# FRUGAL: Efficient and Economic Embedding Model Training with Commodity GPUs

Minhui Xie
Tsinghua University
Beijing, China
Renmin University of China
Beijing, China
xieminhui@ruc.edu.cn

Shaoxun Zeng
Tsinghua University
Beijing, China
zsx21@mails.tsinghua.edu.cn

Hao Guo
Tsinghua University
Beijing, China
gh23@mails.tsinghua.edu.cn

Shiwei Gao
Tsinghua University
Beijing, China
gsw23@mails.tsinghua.edu.cn

Youyou Lu*
Tsinghua University
Beijing, China
luyouyou@tsinghua.edu.cn

## Abstract

Embedding models show superiority in learning representations of massive ID-type features in sparse learning scenarios such as recommendation systems (e.g., user/item IDs) and graph learning (e.g., node/edge IDs). Commodity GPUs are highly favored for their cost-efficient computing power, which is ideally suited for the low computing demand of memory-intensive embedding models. However, directly running embedding model training on commodity GPUs yields poor performance because of their deficient communication resources (including low communication bandwidth and no PCIe P2P support).

This paper presents FRUGAL, an embedding model training system tailored for commodity GPUs. Based on the observation that the communication between commodity GPUs must be bounced on host memory (due to no PCIe P2P support), the key idea of FRUGAL is *proactively flushing*, where each GPU proactively flushes its own parameters that other GPUs will access into host memory, thereby decoupling half of the communication overhead to non-critical paths. To alleviate the communication contention of proactively flushing on foreground training processes, FRUGAL assigns *priorities* to each flush operation, and prioritizes flushing parameters that GPUs will access while deferring others. Further, FRUGAL tailors a two-level priority queue to ensure high scalability for operations involving priorities. FRUGAL has been applied to train embedding models including recommendation models and graph embedding. Experiments indicate that FRUGAL can significantly increase training throughput on commodity GPUs, and achieve similar throughput compared to existing systems on datacenter GPUs with 4.0-4.3× improvement in cost-effectiveness.

***CCS Concepts:*** • **Computer systems organization** → **Heterogeneous (hybrid) systems**; *Neural networks*; • **Computing methodologies** → *Machine learning*; • **Networks** → *Bus networks.*

***Keywords:*** Deep Learning Model Training; Machine Learning System; Embedding Models; Heterogeneous Computing

**ACM Reference Format:**
Minhui Xie, Shaoxun Zeng, Hao Guo, Shiwei Gao, and Youyou Lu. 2025. FRUGAL: Efficient and Economic Embedding Model Training with Commodity GPUs. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3669940.3707245

## 1 Introduction

Embedding models have become pivotal in the field of machine learning for their ability to effectively learn representations of massive ID-type features in sparse learning scenarios, such as advertising [44], recommendation systems [16, 19, 20, 48] and graph learning [14, 21, 27, 41–43, 50]. These models handle ID-type features, such as user/item IDs or graph node/edge IDs, by mapping them to *embeddings* via huge embedding tables ($O(100)$ GB-scale, on host memory). Different from traditional deep learning models, embedding models show extreme *memory intensity* for massive random lookups on embedding tables, and existing training systems [8, 9, 12, 38, 52] maintain multi-GPU embedding cache by caching hot entries to reduce host memory fetching.

The training of embedding models has always relied heavily on datacenter GPUs (e.g., A100/A30), known for their extensive computational power and high-speed communication capabilities (NVLink). However, datacenter GPUs

*Youyou Lu is the corresponding author.

This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.
*ASPLOS '25, March 30-April 3, 2025, Rotterdam, Netherlands*
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0698-1/25/03
https://doi.org/10.1145/3669940.3707245

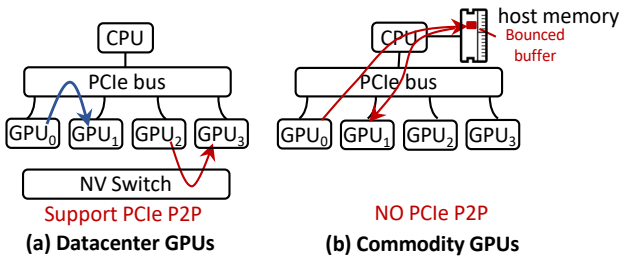


**Figure 1.** Comparison of communication mode between datacenter and commodity GPUs.

are often prohibitively expensive, making them less accessible to medium-sized enterprises, research labs, and individual researchers. In contrast, commodity GPUs (e.g., RTX 4090/3090) are recently more favored [18, 29, 37, 39] for their high accessibility, adequate and higher cost-effective computing power. For instance, the cost-performance ratio of RTX 4090 is 5.4× that of A100.

Ideally, commodity GPUs are well-suited for the low computing demand of embedding model training due to their memory intensity. However, directly running existing training system [12] on commodity GPUs leads to a 46% reduction in throughput (§2.4). Our analysis reveals that the chief culprit is their limited communication resources due to the lack of hardware support for PCIe peer-to-peer (PCIe P2P) communication, which is prone to CPU involvement and additional data bounced on host memory (Figure 1). This causes low bandwidth of GPU collective communication (46% ↓) and CPU-involved software overhead (up to 1.9× ↑), which obstructs training efficiency.

This paper presents Frugal, a communication-efficient training system designed to unlock commodity GPUs' full potential for embedding model training. Based on the observation that communication between commodity GPUs is subject to be bounced on host memory due to the absence of PCIe P2P support, the key idea of Frugal is *proactively flushing* mechanism. Specifically, unlike existing systems where all GPUs passively wait for queries from other GPUs and all communication is on the critical path of training, each GPU in Frugal proactively flushes its own parameters, which other GPUs will access, into host memory. Thus, this mitigates half of the communication overhead from the critical path and eliminates GPU collective communication, so as to alleviate communication overhead. Further, given the latest parameters have been flushed into host memory, GPUs can directly retrieve parameters (missed in the local GPU cache) from host memory, without the coordination of CPU, thereby reducing CPU software overhead.

Based on this key idea, we propose *priority-based proactively flushing algorithm* (§3.3), $P^2F$ algorithm for short. It assigns a *priority* to each flush operation and prioritizes those that are about to be accessed while selectively deferring others. Specifically, priority is defined as the next-to-be accessed step number by anticipating future access of the corresponding parameters. Frugal records metadata (referred to as *g-entry*) for each parameter update, and maintains all g-entries in a global *priority queue (PQ)*. Frugal also runs multiple background threads to continuously dequeue g-entries from this PQ and flush the corresponding embedding updates into host memory.

Since proactively flushing runs asynchronously in the background, to prevent GPUs from reading outdated embeddings on host memory (i.e., **ensuring consistency**[1]), $P^2F$ algorithm utilizes the numerical relationship between the next-to-train step number and the front of PQ, to appropriately block foreground training processes when necessary. We formally prove our algorithm satisfies synchronous consistency in §3.3.

Since the throughput of proactively flushing directly determines the stall time of training processes, Frugal introduces *parallel flushing* mechanism (§3.4) to **improve efficiency** of flushing. Specifically, we find that the scalability of PQ determines flushing efficiency. Thus, Frugal tailors a two-level PQ data structure for our scenario. Based on the observation that $P^2F$ algorithm has a finite range of priority values, PQ is characterized by a layer of priority index pointing to a lock-free g-entry hash table whose entries share the same priority. Thus, Frugal enjoys great scalability and time complexity of $O(1)$ for all PQ-related operations, in contrast to $O(\log N)$ complexity and severe near-root conflicts in traditional tree heap, where $N$ is the number of parameters.

We evaluate Frugal with typical embedding models including graph embedding models and recommendation models on an 8× RTX 3090 server. Compared with SOTA training systems in their respective fields (i.e., DGL-KE [1] and NVIDIA HugeCTR [12]), Frugal boosts throughput by 1.2-1.4× and 6.1-8.7× respectively. Compared with existing systems on datacenter GPUs, Frugal can achieve similar throughput using merely RTX 3090s, with a 4.3× cheaper cost.

Overall, this paper makes the following contributions:

- It analyzes two pitfalls of existing training systems on commodity GPUs, i.e., low collective communication bandwidth and CPU involvement overhead.
- It introduces Frugal, an embedding model training system on commodity GPUs, with goals of alleviating GPU-GPU communication and reducing CPU-involvement overhead by parallel flushing GPU-GPU communication to host memory proactively.
- Comprehensive experiments demonstrate the effectiveness and efficiency of Frugal's design.

Code of Frugal is available at https://github.com/thustorage/Frugal.

[1] Frugal achieves synchronous consistency [40, 49], i.e., never read/modify an outdated parameter. Thus, Frugal does not affect model convergence.

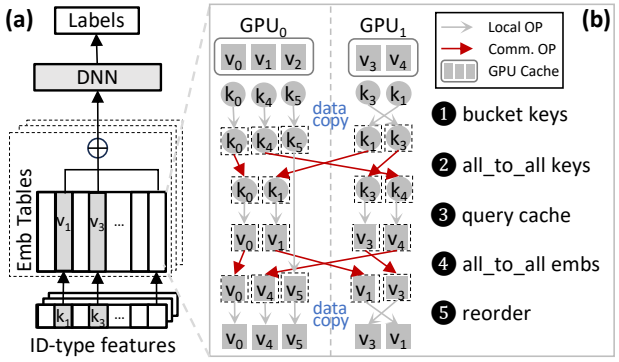


**Figure 2.** (a) Simplified embedding model structure. (b) Architecture of existing multi-GPU training systems.

## 2 Background and Motivation

### 2.1 Embedding Model

**Model structure.** Embedding models drive the main revenues of technology companies, leading to substantial investments in computational resources for their development and deployment. For example, Meta states that around 79% of the computation resources of their AI datacenter are spent on embedding models [31, 35].

Unlike classic deep learning models used in CV and NLP, embedding models are primarily applied to learning tasks on high-dimensional ID-type features (IDs) such as user IDs and graph node IDs. Since IDs can not be fed into DNN directly, embedding models leverage the embedding technique to bridge IDs to DNN. As illustrated in Figure 2a, embedding models consist of two parts: embedding layer (embedded tables) and DNN. 1) The embedding layer is used to process IDs by mapping original high-dimensional IDs to low-dimensional dense vector representations (called *embeddings*), which can then be fed into DNN for learning. Specifically, for each type of ID, the embedding layer maintains a two-dimensional matrix called *embedding table* with a shape of $c \times d$ where each row represents an embedding vector. Here, $c$ denotes the size of the value domain space for this type of feature, and $d$ denotes the embedding dimension. With embedding tables, the embedding layer treats IDs as keys and looks up dense low-dimensional embedding vectors (values) on them. 2) The DNN layer handles both dense inputs (omitted in this figure) and embedding vectors. It combines dense inputs with all embedding vectors, aggregating them as inputs to DNN for the final prediction.

**Embedding bottleneck.** Extensive works from both academy [46, 47] and industry[22, 23, 25, 55] report that the embedding layer tends to become the performance bottleneck. For example, Alibaba [22, 23] and Facebook [25] reported that the embedding layer of their production models account for over 60% time. Thus, FRUGAL mainly focuses on optimizing embeddings. In this paper, we interchangeably use the term *embedding* and *parameter*.

| | Datacenter GPU (A100) | Commodity GPU (RTX 4090) |
|---|---|---|
| Tensor FP16 | 312 TFLOPS | 330 TFLOPS |
| Tensor FP32 | 156 TFLOPS | 83 TFLOPS |
| Memory Capacity | 80 GB | 24 GB |
| Link Bandwidth | 900 GB/s (NVLINK) | 64 GB/s (PCIe 4.0) |
| Price | $16000 | $1600 |
| Dollar per FP32-TFLOPS | 103 $/TFLOPS | 19 $/TFLOPS |

**Table 1.** Main characteristics comparison between commodity GPUs and datacenter GPUs.

**Multi-GPU embedding model training.** As shown in Figure 2b, existing multi-GPU embedding model training systems [12, 38] tend to maintain multi-GPU embedding cache to absorb accesses to host memory. In these systems, to query or modify the latest parameters on other GPUs, it is subject to i) collective communication overhead (❷❹), and ii) CPU-involvement software overhead (❶❺). These overheads get more severe on commodity GPUs, as we will describe later.

### 2.2 Commodity GPU

GPUs can be classified into two categories: data center GPUs (e.g., NVIDIA A100, A30) and commodity GPUs (e.g., NVIDIA RTX 3090, 4090). Table 1 lists the main characteristics of a representative GPU from each category. Datacenter GPUs tend to be overpriced and some of the top-tier models (e.g., A100-SXM) are sold in 8-card bundles only. Therefore, for medium-sized companies, research labs, or individual researchers, commodity GPUs are often a more cost-effective choice. Commodity GPUs bring both opportunities and challenges to embedding model training:

- **Opportunities: Adequate and cost-effective computational power.** As shown in Table 1, commodity GPUs such as RTX 4090 now offer comparable FP32 TFLOPS to datacenter GPUs such as A100, with even greater FP16 TFLOPS surpassing the A100, yet priced at only a tenth of A100's cost. In terms of the cost-performance ratio ($/TFLOPS), RTX 4090 is only 18.4% of A100's cost.
- **Challenges: Limited communication resources.** However, compared to datacenter GPUs, commodity GPUs suffer from limited communication resources. As shown in Figure 1b, new commodity GPUs do not support NVLink and PCIe P2P [28]. They need CPU to coordinate cross-GPU communication and bounce the data on host memory. The GPU→host memory→GPU copy not only increases the communication latency but may lead to bandwidth bottlenecks at the CPU root complex [18].

**Discussion: Speculation on communication resources of future commodity GPUs.** Over time, NVIDIA has gradually weakened the communication capabilities in commodity GPUs to differentiate them from datacenter GPUs. Early models (e.g., TITAN) supported both additional link (NVIDIA

SLI [5], which can link together 4 GPUs) and PCIe P2P, then only 2-GPU NVBridge [6] (10/20-series), and now no P2P supported (30/40-series). Based on this historical trend, we think that it is highly likely that support for P2P on commodity GPUs will not be reinstated.

### 2.3 Unified Virtual Addressing (UVA)

CUDA provides the Unified Virtual Addressing (UVA) [2] feature, which allows mapping of the current GPU memory, other GPU memories, and host memory to the same virtual address space, so that GPU kernels can directly access the entire virtual address space using load/store instructions. For fine-grained data access, UVA can achieve much lower latency compared to DMA engine copies (cudaMemcpy), as the entire process does not require CPU involvement [51].

Due to hardware limitations, **the UVA feature on current commodity GPUs is restricted**. Unlike datacenter GPUs supporting direct access to both host memory and other GPUs, commodity GPUs only support direct access to host memory, but not to other GPUs.

### 2.4 Motivation

**Experiment: impact of commodity GPUs on embedding model training.** While there have been numerous research works [4, 8, 38] on embedding model training, they are specially designed for datacenter GPUs, relying on PCIe P2P features that are not available on commodity GPUs. We take NVIDIA HugeCTR [4] as an example and run a microbenchmark to analyze the performance difference of existing systems between datacenter and commodity GPUs. HugeCTR is an NVIDIA-customized embedding model training framework for recommendation and advertising models, which integrates distributed GPU caching to reduce host memory access. We train a DLRM [35] model on a real-world dataset (Avazu [10]) with 4 RTX 3090 and 4 A30 GPUs respectively; Both types of GPUs are linked with the same PCIe 4.0 bus (with a link bandwidth of 32 GB/s); please see §4 for the detailed server configurations.

Figure 3a illustrates that the training throughput on commodity GPUs decreases by up to 37% compared to datacenter GPUs. To analyze the underlying reasons behind this performance gap, we decouple the time spent on one iteration (including forward, backward, and optimizing steps) into the following components: collective communication (comm.), host memory access (host DRAM), local GPU cache access (cache), and other operations like DNN computation (other). As shown in Figure 3c, the performance gap between two types of GPUs mainly arises from 1) collective communication, and 2) cache miss processing (i.e., host DRAM time).

**Analysis.** Based on the above results, we find that there are two reasons leading to the performance gap.

1) *Low collective communication bandwidth.* A high percentage (54 – 72%) of time difference is collective communication. As stated in §2.1, the main communication pattern of distributed embedding model training is all_to_all primitive, which is used for exchanging inputs (embedding keys) and outputs (embeddings) of GPU cache queries. However, according to our benchmark (Figure 3b), the all_to_all communication bandwidth on commodity GPUs is only 54% of that on datacenter GPUs. The primary reason is the lack of support for PCIe P2P communication in commodity GPUs, which introduces additional copying overhead with data transfer through the host memory's bounced buffer.

2) *CPU involvement overhead during GPU-GPU communication.* Since commodity GPUs lack the support of both PCIe P2P and UVA features during GPU-GPU communication, missing parameters in local GPU caches must be intermediated by CPU software, which introduces multiple additional data copies [51] (Figure 2b). This results in cache miss path processing time accounting for up to 43% of the entire caching system.

We summarize §2.2 and this section: with adequate and cost-effective computational power, commodity GPUs should have been the ideal choice for memory-intensive embedding model training. However, existing systems are designed only for datacenter GPUs. Without consideration of the limited communication resources of commodity GPUs, existing systems are bottlenecked by communication and can not fully unleash the potential of commodity GPUs.

## 3 Design & Implementation

We introduce Frugal, a multi-GPU embedding model training system tailored for commodity GPUs, with the purpose of overcoming the deficiency of communication resources and fully leveraging the cost-effectiveness advantages of commodity GPUs. As shown in Figure 5, Frugal follows the basic structure of existing systems, organizing commodity GPU memory and host memory into a two-tiered structure, where GPU memory caches hot parameters.

Frugal focuses on synchronous training instead of asynchronous training for the following reasons. First, asynchronous training is notorious for poor model convergence, which may bring significant revenue losses and is unacceptable in key applications of embedding models (e.g., recommendation, advertising). For example, prior work has shown that asynchronous training can cause up to an 8% drop in AUC [32], while Alibaba reports that even a modest 0.1% decline in AUC can translate into a significant revenue loss for commercial systems [56]. Second, synchronous training is the mainstream training paradigm in industrial applications. Prominent industrial embedding training frameworks only support synchronous training (e.g., NVIDIA HugeCTR).

In this section, before describing the detailed designs of Frugal, we first outline its key idea.

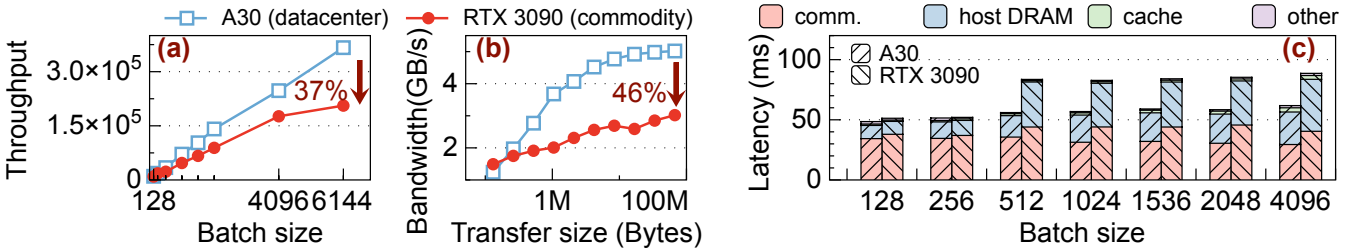


**Figure 3. Motivation.** *(a) Training throughput on 4 NVIDIA A30 (datacenter) and RTX 3090 (commodity) GPUs. (b) All-to-all collective communication bandwidth on A30 and RTX 3090. (c) Time breakdown of one training iteration.*

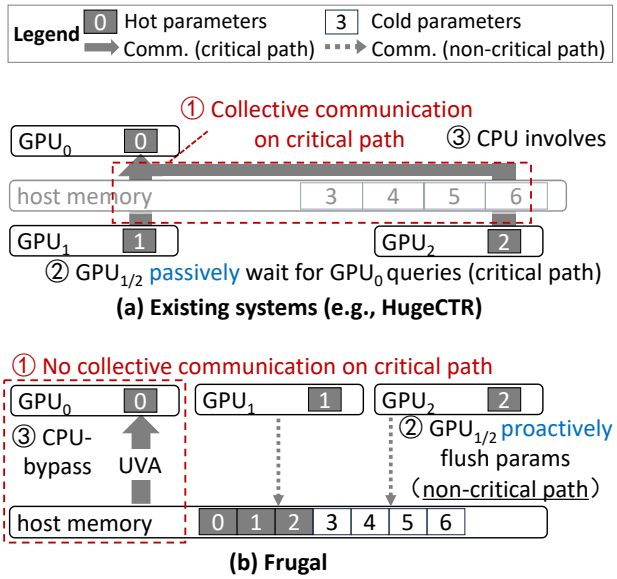


**Figure 4. Key idea of FRUGAL.** *Three design differences between FRUGAL and existing systems are denoted as ①-③.*

### 3.1 Key Idea

The observation of FRUGAL is that the communication between commodity GPUs must be bounced on host memory, which can be retrofitted for maintaining consistency among multiple GPU caches at low cost. Based on this observation, FRUGAL introduces the key idea of *proactively flushing*.

**Key idea: proactively flushing.** As shown in Figure 4, unlike existing systems where all GPUs passively wait for queries from other GPUs and all communication is on the critical path of training (Figure 4a), FRUGAL decouples GPU-GPU communication (e.g., $GPU_1 \rightarrow GPU_0$) into two steps,

1) $GPU_1 \rightarrow$host memory (on the non-critical path): where $GPU_1$ proactively flushes the latest parameters that $GPU_0$ needs to access into host memory, by foreseeing future parameter access traces.

2) host memory$\rightarrow GPU_0$ (on the critical path): where $GPU_0$ directly reads all non-local parameters on host memory.

With such decoupling design, FRUGAL can reduce the communication latency exposed in the critical path by hiding half of the communication in the non-critical path. Note that Frugal does not introduce any additional data movement in terms of GPU communication, since for commodity GPUs, the inter-GPU communication is inherently required to bounce on the host memory.

Aside from hiding communication overhead, proactively flushing provides another benefit: alleviating CPU (software) overhead by UVA-enabled host memory retrieving. Given that proactively flushing has ensured the latest parameters can always be flushed into host memory before GPU retrieving, FRUGAL can eliminate the coordination of CPU and transfer the cache miss path processing from CPU-side software to GPU-side hardware. Specifically, FRUGAL fuses the local GPU cache access and host memory access into one GPU kernel by utilizing the limited UVA features of commodity GPUs. It enables GPUs to directly access parameters in host memory with a zero-copy manner, fully bypassing CPU and eliminating the additional data copies (§2.4).

**Challenges.** Although the key idea of proactively flushing is conceptually simple, the real challenge lies in a crucial question: how to maintain the consistency of proactively flushing with low overhead? It can be boiled down to two following challenges.

First, we need to ensure the correctness of training consistency. In other words, proactively flushing needs to be done in a timely manner, so as to prevent GPUs from reading outdated versions of parameters from host memory. As stated in §3, inconsistency may result in reduced model accuracy or even non-convergence of training. Second, we need to minimize the cost of flushing. A straightforward write-through flushing policy for all the updates leads to long stalls in GPU computation (§4.3).

**Our solution.** To achieve the two aforementioned goals, FRUGAL introduces *priority-based proactively flushing algorithm* ($P^2F$ algorithm, §3.3). Its key idea is to prioritize flushing parameter updates that are about to be accessed to host memory, while selectively deferring others. We define a parameter update's priority as the training step number at which each parameter will be accessed next. Then, FRUGAL organizes all lingering updates using PQ.

To ensure correctness, FRUGAL utilizes the numerical relationship of the front of the PQ, to stall foreground training processes when necessary, and prevent them read outdated parameters on host memory. We prove that our algorithm satisfies synchronous training consistency.

To achieve a high efficiency, we introduce *parallel flushing* mechanism (§3.4) to speedup priority-related operations in $P^2F$ algorithm. We find that the PQ scalability is the key to flushing efficiency. Thus, FRUGAL tailors a high-performance two-level concurrent PQ data structure and adopts scenario-specific optimization to improve flushing efficiency.

### 3.2 FRUGAL Overview

As shown in Figure 5, FRUGAL primarily consists of two types of processes: training process and controller process. There are a total of $n$ training processes, where $n$ represents the GPU count. Each training process handles model training while maintaining private embedding caches of each GPU. The controller process manages the complete set of parameters in host memory, and exposes them to all training processes via a shared memory interface. The controller process proactively flushes parameter updates (first buffered in the *update staging queue*) into host memory by running $P^2F$ algorithm (§3.3), with the help of a customized two-level PQ (§3.4).

The controller process comprises four main components.

1) *Sample queue.* FRUGAL prefetches all IDs of $L$ steps in the future, and stores them in this queue. Here, $L$ is a hyper-parameter that is set to 10 by default.

2) *Update staging queue.* This queue maintains all parameter updates that will be flushed to host memory. After a training process updates the cache in its local CPU, it also inserts the updates into the update staging queue.

3) *Two-level priority queue (PQ).* A customized PQ supports high-concurrency operations such as enqueue, dequeue, and adjusting the priority of an existing element (adjustPriority). It maintains delayed updates according to their priority. Updates in the PQ are sorted based on their next access time (i.e., priority); updates to items accessed earlier have a higher flush priority (§3.3).

4) *Flushing threads.* The controller process maintains several background flushing threads, each independently fetching the highest-priority parameter updates from the PQ and flushing them to host memory (§3.4).

With the assistance of these components, the parameter query and update operations in FRUGAL proceed as follows:

For parameter query operations in forward, the training process first checks its local GPU cache. If not found, FRUGAL utilizes the UVA feature of GPU to directly access the complete set of parameters in host memory, achieving CPU-bypass and zero-copy retrieval of the missing parameter.

For parameter update operations in backward, the training process first updates the local GPU cache and then inserts the parameter update into update staging queue while notifying

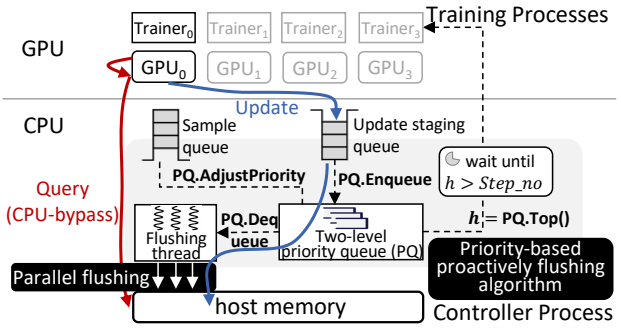


**Figure 5.** FRUGAL design overview.

the controller process. In the background, the controller process records them in the corresponding g-entries, and adjusts their priorities in the PQ. Finally, when the updates dequeue from PQ, flushing threads flush them into host memory and finish updates.

### 3.3 Priority-based Proactively Flushing ($P^2F$) Algorithm

FRUGAL designs $P^2F$ algorithm to timely flush updates to host memory before GPU reads in priority. In this section, first, we introduce the structure of metadata (i.e., g-entry) maintained for each parameter to calculate priority. Then, we give a formal definition of priority. Next, we describe the specific algorithm and provide an example illustrating its detailed process and advantages over the write-through strategy. Finally, we prove that our algorithm satisfies synchronous training consistency.

**Metadata of parameters (g-entry).** To calculate the priority, FRUGAL maintains metadata (referred to as *g-entry*) in PQ for two categories of parameters: i) Parameters soon to be accessed (i.e., those in the sample queue). ii) Parameters with pending updates not yet flushed to host memory (i.e., those in the update staging queue).

Specifically, each g-entry includes four following fields:

- *key*: the key of parameter.
- $R$ set: read set, a collection of training step numbers which the parameter will soon be accessed.
- $W$ set: write set, a collection of ⟨step_no, Δ ⟩ pairs recording all pending parameter updates not yet flushed to host memory, where Δ represents gradients.
- *priority*: priority of this g-entry. A numerically smaller priority means the target item is accessed earlier, and needs to be flushed earlier. Formally, the priority of each parameter's g-entry is defined as in Equation (1):

$$priority = \begin{cases} \min\{R \text{ set }\}, & \text{when } W \text{ set } \neq \emptyset \\ \infty, \text{when } R \text{ set} = \emptyset, \text{ or } W \text{ set} = \emptyset \end{cases} \quad (1)$$

**Algorithm details.** For training processes, assume the current training step number (step_no) to be trained is $s$. 1) The condition for starting training at step $s$ is: the priority

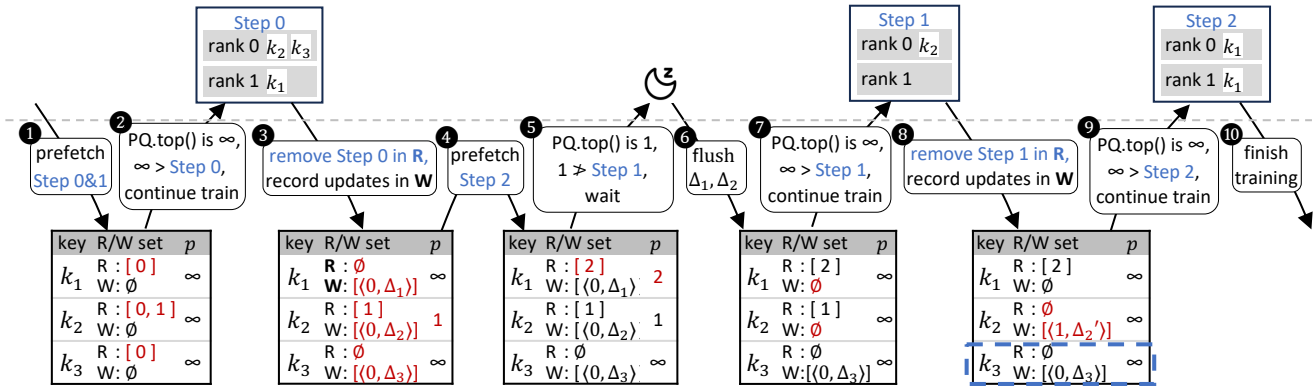


**Figure 6. Example of $P^2F$ algorithm.** The prefetching training step length $L = 2$. All modifications to g-entries are highlighted in red. The below table shows the contents of PQ, where $p$ represents the priority of g-entries.

value at the front of PQ is strictly greater than $s$. 2) Once the condition is met, the training process completes the forward and backward computations of the neural network. The gradients produced by the backward process are transferred to the update staging queue. 3) The controller process removes $s$ from the $R$ set of g-entry corresponding to the batch parameters, and inserts s and the gradient into the $W$ set;

The controller process is primarily responsible for sample preloading and transferring elements from the update staging queue to the PQ. 1) It maintains a separate prefetch thread that, whenever the depth of the sample preload queue is less than $L$ (assuming the current depth is $L\_now$), prefetches parameter keys to be accessed in the next ($L-L\_now$) steps and stores them in the sample queue. For each prefetched parameter, assuming its access step number is $s$, $s$ is inserted into its g-entry's $R$ set. 2) It continuously retrieves elements from the update staging queue (in the form $\langle key, step\_no, \Delta\rangle$), and inserts the gradient $\Delta$ and training step number step_no into the $R$ set of the corresponding g-entry.

In addition, the controller process also maintains several background flushing threads. They continuously retrieve the g-entry with the highest priority (i.e., numerically smallest priority) from the PQ and flush the parameter updates recorded in its $W$ set to host memory.

**Example.** Figure 6 illustrates a specific example of the $P^2F$ algorithm. ❶ Before training step 0, the controller process performs a lookahead for parameter keys to be accessed at step 0 ($k_2, k_3, k_1$) and step 1 ($k_2$) respectively. It sets the $R$ set of the corresponding g-entry in the PQ and updates their priorities. ❷ At this point, the priority at the front of the queue is $\infty$, which is numerically greater than step 0, so training step 0 proceeds directly without block. ❸ After completing training for step 0, the training process generates the gradient $\Delta_{\{2,3,1\}}$. Subsequently, the controller process removes step 0 from the $R$ set of $\Delta_{\{2,3,1\}}$'s g-entry and inserts $<0, \Delta_{\{2,3,1\}}>$ into the $W$ set of $\Delta_{\{2,3,1\}}$'s g-entry. ❹ Before training step 1, the controller process performs a lookahead for parameter keys to be accessed at step 2 and sets the R set, updating the priorities. ❺ At this point, the priority at the front of the queue is 1, which is not greater than step 1, so training processes are blocked. ❻-❼ Training for step 1 does not begin until the flushing thread writes $\Delta_{1,2}$ to host memory. ❽-❿ Training of step 2 proceeds as steps 0 and 1. After training, the system waits for flushing threads to write all deferred parameter updates to host memory.

In this example, because parameter $k\_3$ is not accessed after step 0, the $P^2F$ algorithm delays flushing the update of the gradient generated for $k\_3$ at step 0, until after step 2 (as indicated by the blue dashed box in Figure 6). This demonstrates the superiority of $P^2F$ algorithm, which prioritizes flushing parameters that are about to be accessed soon while deferring updates for other parameters. In this way, our algorithm can effectively reduce the computation stall, caused by flushing, of foreground training processes, thus improving training efficiency.

**Proof of synchronous consistency.** Here, we prove that the $P^2F$ algorithm adheres to synchronous training consistency. The proof is divided into two components: i) From synchronous training consistency, we prove that synchronous training consistency is equivalent to invariant (2). ii) From the $P^2F$ algorithm, we prove that it guarantees the fulfillment of invariant (2).

At step $s$, there does not exist any g-entry that simultaneously satisfies: $\begin{cases} W \text{ set} \neq \emptyset \\ s \in R \text{ set} \end{cases}$ (2)

For i), from synchronous training consistency, it is equivalent to the following statement that at any given moment, there should be no read operations on "stale parameters", which refers to parameters with pending updates. Given that

the $R$ set of g-entry records future read sets and the $W$ set records pending updates, this is equivalent to invariant (2).

For ii), from the process of the $P^2F$ algorithm, note that the condition of step $s$ is that the priority value of the frontmost g-entry in the PQ strictly exceeds $s$. According to the definition of priority (Equation (1)), at this point, no g-entry satisfies both a non-empty $W$ set and the existence of elements in the $R$ set less than or equal to $s$. This ensures the fulfillment of invariant (2).

In conclusion, the $P^2F$ algorithm adheres to synchronous training consistency. □

### 3.4 Parallel Flushing

As described in §3.3, flushing threads involve numerous high-concurrency operations of PQ, including enqueue, dequeue, and adjust priorities. In practice, the PQ performance significantly impacts training performance for two main reasons: 1) The performances of enqueue and adjusting priorities affect the time of backward pass. During the backward pass, adjusting priority is on the critical path. 2) The performance of dequeue affects the throughput of flushing threads, thereby determining the training processes' stall time.

The straightforward implementation of a PQ is using a classic binary tree min-heap. However, its performance is suboptimal in our practice (please see Exp #4). It suffers from O(log $N$) operation complexity (where $N$ is the number of parameters) and limited concurrency caused by near-root contention.

**Two-level concurrent priority queue.** Based on the characteristics of $P^2F$ algorithm, Frugal customizes a two-level PQ. The idea is primarily based on the following observations: i) According to the definition of priorities of Equation (1), the priority values range within a finite set which contains integers from 0 to $max\_step$. Here $max\_step$ is the maximum number of training steps. ii) The priority value of a specific g-entry never decreases.

Figure 7 illustrates the structure of two-level PQ. It consists of two levels: the first level is a priority index, which is a pointer array of length ($max\_step + 2$). Each slot represents a priority value (ranging from 0 to $max\_step$, or ∞), pointing to the second level, a hash table of g-entries with the same priority value. The second level employs a lock-free dynamic scalable hash table structure [34], offering excellent concurrency performance and efficient memory usage facing dynamic capacity changes.

The operations in the two-level PQ are as follows:

- *Enqueue*: Based on the priority of the g-entry, insert it directly into the corresponding g-entry hashtable.
- *Dequeue*: Sequentially scan the priority index from priority 0, until finding a non-empty hashtable, and dequeue one g-entry from it. Dequeue can be batched to remove the repeated scanning overhead.
- *AdjustPriority*: Suppose the priority of g-entry changes from $p\_old$ to $p\_new$. To prevent concurrency errors where readers (dequeue operations) might not find the g-entry in two corresponding hash tables, Frugal first inserts the g-entry into the hash table corresponding to $p\_new$, then deletes it from the hashtable corresponding to $p\_old$. Dequeue operations can identify an inconsistent g-entry by comparing its priority with the priority of the hash table in which it resides.

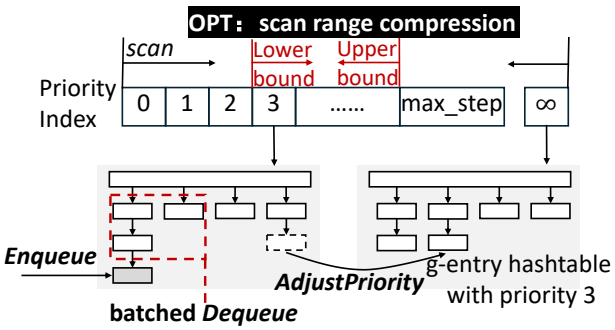


**Figure 7.** Structure of two-level priority queue tailored for $P^2F$ algorithm.

Frugal's PQ design overcomes issues with the tree heap structure. First, the time complexity of PQ-related operations reduces to O(1). Moreover, it exhibits superior concurrency performance (as demonstrated in §4), significantly reducing the stall time of training processes. Additionally, the two-level PQ is further optimized for Frugal's scenario with the following customization.

**Optimization: scan range compression.** In the original design, the dequeue operation scanned from priority 0 sequentially, which could be costly given the length of the priority index (i.e., step count during model training). This optimization improves dequeue efficiency by compressing the priority range that needs scanning. Its key observation is that the priority value of a specific g-entry never decreases. Specifically, Frugal maintains two global variables that record the lower and upper bounds of all g-entry priority values (excluding ∞). During dequeue scanning, it only scans two intervals: ① the interval determined by the upper and lower bounds, and ② ∞. Frugal ensures that these intervals cover all g-entry priority values (not violate correctness) but does not guarantee they are the exact supremum and infimum (not ensuring accuracy).

The values for the bounds are determined as follows: assuming the current step is $s$, for the lower bound, as the priority value of each g-entry only increases, whenever an item is dequeued, the lower bound is updated to its priority value. For the upper bound, since the control process only prefetches parameters for the next $L$ steps, the upper bound is set to $(s + L)$. In experimental evaluations, for PQ with millions of entries, this optimization can reduce the time of dequeue operation by 28%.

| | Dataset | #Vertexes | #Edges | #Relations | Model Size |
|---|---|---|---|---|---|
| KG | FB15k | 592k | 15k | 1.3k | 52 MB |
| | Freebase | 338M | 86.1M | 14.8K | 68.8 GB |
| | WikiKG | 87M | 504 M | 1.3k | 34 GB |
| | Dataset | #Features | #IDs | #Samples | Model Size |
| REC | Avazu | 22 | 49 M | 40M | 5.8 GB |
| | Criteo | 26 | 34 M | 45M | 4.1 GB |
| | CriteoTB | 26 | 882 M | 4.37B | 110.3 GB |

**Table 2.** Datasets used in the real-world applications.

# 4 Evaluation

## 4.1 Experimental Setup

**Testbed.** We run experiments on a server with two Intel Gold 6130 CPUs at 2.1 GHz, 1.5 TB of DRAM, and 8 NVIDIA RTX 3090 commodity GPUs. Each GPU is connected to the host via a PCIE 4.0×16 link.

**Workloads.** We use the following two workloads.
Synthetic workloads. We synthetic different embedding model traces to test FRUGAL and other baseline systems under different embedding ID (key) distributions. We generate keys using three types of distributions: uniform distribution, and Zipfian skewed distributions with parameters 0.9 and 0.99. Unless otherwise specified, the embedding key space size is 10 million, the embedding dimension is 32. In this workload, we only test the embedding part in the forward and backward pass, and eliminate the DNN computation part.
Real-world workloads. We use two representative types of real-world embedding models: knowledge graph (KG) and recommendation models (REC). i) For the KG model, the experiments use FB15k [15], Freebase [13] and WikiKG [7] as the datasets. The model tested is TransE [15], with an embedding dimension of 400 and a negative sampling batch size of 200. Unless otherwise specified, the training batch sizes for the two datasets are 1200 and 2000, respectively. All these hyperparameter settings are consistent with those in the original DGL-KE paper[54]. ii) For the REC model, we use three real-world datasets, Avazu [10], Criteo and CriteoTB [11]. The tested model is Facebook DLRM[35], with an embedding dimension of 32. The DNN part employs a fully connected network with the structure of 512-512-256-1. All hyperparameter settings are consistent with those used in the DLRM code repository [3]. Unless otherwise specified, the default batch size is 1024.

We primarily evaluate only one representative model per application, as different models typically share a similar embedding layer, and all of our techniques focus on optimizing this part. We will evaluate the sensitivity of all competitor systems to different models in Exp #11.

**Competitor systems.** We compare FRUGAL with SOTA training systems in their respective fields. i) For KG, we compare FRUGAL with DGL-KE (without caching) and DGL-KE-cached (a variant integrated with a multi-GPU cache). ii) For REC, we compare FRUGAL with PyTorch (without caching) and HugeCTR (with caching). Since the multi-GPU caching feature of HugeCTR has not yet been integrated into PyTorch, and DGL-KE is implemented based on PyTorch, to ensure a fair comparison, we re-implement its multi-GPU cache within PyTorch. Additionally, the experiments also evaluate a baseline version of FRUGAL, called FRUGAL-SYNC, which uses a write-through strategy to synchronously flush all updates to host memory.

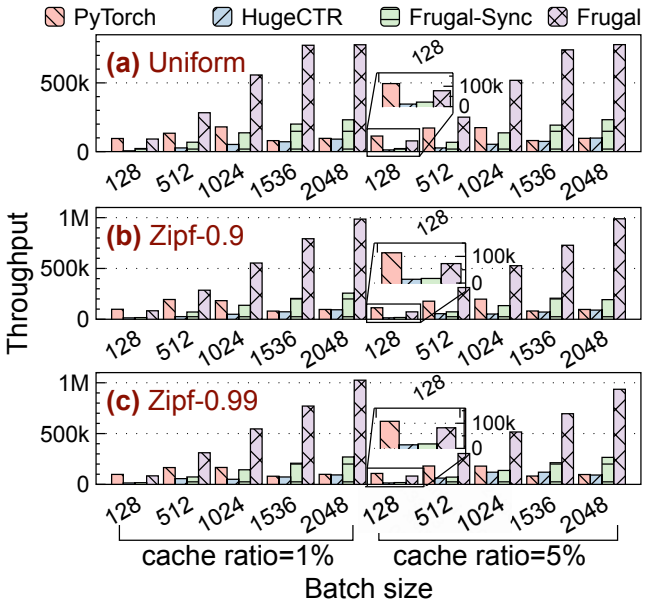


**Figure 8. (Exp #1) Microbenchmark.**

Unless otherwise specified, all experiments use the following configurations by default: the cache size (ratio) is set to 5% of the total parameters for the GPU-cached model, all throughputs refer to samples per second, all GPUs on the server are utilized by default, and 8 flushing threads are used.

Accuracy is not tested in the evaluation, as all competitor systems meet the synchronous training consistency (§3.3). We also omit the testing of cache hit ratios, as all competitor systems follow the existing cache strategy in HugeCTR, thus they share a similar hit ratio.

## 4.2 Microbenchmark

**Exp #1: Microbenchmark.** FRUGAL focuses on the embedding process. To understand its effectiveness and eliminate interference from DNN computations and KG-specific operations (graph sampling), we use synthetic workloads to evaluate the throughput of FRUGAL and other systems. We also evaluate a UVM-capble [26] baseline system, called PyTorch-UVM. It leverages the Unified Virtual Memory (UVM) interface provided by CUDA to unify all GPUs' memory and host memory in one global unified virtual memory space.

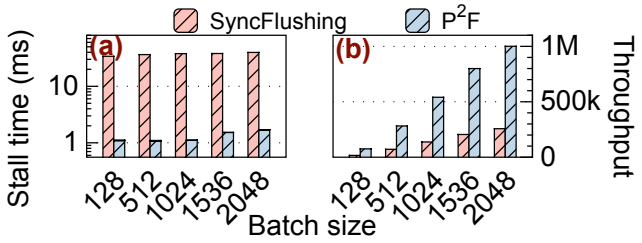


**Figure 9. (Exp #2) Effect of priority-based proactively flushing algorithm.** *SyncFlushing denotes the case of adopting write-through policy. (a) Stall time reduction of training processes; please note that the y-axis is log-scale. (b) Training throughput.*

However, the performance of PyTorch-UVM is two orders of magnitude slower than other systems. The reason is that the minimum migrating granularity (page size) of UVM is 4KB, while the granularity of model updating is an embedding (~512B). Thus, UVM's excessively large granularity will cause a lot of additional data movement overhead. For this reason, we emit its results in the following experiments.

From Figure 8, we observe that: first, FRUGAL's throughput is significantly higher than that of PyTorch, HugeCTR, and FRUGAL-SYNC, respectively reaching 1.5-10.2×, 4.3-11.3×, and 3.3-5.1×. The only exception is that when the batch size is extremely small, such as 128, the throughput of cache-enabled systems is lower than that of PyTorch. This is because the benefits brought by multi-GPU caching are outweighed by the collective communication overhead.

Second, as the batch size increases, all cache-enabled systems can better leverage the advantages brought by GPU parallelism, yielding better performance. Third, straightforward multi-GPU caching (HugeCTR) shows little performance variance under different distributions. This is mainly due to its communication overhead and CPU-involved software overhead becoming the bottleneck for cache queries, while the impact of cache hit rate remains minimal. Fourth, compared to HugeCTR, FRUGAL-SYNC and FRUGAL consistently exhibit better throughput across different distributions and cache sizes. We will analyze them detailedly in Exp #5.

### 4.3 Techniques

We evaluate the effectiveness of our proposed techniques and show how much they contribute to the final performance.

**Exp #2: Priority-based proactively flushing algorithm.** In Figure 9, we compare $P^2F$ algorithm with the write-through flushing scheme (denoted as *SyncFlushing*). We evaluate the stall time of training processes and end-to-end training throughput, the metrics that the flushing schemes mainly affect. This experiment uses a synthetic workload with a Zipfian-0.9 distribution and a 1% cache ratio; similar results are observed with other settings.

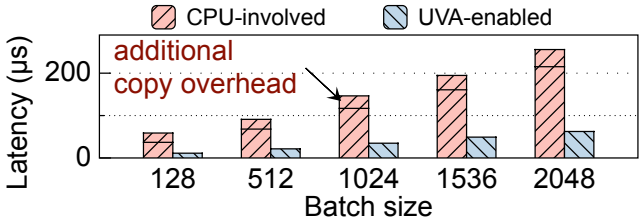


**Figure 10. (Exp #3) Effect of UVA-enabled host memory access.**

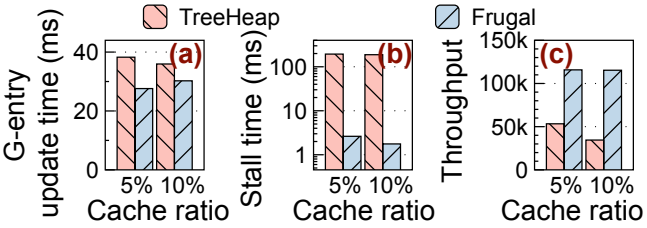


**Figure 11. (Exp #4) Effect of two-level priority queue.** *(a) Mean time of updating g-entries in a batch. (b) Stall time of training processes. (c) Training throughput.*

We make the following two observations. 1) From Figure 9a, we can see that $P^2F$ algorithm can reduce the training stall time by 34-101×, compared to SyncFlushing. The reason is that $P^2F$ algorithm decouples some embedding updates to non-critical paths. In contrast, SyncFlushing immediately commits all the updates in the critical path, resulting in a longer stall. 2) The reduction of stall time can improve the end-to-end throughput by 3.5-5.3×, as shown in Figure 9b.

**Exp #3: UVA-enabled host memory access.** FRUGAL leverages UVA-enabled host memory access to circumvent CPU overheads when accessing host memory. This experiment compares the performance of UVA-enabled host memory access (FRUGAL) with CPU-involved host memory access (PyTorch and HugeCTR).

Figure 10 shows the query latency of CPU-involved access and UVA-enabled access across different batch sizes. For the same batch size, UVA-enabled access lowers the host memory access latency by 3.1-3.4×. This improvement is partly due to the high concurrency of GPU, which enhances memory access parallelism, and partly due to UVA can avoid extra copy overhead on GPU (indicated by the arrow).

**Exp #4: Two-level priority queue.** This experiment compares two different concurrent PQ designs: TreeHeap and FRUGAL's two-level PQ. TreeHeap is a concurrent binary tree heap using per-node spinlocks. Both PQs are integrated into the FRUGAL system to evaluate their real performance impact. The experiment is conducted on KG model using Freebase dataset; results on REC models share a similar conclusion.

We measure three metrics: a) The mean time to complete all g-entries updates of a batch, which reflects the

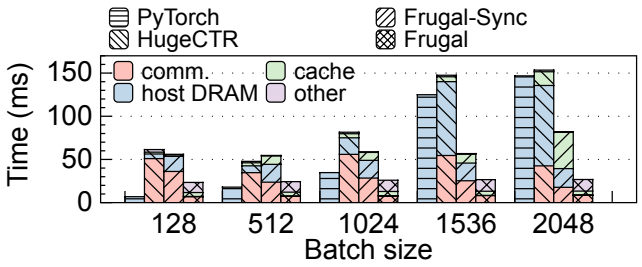


**Figure 12. (Exp #5) Contributions of techniques to performance.** *'comm.' denotes communication.*

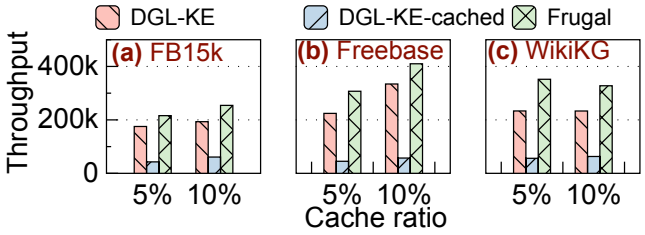


**Figure 13. (Exp #6) Training throughput of knowledge graph models.**

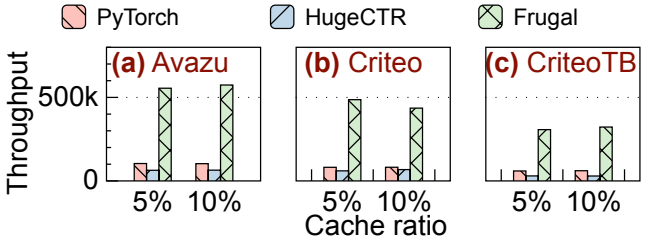


**Figure 14. (Exp #7) Training throughput of recommendation models.**

efficiency of concurrent enqueue and `adjustPriority` operations. b) Stall time of training processes, which reflects the efficiency of concurrent dequeue operations. c) End-to-end training throughput.

Figure 11 shows the results. a) FRUGAL is 1.2-1.4× faster than TreeHeap for completing g-entry updates in a batch. b) FRUGAL reduces stall time by 74.0-106.8×, lowering the stall time to milliseconds. This is because the $O(1)$ dequeue in two-level PQ is more efficient than $O(\log n)$ in TreeHeap. c) The efficiency of two-level PQ increases the training throughput of FRUGAL by 2.1-3.3× compared to TreeHeap.

**Exp #5: Contributions of techniques to performance.** To further analyze the contribution of the aforementioned techniques to the final performance, we break down the time of a single training step under a synthetic workload with a Zipfian-0.9 distribution. The meaning of each metric is the same as in §2.4. Figure 12 shows the results. First, compared to PyTorch, HugeCTR introduces GPU cache and significantly reduces host memory time by 22-55%. However, this reduction comes at the cost of a 0.3-5.8× increase in the cache update and collective communication.

Second, the introduction of synchronous flushing (FRUGAL-SYNC) reduces collective communication overhead in the forward pass by approximately 29-53%, since it only accesses local GPU cache. Additionally, FRUGAL-SYNC accelerates host embedding access by leveraging the UVA feature, reducing host memory time by up to 76%. As a result, FRUGAL-SYNC enhances training throughput by 1.2-2.8× compared to HugeCTR. However, the performance improvement of FRUGAL-SYNC remains limited. The reason is that its write-through policy may cause a long stall to aggregate all gradients and update parameters on DRAM.

Third, based on FRUGAL-SYNC, FRUGAL introduces priority-based proactively flushing algorithm that further decouples the collective communication for host memory writes to the background. Data shows that FRUGAL reduces collective communication time by approximately 60-85% and reduces host access time by about 98%. Consequently, the end-to-end training throughput is further improved by 3.5-5.1×.

### 4.4 Overall Performance

**Exp #6: Knowledge graph models (KG).** Figure 13 illustrates the training throughput across different systems for KG models. The cache ratios are set to 5% and 10%. For various datasets and cache ratios, FRUGAL consistently achieves significantly higher training throughput 1.2-1.5×, and 4.1-7.1×, compared to DGL-KE and DGL-KE-cached respectively. This demonstrates the efficacy of the FRUGAL technique, where the proactively flushing design eliminates collective communication and enhances training performance. Specifically, FRUGAL can boost performance more on the larger dataset (Freebase or WikiKG) since a larger ID space allows more optimization room for asynchronous flushing. Notably, the training throughput of DGL-KE-cached is up to 15.2% lower than the vanilla DGL-KE. This suggests that simply using existing multi-GPU caching on commodity GPUs can lead to a slight decrease in throughput.

**Exp #7: Recommendation models (REC).** Figure 14 shows the training throughput of recommendation models. Similarly, FRUGAL consistently achieves significantly higher training throughput compared to PyTorch and HugeCTR, respectively reaching 4.9-7.4× and 6.1-8.7×. The performance improvement is more pronounced for REC than for KG models, which is expected due to the more memory-intensive nature.

**Exp #8: Scalability.** Figure 15 shows the training throughput of all competitor systems across different numbers of GPUs. First, the performance of systems with straightforward caching (DGL-KE-cached/HugeCTR) is comparable to or even worse than that of systems without caching (DGL-KE/PyTorch). It indicates that the benefits of caching are outweighed by the overhead introduced by collective communication. Second, after increasing the number of GPUs

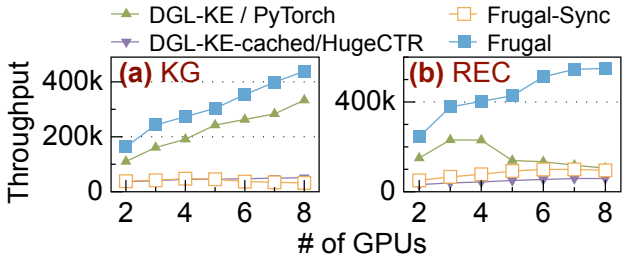


**Figure 15. (Exp #8) Scalability of training.** *(a) KG: Freebase dataset, (b) REC: Avazu dataset.*

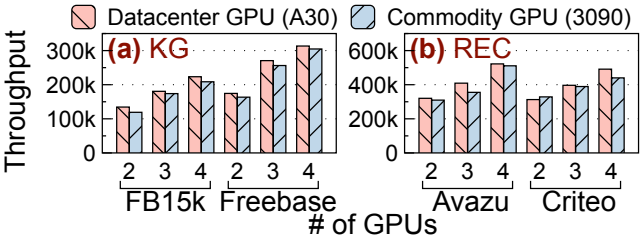


**Figure 16. (Exp #9) Cost efficiency of Frugal.**

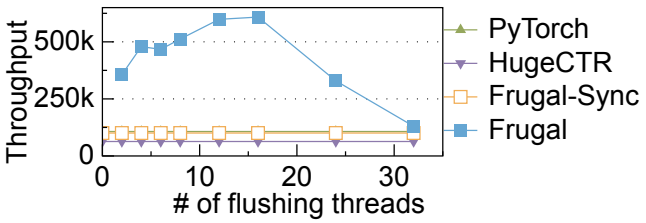


**Figure 17. (Exp #10) Sensitivity to the number of flushing threads.**

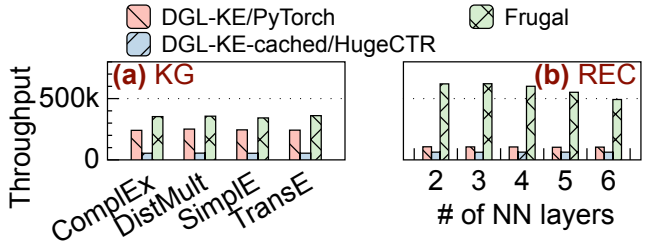


**Figure 18. (Exp #11) Sensitivity to embedding models.**

to a certain level (e.g., 4 in REC), systems without caching are limited by the bandwidth of the CPU root complex and the performance cannot scale up. Third, in contrast, Frugal exhibits superior scalability on two workloads. Frugal can enhance throughput by 1.2-4.9× across various numbers of GPUs. However, due to the hardware limitations of commodity GPU communication bandwidth, Frugal still can not achieve fully linear scalability.

We currently do not consider cross-server distributed training, since commodity GPUs are usually not equipped with high-end NICs, which will cause the network to become a bottleneck.

### 4.5 Cost Efficiency of Frugal

**Exp #9: Cost efficiency vs. datacenter GPUs.** To demonstrate Frugal's cost-effectiveness on commodity GPUs, this experiment compares its performance on RTX 3090 GPUs with existing systems on NVIDIA A30 GPUs (only showing the best performance). Both A30 and 3090 are interconnected with the same PCIe 4.0x16 link. Due to GPU resource constraints, we only evaluate up to 4 GPUs temporarily.

Figure 16 reveals that Frugal achieves comparable throughput (89 – 97%) to datacenter GPUs. Considering the price difference ($5,885 per A30 and $1,310 per RTX 3090), Frugal improves the cost-performance ratio by 4.0-4.3×.

### 4.6 Sensitivity Analysis

**Exp #10: Different flushing thread numbers.** Figure 17 shows the REC training throughput of Frugal under different numbers of flushing threads. The dataset is Avazu. The training throughput initially increases with the number of flushing threads (from 2 to 12), but then begins to decline (since 14). This is because the throughput of Frugal's flushing dictates whether the foreground training processes can proceed to the next training step. If there are too few threads, the foreground process experiences extended stalls. On the other side, too many flushing threads will divert CPU resources that would otherwise be used for model computation, causing a drop in performance. In practice, we set the number of flushing threads to 12, which achieves an optimal balance between high training throughput and CPU efficiency.

**Exp #11: Different DNN models.** For KG, we evaluate 4 different graph embedding models, including ComplEx [42], DistMult [50], SimplE [24], and TransE [15]. For REC, we evaluate the sensitivity by deepening the number of DNN layers in DLRM. Figure 18 shows the results. We observe that the performance of Frugal is consistently better than other systems across different models. Since our design mainly targets the embedding part, it has no side effects on the neural network part. Thus, changes in different DNN model will only affect how much performance gain we achieve.

## 5 Related Work

**Multi-GPU systems built for embedding models.** Existing works commonly cache hot embeddings into multiple GPUs to form a distributed caching [4, 9, 30, 36, 53]. They can be classified by caching policy and access method.

For the caching policy, they can be classified into three categories: replication [45, 52], sharding [4, 9, 30, 33, 36, 51, 53], and replication-sharding hybrid [8, 31, 38]. Replication policy replicates cache across GPUs, which is simple but can not fully utilize the aggregated memory capacity of multiple GPUs. Sharding policy shards hot embeddings to GPUs,

which tends to result in some hot parameters being partitioned to remote GPUs. Thus, the marginal performance gain (from the increase of hit rates) from multi-GPU cache capacity expansion is diminishing. To this end, replication-sharding hybrid policy combines these two policies and selectively replicates embeddings by solving an optimization problem. However, both replication as well as replication-sharding hybrid policies are only applicable to scenarios with frozen embeddings (such as inference), due to the heavy synchronization overhead imposed by replicas. FRUGAL pertains to a sharding policy in essence. It reduces the overhead of cross-GPU parameter access by proactively flushing.

For the access method, they can be categorized as message-based [4, 9] and unified address-based [33, 51]. Message-based systems query distributed cache via `all_to_all` collective communication primitive, which becomes the bottleneck on commodity GPUs for limited communication bandwidth (§2.1). Unified address-based systems allow each GPU to load/store the data on other GPUs, eliminating redundant data copies in the message-based systems. These systems rely on unthrottled UVA features of datacenter GPUs and will not work on commodity GPUs. In contrast, FRUGAL takes full advantage of the restricted UVA feature on commodity GPUs to save communication overhead based on the proactively flushing design.

**Deep learning systems on commodity GPUs.** Commodity GPUs have been widely applied to the training and inference systems of deep learning models [17, 18, 29, 37, 39]. For the training scenario, FTPipe [17], Harmony [29] and Mobius [18] design sophisticated plans of pipeline parallelism to achieve well load balancing across multiple commodity GPUs [17] or hide the CPU-GPU communication overhead caused by heterogeneous memory [18, 29]. Like FRUGAL, Mobius also considers the collective communication bandwidth bottleneck of commodity GPUs caused by the CPU root complex, but alleviates it by mapping different pipeline stages to different NUMAs. On the other hand, FlexGen [37] and PowerInfer [39] focus on the inference scenario and enable a single GPU to run ultra-large LLM with host memory offloading. They mainly focus on reducing the model swap overhead between the host and GPU, by utilizing reasonable scheduling [37] or the sparsity of model structure [39].

Unlike these works, which focus on dense models (e.g., LLM) that require dense parameter access, FRUGAL concentrates on embedding models whose parameter access is sparse. To the best of our knowledge, no prior works consider embedding models, including recommendation models and knowledge graphs, on commodity GPUs.

## 6 Conclusion

Commodity GPUs are highly favored for cost-efficient computing power. Ideally, they are well suited for the relatively low computational demands of memory-intensive embedding models. However, the limited communication resources hinder current systems to unleash their potential. We introduce FRUGAL, a system designed for embedding model training on commodity GPUs. By leveraging the proactive flushing mechanism, FRUGAL effectively decouples critical communication paths, significantly reducing GPU-GPU communication overhead and CPU involvement. Through comprehensive experiments on recommendation and graph embedding models, FRUGAL demonstrates more efficiency and cost-effectiveness compared to SOTA systems.

## Acknowledgements

We sincerely thank our shepherd Prof. Jason Lowe-Power for helping us improve the paper. We also thank the anonymous reviewers for their feedback. This work is supported by the National Natural Science Foundation of China (Grant No. 62332011).

## References

[1] awslabs/dgl-ke: High Performance, Easy-to-use, and Scalable Package for Learning Large-scale Knowledge Graph Embeddings. https://github.com/awslabs/dgl-ke. (Accessed on 01/30/2024).

[2] Cuda Driver API:: Cuda Toolkit Documentation. https://docs.nvidia.com/cuda/cuda-driver-api/group__CUDA__VA.html. (Accessed on 01/15/2024).

[3] FacebookResearch/DLRM: An Implementation of a Deep Learning Recommendation Model (DLRM). https://github.com/facebookresearch/dlrm. (Accessed on 01/31/2024).

[4] NVIDIA-Merlin/HugeCTR: HugeCTR Is a High Efficiency GPU Framework Designed for click-through-rate (CTR) Estimating Training. https://github.com/NVIDIA-Merlin/HugeCTR. (Accessed on 01/28/2024).

[5] Nvidia sli. https://docs.nvidia.com/gameworks/content/technologies/desktop/sli.htm. (Accessed on 10/20/2024).

[6] Nvlink high-speed gpu interconnect | nvidia quadro. https://www.nvidia.com/en-us/design-visualization/nvlink-bridges/. (Accessed on 10/20/2024).

[7] Ogb-lsc @ kdd cup 2021 | open graph benchmark. https://ogb.stanford.edu/kddcup2021/wikikg90m/#dataset. (Accessed on 10/21/2024).

[8] quiver-team/torch-quiver: PyTorch Library for Low-Latency, High-Throughput Graph Learning on GPUs. https://github.com/quiver-team/torch-quiver. (Accessed on 01/28/2024).

[9] Sparse Operation KIT — Documentation. https://nvidia-merlin.github.io/HugeCTR/sparse_operation_kit/master/intro_link.html. (Accessed on 01/28/2024).

[10] Click-Through Rate Prediction | Kaggle. https://www.kaggle.com/c/avazu-ctr-prediction, 2021.

[11] Display Advertising Challenge | Kaggle. https://www.kaggle.com/c/criteo-display-ad-challenge, 2021.

[12] NVIDIA/HugeCTR: HugeCTR Is a High Efficiency GPU Framework Designed for Click-through-rate (CTR) Estimating Training. https://github.com/NVIDIA/HugeCTR, 2021.

[13] Kurt Bollacker, Colin Evans, Praveen Paritosh, Tim Sturge, and Jamie Taylor. Freebase: a collaboratively created graph database for structuring human knowledge. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, page 1247–1250, New York, NY, USA, 2008. Association for Computing Machinery.

[14] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. Translating Embeddings for Modeling Multi-relational Data. *NeurIPS'13: Advances in Neural Information Processing Systems*, 26, 2013.

[15] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Durán, Jason Weston, and Oksana Yakhnenko. Translating embeddings for modeling multi-relational data. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'13, page 2787–2795, Red Hook, NY, USA, 2013. Curran Associates Inc.

[16] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, et al. Wide & Deep Learning for Recommender Systems. In *DLRS'16: Proceedings of the 1st Workshop on Deep Learning for Recommender Systems*, pages 7–10, Boston, MA, 2016. ACM.

[17] Saar Eliad, Ido Hakimi, Alon De Jagger, Mark Silberstein, and Assaf Schuster. Fine-tuning giant neural networks on commodity hardware with automatic pipeline model parallelism. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 381–396, 2021.

[18] Yangyang Feng, Minhui Xie, Zijie Tian, Shuo Wang, Youyou Lu, and Jiwu Shu. Mobius: Fine Tuning Large-scale Models on Commodity GPU Servers. In *ASPLOS'23: Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 489–501, Vancouver, Canada, 2023.

[19] Huifeng Guo, Ruiming Tang, Yunming Ye, Zhenguo Li, and Xiuqiang He. DeepFM: A Factorization-machine Based Neural Network for CTR Prediction. In *IJCAI'17: Proceedings of the 26th International Joint Conference on Artificial Intelligence*, page 1725–1731, Melbourne, Australia, 2017. AAAI Press.

[20] U. Gupta, C. Wu, X. Wang, M. Naumov, B. Reagen, D. Brooks, B. Cottel, K. Hazelwood, M. Hempstead, B. Jia, H. S. Lee, A. Malevich, D. Mudigere, M. Smelyanskiy, L. Xiong, and X. Zhang. The Architectural Implications of Facebook's DNN-based Personalized Recommendation. In *HPCA'20: IEEE International Symposium on High Performance Computer Architecture*, pages 488–501, San Diego, CA, Feb 2020.

[21] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive Representation Learning on Large Graphs. *NeurIPS'17: Advances in Neural Information Processing Systems*, 30, 2017.

[22] Wenqi Jiang, Zhenhao He, Shuai Zhang, Thomas B Preußer, Kai Zeng, Liang Feng, Jiansong Zhang, Tongxuan Liu, Yong Li, Jingren Zhou, et al. MicroRec: Efficient Recommendation Inference by Hardware and Data Structure Solutions. *MLSys'21: Proceedings of Machine Learning and Systems*, 3, 2021.

[23] Wenqi Jiang, Zhenhao He, Shuai Zhang, Kai Zeng, Liang Feng, Jiansong Zhang, Tongxuan Liu, Yong Li, Jingren Zhou, Ce Zhang, et al. FleetRec: Large-scale Recommendation Inference on Hybrid GPU-FPGA Clusters. In *KDD'21: Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, pages 3097–3105, Virtual Event, 2021.

[24] Seyed Mehran Kazemi and David Poole. Simple embedding for link prediction in knowledge graphs. *Advances in neural information processing systems*, 31, 2018.

[25] Liu Ke, Udit Gupta, Carole-Jean Wu, Benjamin Youngjae Cho, Mark Hempstead, Brandon Reagen, Xuan Zhang, David Brooks, Vikas Chandra, Utku Diril, Amin Firoozshahian, Kim Hazelwood, Bill Jia, Hsien-Hsin S. Lee, Meng Li, Bert Maher, Dheevatsa Mudigere, Maxim Naumov, Martin Schatz, Mikhail Smelyanskiy, and Xiaodong Wang. RecNMP: Accelerating Personalized Recommendation with Near-Memory Processing. *arXiv:1912.12953 [cs]*, December 2019.

[26] Hyojong Kim, Jaewoong Sim, Prasun Gera, Ramyad Hadidi, and Hyesoon Kim. Batch-aware unified memory management in gpus for irregular workloads. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1357–1370, 2020.

[27] Thomas N Kipf and Max Welling. Semi-supervised Classification with Graph Convolutional Networks, 2017.

[28] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R Tallent, and Kevin J Barker. Evaluating modern GPU interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect. *TPDS'19: IEEE Transactions on Parallel and Distributed Systems*, 31(1):94–110, 2019.

[29] Youjie Li, Amar Phanishayee, Derek Murray, Jakub Tarnawski, and Nam Sung Kim. Harmony: Overcoming the hurdles of gpu memory capacity to train massive dnn models on commodity servers. *Proceedings of the VLDB Endowment*, 15(11):2747–2760, 2022.

[30] Tianfeng Liu, Yangrui Chen, Dan Li, Chuan Wu, Yibo Zhu, Jun He, Yanghua Peng, Hongzheng Chen, Hongzhi Chen, and Chuanxiong Guo. BGL: GPU-Efficient GNN Training by Optimizing Graph Data I/O and Preprocessing. In *NSDI'23: 20th USENIX Symposium on Networked Systems Design and Implementation*, pages 103–118, Boston, MA, 2023.

[31] Kaihao Ma, Xiao Yan, Zhenkun Cai, Yuzhen Huang, Yidi Wu, and James Cheng. FEC: Efficient Deep Recommendation Model Training with Flexible Embedding Communication. *SIGMOD'23: Proceedings of the ACM on Management of Data*, 1(2):1–21, 2023.

[32] Xupeng Miao, Hailin Zhang, Yining Shi, Xiaonan Nie, Zhi Yang, Yangyu Tao, and Bin Cui. HET: Scaling out Huge Embedding Model Training via Cache-enabled Distributed Framework. *VLDB'22: Proc. VLDB Endow.*, 15(2):312–320, 2022.

[33] Seung Won Min, Kun Wu, Sitao Huang, Mert Hidayetoğlu, Jinjun Xiong, Eiman Ebrahimi, Deming Chen, and Wen-mei Hwu. Large Graph Convolutional Network Training with GPU-oriented Data Communication Architecture, 2021.

[34] Moohyeon Nam, Hokeun Cha, Young ri Choi, Sam H. Noh, and Beomseok Nam. Write-optimized Dynamic Hashing for Persistent Memory. In *FAST'19: 17th USENIX Conference on File and Storage Technologies*, pages 31–44, Boston, MA, February 2019.

[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. Deep Learning Recommendation Model for Personalization and Recommendation Systems, 2019.

[36] Geet Sethi, Bilge Acun, Niket Agarwal, Christos Kozyrakis, Caroline Trippel, and Carole-Jean Wu. RecShard: Statistical Feature-based Memory Optimization for Industry-scale Neural Recommendation. In *ASPLOS'22: Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 344–358, Lausanne, Switzerland, 2022.

[37] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. FlexGen: High-throughput Generative Inference of Large Language Models with a Single GPU. In *International Conference on Machine Learning*, pages 31094–31116, New Orleans, LA, 2023. PMLR.

[38] Xiaoniu Song, Yiwen Zhang, Rong Chen, and Haibo Chen. UGache: A Unified GPU Cache for Embedding-based Deep Learning. In *SOSP'23: Proceedings of the 29th Symposium on Operating Systems Principles*, pages 627–641, Banff, Canada, 2023.

[39] Yixin Song, Zeyu Mi, Haotong Xie, and Haibo Chen. PowerInfer: Fast Large Language Model Serving with a Consumer-grade GPU, 2023.

[40] Wenbo Su, Yuanxing Zhang, Yufeng Cai, Kaixu Ren, Pengjie Wang, Huimin Yi, Yue Song, Jing Chen, Hongbo Deng, Jian Xu, et al. GBA: A Tuning-free Approach To Switch Between Synchronous and Asynchronous Training for Recommendation Models. *NeurIPS'22: Advances in Neural Information Processing Systems*, 35:29525–29537, 2022.

[41] Zhiqing Sun, Zhi-Hong Deng, Jian-Yun Nie, and Jian Tang. Rotate: Knowledge Graph Embedding by Relational Rotation in Complex Space, 2019.

[42] Théo Trouillon, Johannes Welbl, Sebastian Riedel, Éric Gaussier, and Guillaume Bouchard. Complex Embeddings for Simple Link Prediction.

In *ICML'16: International conference on machine learning*, pages 2071–2080, New York, NY, 2016. PMLR.

[43] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph Attention Networks, 2018.

[44] Ruoxi Wang, Bin Fu, Gang Fu, and Mingliang Wang. Deep & Cross Network for Ad Click Predictions. In *Proceedings of the ADKDD'17*, pages 1–7. 2017.

[45] Yingcan Wei, Matthias Langer, Fan Yu, Minseok Lee, Jie Liu, Ji Shi, and Zehuan Wang. A GPU-specialized Inference Parameter Server for Large-scale Deep Recommendation Models. In *RecSys'22: Proceedings of the 16th ACM Conference on Recommender Systems*, pages 408–419, Seattle, WA, 2022.

[46] Minhui Xie, Youyou Lu, Jiazhen Lin, Qing Wang, Jian Gao, Kai Ren, and Jiwu Shu. Fleche: An Efficient GPU Embedding Cache for Personalized Recommendations. In *EuroSys'22: Proceedings of the Seventeenth European Conference on Computer Systems*, pages 402–416, Rennes, France, 2022.

[47] Minhui Xie, Youyou Lu, Qing Wang, Yangyang Feng, Jiaqiang Liu, Kai Ren, and Jiwu Shu. PetPS: Supporting Huge Embedding Models with Persistent Memory. *VLDB'23: Proceedings of the VLDB Endowment*, 16(5):1013–1022, 2023.

[48] Minhui Xie, Kai Ren, Youyou Lu, Guangxu Yang, Qingxing Xu, Bihai Wu, Jiazhen Lin, Hongbo Ao, Wanhong Xu, and Jiwu Shu. Kraken: Memory-efficient Continual Learning for Large-scale Real-time Recommendations. In *SC'20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–17, Virtual Event, 2020. IEEE.

[49] Eric P Xing, Qirong Ho, Wei Dai, Jin-Kyu Kim, Jinliang Wei, Seunghak Lee, Xun Zheng, Pengtao Xie, Abhimanu Kumar, and Yaoliang Yu. Petuum: A New Platform for Distributed Machine Learning on Big Data. In *KDD'15: Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1335–1344, Sydney, Australia, 2015.

[50] Bishan Yang, Wen tau Yih, Xiaodong He, Jianfeng Gao, and Li Deng. Embedding Entities and Relations for Learning and Inference in Knowledge Bases, 2015.

[51] Dongxu Yang, Junhong Liu, Jiaxing Qi, and Junjie Lai. Wholegraph: A Fast Graph Neural Network Training Framework with Multi-GPU Distributed Shared Memory Architecture. In *SC'22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, New Orleans, LA, 2022. IEEE.

[52] 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'22: Proceedings of the Seventeenth European Conference on Computer Systems*, pages 417–434, Rennes, France, 2022.

[53] Weijie Zhao, Deping Xie, Ronglai Jia, Yulei Qian, Ruiquan Ding, Mingming Sun, and Ping Li. Distributed Hierarchical GPU Parameter Server for Massive Scale Deep Learning Ads Systems. *MLSys'20: Proceedings of Machine Learning and Systems*, 2:412–428, 2020.

[54] Da Zheng, Xiang Song, Chao Ma, Zeyuan Tan, Zihao Ye, Jin Dong, Hao Xiong, Zheng Zhang, and George Karypis. DGL-KE: Training Knowledge Graph Embeddings At Scale. In *SIGIR'20: Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 739–748, Xi'an, China, 2020.

[55] Guorui Zhou, Na Mou, Ying Fan, Qi Pi, Weijie Bian, Chang Zhou, Xiaoqiang Zhu, and Kun Gai. Deep Interest Evolution Network for Click-through Rate Prediction. In *AAAI'19: Proceedings of the AAAI conference on artificial intelligence*, volume 33, pages 5941–5948, Honolulu, HI, 2019.

[56] Guorui Zhou, Xiaoqiang Zhu, Chenru Song, Ying Fan, Han Zhu, Xiao Ma, Yanghui Yan, Junqi Jin, Han Li, and Kun Gai. Deep Interest Network for Click-through Rate Prediction. In *KDD'18: Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1059–1068, London, UK, 2018. ACM.