OPER: Optimality-Guided Embedding Table Parallelization for Large-scale Recommendation Model

Zheng Wang, *University of California, San Diego;* Yuke Wang, Boyuan Feng, and Guyue Huang, *University of California, Santa Barbara;* Dheevatsa Mudigere and Bharath Muthiah, *Meta;* Ang Li, *Pacific Northwest National Laboratory;* Yufei Ding, *University of California, San Diego*

https://www.usenix.org/conference/atc24/presentation/wang

This paper is included in the Proceedings of the 2024 USENIX Annual Technical Conference.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-41-0

Open access to the Proceedings of the 2024 USENIX Annual Technical Conference is sponsored by

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

OPER: Optimality-Guided Embedding Table Parallelization for Large-scale Recommendation Model

Zheng Wang[1], Yuke Wang[2], Boyuan Feng[2], Guyue Huang[2], Dheevatsa Mudigere[3], Bharath Muthiah[3], Ang Li[4], Yufei Ding[1]
[1]*University of California, San Diego* [2]*University of California, Santa Barbara*
[3]*Meta* [4]*Pacific Northwest National Laboratory*

Abstract

The deployment of Deep Learning Recommendation Models (DLRMs) involves the parallelization of extra-large embedding tables (EMTs) on multiple GPUs. Existing works overlook the input-dependent behavior of EMTs and parallelize them in a coarse-grained manner, resulting in unbalanced workload distribution and inter-GPU communication.

To this end, we propose **OPER**, an algorithm-system co-design with **OP**timality-guided **E**mbedding table parallelization for large-scale **R**ecommendation model training and inference. The core idea of OPER is to explore the connection between DLRM inputs and the efficiency of distributed EMTs, aiming to provide a near-optimal parallelization strategy for EMTs. Specifically, we conduct an in-depth analysis of various types of EMTs parallelism and propose a heuristic search algorithm to efficiently approximate an empirically near-optimal EMT parallelization. Furthermore, we implement a distributed shared memory-based system, which supports the lightweight but complex computation and communication pattern of fine-grained EMT parallelization, effectively converting theoretical improvements into real speedups. Extensive evaluation shows that OPER achieves 2.3× and 4.0× speedup on average in training and inference, respectively, over state-of-the-art DLRM frameworks.

1 Introduction

Deep learning recommendation models (DLRMs) play a central role in many industry-scale deep learning applications, such as content recommendation [11, 24], personalized advertisement [10, 18], and rankings [12, 36]. Driven by the sharp increase in user demand, DLRMs have witnessed exponential growth in the number of parameters [7, 25]. Moreover, more than 50% of AI training cycles and about 79% of AI inference cycles in a production-scale data center at Meta are devoted to recommendation models [5, 15].

The key to the stunning algorithmic performance of DLRMs lies in the embedding learning mechanism, which maps sparse categorical input into dense features. The embedding tables (EMTs) of DLRMs are memory-intensive, demanding both high memory capacity and high memory bandwidth. In production-level DLRMs, EMTs may encompass billions of parameters [7], resulting in significant memory consumption [23, 49]. Additionally, the embedding lookup involves a large amount of random memory access, demanding high memory bandwidth [25]. This drives the need for scaling out the training of EMTs on multi-GPU/multi-node platforms that can scale both the memory capacity and the aggregated bandwidth at the same time.

A common practice (e.g., TorchRec [4], HugeCTR [41], and NEO [25]) is to distribute EMTs across multiple GPUs and nodes through a combination of *data parallelism* (where smaller EMTs are duplicated on all GPUs) and *model parallelism* (where larger EMTs are equally sharded into smaller partitions and scattered across multiple GPUs). Such parallelization strategies treat all embeddings of an EMT equally for seeking memory balance while overlooking the access frequency variation among embeddings. Therefore, they suffer from inferior runtime performance due to largely unbalanced computation/communication across GPUs [46, 47]. This drives the need for finding the ideal parallelism for EMTs in DLRMs that comprehensively balances memory consumption, computation, and communication across GPUs.

Despite the recent effort of exploring the optimal parallelism has achieved great success in the context of distributed DNN training [38, 39, 51], these practices could hardly be transferred and applied to EMTs in DLRMs, due to two major reasons. First, the access pattern of EMTs is highly *sparse* and *irregular*. A large EMT may have millions of embedding rows, while only thousands of embeddings will be accessed by the input samples in one training iteration. Additionally, the distribution of the accessed embeddings is irregularly scattered among GPUs. This makes it difficult to achieve a balanced EMT partitioning for parallelism. In contrast, for dense DNN operators (e.g., convolutions), computation intensity is consistent for all parameters. Second, the computation and communication of EMTs are *input-dependent*. Different

input batches may access different embedding rows leading to largely varied amounts of computation and communication workload for the distributed EMTs. In contrast, the computation flow of dense DNN models is abstracted as static computation graphs regardless of their inputs.

To address these challenges, this paper presents both theoretical and practical solutions to gain a firm understanding of the optimal parallelism approach in a typical input-dependent, multi-GPU deep learning workload like the DLRMs. The primary focus of our theoretical investigation is on two research problems. The first research problem focuses on identifying the input information that influences the optimal parallelization decision to minimize the computation and communication of EMTs for multi-GPU DLRM. Our goal is to ascertain whether employing a fine granularity of EMTs, i.e., processing each row of an EMT separately, is indispensable, or if a coarser granularity, treating the entire table as a whole, is already sufficient for optimal placement. Once we establish the preferred granularity, the second research problem involves examining the overhead associated with effectively processing the extensive input information to generate the best optimal parallelism solutions. Given the large number of embedding rows and the numerous potential placements of all embeddings across multiple GPUs, how to efficiently search for the optimal parallelization plan becomes a significant challenge.

From the practical side, it is essential to recognize that the theoretically optimal solution may not necessarily translate to the best performance in practice. Real-world implementations may face various system-level and hardware implications. Factors such as inter-GPU topology, interconnection bandwidth, and communication protocol can significantly impact the performance and effectiveness of the proposed theoretical solutions. For example, existing distributed EMT training systems typically rely on the collective communication APIs (e.g., AlltoAll()) provided by NCCL [27] which can only express certain type of communication pattern. However, in pursuit of optimal EMT parallelization, EMTs are shared, scattered, and duplicated across different GPUs at an extremely fine granularity. This results in complex inter-GPU communication pattern for both embedding lookup and gradient synchronization, which is not efficiently supported by the coarse-grained collective communication available in current systems. This need for system-level optimization to facilitate the novel EMTs parallelization strategy.

To this end, we propose **OPER**, an **OP**timality-guided **E**mbedding table parallelization scheme and end-to-end system implementation for large-scale **R**ecommendation model training and inference. *At the theoretical level,* we address the question of what the optimal parallelization strategy is for large EMTs in DLRMs and propose an efficient algorithm for finding an empirically near-optimal parallelization strategy. To efficiently evaluate the cost of different EMT parallelization strategies, we exploit the per-embedding access frequency from the input dataset to estimate the memory consumption, embedding lookup workload, and embedding communication amount on GPUs for a given parallelization. To effectively approximate optimal EMT parallelization, we adaptively group the embeddings into EMT partitions with balanced size and access frequency to shrink the huge search space of EMT parallelization. Additionally, we design a heuristic hierarchical search algorithm that decomposes the original optimization problem into two separate problems for optimizing model- and data-parallelism and then solve them separately.

At the system implementation level, we explore system support to translate the theoretical saving to real performance speedup. The near-optimal fine-grained EMT parallelization found by our algorithm requires new system support to handle a large number of lightweight EMT computation and communication workloads. First, to facilitate flexible, fine-grained EMT parallelization, OPER maps all embeddings into a distributed shared memory (NVSHMEM [29]), allowing for a unified implementation of distributed embeddings. Second, to mitigate the overhead of remote embedding fetching and sparse gradient synchronization, OPER introduces a series of system-level optimizations that not only enable the overlapping of embedding communication and computation, but also support efficient sparse gradient AllReduce.

Overall, we make the following contributions in this paper:

- We conduct a thorough investigation of how parallelization affects the efficiency of distributed EMTs and provide a formal definition of the optimal EMT parallelization for DLRM training and inference (§3).
- We propose an optimality-guided heuristic algorithm to efficiently search for near-optimal EMT parallelization that comprehensively balances the memory, computation, and embedding communication across GPUs (§4).
- We implement a unified EMT abstraction built on a distributed shared memory architecture to better support the computation and communication pattern of fine-grained EMT parallelization (§5).
- Comprehensive experiments show that OPER outperforms state-of-the-art DLRM frameworks in both training and inference across various datasets.

2 Background and Motivation

In this section, we will provide the background of DLRM and the mainstream parallelization strategy for EMTs. We will then discuss the distinct characteristics of DLRM EMTs and the multi-GPU communication support for distributed EMTs.

2.1 DLRM and EMT Parallelization

DLRMs take user and item data as input and output a prediction of the user's likelihood of clicking on a specific item, called click-through rate (CTR). Figure 1 shows an overview

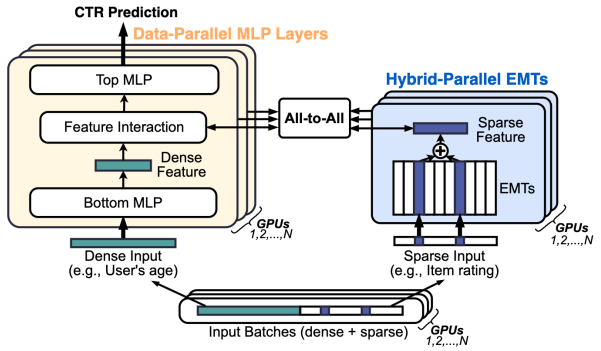


Figure 1: Overview of multi-GPU DLRM training. The MLP layers (bottom MLP and top MLP) are trained with data parallelism, while the EMTs are trained with model parallelism.

of DLRM training on a multi-GPU platform [26,34,41,46,47]. DLRMs are typically composed of two main modules: MLP layers (bottom MLP and top MLP) and EMTs that are responsible for processing dense inputs and sparse inputs, respectively. As shown in Figure 1, the compute-intensive MLP layers are replicated on all GPUs for data parallelism. For the memory-intensive EMTs, state-of-the-art works [4, 25] combine multiple types of parallelism to jointly optimize the performance of EMT operators. Several different parallelism approaches (e.g., table-wise, row-wise) have been proposed in previous works [25,26]. These approaches can be broadly classified into two main categories:

Model Parallelism: The EMTs are typically divided into multiple subsets and then distributed across multiple GPUs. There are different ways to divide the EMTs, which can lead to different sub-types of model parallelism. For example, table-wise parallelism [26] treats the EMT holistically, allocating an equal number of EMTs to each GPU, while row-wise parallelism [25] equally partitions large EMTs into small EMT shards and scatters different shards to different GPUs. In addition to employing different types of model parallelism, another dimension for optimization is how specific parallelism is applied. For example, an EMT can be partitioned into different numbers of EMT shards when performing row-wise parallelism. These two optimization dimensions lead to a very large design space, making it challenging to determine the optimal model parallelism strategy for EMTs.

Data Parallelism: In data parallelism, the EMTs are duplicated to several or all GPUs [25]. This can mitigate the all-to-all communication for fetching embeddings in the forward pass, but it also introduces additional gradient synchronization (AllReduce [27]) overhead in the backward pass. Thus, to improve overall performance through data parallelism, the choice of embeddings for data parallelism must be carefully considered. However, how to select the right embeddings for data parallelism is still an open problem that has not been fully addressed in previous research.

2.2 Multi-GPU Communication Support

Collective Communication: The parallelization of EMTs requires inter-GPU communication to fetch embeddings from remote GPUs and synchronize the gradients. Existing DLRM frameworks [4, 25, 41] leverage the collective communication APIs (e.g., AlltoAll, ReduceScatter, AllReduce) provided by NCCL [27] to implement the communication of distributed EMTs. Although the collective communication APIs of NCCL are well-optimized for throughput, they exhibit some limitations when applied to EMT parallelization tasks. Different EMT parallelism leads to different communication patterns, each typically requiring a unique embedding lookup kernel and its own corresponding collective communication call (e.g., row-wise parallel needs `ReduceScatter()`, table-wise parallel needs `AlltoAll()`) [25]. This necessitates multiple launches of both kernels and communication calls, leading to significant kernel launching and communication initialization overhead when more complex parallelization strategies are applied.

Distributed Shared Memory: Another more promising approach to support the communication of distributed EMTs is to use a *distributed shared memory architecture* [8, 31]. Many libraries provide interfaces for this shared memory abstraction, like OpenSHMEM [8] and NVSHMEM [29]. It offers a global memory address space used by all GPUs and features one-sided communication primitives that can be initialized inside GPU kernel functions [9,40]. These one-sided P2P communication primitives are particularly well-suited for the intensive, lightweight inter-GPU communication introduced by fine-grained EMT parallelization. Therefore, OPER uses NVSHMEM as its primary communication backend.

3 Theoretical Modeling for Parallelization

This section provides theoretical modeling for EMT parallelization problem and gives a formal definition of the optimal EMT parallelization by addressing the following research questions: **(1)** How does EMT parallelization influence the lookup workload distribution and communication (§3.1)? **(2)** How can we formulate EMT parallelization and model its impact on efficiency (§3.2)? **(3)** What is the complexity and scope of the EMT parallelization problem (§3.3)?

3.1 Impact of Parallelization on Efficiency

To demonstrate how EMT parallelization influences the lookup workload distribution and inter-GPU communication amount, we give several examples in Figure 2. Assuming we have three EMTs (labeled with different colors) that need to be placed on two GPUs, each with limited memory for 7 embeddings. In distributed training, the dataset is usually randomly divided and assigned to different devices. So in this example, we assume that the input batches on both GPUs have

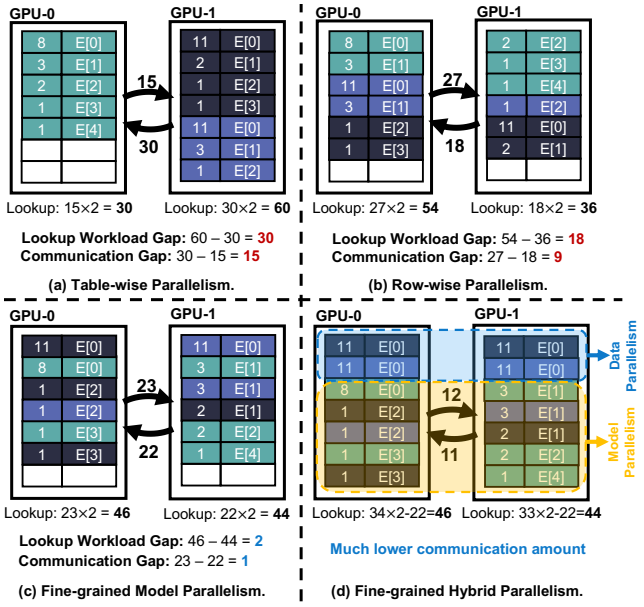


Figure 2: An example illustrating the complexity of the EMT placement problem. We compare four different parallelization strategies. Different color means different EMTs. The number next to the embedding indicates the average access count. The number on the arrows measures the communication amount.

the same number of access to embeddings, and the average access count of each embedding is given in the first column of the EMTs. Then, the amount of embedding communication equals the sum of the access count and the lookup workload could be computed by doubling the total access count since each GPU also needs to lookup embedding for another GPU.

As shown in Figure 2 (a)(b), table-wise parallelization [26] and row-wise parallelization both exhibit significant asymmetrical communication and a large gap in lookup workload. The main reason is that table-wise and row-wise parallelization are both too coarse-grained to handle the highly skewed embedding access frequency. To overcome this issue, we leverage a more fine-grained parallelization. As shown in Figure 2 (c), we partition the EMTs at the embedding level. Such fine-grained parallelization achieves both lookup and communication balance. In Figure 2 (d), we utilize the extra memory space on GPUs by duplicating the frequently accessed embeddings through data parallelism, which significantly reduces the amount of embedding communication required. This simple example demonstrates that the parallelization of EMTs greatly influences the efficiency of distributed EMTs. It also conveys that both fine-grained and hybrid parallelism are necessary for optimal EMT parallelization.

3.2 Formulating the Parallelization Problem

To find the optimal parallelization, this section provides concrete theoretical modeling that describes the relationship between EMT parallelization and memory consummation, lookup workload, and inter-GPU communication.

Given a set of embeddings $E = \{e_1, e_2, \cdots, e_N\}$, and a multi-GPU platform that consists of M GPUs, there are two things that should be determined by an EMT parallelization: 1) where to store a specific embedding; 2) where to get an embedding when this embedding is not in the local GPU memory (an embedding may be stored on several GPUs). To describe the EMT parallelization, for each embedding e_k, we use a $M \times M$ binary matrix P_k to not only represent the placement of e_k but also indicate the embedding fetch path of e_k. Specifically, $P_k[i][j] = 1$ indicates that GPU-i will fetch embedding e_k from GPU-j during training and inference. Additionally, we use a matrix L, which is also $M \times M$, to record the per-embedding lookup and communication cost. This cost can be obtained by conducting offline profiling on the hardware. Specifically, $L[i][j]$ represents the average cost of GPU-i fetching one embedding from GPU-j. When $i = j$, it measures the average cost of looking up the one embedding that is locally owned. For an arbitrary EMT parallelization $\mathcal{P} = \{P_1, P_2, \cdots, P_N\}$, the memory consumption, embedding lookup workload, and inter-GPU communication on GPUs can be computed as:

Memory Consumption: If $\sum_{i=0}^{M-1} P_k[i][j] > 0$, it means that GPU-$j$ owns a copy of embedding e_k. Such that, the memory consumption of embeddings on GPU-j could be obtained by:

$$M_j = \sum_{k=1}^{N} (S_k \cdot \mathbf{H}(\sum_{i=0}^{M-1} P_k[i][j])) \tag{1}$$

in which S_k is the footprint of embedding e_k and $\mathbf{H}()$ is the Heaviside step function whose output is 1 if the input is larger than 0, otherwise the output would be 0.

Embedding Lookup Workload: $P_k[i][j] > 0$ means GPU-i need to fetch embedding e_k from GPU-j. In other word, the lookup of embedding e_k for GPU-i takes place on GPU-j. Such that we can obtain the overall embedding lookup latency on an arbitrary GPU-j by:

$$E_j = \sum_{k=1}^{N} (S_k \cdot A_k \cdot \sum_{i=0}^{M-1} P_k[i][j]) \cdot L[j][j] \tag{2}$$

where A_k is the average access count among GPUs of embedding e_k. A_k can be obtained by profiling the training dataset.

Embedding Communication: As mentioned before, $P_k[i][j] > 0$ indicates that GPU-i will fetch embedding e_k from GPU-j. Thus, the communication latency from GPU-j to GPU-i can be calculated by:

$$C_{i,j} = \sum_{k=1}^{N} (S_k \cdot A_k \cdot P_k[i][j]) \cdot L[i][j] \tag{3}$$

here we only consider the embedding communication amount in the forward pass. We discuss the gradient communication amount in the DLRM training in §4.3.

3.3 Complexity of Parallelization Problem

Based on the theoretical modeling presented in the last subsection, the problem of finding the optimal parallelization can be formulated as a constrained optimization problem like mixed-integer linear programming (MILP). The optimal EMT parallelization can be obtained by traversing all valid parallelizations $\mathcal{P}$, ensuring that the memory consumption computed by Equation 1 does not exceed the memory capacity of GPUs, with the goal of minimizing the lookup workload and communication overhead computed by Equation 2 and 3. However, such a brute-force approach is infeasible due to the extra large search space. For each embedding, it has a $M \times M$ binary matrix. Such that, we have $N \times M^2$ binary variables in total, and the search space contains 2^{NM^2} possible solution. N is on the order of billions and M is on the order of hundreds in the industry scale DLRM. It is challenging to search for the optimal inside such a huge search space, even using the commercial MILP solver.

Facing such a large search space, we wonder if there exists an algorithm that can solve the EMT parallelization problem efficiently rather than iterating through all possible solutions. Unfortunately, the EMT parallelization problem is an NP-hard problem, which means no polynomial algorithm exists for computing optimal EMT placement unless NP=P. To prove the NP-hardness of EMT, we start from the simplest case of the EMT parallelization problem: place EMTs on two GPUs with equal memory capacity and the total memory capacity can just fit all embeddings. Since there is no extra memory space, there is no need to consider data parallelism. Such that, the simplest case of the EMT parallelization problem is equivalent to a classic NP-complete problem, equal-cardinality PARTITION [17]. This indicates that the EMT parallelization problem is at least as difficult as the equal-cardinality PARTITION problem which guarantees the NP-hardness. Given the proven NP-hardness, the polynomial-time solutions are unlikely to exist. This calls for an efficient heuristic approach (§4) to approximate the optimal EMT parallelization.

4 Approximating Near-Optimal EMT Parallelization

In this section, we will detail our optimality-guided heuristic algorithm design for searching for an empirically near-optimal EMT parallelization for DLRM training and inference. As discussed in the last section, the heterogeneity in EMT size and embedding access frequency makes it difficult to achieve balance in all three aspects based on a coarse-grained EMT partition. Meanwhile, the NP-hard complexity and a large number of embeddings make it impractical to perform a fine-grained parallelization at the embedding level. To overcome this issue, we propose to partition the EMTs adaptively and find the "sweet point" between parallelization granularity and optimality.

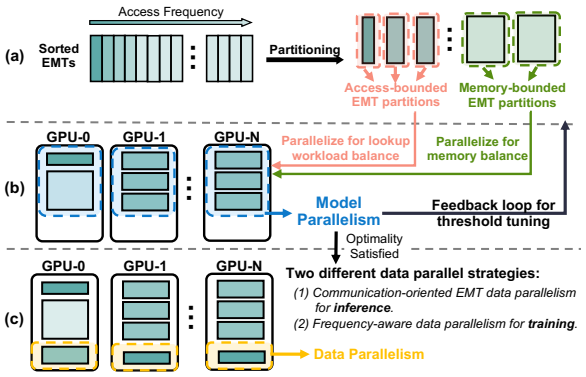


Figure 3: Three steps of our optimality-guided heuristic algorithm for optimal EMT parallelization: (a) Access- and memory-aware EMT partitioning to shrink the large search space; (b) Backtrack searching for balanced EMT model parallelism and partitioning threshold selection; (c) Minimizing inter-GPU communication through EMT data parallelism.

4.1 Access- and Memory-aware Partitioning

Due to the highly skewed access pattern of EMTs, there is a trade-off between placement balance and search complexity. The key tuning knob is partition granularity. Finer-grained partitions could yield better parallelization, but also increase the complexity of the problem. Thus, we propose an *Access- and Memory-aware Adaptive EMT Partition* method to divide the EMTs into smaller partitions that have balanced memory footprint and access frequency. The key insight of our method is the adaptive division of EMTs, taking into account both memory and access frequency, rather than equally dividing EMTs along one dimension.

As shown in Figure 3(a), we first sort the embeddings based on the average access frequency on all GPUs. And then we go through all the embeddings to divide them into small partitions. Here, we introduce a hyperparameter *threshold* to control the partition granularity. The *threshold* constrains the maximum footprint and access frequency of an EMT partition. For example, the EMT partitions marked with red boxes in Figure 3(a) are bound by access frequency, indicating that this partition's access frequency exceeds the *threshold*. Similarly, the EMT partitions marked with green boxes are bound by the memory constraint. This approach helps achieve a balanced partition size and access frequency, which is crucial for finding a near-optimal parallelization.

4.2 Algorithm for Balanced Model Parallelism

In this step, we tried to find an optimal model parallelism plan for EMTs. The primary goal of EMT model parallelism is to balance the memory consumption and embedding lookup workloads on GPUs. To approximate the optimal solution for this two-objective optimization problem, we propose sepa-

rating all the EMT partitions into two distinct types: access-bounded partitions and memory-bounded partitions. We then parallelize these partitions in a greedy way with the access-balance objective and memory-balance objective, respectively. The key idea is to leverage the highly skewed access pattern of DLRM EMTs. Our profiling results show that access-bounded partitions account for over 90% of the total embedding access but less than 5 of the EMT footprint. This means that parallelizing access-bounded partitions significantly affects the embedding lookup workload balance, but the impact on memory balance is relatively minor.

Based on this observation, we first parallelize the access-bounded EMT partitions to balance the embedding lookup workload. We do this iteratively by assigning partitions to the GPU with the lowest embedding lookup workload E_i, computed using Equation 2. Then, we parallelize the memory-bounded partitions in a similar manner to balance memory consumption. This is achieved by assigning partitions to the GPU with the lowest memory consumption M_i, computed using Equation 1. After obtaining the model parallelization plan, its optimality can be evaluated by calculating the degree of balance of memory consumption and embedding lookup workload. If the degree of balance does not meet the expectations, it indicates that the threshold chosen for EMT partitioning (§4.1) was not small enough. In this case, we will re-generate new partitions and a model parallelism plan with a smaller threshold until we obtain a near-optimal parallelization plan that satisfies the balance requirements.

4.3 EMT Data Parallelism Tailored for Inference and Training

As shown in Figure 3 (3), in this step, we exploit EMT data parallelism by duplicating the EMT partitions to multiple GPUs. To maximize communication reduction, we design different parallelization approaches for training and inference, as they have distinct communication patterns.

Communication-oriented EMT data parallelism for inference: In inference, we only need to focus on reducing the amount of communication between each GPU, without worrying about the AllReduce communication for gradients. Therefore, we design a communication-oriented EMT data parallelism algorithm for inference. As shown in Algorithm 1, in each iteration, we first identify the target GPU with the highest communication cost (Line 4). We then examine all EMT partitions placed on the target GPU (Line 6) and attempt to find a partition that can be accommodated within the local GPU memory (Line 7). If a suitable partition is found, we update the EMT partition's communication path and modify the communication latency between GPUs accordingly (Lines 9-12). To maximize communication latency reduction, it is crucial to assess whether this data parallelism can benefit other GPUs (Lines 13-20). For instance, if an EMT partition is duplicated from another node, all GPUs on the same node could retrieve this EMT partition from the intra-node GPU rather than the original inter-node GPU. It would greatly reduce the traffic along the inter-node interconnection which is the key bottleneck in multi-node DLRMs.

Algorithm 1: EMT Data Parallelism for inference.

```
   input  : #GPU: M, #Partitions: N, EMT Partitions: Par,
            communication latency between GPUs: CL_{M×M},
            available memory on GPUs: Ava_Mem_{1×M},
            per-embedding fetching latency: L_{M×M},
   output : Updated EMT partitions using data parallelism: Par
1  Stop_gpu[M] = {0};
2  while sum(Stop_gpu) < M do
3      for gid in {0,1,...,M} and Stop_gpu[gid] == 0 do
           /* Find the interconnect with highest comm. latency   */
4          tid = argmax(CL[gid]);
5          find_p = 0;
6          for each p in Par and p.comm_path[gid] == tid do
7              if p.mem < Ava_Mem[gid] then
                   /* Data parallel partition p to GPU-gid.      */
8                  find_p = 1;
9                  p.comm_path[gid] = gid;
10                 CL[gid][tid] -= g.access × L[gid][tid];
11                 CL[gid][gid] += g.access × L[gid][gid];
12                 Ava_Mem[gid] -= p.mem;
                   /* Update all GPUs with lower comm. cost.     */
13                 for i in {0,1,...,M} and i != gid do
14                     oid = g.comm_path[i];
15                     if L[i][gid]<L[i][oid] then
16                         g.comm_path[i] = gid;
17                         CL[i][oid] -= g.access × L[i][oid];
18                         CL[i][gid] += g.access × L[i][gid];
19                     end
20                 end
21                 Break;
22             end
23         end
24         if find_p == 0 then
25             Stop_gpu[gid] = 1;
26         end
27     end
28 end
```

Frequency-aware data parallelism for training: If the embedding is duplicated for data parallelism in DLRM training, gradients AllReduce communication is required to make sure the embedding parameter is updated correctly which incurs additional inter-GPU communication. Therefore, it is necessary to take into account the impact of gradient AllReduce communication for distributed EMT training.

Assuming we have an embedding e_k, whose access frequency is f and footprint is S_k. We want to know whether duplicating embedding e_k will reduce the overall communication amount. If the embedding e_k is not duplicated and only be placed on a single GPU, then the average communication latency for e_k in each iteration will be:

$$T_{non-dup} = 2 \cdot f \cdot Batch\_Size \cdot \frac{(M-1)}{M} \cdot S_k \;/\; BW_{P2P} \qquad (4)$$

M is the number of GPUs, $f \cdot Batch\_Size$ measures the average access count for e_k in each batch. The factor $\frac{M-1}{M}$ is multiplied

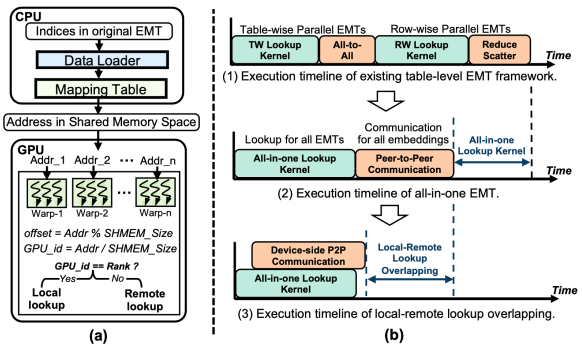


Figure 4: The design of all-in-one EMT abstraction: (a) The workflow of index mapping and decoding of all-in-one EMT; (b) The execution timeline comparison between all-in-one EMT and existing EMT frameworks based on collective communication.

since only the access from other GPUs needs communication. And BW_{P2P} is the bandwidth of P2P communication.

If the embedding e_k is duplicated to all GPUs, there will be no communication in the forward process, the communication only comes from the gradient AllReduce for e_k:

$$T_{dup} = 2 \cdot \frac{M-1}{M} \cdot S_k \;/\; BW_{AR} \tag{5}$$

Here, $2 \cdot \frac{M-1}{M} \cdot S_k$ represents the communication amount of each GPU when using the ring-based AllReduce algorithm. BW_{AR} is the bandwidth of AllReduce. To minimize the communication amount, the embedding e_k should be duplicated only when $T_{dup} < T_{non-dup}$. Solving this equation yields $f > \frac{BW_{P2P}}{Batch\_Size \cdot BW_{AR}}$, which indicates that, for DLRM training, we should only duplicate the embeddings whose access frequency is greater than $\frac{BW_{P2P}}{Batch\_Size \cdot BW_{AR}}$.

5 Distributed Shared Memory-based EMT

As described in previous sections (§2.2), existing EMT training systems typically rely on collective communication APIs, which only support specific types of communication patterns. This limits their capability in representing more complex and fine-grained communication patterns, which are exactly what the theoretically optimal EMT parallelization (§4) introduced. To overcome these challenges, we designed a distributed shared memory-based EMT training system, incorporating an array of system-level optimizations tailored for EMT computation and communication. This effectively translates the theoretical savings into real performance speedup.

5.1 All-in-one EMT Abstraction

We first address the challenge of how to efficiently represent fine-grained EMT parallelization. Existing DLRM frameworks [4, 41] leverage a "high-level" parallelization representation, based on table-level EMT primitives and collective communication APIs. This approach lacks the flexibility to coordinate the parallelization of different embedding rows separately. To overcome this limitation, we propose an *All-in-one EMT Abstraction* that enables "low-level" parallelization representation on a per-embedding basis. Follow the parallelization plan searched by our heuristic algorithm, we map all embeddings into a distributed global memory space that is shared by all GPUs. Here, each GPU allocates an equal amount of consecutive memory within a global virtual memory space. The memory size on each GPU is computed as $SHMEM\_Size = \frac{EMT\_Size}{\#GPU}$. This results in a distributed shared memory space, which can be conceptualized as a large distributed embedding table encompassing all EMTs.

Besides where to place the embeddings, another essential aspect of EMT parallelization is how to communicates embeddings. In our all-in-one EMT abstraction, this information is encoded into the address of the embeddings. As shown in Figure 4 (a), the dataloader first converts EMT indices to addresses in the global shared memory space by looking up a mapping table maintained in host memory. Each address will be mapped to a warp of GPU threads and decoded to obtain the target GPU ID (*GPU_id*) and the offset in the shared memory (*offset*) at runtime. With the target GPU ID and offset information, the embedding lookup kernel can determine whether it needs to initiate a communication request to retrieve the embeddings from remote memory.

In summary, our all-in-one EMT abstraction offers a lightweight and efficient way to represent any EMT parallelization at a per-embedding granularity. The address mapping table is stored in the host memory, ensuring that GPU memory consumption does not increase. Furthermore, the CPU-side index conversion can be seamlessly pipelined with EMT training, resulting no additional overhead.

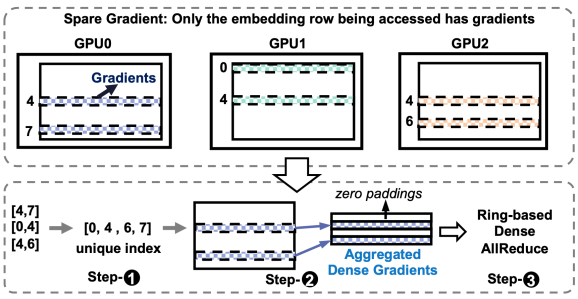


Figure 5: The workflow of local-aggregated sparse AllReduce.

5.2 Local-remote EMT Lookup Overlapping

Due to the limitations of collective communication, existing DLRM frameworks utilize separate embedding lookup kernels and communication APIs to support various EMT parallelizations. Figure 4 (b)(1) illustrates the execution timeline

of the lookup process for a combination of table-wise and row-wise EMT parallelization. Since these two parallelization strategies require different communication patterns, two lookup kernels and two communication calls are invoked to complete the embedding lookup. If we consider more fine-grained and complicated EMTs parallelization, the overhead of kernel launching and communication initialization will become a bottleneck.

To overcome this challenge, we first implement an *All-in-one Lookup Kernel* to perform the lookup of all EMTs within a single kernel. This design is facilitated by our address encoding technique (§5.1), which provides an efficient way to access all necessary information at runtime regarding where the embedding is placed and how to communicate the embedding. Furthermore, we propose a novel *Local-remote Embedding Lookup Overlapping* technique to overlap the computation and communication of embedding lookup. In the distributed shared memory-based architecture, the GPU memories are mapped to the NIC, making GPU memory directly accessible by remote GPU threads without local or remote CPU involvement. When the lookup kernel encounters an embedding stored in remote GPU memory, it can directly initialize a remote memory get request to fetch the embedding from other GPUs, thereby hiding part of the communication latency with the embedding lookup computation. As shown in Figure 4 (b), the local-remote lookup overlapping significantly reduces the overall EMT lookup latency.

5.3 Sparse AllReduce for Data-Parallel EMT

To synchronize gradients, AllReduce communication is required for data-parallel EMTs. Existing methods either treat gradients as dense tensors and use ring-based Dense AllReduce [30], or leverage AllGather-based Sparse AllReduce [32], which involves sending sparse gradients to all GPUs and completing gradient reduction locally on each GPU. However, neither approach considers the sparse and input-dependent nature of data-parallel EMTs, resulting in inferior performance.

To overcome it, we propose a novel *Local-aggregated Sparse AllReduce* to reduce the sparse gradient communication overhead. Our key insight is that some embedding rows are much more popular than others, these popular embeddings may be accessed multiple times in an input batch. We could first aggregate these replicated gradients locally to reduce the amount of communication. As shown in Figure 5, the step-❶ is to find the unique indices from all input batches. Then we aggregate the gradient locally based on the unique indices. If some unique indices are not accessed locally, we perform zero-padding to ensure that the aggregated gradients on GPUs have the same size (step-❷). Once the local aggregation is complete, the aggregated gradients on each GPU can be treated as dense gradients. Then we leverage the ring-based AllReduce primitive (e.g. NCCL) to communicate the dense aggregated gradients(step-❸).

Table 1: Details of Datasets and Model Architecture.

Dataset	Embedding Tables			Model Architecture	
	#Rows	Dim.	Size	Bottom MLP	Top MLP
Avazu	8.9M	16	0.55GB	512-256-64-16	512-256-1
Criteo Kaggle	30.8M	16	1.9GB	512-256-64-16	512-256-1
Criteo Terabyte	242.5M	64	59.2GB	512-256-64-64	512-256-1
Syn_Small	838.9M	64	200GB	/	/
Syn_Large	2.5B	64	600GB	/	/

6 Evaluation

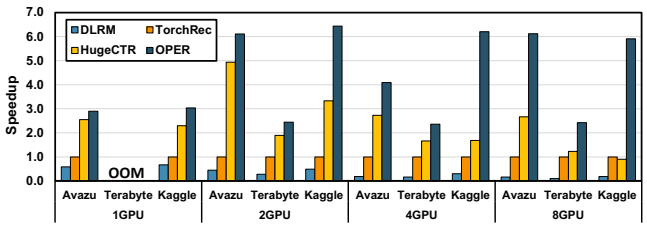
In this section, we provide a comprehensive evaluation of *OPER* in terms of training and inference efficiency. Additionally, we assess the optimality of the parallelization plan generated by *OPER*.

6.1 Experimental Setup

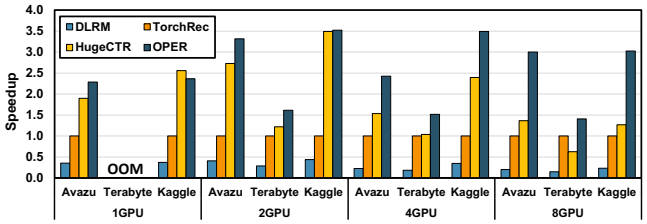
Benchmark and Dataset: We choose three widely adopted real-world DLRM datasets for single-node evaluation. Since the data-scale of these publicly available DLRM datasets is relatively small compared to industry-scale models. We generate two larger datasets that follow a power-law embedding access distribution to validate the performance of *OPER* in a large-scale and multi-node setting: **Avazu** [1] is an open-source dataset of a click-through rate competition. It consists of 11 days of the Avazu users' behavior data. Each sample in Avazu has 20 categorical features and 1 numerical feature. **Criteo Terabyte** [3] is the largest publicly available DLRM dataset. It contains 24 days data records which have over four billion training samples that consist of feature values and click feedback of display ads. Each training sample contains 26 sparse features and 13 numerical features. **Criteo Kaggle** [2] is the dataset for Criteo Kaggle Display Advertising Challenge. It is a subset of Criteo Terabyte which contains the records of Criteo's traffic spanning 7 days. **Syn_Small** and **Syn_Large** are two larger datasets that we generated for multi-node experiments. Both are generated using the data generation script provided by FBGEMM [20]. The details of the datasets (e.g., dimensions, footprints) and the corresponding network architectures can be found in Table 1.

Hyperparameter Setting: For most of the experiments, we set the extra memory ratio for EMTs data parallelism to 5%, which means the total memory space allocated for EMTs is $(1+5\%)\times$ the footprint of EMTs. Both the access- and memory-partitioning thresholds in §4.1 are initially set to 0.1%. We conduct a sensitivity study of the extra memory ratio and partitioning threshold in §6.5 to demonstrate the impact of hyperparameter settings.

Baseline: To demonstrate the advantages of *OPER*, we choose several state-of-the-art DLRM frameworks as baselines for comparison. **DLRM** [26], an open-source recom-

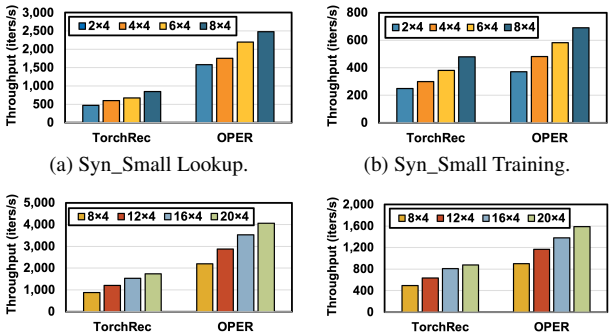


(a) Inference latency comparison.



(b) Training throughput comparison.

Figure 6: End-to-end DLRM training and inference performance on a single node (DGX-A100). Both training and inference speedups are normalized to *TorchRec*.



(a) Syn_Small Lookup.

(b) Syn_Small Training.

(c) Syn_Large Lookup.

(d) Syn_Large Training.

Figure 7: Multi-node embedding table lookup and training scaling test on two large synthetic datasets Syn_Small and Syn_Large ($n \times 4$ means n nodes in total and each node has 4 GPUs).

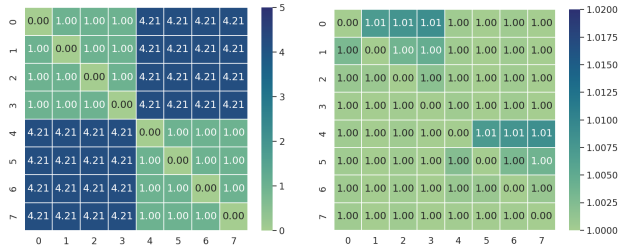
mendation model training framework proposed by Meta. It supports multi-GPU DLRM training with table-wise EMT parallelization which does not further split EMT and place the whole EMT on a single GPU. **TorchRec** [4] was recently released by Meta. It provides the different types of EMT parallelism including table-wise, row-wise, column-wise, and data-parallel for training massive embedding operators in DLRMs. An EMT parallelization strategy will be generated automatically based on the information of the given dataset and hardware. **HugeCTR** [41] is a highly-optimized DLRM training framework proposed by Nvidia. *HugeCTR* offers different implementations of EMTs. In our comparison, we choose LocalizedSlotEmbeddingHash which parallelizes EMTs in a table-wise way, as it achieves higher efficiency in compared with other EMT implementations of *HugeCTR*.

Platform & Tools: We implement *OPER* mostly with C++ and CUDA, and the front-end part of our parallelization search algorithm is implemented with Python to make it more user-friendly. Our single-node experiments are conducted on **Nvidia DGX-A100** [28] which incorporates 8× Nvidia A100 GPUs (40 GB). The GPUs are fully connected through NVSwitch and NVLink with high P2P communication bandwidth. Our multi-node experiments are conducted on a GPU-based HPC cluster. Each node contains 4× Nvidia A100 GPUs (40 GB) with NVLink for intra-node connections and HPE Slingshot network for inter-node connections.

6.2 Overall Performance

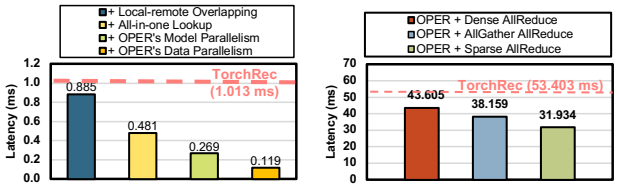
Single-node DLRM inference and training efficiency: In single-node evaluation, we run *OPER* and all baseline works on diverse numbers of GPUs (from 1 to 8) with three real-world DLRM datasets. The end-to-end DLRM inference and training speedup over baseline frameworks has been shown in Figure 6 (a) and (b), respectively. Overall, *OPER* achieves significant improvement over *DLRM* and *TorchRec* in all settings. And *OPER* also beats *HugeCTR* in most cases, typically in multi-GPU settings (e.g., 4 GPUs and 8 GPUs). Compared to *TorchRec*, *OPER* attains 2.4× to 6.4× speedup in inference latency and 1.9× to 3.5× improvement in training throughput. We further observe up to 6.6× speedup in inference and up to 2.4× speedup in training over *HugeCTR*. From the perspective of the number of GPUs, the performance speedup remains low when using only one GPU. This is because, with a single GPU, there is no inter-GPU communication, and the speedup solely comes from our system-level optimization. As the number of GPUs increases, we achieve higher speedup, particularly when compared to *HugeCTR*.

Multi-node scaling test: To validate the effectiveness of *OPER* on large-scale distributed platforms with multiple machines, we compare the embedding lookup and training throughput of *OPER* and *TorchRec* across different numbers of computing nodes using two large synthetic datasets. Figure 7 shows that when the number of nodes increases, *OPER* exhibits a similar scaling trend to *TorchRec*, and the absolute throughput consistently outperforms *TorchRec* in all experimental settings. On average, *OPER* achieves 2.76× and 1.65× improvements in embedding lookup and training throughput, respectively. The speedup in the multi-node setting is relatively lower than that in the single-node experiments. This is because, rather than using collective communication to synchronize all embeddings at once, *OPER* communicates the embeddings in a fine-grained, per-embedding manner that makes it hard to achieve the peak bandwidth of the inter-node connection. However, the speedup we achieved demonstrates that fine-grained parallelization is worthwhile since it can greatly reduce the overall communication volume and achieve better end-to-end performance.



(a) (b)

Figure 8: Adaptability for heterogeneous bandwidths: (a) Normalized per embedding communication latency on 2×4 computing cluster. (b) Normalized embedding communication cost between GPUs (normalized to lowest cost).



(a) (b)

Figure 9: Optimization Analysis: (a) Embedding lookup and communication speedup breakdown on DGX-A100. (b) Per iteration embedding training latency comparison with different AllReduce methods on 4×4 computing cluster.

6.3 Optimization Analysis

Adaptability for heterogeneous bandwidth: The bandwidth of inter-GPU connection may not be identical. Figure 8 (a) shows the normalized per-embedding communication latency on a 2×4 computing cluster. The embedding communication latency exhibits a hierarchical pattern, with inter-node connections having a much higher communication cost than intra-node connections. Our heuristic algorithm design (§4) considers the heterogeneous bandwidth between different interconnections and automatically adjusts the embedding parallelization. Figure 8 (b) shows that the EMT parallelization plan generated by *OPER* achieves balanced embedding communication among GPUs. Despite the per-embedding communication latency between the intra-node and inter-node connection having a gap of about 4.21×, *OPER* achieves about a 1% gap in terms of embedding communication cost. This highlights the effectiveness of *OPER* in handling heterogeneous bandwidths.

Speedup breakdown: To better understand the benefits of different optimizations, we show the speedup breakdown of EMT computation and communication in Figure 9 (a). Here, we evaluate the embedding lookup and the embedding communication latency in the forward pass on DGX-A100 server using the Syn_Small dataset. We start from a baseline implementation of EMT and incrementally add individual optimizations to assess their impact on performance. Overall, *OPER* achieves 8.5× speedup over *TorchRec* in this setting, in which 2.11× speedup from our all-in-one lookup kernel, 1.79× speedup from EMT model parallelization optimization, and 2.26× speedup from EMT data parallelization optimization. These results clearly show the contribution of individual optimizations to the overall performance improvement.

Sparse AllReduce optimization analysis: To demonstrate the effectiveness of our local-aggregated sparse AllReduce design (§5.3), we test *OPER* with two other AllReduce implementations: ring-based dense AllReduce (Dense AllReduce) [30] and AllGather-based sparse AllReduce (AllGather AllReduce) [32]. The experiment used a 4×4 computing cluster and the Syn_Small dataset. As shown in Figure 9 (b), *OPER* with the local-aggregated sparse AllReduce outperforms all other AllReduce implementations. Compared to Dense AllReduce, the local-aggregation method greatly reduces the communication amount for data-parallel EMTs and achieves a 1.37× improvement. Our local-aggregated sparse AllReduce also outperforms the AllGather-based sparse AllReduce. The communication volume of AllGather is proportional to the number of GPUs, making it unsuitable for large-scale DLRM training that requires many of GPUs.

6.4 Placement Optimality Analysis

We evaluate the near-optimality of *OPER*'s EMT parallelization by conducting a comprehensive comparison with two methods from state-of-the-art DLRM frameworks and two greedy strategies specifically optimized for memory and communication balance, respectively.

1. **Memory-Greedy Parallelization (MG)**: only focusing on memory balance which places the equal size of EMTs on each GPU for model parallelism.

2. **Access-Greedy Parallelization (AG)**: starts from the embedding with the highest access frequency and assigns embeddings one by one to the GPUs for model parallelism to balance the embedding access on GPUs.

3. **Table-wise (TW)** [26, 41]: the EMTs are equally scattered across GPUs without further partitioning. The GPUs will have roughly the same number of EMTs.

4. **TorchRec** [4, 25]: the EMT parallelization of *TorchRec* combines different types of EMT parallelism (e.g., table-wise, model-parallel, data-parallel) to balance the memory consumption and lookup workload.

We use the EMT parallelization strategies described above to generate EMT parallelization plans for three real-world DLRM datasets on a DGX-A100 server (8 GPUs). We then evaluate the optimality of the plans in three aspects: **Memory balance**: the gap between the largest and smallest memory

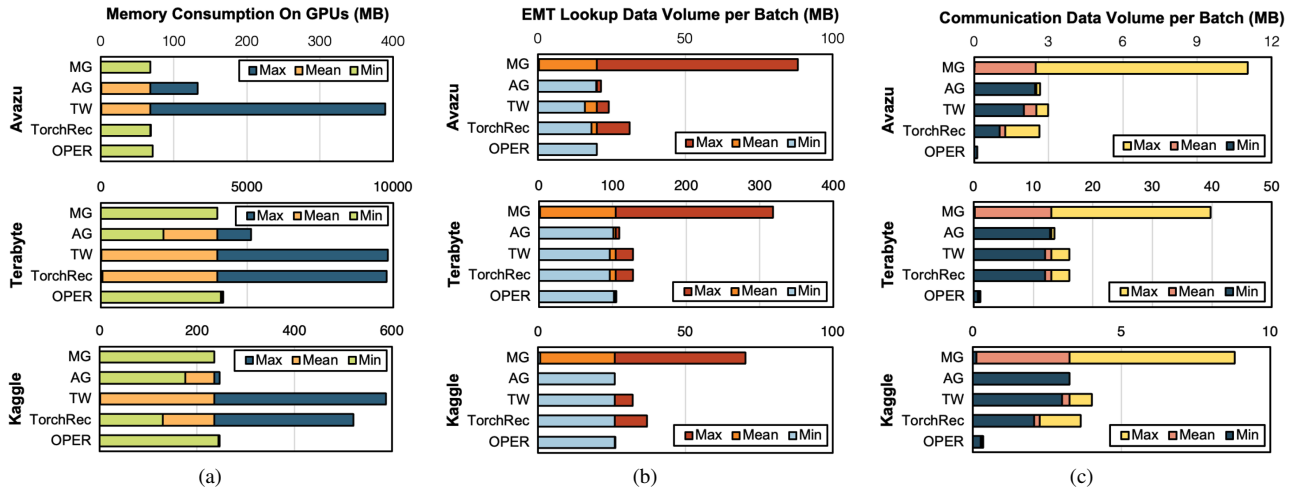


Figure 10: Parallelization optimality analysis: (a) Embedding table memory consumption comparison on GPUs. (b) Embedding Lookup workload comparison on GPUs. (c) Inter-GPU embedding communication amount comparison between GPUs.

consumption for EMTs among all GPUs. **Workload balance**: the gap between the largest and smallest amount of embedding lookup workload among all GPUs. **Communication balance**: the gap between the largest and smallest amount of embedding communication between GPUs.

Memory balance analysis: We compute the footprint of EMTs on GPUs and show the maximum, average, and minimal memory consumption in Figure 10 (a). *Memory-Greedy Parallelization* shows the highest balance in memory consumption. *OPER* also achieves balanced memory consumption, showing our heuristic EMT placement is near-optimal from the memory perspective. It has slightly higher memory use than the *Memory-Greedy Placement* method due to duplicating some embeddings for data parallelism. However, *Access-Greedy Parallelization*, *Table-wise*, and *TorchRec* show a high variance in the memory consumption on different GPUs. This is mainly due to the diversity in EMT sizes, some EMTs are extremely large and some EMTs only contain tens of embeddings. Such significant diversity in EMT size makes it easily fall into memory unbalance.

Workload balance analysis: We analyze the lookup workload by calculating the average data volume of embeddings lookup in each iteration on GPUs. As shown in Figure 10 (b), *Memory-Greedy Parallelization* has large variance in workload distribution due to its focus on memory balance. *Table-wise* and *TorchRec* perform better, but still, show a considerable gap between the maximum and minimal lookup workload which will inevitably increase the overall EMT lookup latency. *Access-Greedy Parallelization* and *OPER* both show near-optimal performance in lookup workload balance. These results demonstrate the effectiveness of our EMT partitioning and parallelization algorithm.

Communication balance analysis: We count the average embedding communication amount among all interconnections in each iteration. Figure 10 (c) shows that *Memory-Greedy Parallelization* has high variance in communication. *Access-Greedy Parallelization*, *Table-wise*, and *TorchRec* show balanced communication, but the average amount of communication still stays high. *OPER* shows not only the lowest variance but also the minimal average communication amount, indicating our frequency-aware data parallelism effectively reduces the total communication amount. Although *TorchRec* also employs data parallelism for small EMTs, it duplicates the entire EMT, which contains many non-popular embeddings, leading to more overall communication.

In conclusion, *OPER* is the only one that achieves near-optimal performance in all three aspects.

6.5 Sensitive Study

In this section, we conduct a sensitive study of data parallelism memory consumption and EMT partition granularity. We also compare our heuristic algorithm with MILP solver to demonstrate the efficiency of *OPER*.

Memory Consumption for Data-parallel EMTs: Although data parallelize EMTs will greatly reduce the embedding communication, they also introduce additional memory consumption. To better understand the memory consumption overhead of data-parallel EMTs, we experiment on all three datasets with different ratios of extra memory space (the ratio between the extra memory space and the total EMT footprint) for data parallelism from 0% to 20%. As shown in Table 2, only 1% extra memory for data parallelism achieves more than 10× communication amount reduction on average. Continue to increase the extra memory space will reduce the communication amount and variance but the effect becomes limited. This is due to the highly skewed access pattern of EMTs. When the popular embeddings are already duplicated,

Table 2: Sensitive study of memory consumption for data-parallel EMTs (**Mean**: average embedding communication amount (MB) in each iteration, **Var.**: the variance of embedding communication among all GPU-GPU interconnections).

Extra Memory Ratio	Avazu		Terabyte		Kaggle	
	Mean	Var.	Mean	Var.	Mean	Var.
0%	2.50	1.4E-1	13.00	8.4E-4	3.25	2.0E-7
1%	0.16	2.0E-5	0.95	4.2E-3	0.46	6.9E-2
2%	0.15	6.4E-5	0.84	1.5E-3	0.30	8.9E-4
5%	0.13	6.1E-5	0.73	1.6E-3	0.24	1.3E-3
10%	0.12	1.0E-4	0.66	1.2E-3	0.20	4.6E-5
15%	0.10	1.3E-4	0.57	2.7E-3	0.19	2.8E-5
20%	0.10	7.3E-5	0.52	5.6E-3	0.18	1.3E-5

increasing the memory will bring limited improvement.

EMT Partition Granularity: We evaluate the impact of partition granularity of our access- and memory-ware EMT partition method on the performance of distributed EMTs. We experiment on the Kaggle dataset, fixing other settings while only adjusting the access and memory threshold (§4.1). As shown in Table 3, a lower threshold leads to less inter-GPU embedding communication and a higher degree of communication balance (DoB). We also tried the commercial mixed-integer linear programming (MILP) solver [16] to solve the EMT parallelization problem. As shown in Table 3, when the partitioning threshold is set to 5%, the solver delivers a better DoB than *OPER*. However, the solving time was extremely high, taking 3.8 hours. With a lower partitioning threshold, the solver could not find a solution within a reasonable time. In contrast, the heuristic algorithm of *OPER* efficiently searched for an empirically near-optimal parallelism plan within several minutes, achieving a 99.1% DoB, which is better than the solver under the 5% partitioning threshold.

7 Related Work

System Design for DLRM: Existing system designs for DLRM largely fall into three categories. The first category is built on the Parameter Server architecture [21, 22] which capitalizes on the host memory for maintaining the embedding parameters of large size [14, 19, 33, 50]. This approach often suffers from CPU-side computation and CPU-GPU communication. The second type of work scaling out the training of EMTs on multi-GPU/multi-node platforms to better utilize the high-bandwidth memory on GPUs [25, 41]. This approach demonstrates higher efficiency compared to other DLRM system designs but still suffers from high embedding communication overhead due to suboptimal EMT parallelization. Another direction of work leverages embedding table compression to decrease the footprint of EMTs. Jie et al. [44] and Hui et al. [13] represent embeddings with fewer bits. TT-Rec [45] and EL-Rec [43] leverages tensor-train decomposition to compress the EMTs. CAFE [48] compresses the embedding tables by sharing embeddings between several non-hot features through hashing. Our proposed method is orthogonal to these embedding compression techniques and can be potentially applied in conjunction with them to better utilize the limited capacity of GPU HBM.

Table 3: Sensitive study of EMT partition granularity (**Thre.**: the access and memory threshold for partitioning; **Max/Min Comm.**: the maximum/minimum communication cost per iteration; **DoB**: degree of balance, defined as $\frac{Min\_Comm.}{Max\_Comm.}$).

Method	Thre.	Time	Max Comm.	Min Comm.	DoB
OPER	5%	15.8 s	48.6	34.8	71.6%
	1%	28.4 s	42.3	33.9	80.1%
	0.5%	41.5 s	38.4	33.9	88.3%
	0.1%	120.3 s	34.1	33.8	99.1%
	0.05%	235.2 s	34.2	33.9	99.1%
Solver	5%	3.8 hr	35.2	32.9	93.5%

Optimization for Embedding Placement and Parallelization: Early works focus on embedding communication overhead between the host memory and GPU. FAE [6] and RecShard [34] split EMTs based on the embedding access frequency and put the popular embeddings on the GPU to reduce the communication between host memory and GPU. The recent trend of DLRM system design is to distribute EMTs to multiple GPUs which incurs the EMT parallelization problem. The complex combination of multiple parallelisms leads to a significantly larger search space for the EMT parallelization problem. Compromising on the complexity of the multi-GPU EMT parallelization problem, exist works [4, 25, 26, 41] usually employ an empirical and coarse-grained way which either places the whole EMT without partitioning [26, 41] or equally splitting the EMT along one dimension [25]. Such that, their parallelization strategy could hardly achieve memory and communication balance simultaneously. UGACHE [37] designs a multi-GPU embedding cache and optimizes the embedding placement in a fine-grained manner. However, it focuses on settings where the EMT is not updated (e.g., GNN training, DLRM inference), which cannot comprehensively address the EMT parallelism problem in both DLRM training and inference. Recent works leverage reinforcement learning (RL) for EMT parallelization [46, 47]. However, these works focus solely on the EMT lookup workload balance through model parallelism, without considering the use of data parallelism for communication optimization.

8 Discussion

Handling Input Distribution Shift: The serving of DLRM inference and DLRM online training [42] continuously incorporates new data. The distribution of new input data may shift over time and may not align with the previous data distribution used for deciding EMT parallelism. The input distribution shift may affect the balance of the EMT parallelism plan and potentially impact the performance of OPER. To address this

issue, OPER can continuously monitor the data distribution at runtime and evaluate the optimality of the existing parallelism using the formulation provided in Section 3.2. If the input distribution shift is significant enough to degrade optimality, OPER can generate a new parallelism plan and redistribute the EMTs accordingly. Thanks to our efficient heuristic algorithm design, the regeneration process takes only a few minutes, minimizing disruption to the system.

Design Choices for EMT CUDA Kernel Implementation: To support fine-grained EMT parallelism, OPER leverages NVSHMEM to map all embeddings onto a distributed shared memory space accessible by all GPUs. Building on top of the distributed shared memory, there are two major design choices for implementing the CUDA kernel for EMT-related operations. Using the embedding table lookup kernel as an example, the first implementation is **Retrieve-based**: each GPU retrieves the required embeddings from remote GPUs and then performs pooling locally. The second implementation is **Send-based**: each GPU conducts embedding pooling according to requests from remote GPUs and then sends the pooled embeddings back to the requesting GPUs. OPER follows the retrieve-based design for kernel implementation. This approach supports more general pooling operations beyond traditional sum-based pooling, such as sequence-based pooling or attention-based pooling [35], which require all embedding vectors to perform the pooling operation.

9 Conclusion

In this work, we propose *OPER*, a systematic framework for accelerating large-scale DLRM training and inference through input-aware fine-grained EMT parallelization. Specifically, *OPER* conducts a thorough investigation of how parallelization affects the efficiency of distributed EMTs, and gives a formal definition of the optimal EMT parallelization. *OPER* designs a heuristic algorithm that can efficiently generate empirically near-optimal EMT parallelization for training and inference which comprehensively balances the memory, computation, and communication across GPUs. *OPER* implements a distributed shared memory-based EMT training system to better support fine-grained parallelization and achieve end-to-end performance improvement. Comprehensive experiments demonstrate that *OPER* outperforms the existing DLRM framework in both training and inference.

10 Acknowledgment

We would like to express our appreciation for the great help and invaluable suggestions from the ATC anonymous reviewers. This work was supported in part by NSF 2124039. Additionally, this research was partially supported by the U.S. DOE Office of Science, Office of Advanced Scientific Computing Research, under award 66150: "CENATE - Center for Advanced Architecture Evaluation". Also, we would like to thank the generous help and support from Meta for their grant in support of UCSB IEE for energy efficiency research.

References

[1] Avazu mobile ads ctr. https://www.kaggle.com/c/avazu-ctr-prediction.

[2] Criteo display ad challenge. https://www.kaggle.com/c/criteodisplay-ad-challenge.

[3] Terabyte click logs. https://labs.criteo.com/2013/12/downloadterabyte-click-logs.

[4] Torchrec. github.com/pytorch/torchrec/, 2022.

[5] Bilge Acun, Matthew Murphy, Xiaodong Wang, Jade Nie, Carole-Jean Wu, and Kim Hazelwood. Understanding training efficiency of deep learning recommendation models at scale. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 802–814. IEEE, 2021. https://doi.org/10.1109/HPCA51647.2021.00072.

[6] Muhammad Adnan, Yassaman Ebrahimzadeh Maboud, Divya Mahajan, and Prashant J Nair. Accelerating recommendation system training by leveraging popular choices. *Proceedings of the VLDB Endowment*, 15(1):127–140, 2021. http://www.vldb.org/pvldb/vol15/p127-mahajan.pdf.

[7] Newsha Ardalani, Carole-Jean Wu, Zeliang Chen, Bhargav Bhushanam, and Adnan Aziz. Understanding scaling laws for recommendation models. *arXiv preprint arXiv:2208.08489*, 2022. https://doi.org/10.48550/arXiv.2208.08489.

[8] Barbara Chapman, Tony Curtis, Swaroop Pophale, Stephen Poole, Jeff Kuehn, Chuck Koelbel, and Lauren Smith. Introducing openshmem: Shmem for the pgas community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, pages 1–3, 2010. https://doi.org/10.1145/2020373.2020375.

[9] Yuxin Chen, Benjamin Brock, Serban Porumbescu, Aydın Buluç, Katherine Yelick, and John D Owens. Scalable irregular parallelism with gpus: getting cpus out of the way. In *2022 SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 708–723. IEEE Computer Society, 2022. https://doi.org/10.1109/SC41404.2022.00055.

[10] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, Rohan Anil, Zakaria Haque, Lichan Hong, Vihan Jain, Xiaobing Liu, and Hemal Shah. Wide & deep learning for recommender systems. In Alexandros Karatzoglou, Balázs Hidasi, Domonkos Tikk, Oren Sar Shalom, Haggai Roitman, Bracha Shapira, and Lior Rokach, editors, *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems, DLRS@RecSys 2016, Boston, MA, USA, September 15, 2016*, pages 7–10. ACM, 2016. https://doi.org/10.1145/2988450.2988454.

[11] Paul Covington, Jay Adams, and Emre Sargin. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM conference on recommender systems*, pages 191–198, 2016. https://doi.org/10.1145/2959100.2959190.

[12] Carlos A Gomez-Uribe and Neil Hunt. The netflix recommender system: Algorithms, business value, and innovation. *ACM Transactions on Management Information Systems (TMIS)*, 6(4):1–19, 2015. https://doi.org/10.1145/2843948.

[13] Hui Guan, Andrey Malevich, Jiyan Yang, Jongsoo Park, and Hector Yuen. Post-training 4-bit quantization on embedding tables. *arXiv preprint arXiv:1911.02079*, 2019. http://arxiv.org/abs/1911.02079.

[14] Huifeng Guo, Wei Guo, Yong Gao, Ruiming Tang, Xiuqiang He, and Wenzhi Liu. Scalefreectr: Mixcache-based distributed training system for ctr models with huge embedding table. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 1269–1278, 2021. https://doi.org/10.1145/3404835.3462976.

[15] Udit Gupta, Carole-Jean Wu, Xiaodong Wang, Maxim Naumov, Brandon Reagen, David Brooks, Bradford Cottel, Kim M. Hazelwood, Mark Hempstead, Bill Jia, Hsien-Hsin S. Lee, Andrey Malevich, Dheevatsa Mudigere, Mikhail Smelyanskiy, Liang Xiong, and Xuan Zhang. The architectural implications of facebook's dnn-based personalized recommendation. In *IEEE International Symposium on High Performance Computer Architecture, HPCA 2020, San Diego, CA, USA, February 22-26, 2020*, pages 488–501. IEEE, 2020. https://doi.org/10.1109/HPCA47549.2020.00047.

[16] LLC Gurobi Optimization. Gurobi optimizer reference manual, 2021. https://www.gurobi.com/documentation/current/refman/index.html.

[17] Juris Hartmanis. Computers and intractability: a guide to the theory of np-completeness (michael r. garey and david s. johnson). *Siam Review*, 24(1):90, 1982.

[18] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. Neural collaborative filtering. In *Proceedings of the 26th international conference on world wide web*, pages 173–182, 2017. https://doi.org/10.1145/3038912.3052569.

[19] Ranggi Hwang, Taehun Kim, Youngeun Kwon, and Minsoo Rhu. Centaur: A chiplet-based, hybrid sparse-dense accelerator for personalized recommendations. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 968–981. IEEE, 2020. https://doi.org/10.1109/ISCA45697.2020.00083.

[20] Daya Khudia, Jianyu Huang, Protonu Basu, Summer Deng, Haixin Liu, Jongsoo Park, and Mikhail Smelyanskiy. Fbgemm: Enabling high-performance low-precision deep learning inference. *arXiv preprint arXiv:2101.05615*, 2021. https://arxiv.org/abs/2101.05615.

[21] Soojeong Kim, Gyeong-In Yu, Hojin Park, Sungwoo Cho, Eunji Jeong, Hyeonmin Ha, Sanha Lee, Joo Seong Jeong, and Byung-Gon Chun. Parallax: Sparsity-aware data parallel training of deep neural networks. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–15, 2019. https://doi.org/10.1145/3302424.3303957.

[22] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 583–598, 2014. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/li_mu.

[23] Michael Lui, Yavuz Yetim, Özgür Özkan, Zhuoran Zhao, Shin-Yeh Tsai, Carole-Jean Wu, and Mark Hempstead. Understanding capacity-driven scale-out neural recommendation inference. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 162–171. IEEE, 2021. https://doi.org/10.1109/ISPASS51385.2021.00033.

[24] Yifei Ma, Balakrishnan Narayanaswamy, Haibin Lin, and Hao Ding. Temporal-contextual recommendation in real-time. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2291–2299, 2020. https://doi.org/10.1145/3394486.3403278.

[25] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Zhihao Jia, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, Liang Luo, Jie Amy Yang, Leon Gao, Dmytro Ivchenko, Aarti Basant, Yuxi Hu, Jiyan Yang, Ehsan K. Ardestani, Xiaodong Wang, Rakesh Komuravelli, Ching-Hsiang Chu, Serhat Yilmaz, Huayu Li, Jiyuan Qian, Zhuobo Feng, Yinbin Ma, Junjie Yang, Ellie Wen, Hong Li, Lin Yang, Chonglin Sun, Whitney Zhao, Dimitry Melts, Krishna Dhulipala, K. R. Kishore, Tyler Graf, Assaf Eisenman, Kiran Kumar Matam, Adi Gangidi, Guoqiang Jerry Chen, Manoj Krishnan, Avinash Nayak, Krishnakumar Nair, Bharath Muthiah, Mahmoud khorashadi, Pallab Bhattacharya, Petr Lapukhov, Maxim Naumov, Ajit Mathews, Lin Qiao, Mikhail Smelyanskiy, Bill Jia, and Vijay Rao. Software-hardware co-design for fast and scalable training of deep learning recommendation models. In Valentina Salapura, Mohamed Zahran, Fred Chong, and Lingjia Tang, editors, *ISCA '22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022*, pages 993–1011. ACM, 2022. https://doi.org/10.1145/3470496.3533727.

[26] 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. *CoRR*, abs/1906.00091, 2019. http://arxiv.org/abs/1906.00091.

[27] Nvidia. Nvidia collective communication library (nccl). developer.nvidia.com/nccl.

[28] Nvidia. Nvidia dgx a100. www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-dgx-a100-datasheet.pdf.

[29] Nvidia. Nvshmem communication library. developer.nvidia.com/nvshmem.

[30] Pitch Patarasuk and Xin Yuan. Bandwidth optimal all-reduce algorithms for clusters of workstations. *Journal of Parallel and Distributed Computing*, 69(2):117–124, 2009. https://doi.org/10.1016/j.jpdc.2008.09.002.

[31] Jelica Protic, Milo Tomasevic, and Veljko Milutinovic. Distributed shared memory: Concepts and systems. *IEEE Parallel & Distributed Technology: Systems & Applications*, 4(2):63–71, 1996. https://doi.org/10.1109/88.494605.

[32] Cèdric Renggli, Saleh Ashkboos, Mehdi Aghagolzadeh, Dan Alistarh, and Torsten Hoefler. Sparcml: High-performance sparse communication for machine learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2019. https://doi.org/10.1145/3295500.3356222.

[33] Haidong Rong, Yangzihao Wang, Feihu Zhou, Junjie Zhai, Haiyang Wu, Rui Lan, Fan Li, Han Zhang, Yuekui Yang, Zhenyu Guo, and Di Wang. Distributed equivalent substitution training for large-scale recommender systems. In Jimmy X. Huang, Yi Chang, Xueqi Cheng, Jaap Kamps, Vanessa Murdock, Ji-Rong Wen, and Yiqun Liu, editors, *Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval, SIGIR 2020, Virtual Event, China, July 25-30, 2020*, pages 911–920. ACM, 2020. https://doi.org/10.1145/3397271.3401113.

[34] 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 Babak Falsafi, Michael Ferdman, Shan Lu, and Thomas F. Wenisch, editors, *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, pages 344–358. ACM, 2022. https://doi.org/10.1145/3503222.3507777.

[35] Geet Sethi, Pallab Bhattacharya, Dhruv Choudhary, Carole-Jean Wu, and Christos Kozyrakis. Flexshard: Flexible sharding for industry-scale sequence recommendation models. *arXiv preprint arXiv:2301.02959*, 2023.

[36] Brent Smith and Greg Linden. Two decades of recommender systems at amazon. com. *Ieee internet computing*, 21(3):12–18, 2017. https://doi.org/10.1109/MIC.2017.72.

[37] Xiaoniu Song, Yiwen Zhang, Rong Chen, and Haibo Chen. Ugache: A unified gpu cache for embedding-based deep learning. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 627–641, 2023.

[38] Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Patrick S. McCormick, Jamaludin Mohd-Yusof, Xi Luo, Dheevatsa Mudigere, Jongsoo Park, Misha Smelyanskiy, and Alex Aiken.

Unity: Accelerating DNN training through joint optimization of algebraic transformations and parallelization. In Marcos K. Aguilera and Hakim Weatherspoon, editors, *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 267–284. USENIX Association, 2022. https://www.usenix.org/conference/osdi22/presentation/unger.

[39] Minjie Wang, Chien-chin Huang, and Jinyang Li. Supporting very large models using automatic dataflow graph partitioning. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–17, 2019. https://doi.org/10.1145/3302424.3303953.

[40] Yuke Wang, Boyuan Feng, Zheng Wang, Tong Geng, Kevin Barker, Ang Li, and Yufei Ding. {MGG}: Accelerating graph neural networks with {Fine-Grained}{Intra-Kernel}{Communication-Computation} pipelining on {Multi-GPU} platforms. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 779–795, 2023. https://www.usenix.org/conference/osdi23/presentation/wang-yuke.

[41] Zehuan Wang, Yingcan Wei, Minseok Lee, Matthias Langer, Fan Yu, Jie Liu, Shijie Liu, Daniel G Abel, Xu Guo, Jianbing Dong, et al. Merlin hugectr: Gpu-accelerated recommender system training and inference. In *Proceedings of the 16th ACM Conference on Recommender Systems*, pages 534–537, 2022.

[42] Zheng Wang, Yuke Wang, Jiaqi Deng, Da Zheng, Ang Li, and Yufei Ding. Rap: Resource-aware automated gpu sharing for multi-gpu recommendation model training and input preprocessing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 964–979, 2024.

[43] Zheng Wang, Yuke Wang, Boyuan Feng, Dheevatsa Mudigere, Bharath Muthiah, and Yufei Ding. El-rec: Efficient large-scale recommendation model training via tensor-train embedding table. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14. IEEE, 2022.

[44] Jie Amy Yang, Jianyu Huang, Jongsoo Park, Ping Tak Peter Tang, and Andrew Tulloch. Mixed-precision embedding using a cache. *arXiv preprint arXiv:2010.11305*, 2020. https://arxiv.org/abs/2010.11305.

[45] Chunxing Yin, Bilge Acun, Carole-Jean Wu, and Xing Liu. Tt-rec: Tensor train compression for deep learning recommendation models. *Proceedings of Machine Learning and Systems*, 3, 2021. https://proceedings.mlsys.org/paper/2021/hash/979d472a84804b9f647bc185a877a8b5-Abstract.html.

[46] Daochen Zha, Louis Feng, Bhargav Bhushanam, Dhruv Choudhary, Jade Nie, Yuandong Tian, Jay Chae, Yinbin Ma, Arun Kejariwal, and Xia Hu. Autoshard: Automated embedding table sharding for recommender systems. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 4461–4471, 2022. https://doi.org/10.1145/3534678.3539034.

[47] Daochen Zha, Louis Feng, Qiaoyu Tan, Zirui Liu, Kwei-Herng Lai, Bhargav Bhushanam, Yuandong Tian, Arun Kejariwal, and Xia Hu. Dreamshard: Generalizable embedding table placement for recommender systems. *Advances in Neural Information Processing Systems*, 35:15190–15203, 2022.

[48] Hailin Zhang, Zirui Liu, Boxuan Chen, Yikai Zhao, Tong Zhao, Tong Yang, and Bin Cui. Cafe: Towards compact, adaptive, and fast embedding for large-scale recommendation models. *Proceedings of the ACM on Management of Data*, 2(1):1–28, 2024.

[49] 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. *arXiv preprint arXiv:2003.05622*, 2020. https://proceedings.mlsys.org/book/315.pdf.

[50] Weijie Zhao, Jingyuan Zhang, Deping Xie, Yulei Qian, Ronglai Jia, and Ping Li. Aibox: Ctr prediction model training on a single node. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, pages 319–328, 2019. https://doi.org/10.1145/3357384.3358045.

[51] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. Alpa: Automating inter- and Intra-Operator parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 559–578, 2022. https://www.usenix.org/conference/osdi22/presentation/zheng-lianmin.