FUSEDREC: *Fus*ed Embedding Communication for Distribut*ed Rec*ommendation Training on GPUs

Xuanteng Huang[1], Fan Li[2], Riyang Hu[2], Jianchang Zhang[2], Yuan Peng[2], Yang Zhou[2]
Fangying Chen[2], Xianwei Zhang[1]*
[1]School of Computer Science and Engineering, Sun Yat-sen University, Guangzhou, China
[2]Tencent Inc., Guangzhou, China
huangxt57@mail2.sysu.edu.cn, zhangxw79@mail.sysu.edu.cn

Abstract

Recent years have witnessed the wide adoption of deep learning recommendation models (DLRMs) for many online services. Unlike traditional DNN training, DLRMs leverage massive embeddings to represent sparse features, which are stored in distributed GPUs following the model parallel paradigm. Existing approaches adopt deduplication to eliminate replicated embeddings involved in `AlltoAll` transfers to avoid unnecessary communication. In our practices, we have observed that such a deduplication design exacerbates interconnect inefficiency due to the fragmented embedding transfers with reduced message sizes, hindering the performance of distributed DLRM training.

This paper introduces FUSEDREC, a fused embedding communication and lookup mechanism to tackle the inefficiency due to deduplication. By seeking the opportunities to fuse embeddings from multiple categories into a group, FUSEDREC conducts the communication in a combined shot to alleviate bandwidth under-utilization. Meanwhile, a categorical-aware deduplication algorithm is integrated into FUSEDREC to retain the category information during lookup without extra communication. Combining with efficient recovery procedure, comprehensive results show FUSEDREC achieves a 37.8% throughput speedup in average compared to the SOTA industry implementation, without hurting the recommendation qualities of our in-house models used in online production environments.

1 Introduction

Deep learning recommendation models (DLRMs) have been widely adopted for content recommendation in online platforms of Meta (Firoozshahian et al. 2023; Naumov et al. 2019), Amazon (ama 2025-03-23; Smith and Linden 2017), Alibaba (dee 2025-03-23; Pan et al. 2023) and Tencent (Sima et al. 2022). To cope with the continuously pouring users/items and capture the ever-changing trends, vendors incline to invest a large portion of resources for online DLRM training with timely statistics and samples. For example, Meta spends more than 80% of cycles in their training clusters for DLRMs (Gupta et al. 2020). And there are numerous software frameworks (Wang et al. 2022b; Ivchenko et al. 2022; tfr 2025-03-23) and hardware designs (Firoozshahian et al. 2023; Guo, Hao, and et al. 2023) to expedite the DLRM training.

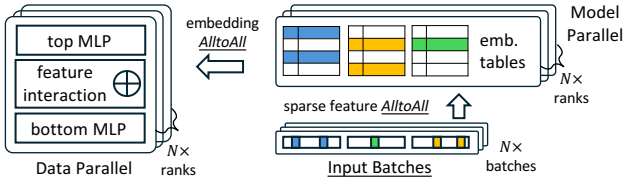

Figure 1: Distributed DLRM training with N workers.

Unlike traditional DNN models receiving dense features in the continuous domain, DLRM accepts vectorized features (i.e., embeddings) indexed by sparse inputs to represent the high-level semantics, e.g., item characteristics and user profiles (Fig. 1) . Hence, DLRM follows the model parallel (MP) paradigm where embedding tables are scattered in multiple distributed GPUs to host the massive embedding entries. During training, each worker (i.e., rank) concurrently receives batches of inputs (i.e., index keys [1]) to retrieve the demanding embeddings from other distributed peers via `AlltoAll`s. In backward, the embedding rows are updated by the sparse gradients communicated by `AlltoAll`s, while the dense parameters (top/bottom MLPs) are synchronized with `AllReduce`.

With large volume of data being transmitted and the number of involved workers increasing (Park et al. 2018; Firoozshahian et al. 2023; Zhang et al. 2022), the embedding lookup stage tends to propose a bottleneck on DLRM training. With our in-house production models, we observe that the embedding lookup process accounts for up-to 60% duration in a training iteration. There are massive explorations trying to speedup and optimize the distributed embedding lookups. Some works (Wang et al. 2022b; Ivchenko et al. 2022; Sethi et al. 2022; Wang et al. 2024; Luo et al. 2024) propose hybrid embedding table parallel via replicating frequently-accessed embeddings in multiple workers based on historical statistics and hardware topologies. On the other hand, there are some approaches aiming to reduce the message sizes involved in the communications for

*Corresponding author.

[1]In this work, we use *keys/tokens* to denote the numeric IDs to index the rows in embedding tables, and we use embedding *rows/entries* interchangeably to represent the trainable vectors with semantic features.

distributed embedding lookup. QuickUpdate (Matam et al. 2024) and AdaEmbed (Lai et al. 2023) selectively update the "important" embeddings having large impacts on model parameters and prune the less crucial embeddings from communications. The adaptive lossy compression mechanism (Feng et al. 2024) is proposed to condense embedding sizes in `AlltoAll` communications based on offline embedding statistics. Additionally, RecD (Zhao et al. 2023), PICASSO (Zhang et al. 2022), EL-Rec (Wang et al. 2022a) and LS-PLM (Gai et al. 2017) target on eliminating the potential duplications of samples within a batch in order to avoid needless embedding exchanges in the distributed lookups. Although the number of embeddings and message sizes per transfer are reduced, these methods lead to **increasingly frequent fragmented communications** as recommendation models tend to incorporate more diverse sparse features, making the interconnect bandwidth under-utilized during communication (§2).

To address this network inefficiency, we propose FUSEDREC, a fused embedding communication and lookup workflow for large-scale distributed DLRM training on GPUs. The key design of FUSEDREC is to seek and fully leverage the opportunities to group the sparse features from multiple categories, and conduct the distributed embedding lookup for the fused embedding groups in a combined manner. FUSEDREC also achieves a lossless lookup with the category level information retained by deferring the hashing the just before the embedding retrieval at the remote side. To accomplish this, FUSEDREC exploits a segment unique operator implemented in GPU to efficiently eliminate the duplicated entries on the fused embedding groups. With the runtime information produced in the forward pass, FUSEDREC recovers and stitches the fused embedding into the dense format for proceeding computation. The contributions of this paper can be summarized as follows:

- With the ample adopted embedding categories, we observe interconnect bandwidth inefficiency due to fragmented communications with reduced message sizes.
- We propose FUSEDREC, a distributed embedding lookup approach which executes the communication and retrieval in semantic-preserving fused manner to better utilize the network bandwidth.
- Extensive experiment results show FUSEDREC can achieve significant end-to-end training throughput speedups without hurting model qualities in our production environments.

2 Motivation

Embedding Deduplication and Network Inefficiency. In DLRM trainings, embedding accesses tend to express locality and skewness. For example, users may watch the same clips with the recently hot topics in video platforms, and people in the same city may share similar interests in some items. Thus, there will be a considerable number of duplicated embedding keys within a batch during training (Liu et al. 2024; Wang et al. 2024; Sethi et al. 2023, 2022; Song et al. 2023; Zhang et al. 2022), leading to skew accesses on the hot embeddings. To avoid unnecessary embedding transfers, existing DLRM frameworks tend to deduplicate the sparse keys by applying the unique operator on the embedding token tensor before distributed lookup. However, we have observed that such a deduplication design exacerbates the interconnect inefficiency (NVLink/InfiniBand). In Fig. 2 (a), we count the message sizes involved in `AlltoAll`s for deduplicated embeddings. It can be captured that the vast majority of embedding message sizes involved in `AlltoAll` is small. In Fig. 2(b), we conduct a performance benchmark for `AlltoAll` communication between 16 GPUs in 2 nodes (hardware specification in §4) to draw the relationship between achieved bandwidth and the message size. We highlight the message size range in Fig. 2(b) where the majority of `AlltoAll` operations in Fig. 2(a) span, indicating such category-wise communications with reduced message sizes can only achieve less than half of the peak bandwidth in distributed environments. The inefficiency comes from two folds: 1) the reduced message size due to the removal of redundant sparse keys within each feature category; b) the feature-wise communication pattern where the `AlltoAll` communication is invoked to lookup embeddings only for one individual sparse feature category.

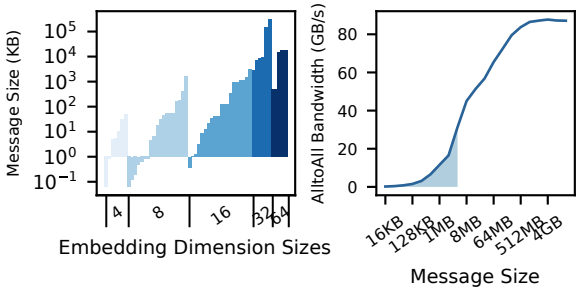

Figure 2: (a) *Left*: The message size involved in `AlltoAll`s with different embedding dimensions. (b) *Right*: The observed `AlltoAll` bandwidth by NCCL test running on 2 nodes each with 8 GPUs.

Category-aware Lossless Lookup. An embedding token is composed of two levels of information: category (i.e., the sparse feature name like *User Profile*) and instance ID (e.g., a specific user with ID *#1234*). For distributed DLRM training jobs in production environments, it is demanding to track and record both the category and instance statistics during embedding lookup, which are used by distributed workers to periodically interact the remote parameter server for access statistics analysis and evict policy decisions. Besides, some DLRMs may refer to embeddings from other models, which also depends both the category and instance information to fetch the external embeddings. With the fused embedding lookup scheme, one has to encode the embedding tokens with categorical information and conduct hash functions to eliminate the intra-category duplications on the fused groups. As the hash is an irreversible operation, the category and instance information is unavailable during the distributed lookup process. Thus, it is demanding to propose a lossless fused embedding lookup scheme with both the category and instance information retained after fusion.

3 Design

Problem Formulation & Overview

We formulate the embedding lookup process of N involved distributed workers (ranks/GPUs) on M sparse categories as follows. For clarity, we exemplify with 1 rank but it applies to all peer ranks. With sparse category i with m_i tokens $\mathbf{e}_i = [e_1, \ldots, e_{m_i}]$ from the input batch, the distributed embedding lookup aims to retrieve embeddings entries $\mathcal{E}_i = [\mathbf{E}_1, \ldots, \mathbf{E}_{m_i}]$ from N distributed workers with `AlltoAll` communications, where $\mathbf{E}_j (1 \leq j \leq m_i)$ may locate in one of the N distributed workers. FUSEDREC integrates a fusion criteria to fuse M embedding categories into K groups and initiates the `AlltoAll` communications at group granularity. Besides, FUSEDREC also proposes a lossless category-aware lookup mechanism by deferring the token hashing after the `AlltoAll` communication in the destination side. Combined with the segment unique operator for grouped token deduplication and efficient embedding format recovery, FUSEDREC ships an end-to-end distributed embedding lookup workflow for DLRM training.

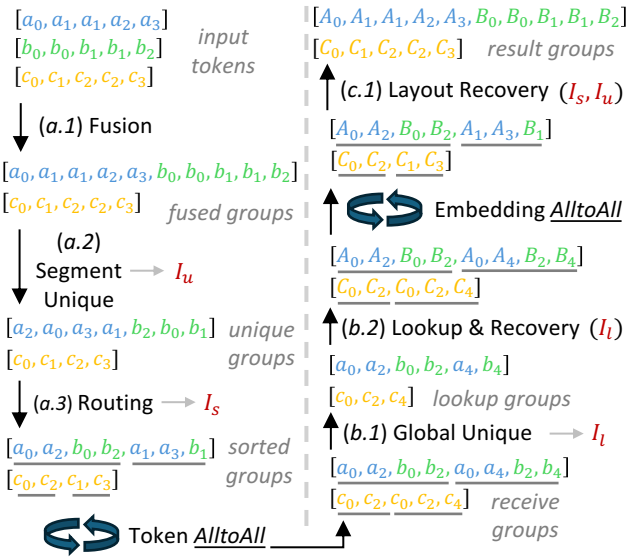

Figure 3: The forward workflow in FUSEDREC, exemplified with rank 0.

Fig. 3 depicts the steps to process the embeddings in the forward pass. In this section, we represent the sparse embedding tokens with lower case letters as the category followed by the numeric instance IDs (e.g., a_0), and the embedding entries (rows) indexed by the tokens are expressed with the corresponding upper case letters (A_0).

Embedding Fusion and Deduplication

Upon receiving the sparse embedding tokens in an input batch, FUSEDREC firstly fuses the applicable keys from M isolated categories into K groups based on their specifications (detailed below), where each group incorporates M_k features and $\sum_{k=1}^{K} M_k = M$. Within each group, there are potential occurrences of *conflicting keys* with the same IDs but originating from different categories (e.g., a_0 and b_0 share the same instance ID 0, but represent different sparse features a and b). Then for each fused group, FUSEDREC initiates the segment unique operator at the group granularity, to remove the redundant keys from the same feature category while allowing conflicting keys to co-exist, meanwhile containing the category and instance statistics for lookup.

Embedding Fusion. Given an embedding $\mathcal{E}_i$ out of the M involved categories, FUSEDREC aims to fuse the keys from categories with the same embedding specifications, which is defined as:

$$\text{spec}(\mathcal{E}_i) = (d_{\mathcal{E}_i}, o_{\mathcal{E}_i}, i_{\mathcal{E}_i}), \tag{1}$$

where $d_{\mathcal{E}_i}$, $o_{\mathcal{E}_i}$ and $i_{\mathcal{E}_i}$ stand for dimension size (e.g., 32, 64), optimizer (e.g., `AdaGrad`, `Adam`), and initializer (e.g., normal or uniform distribution) for $\mathcal{E}_i$. By fusing embedding with the same specification into a group, FUSEDREC could conduct the lookup, gradient and initialization operations for embeddings in a combined shot at the group granularity. For example, as all the embedding entries contained in a group have the same dimension, one can easily store them in a continuous GPU buffer without further complicated management. Given the same dimension and optimizer type, FUSEDREC can invoke common GPU kernels to calculate and accumulate the gradients in the backward pass for all embeddings within the group. Besides, FUSEDREC can also initialize the missing embeddings during lookup (e.g., when new user registers or new video is uploaded) with the same distribution if the embeddings within the group share the same initialization configurations.

In Fig. 3 step (*a.1*), given the embedding keys from 3 categories (a, b, c, and suppose a and b share the same specification), FUSEDREC fuses keys from a and b into a group by directly concatenating them into a flat tensor, and treats keys from c as another standalone group.

Embedding Deduplication. In the fused group, there exist redundant embedding keys within each category, e.g., there are more than one occurrences of a_1, b_0 and b_1 in the fused groups in Fig. 3. Existing unique operators in mainstream DL frameworks (e.g., `torch.unique`) only support to remove the redundant elements at the whole array. In order to eliminate the duplication and avoid needless traffics in communication, one has to either invoke the unique operators M_k times for tokens from each category, resulting in non-negligible runtime overhead due to frequent kernel launches (Pan et al. 2023, 2024) and dynamic memory allocations since the number of unique elements can not be inferred in advance. Or apply the hashing function with both the category and instance ID as the hashing keys to filter out conflicting keys. As hashing operation is irreversible, the category and instance information of each token is unavailable during lookup, making the training system malfunctioned in production environments.

To overcome the defects of both strawman schemes, FUSEDREC is designed to conduct the deduplicate operation on the fused group in a single shot. In Fig. 3 step (*a.2*), we develop a custom *segment unique* operator in GPU (with similar semantic to segment sort (Hou et al. 2017; Kobus et al. 2023)) to remove the redundant keys within the category boundary, while retaining the conflicting keys with

same IDs but from different categories (e.g., a_0 and b_0). Similar to existing unique operators (e.g., `torch.unique`), inverse indices I_u mapping from the keys in fused group to their occurrences in the unique group are also returned by the segment unique operator. We elaborate the implementation details for segment unique later in section 3. After the segment unique, FUSEDREC ensures keys in each *unique group* are distinct, which avoids the unnecessary transfers of duplicated elements.

Embedding Route

In distributed DLRM training, the embedding entries are stored in tables physically located in the GPU memory. Given the tokens in each unique group, FUSEDREC needs to dispatch them into distributed peers to fetch the corresponding embedding entries (Pan et al. 2023; Jain et al. 2024), and then routes them back to the requesting worker via `AlltoAll` operations. The `AlltoAll` interfaces provided by existing implementations (Sergeev and Balso 2018; Romero et al. 2022) requires elements with the same destination to be stored in continuous locations in the input tensor. Thus, FUSEDREC needs to rearrange each unique group to make keys that will be scheduled to the same worker collocated in adjacent positions in the send buffer. In Fig. 3 (*a.3*), FUSEDREC reorders the tokens in each *unique group* by their destinations, such that keys with the same destination are located consecutively, indicated by underlines in the resulting *sorted groups*. In this rankwise sort stage, the destination of each key is determined by the hashing value of its ID in order to keep a balanced distribution within each feature category, i.e., conflicting keys like a_0 and b_0 will be routed to the same worker as they have the same ID (0).

After rankwise sort, the fused keys in sorted groups will be dispatched to different workers via the `AlltoAll` operations (step *b.2* in Fig. 3). Meanwhile, the rankwise sort operator also returns the indices I_s where $I_s[i]$ records the position of `unique_group[i]` in the `sorted_group`. Together with I_u returned by segment unique (§3), both of indices are used by FUSEDREC to stitch and recovery the dense embedding layout.

Lossless Lookup

After the embedding keys `AlltoAll`, a worker receives the query tokens from distributed peer ranks to retrieve the indexed rows stored in its local GPU memory. Unlike traditional strawman fusion scheme (§2) which applies the irreversible hash operation right after fusion, FUSEDREC defers it just before the lookup stage, where both the category and instance information for received tokens are available in step (*b.1*). The training framework could exploit these information for metric reporting and external embedding reference, resulting in a lossless lookup process.

Although the duplication has been eliminated by segment unique locally within each rank (step *a.2*), it is possible for one embedding entry to be retrieved by more than one workers. Thus, there may be redundant keys from different ranks in the receive buffer (marked by different underlines), e.g., a_0, b_2, c_0, c_2 before step *b.1*. As the redundant embedding lookup may introduce significant runtime overhead due to the irregular and repetitive memory accesses (Jain et al. 2024; Pan et al. 2023, 2024; Xie et al. 2025), it is demanding to further eliminate the redundancy within each category to ensure the oneness of tokens in the resulting *lookup groups* (step *b.1*). As the receive groups are in the format numeric instance IDs, it is necessary to hash them together with the category information to distinguish the conflicting tokens across multiple sparse features. In this step, we *stringify* the received tokens by concatenating the instance ID with the embedding category name and generate the *unified query keys*, e.g., a_2 is converted to `'a/2'`. Then we feed the converted strings into a hash function to filter out the redundant elements, which are stored in the *lookup groups* as the query keys for embedding tables. Note that the embedding name strings (`'a'`, `'b'`) are only recorded once during the compute graph construction stage, which are kept unchanged and can be reused in the subsequent training iterations. Meanwhile, the inverse mapping I_l is also returned for subsequent local embedding entry recovery.

Embedding Recovery

After retrieving the embedding entries indexed by the unified query keys from embedding tables in each distributed ranks, FUSEDREC applies the local recovery in the destination side by invoking `gather` operator on the retrieved deduplicated embedding entries lookup indices I_l and produces the embedding groups with the same order as the tokens in receive groups (step *b.2*). Then each involved distributed worker initiates `AlltoAll` communications to transfer the lookup results back to the corresponding requesting peers.

After the embedding `AlltoAll`, the received lookup results in each worker are organized in a deduplicated, fused and reordered layout, as shown in step (*c.1*). For following DNN computations, FUSEDREC has to recover the entries in a format where the duplication is restored and the fused entries are further split as the categorical input for dense MLP layers. Recall that the previously applied segment unique (*a.2*) and rankwise sort (*a.3*) operations both return the inverse indices: I_u and I_s. FUSEDREC leverage these two indices stored locally in each rank to recover the processed entries back to the originally neat layout.

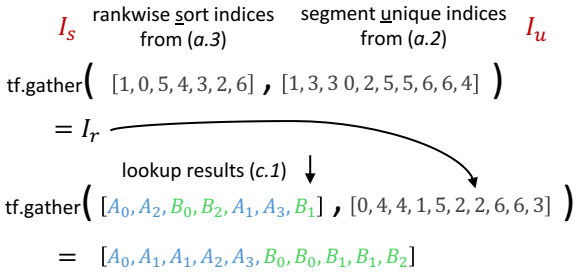

Figure 4: The depiction of embedding recovery (step *c.1* in Fig. 3) after distributed lookup.

Fig. 4 demonstrates the process of embedding entry layout recovery. The indices returned by rankwise sort (I_s)

record the positions mapping from unique groups to the sorted groups. While the indices returned by segment unique (I_u) represent the mapping from keys in the fused group to their occurrences in the unique group, i.e., the ith key in the fused group (step *a.1*) will be placed at position $I_u[i]$ in the unique group after deduplication. With both indices, FUSEDREC then constructs the recover mapping indices (I_r) via the gather operator (e.g., `tf.gather` in TensorFlow). By applying another gather operation on the received lookup results and I_r, FUSEDREC can reproduce the fused entry group without unnecessary intermediate memory copies, where the embeddings from the same category are stored continuously in the same order as the input sparse tokens (step *c.1*). After the second gathering, the produced fused entries are further split into separated categories for following dense model computation.

Putting All Together

Algorithm 1 The fused embedding lookup in FUSEDREC

Input:

N: the number of distributed training workers

$\mathcal{B} = \{\mathbf{e}_1, \ldots, \mathbf{e}_M\}$: embedding keys from M sparse categories in a batch, where $\mathbf{e}_i$ contains all the demanding tokens for the ith category

1: $\mathcal{G} = \text{group}(\mathcal{B})$ ▷ fuse into K groups (step *a.1*)
2: **for** $g \in \mathcal{G}$ **do**
3: Conduct segment unique on g, get unique group g_u (step *a.2*) and indices I_u^g
4: Route embedding to their destinations via rankwise sort (*a.3*), which returns sorted group g_s and indices I_s^g, then route g_s to distributed ranks with `AlltoAll`
5: Apply global unique on the received tokens from peers (*b.1*), and retrieve embedding entries from local tables with the produced unified tokens, followed by the lookup recovery to produce the lookup results G_l (*b.2*)
6: Transfer G_l back to the requesting side, and recover them to the dense layout (*c.1*), based on previously saved I_s^g and I_u^g (Fig. 4)
7: **end for**

Algorithm 1 summarizes the process of embedding group, deduplication, route, lookup and recovery in FUSEDREC. For the batch inputs in each iteration, FUSEDREC firstly merges the sparse embedding keys into fused groups based on their specifications. Then for each fused group g, FUSEDREC conducts the segment unique to eliminate the intra-category redundancy at group granularity, and reorders and routes the unique groups to distributed workers. With deferred hashing, FUSEDREC produces the unified lookup keys after receiving `AlltoAll` tokens from all peers, which are then used to lookup the corresponding embedding entries stored locally in each worker. Finally, FUSEDREC sends the retrieved entries back to the requesting peers with `AlltoAll` communications, and conducts recovery steps with the previously saved runtime information.

For clarity, we only discuss the forward pass for FUSEDREC, the backward acts mirrorily as the forward stage. Since only embeddings with the same specification (§3) can be grouped, it is feasible for FUSEDREC to compute and accumulate the local gradients within each group, and invoke `AlltoAll` operations in a combined manner to update the touched embedding entries in the distributed unified tables, which further helps to reduce the communication and kernel launch overhead.

Implementation

Software. We implement FUSEDREC based on TensorFlow 2.11 (Abadi et al. 2016) and the recommendation add-ons (TFRA (tfr 2025-03-23)) in its graph mode. We leverage Horovod (Sergeev and Balso 2018) backended by NCCL as the library for the communications among distributed peers.

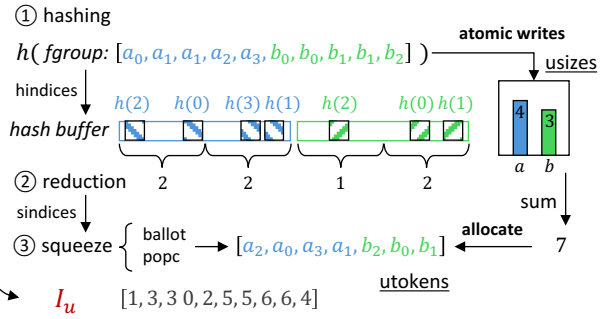

Figure 5: Our custom implementation of segment unique operator used in FUSEDREC which removes the redundant embeddings keys at group level.

Segment unique. With embedding tokens from multiple categories being fused into a group (Fig. 3 step *a.1*), FUSEDREC needs to remove the redundant instance IDs within category boundary. We develop an efficient CUDA implementation for the segment unique operator in GPU to avoid the potential runtime overhead for the per-category deduplication. Fig. 5 presents the segment unique workflow, taking a fused group (*fgroup*) as input, which contains the mixed tokens from multiple sparse feature categories. Given the fused group, FUSEDREC firstly ① allocates a contiguous buffer in GPU memory and hashes the input tokens into this scratch area. Meanwhile, FUSEDREC also counts and updates the number of unique tokens on-the-fly with atomic writes to allocate the output tensor `utokens`. Then FUSEDREC launches ② a warp-level reduction GPU kernel to collect the scattered hashed tokens and ③ squeezes them into a compact format using CUDA warp-level primitives `ballot` and `popc` to produce the output tensor `utokens` incorporating the unique tokens deduplicated within each category. Additionally, FUSEDREC also records two positional mapping `hindices` and `sindices` in steps ① and ③ to generate the inverse indices I_u, which is further exploited to recover the dense embedding layout.

Embedding prefetch and eviction. FUSEDREC is integrated with a similar embedding prefetch design like Bagpipe (Agarwal et al. 2023) and ScratchPipe (Kwon and Rhu 2022) to asynchronously retrieve the touched embeddings in the next few iterations from remote parameter server in ad-

vance, such that the execution of embedding communication and dense MLP calculation can be pipelined.

4 Experiment Setup

Hardware. We evaluate FUSEDREC in a GPU cluster where each node is equipped with 8 NVIDIA H20 GPUs with $96\,\mathrm{GB}$ VRAM capacity connected with $900\,\mathrm{GB/s}$ fast NVLink. And each GPU is attached with a Mellanox ConnectX-7 NIC, providing $400\,\mathrm{Gb/s}$ bidirectional inter-node bandwidth with the support of RDMA. In the following experiments, we scale the number of nodes from 1 to 16 with the number of GPUs from 8 to 128.

Models and Datasets. We evaluate FUSEDREC with 4 representative in-house production models listed in Table 1, which incorporates TB-level embedding entries stored in hierarchical locations: GPU memory, local host memory and remote persistent storage. The training samples are collected from real-time data stream in our platforms with hundreds of millions of daily active users to continuously training the models. We set the batch size as the maximal value to saturate the GPU memory.

Model	Emb. Table (TB)	Model	Emb. Table (TB)
W_0	4.4	W_1	5.6
W_2	8	W_3	12.8

Table 1: Evaluation Model Embedding Table Size.

Baseline. We compare FUSEDREC with `TFRA`[2] , an industry-leading distributed DLRM training framework supporting elastic scaling of dynamic embeddings in GPU storage. We also integrate the prefetch scheme proposed in Bagpipe (Agarwal et al. 2023) and ScratchPipe (Kwon and Rhu 2022) in `TFRA` and FUSEDREC to asynchronously fetch the embedding entries in the next few batches from distributed workers. To validate the effectiveness of FUSEDREC, we also propose a comparable scheme `TFRA-fused` with the same fuse criterion as FUSEDREC with hashing operator in the source side to filter out duplicated tokens the followed by an extra communication to inform the category statistics to the destination side to accomplish the lossless lookup.

5 Results and Analysis

Training Throughput and Scalability

End-to-end throughput. Fig. 6 demonstrates continuously online training throughput speedups of FUSEDREC over `TFRA` and `TFRA-fused`, where FUSEDREC achieves 37.8% throughput elevation over `TFRA`, and 16.7% additional throughput gains over `TFRA-fused`. It can be observed that conducting the embedding lookup in the fused manner brings significant training throughput, while the efficient segment unique implementation further provides performance gains by avoiding repetitive unique kernel launches for multiple fused categories and extra communication for category information during token `AlltoAll`.

[2]https://github.com/tensorflow/recommenders-addons

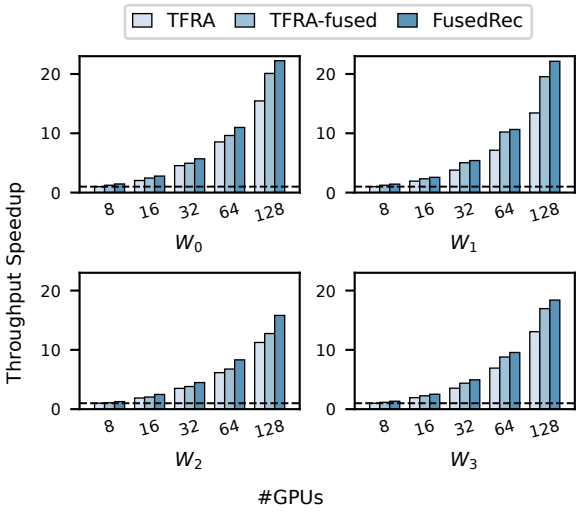

Figure 6: Comparison of the training throughput and scalability with different number of GPUs, which is normalized `TFRA` in single node.

Scalability. FUSEDREC accomplishes a promising scalability with the scaling factor (Zhang et al. 2020) sustained in 0.93 when the number of involved GPUs increases from 8 to 128, which indicates that FUSEDREC could be applied in large scale distributed DLRM training with manageable communication efficiency.

Communication Efficiency

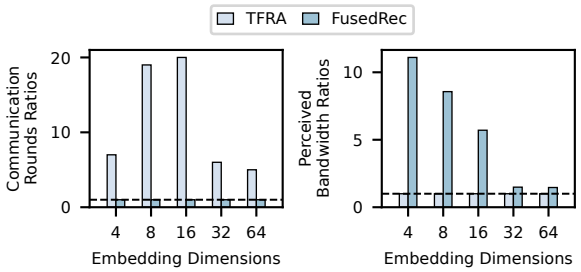

Figure 7: (a) Left: the ratios of `AlltoAll` communication rounds (b) Right: the ratios of perceived interconnection bandwidth for different embedding dimensions in W_0 with 32 GPUs.

Communication rounds. To demonstrate how FUSEDREC reduces the number of `AlltoAll`s required to complete the embedding lookups in distributed training, we count the communication rounds (sending embedding tokens and receiving corresponding entries) in Fig. 7(a), which indicates how many embedding categories with the same specification (equation 1) could be fused into one communication group. In average, FUSEDREC reduces the number of `AlltoAll` rounds by $11.4\times$ for each dimension. And for each `AlltoAll`, FUSEDREC could send/receive $8.16\times$ more embedding entries compared to `TFRA`.

Interconnect bandwidth. We also quantify perceived interconnect bandwidth for communications on embeddings with different dimensions from the execution timeline in NVIDIA *Nsight System* profiler. Fig. 7(b) illustrates the ratios the achieved bandwidth of two schemes, where FUSEDREC possesses $5.66\times$ interconnect bandwidth (NVLink for intra-node and InfiniBand for inter-node communications) compared to TFRA. Besides, it can be noticed that low-dimensional embeddings (i.e., 4, 8, 16) earns higher bandwidth benefits than the high-dimensional ones. Typically, recommendation models tend to employ low-dimension embeddings to represent features with relatively simple semantic information like geographic locations or user genders, while high-dimensional embeddings are used to characterize features with richer semantics like user profiles or video tags. The high-dimensional embeddings are more possible to be unique across samples within a batch, resulting in a relatively larger message size in each transfer compared to low-dimensional embeddings. Therefore, the bandwidth improvements are less pronounced for high-dimensional ones.

Latency Analysis and GPU Utilization

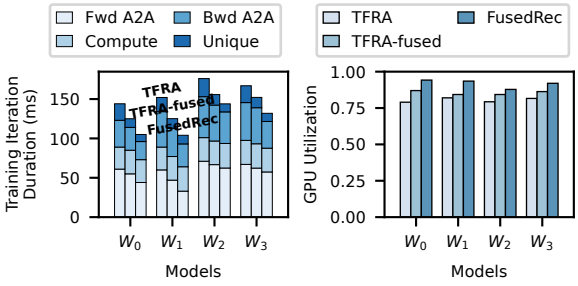

Figure 8: (a) Left: latency breakdown of a training iteration duration for different schemes (b) Right: GPU utilization during training.

Forward and backward `AlltoAll`s. To further investigate the performance improvement, we compare the breakdowns of the duration for TFRA, TFRA-fused and FUSEDREC in a single training iteration in Fig. 8(a). As the computation and communication are already pipelined in both FUSEDREC and TFRA, we only show the exposed part of dense layer computation. FUSEDREC significantly boosts the forward and backward AlltoAll$1.39\times$ compared to TFRA in average. Note that the backward communication follows the similar workflow in the forward embedding exchange, where the computed gradients on the touched embeddings are accumulated locally at first, then routed back to the corresponding ranks in the same fused groups as forward to update the embedding table rows. Overall, FUSEDREC can shorten the latency of one training iteration up-to 20.3% with the fused embedding lookup workflow.

GPU utilization. Fig. 8(b) quantifies ratios of duration in an iteration where the GPU is active for computing as the indicator for hardware utilization. With the improved distributed embedding lookup workflow by FUSEDREC, there is a more compact pipeline to better overlap the dense computation and embedding AlltoAll during training, resulting in a significant 11.4% utilization improvement compared to TFRA.

6 Related Work

Embedding Communication Optimization. To optimize distributed embedding lookup as the bottleneck hindering DLRM training performance, QuickUpdate (Matam et al. 2024) and AdaEmbed (Lai et al. 2023) selectively update the "important" embeddings having large impacts on model parameters and prune the less crucial embeddings from communications. The adaptive lossy compression mechanism (Feng et al. 2024) is proposed to condense embedding sizes in AlltoAll communications based on offline embedding statistics. Additionally, RecD (Zhao et al. 2023), PICASSO (Zhang et al. 2022), EL-Rec (Wang et al. 2022a) and LS-PLM (Gai et al. 2017) target on eliminating the potential duplications of samples within a batch in order to avoid needless embedding exchanges in the distributed lookups. Although the number of embeddings and message sizes per transfer are reduced, these methods lead to increasingly frequent fragmented communications as recommendation models tend to incorporate more diverse sparse features. The fused embedding lookup workflow proposed by FUSEDREC in this work can cooperate with these orthogonal communication optimizations for further embedding lookup improvements.

Embedding Scheduling. With the prior knowledge of the samples in the incoming batches, it is possible to schedule them for the sake of embedding locality. Bagpipe (Agarwal et al. 2023) prefetches the embedding entries from remote PS in advance and holds them in the GPU VRAM if they will be consumed again in the next few steps. Similarly, ScratchPipe (Kwon and Rhu 2022) exploits the samples in future batches to manage a dynamic GPU cache so that the traffic between CPU and GPU can be minimized. Herald (Zeng et al. 2024), permutes the incoming data samples to workers where the used embeddings are resident in the GPU caches to minimize the remote embedding accesses. Although these scheduling mechanisms could bring performance benefits by leveraging the embedding locality, permuting the data samples introduces the risks that the online recommendation quality could be affected (Lai et al. 2023), which is unacceptable for commercial platforms as even 0.01% accuracy downgrade leads to non-negligible revenue loss.

7 Conclusion

In this work, we propose FUSEDREC, a fused embedding lookup and storage workflow for end-to-end distributed DLRM training on GPUs. FUSEDREC fuses applicable embeddings from multiple sparse features into a group, and conducts the embedding lookup in a combined shot. Combined with performant segment unique implementation and efficient embedding layout recovery procedure, FUSEDREC achieves lossless fused embedding lookup without extra communication. Comprehensive experiments show FUSEDREC could bring significant performance improvements for our in-house production models.

Acknowledgments

We sincerely thank the anonymous reviewers for their valuable comments and suggestions. This research is supported by the NSFC grants (62472462, 62461146204).

References

2025-03-23. Amazon Personalize. https://aws.amazon.com/personalize/.

2025-03-23. DeepRec: a high-performance recommendation deep learning framework based on TensorFlow. https://github.com/DeepRec-AI/DeepRec.

2025-03-23. TensorFlow Recommenders Addons. https://github.com/tensorflow/recommenders-addons.

Abadi, M.; Barham, P.; Chen, J.; Chen, Z.; Davis, A.; Dean, J.; Devin, M.; Ghemawat, S.; Irving, G.; Isard, M.; Kudlur, M.; Levenberg, J.; Monga, R.; Moore, S.; Murray, D. G.; Steiner, B.; Tucker, P. A.; Vasudevan, V.; Warden, P.; Wicke, M.; Yu, Y.; and Zheng, X. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation*, 265–283. USENIX Association.

Agarwal, S.; Yan, C.; Zhang, Z.; and Venkataraman, S. 2023. Bagpipe: Accelerating Deep Recommendation Model Training. In *Proceedings of the 29th Symposium on Operating Systems Principles*, 348–363. ACM.

Feng, H.; Zhang, B.; Ye, F.; Si, M.; Chu, C.; Tian, J.; Yin, C.; Deng, S.; Hao, Y.; Balaji, P.; Geng, T.; and Tao, D. 2024. Accelerating Communication in Deep Learning Recommendation Model Training with Dual-Level Adaptive Lossy Compression. In *Proceedings of the International Conference for High Performance Computing*, 89. IEEE.

Firoozshahian, A.; Coburn, J.; Levenstein, R.; Nattoji, R.; Kamath, A.; Wu, O.; Grewal, G.; Aepala, H.; Jakka, B.; Dreyer, B.; Hutchin, A.; Diril, U.; Nair, K.; Ardestani, E. K.; Schatz, M.; Hao, Y.; Komuravelli, R.; Ho, K.; Asal, S. A.; Shajrawi, J.; Quinn, K.; Sreedhara, N.; Kansal, P.; Wei, W.; Jayaraman, D.; Cheng, L.; Chopda, P.; Wang, E.; Bikumandla, A.; Sengottuvel, A. K.; Thottempudi, K.; Narasimha, A.; Dodds, B.; Gao, C.; Zhang, J.; Al-Sanabani, M.; Zehtabioskuie, A.; Fix, J.; Yu, H.; Li, R.; Gondkar, K.; Montgomery, J.; Tsai, M.; Dwarakapuram, S.; Desai, S.; Avidan, N.; Ramani, P.; Narayanan, K.; Mathews, A.; Gopal, S.; Naumov, M.; Rao, V.; Noru, K.; Reddy, H.; Venkatapuram, P.; and Bjorlin, A. 2023. MTIA: First Generation Silicon Targeting Meta's Recommendation Systems. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 80:1–80:13. ACM.

Gai, K.; Zhu, X.; Li, H.; Liu, K.; and Wang, Z. 2017. Learning Piece-wise Linear Models from Large Scale Data for Ad Click Prediction. *CoRR*, abs/1704.05194.

Guo, A.; Hao, Y.; and et al. 2023. Software-Hardware Codesign of Heterogeneous SmartNIC System for Recommendation Models Inference and Training. In *Proceedings of the 37th International Conference on Supercomputing*, 336–347. ACM.

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

Hou, K.; Liu, W.; Wang, H.; and Feng, W. 2017. Fast segmented sort on GPUs. In *Proceedings of the International Conference on Supercomputing*, 12:1–12:10. ACM.

Ivchenko, D.; Staay, D. V. D.; Taylor, C.; Liu, X.; Feng, W.; Kindi, R.; Sudarshan, A.; and Sefati, S. 2022. TorchRec: a PyTorch Domain Library for Recommendation Systems. In *Sixteenth ACM Conference on Recommender Systems*, 482–483. ACM.

Jain, R.; Bhasi, V. M.; Jog, A.; Sivasubramaniam, A.; Kandemir, M. T.; and Das, C. R. 2024. Pushing the Performance Envelope of DNN-based Recommendation Systems Inference on GPUs. In *57th IEEE/ACM International Symposium on Microarchitecture, MICRO Austin, TX, USA*, 1217–1232. IEEE.

Kobus, R.; Nelgen, J.; Henkys, V.; and Schmidt, B. 2023. Faster Segmented Sort on GPUs. In *29th International Conference on Parallel and Distributed Computing*, volume 14100 of *Lecture Notes in Computer Science*, 664–678. Springer.

Kwon, Y.; and Rhu, M. 2022. Training personalized recommendation systems from (GPU) scratch: look forward not backwards. In *The 49th Annual International Symposium on Computer Architecture*, 860–873. ACM.

Lai, F.; Zhang, W.; Liu, R.; Tsai, W.; Wei, X.; Hu, Y.; Devkota, S.; Huang, J.; Park, J.; Liu, X.; Chen, Z.; Wen, E.; Rivera, P.; You, J.; Chen, C. J.; and Chowdhury, M. 2023. AdaEmbed: Adaptive Embedding for Large-Scale Recommendation Models. In *17th USENIX Symposium on Operating Systems Design and Implementation*, 817–831. USENIX Association.

Liu, S.; Zheng, N.; Kang, H.; Simmons, X.; Zhang, J.; Langer, M.; Zhu, W.; Lee, M.; and Wang, Z. 2024. Embedding Optimization for Training Large-scale Deep Learning Recommendation Systems with EMBark. In *Proceedings of the 18th ACM Conference on Recommender Systems, RecSys*, 622–632. ACM.

Luo, L.; Zhang, B.; Tsang, M.; Ma, Y.; Chu, C.; Chen, Y.; Li, S.; Hao, Y.; Zhao, Y.; Lakshminarayanan, G.; Wen, E.; Park, J.; Mudigere, D.; and Naumov, M. 2024. Disaggregated Multi-Tower: Topology-aware Modeling Technique for Efficient Large Scale Recommendation. In *Proceedings of the Seventh Annual Conference on Machine Learning and Systems, MLSys*. mlsys.org.

Matam, K. K.; Ramezani, H.; Wang, F.; Chen, Z.; Dong, Y.; Ding, M.; Zhao, Z.; Zhang, Z.; Wen, E.; and Eisenman, A. 2024. QuickUpdate: a Real-Time Personalization System for Large-Scale Recommendation Models. In *21st USENIX Symposium on Networked Systems Design and Implementation*, 731–744. USENIX Association.

Naumov, M.; Mudigere, D.; Shi, H. M.; Huang, J.; Sundaraman, N.; Park, J.; Wang, X.; Gupta, U.; Wu, C.; Azzolini, A. G.; Dzhulgakov, D.; Mallevich, A.; Cherniavskii, I.; Lu, Y.; Krishnamoorthi, R.; Yu, A.; Kondratenko, V.; Pereira, S.; Chen, X.; Chen, W.; Rao, V.; Jia, B.; Xiong, L.; and Smelyanskiy, M. 2019. Deep Learning Recommendation Model for Personalization and Recommendation Systems. *CoRR*, abs/1906.00091.

Pan, Z.; Zheng, Z.; Zhang, F.; Wu, R.; Liang, H.; Wang, D.; Qiu, X.; Bai, J.; Lin, W.; and Du, X. 2023. RECom: A Compiler Approach to Accelerating Recommendation Model Inference with Massive Embedding Columns. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, 268–286. ACM.

Pan, Z.; Zheng, Z.; Zhang, F.; Xie, B.; Wu, R.; Smith, S.; Liu, C.; Ruwase, O.; Du, X.; and Ding, Y. 2024. RecFlex: Enabling Feature Heterogeneity-Aware Optimization for Deep Recommendation Models with Flexible Schedules. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, 41. IEEE.

Park, J.; Naumov, M.; Basu, P.; Deng, S.; Kalaiah, A.; Khudia, D. S.; Law, J.; Malani, P.; Malevich, A.; Satish, N.; Pino, J. M.; Schatz, M.; Sidorov, A.; Sivakumar, V.; Tulloch, A.; Wang, X.; Wu, Y.; Yuen, H.; Diril, U.; Dzhulgakov, D.; Hazelwood, K. M.; Jia, B.; Jia, Y.; Qiao, L.; Rao, V.; Rotem, N.; Yoo, S.; and Smelyanskiy, M. 2018. Deep Learning Inference in Facebook Data Centers: Characterization, Performance Optimizations and Hardware Implications. *CoRR*, abs/1811.09886.

Romero, J.; Yin, J.; Laanait, N.; Xie, B.; Young, M. T.; Treichler, S.; Starchenko, V.; Borisevich, A. Y.; Sergeev, A.; and Matheson, M. A. 2022. Accelerating Collective Communication in Data Parallel Training across Deep Learning Frameworks. In *19th USENIX Symposium on Networked Systems Design and Implementation*, 1027–1040. USENIX Association.

Sergeev, A.; and Balso, M. D. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. *CoRR*, abs/1802.05799.

Sethi, G.; Acun, B.; Agarwal, N.; Kozyrakis, C.; Trippel, C.; and Wu, C. 2022. RecShard: statistical feature-based memory optimization for industry-scale neural recommendation. In *27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 344–358. ACM.

Sethi, G.; Bhattacharya, P.; Choudhary, D.; Wu, C.; and Kozyrakis, C. 2023. FlexShard: Flexible Sharding for Industry-Scale Sequence Recommendation Models. *CoRR*, abs/2301.02959.

Sima, C.; Fu, Y.; Sit, M.; Guo, L.; Gong, X.; Lin, F.; Wu, J.; Li, Y.; Rong, H.; Aublin, P.; and Mai, L. 2022. Ekko: A Large-Scale Deep Learning Recommender System with Low-Latency Model Update. In *16th USENIX Symposium on Operating Systems Design and Implementation*, 821–839. USENIX Association.

Smith, B.; and Linden, G. 2017. Two decades of recommender systems at Amazon.com. *IEEE Internet Computing*.

Song, X.; Zhang, Y.; Chen, R.; and Chen, H. 2023. UGACHE: A Unified GPU Cache for Embedding-based Deep Learning. In *Proceedings of the 29th Symposium on Operating Systems Principles*, 627–641. ACM.

Wang, Z.; Wang, Y.; Feng, B.; Huang, G.; Mudigere, D.; Muthiah, B.; Li, A.; and Ding, Y. 2024. OPER: Optimality-Guided Embedding Table Parallelization for Large-scale Recommendation Model. In *Proceedings of the USENIX Annual Technical Conference, USENIX ATC*, 667–682. USENIX Association.

Wang, Z.; Wang, Y.; Feng, B.; Mudigere, D.; Muthiah, B.; and Ding, Y. 2022a. EL-Rec: Efficient Large-Scale Recommendation Model Training via Tensor-Train Embedding Table. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 70:1–70:14. IEEE.

Wang, Z.; Wei, Y.; Lee, M.; Langer, M.; Yu, F.; Liu, J.; Liu, S.; Abel, D. G.; Guo, X.; Dong, J.; Shi, J.; and Li, K. 2022b. Merlin HugeCTR: GPU-accelerated Recommender System Training and Inference. In *RecSys: Sixteenth ACM Conference on Recommender Systems, Seattle, WA, USA*, 534–537. ACM.

Xie, M.; Zeng, S.; Guo, H.; Gao, S.; and Lu, Y. 2025. Frugal: Efficient and Economic Embedding Model Training with Commodity GPUs. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, 509–523. ACM.

Zeng, C.; Liao, X.; Cheng, X.; Tian, H.; Wan, X.; Wang, H.; and Chen, K. 2024. Accelerating Neural Recommendation Training with Embedding Scheduling. In *21st USENIX Symposium on Networked Systems Design and Implementation*. USENIX Association.

Zhang, Y.; Chen, L.; Yang, S.; Yuan, M.; Yi, H.; Zhang, J.; Wang, J.; Dong, J.; Xu, Y.; Song, Y.; Li, Y.; Zhang, D.; Lin, W.; Qu, L.; and Zheng, B. 2022. PICASSO: Unleashing the Potential of GPU-centric Training for Wide-and-deep Recommender Systems. In *38th IEEE International Conference on Data Engineering*, 3453–3466. IEEE.

Zhang, Z.; Chang, C.; Lin, H.; Wang, Y.; Arora, R.; and Jin, X. 2020. Is Network the Bottleneck of Distributed Training? In *Proceedings of the Workshop on Network Meets AI & ML, NetAI@SIGCOMM*, 8–13. ACM.

Zhao, M.; Choudhary, D.; Tyagi, D.; Somani, A.; Kaplan, M.; Lin, S.; Pumma, S.; Park, J.; Basant, A.; Agarwal, N.; Wu, C.; and Kozyrakis, C. 2023. RecD: Deduplication for End-to-End Deep Learning Recommendation Model Training Infrastructure. In *Proceedings of the Sixth Conference on Machine Learning and Systems*. mlsys.org.