Daochen Zha, Louis Feng, Bhargav Bhushanam, Dhruv Choudhary, Jade Nie, Yuandong Tian, Jay Chae, Yinbin Ma, Arun Kejariwal, and Xia Hu. 2022. AutoShard: Automated Embedding Table Sharding for Recommender Systems. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '22)*. Association for Computing Machinery, New York, NY, USA, 4461–4471.

[48] Yuanxing Zhang, Langshi Chen, Siran Yang, Man Yuan, Huimin Yi, Jie Zhang, Jiamang Wang, Jianbo Dong, Yunlong Xu, Yue Song, Yong Li, Di Zhang, Wei Lin, Lin Qu, and Bo Zheng. 2022. PICASSO: Unleashing the Potential of GPU-centric Training for Wide-and-deep Recommender Systems. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. 3453–3466. ISSN: 2375-026X.

[49] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. 2020. DistDGL: Distributed Graph Neural Network Training for Billion-Scale Graphs. In *Proceedings of the 10th IEEE/ACM Workshop on Irregular Applications: Architectures and Algorithms (IA3'20)*. 36–44.