EL-Rec: Efficient Large-Scale Recommendation Model Training via Tensor-Train Embedding Table

Zheng Wang
University of California, Santa Barbara
Santa Barbara, CA, USA
zheng_wang@ucsb.edu

Yuke Wang
University of California, Santa Barbara
Santa Barbara, CA, USA
yuke_wang@cs.ucsb.edu

Boyuan Feng
University of California, Santa Barbara
Santa Barbara, CA, USA
boyuan@cs.ucsb.edu

Dheevatsa Mudigere
Meta
San Francisco, CA, USA
dheevatsa@fb.com

Bharath Muthiah
Meta
Santa Clara, CA, USA
bharathm@fb.com

Yufei Ding
University of California, Santa Barbara
Santa Barbara, CA, USA
yufeiding@cs.ucsb.edu

***Abstract*—Deep learning Recommendation Models (DLRMs) plays an important role in various application domains. However, existing DLRM training systems require a large number of GPUs due to the memory-intensive embedding tables. To this end, we propose EL-Rec, an efficient computing framework harnessing the Tensor-train (TT) technique to democratize the training of large-scale DLRMs with limited GPU resources. Specifically, EL-Rec optimizes TT decomposition based on key computation primitives of embedding tables and implements a high-performance compressed embedding table which is a drop-in replacement of Pytorch API. EL-Rec introduces an index reordering technique to harvest the performance gains from both local and global information of training inputs. EL-Rec also highlights a pipeline training paradigm to eliminate the communication overhead between the host memory and the training worker. Comprehensive experiments demonstrate that EL-Rec can handle the largest publicly available DLRM dataset with a single GPU and achieves 3× speedup over the state-of-the-art DLRM frameworks.**

***Index Terms*—Recommender systems, High performance computing, Deep learning**

I. Introduction

Over the recent years, deep learning recommendation models (DLRMs) [1]–[5] attract a lot of attention from the research and industry. DLRMs combine sparse input embedding learning [6], [7] with neural networks and has demonstrated significant improvement compared with the collaborative filtering based recommendation model [8]. DLRMs have accounted for a significant proportion of deep learning instances in the industry, including product recommendations from Amazon [9], personalized advertisements from Google [10], and e-commerce recommendations from Alibaba [11].

Different from traditional compute-intensive neural network architectures [12], [13], DLRMs consist of not only the compute-intensive multi-layer perceptron (MLP) but also the memory-intensive embedding tables. In the industry-scale DLRM workload, the footprint of embedding tables is on the order of terabytes [14]–[16] which easily surpasses the limited High Bandwidth Memory (HBM) of a single GPU device (several tens of GBs) [17]. To address this challenge, a hybrid-parallel distributed training system is leveraged by many industrial companies, such as Nvidia HugeCTR [18] and Facebook NEO [16]. In these systems, the compute-intensive MLP layers are replicated across all GPUs and trained in a data-parallel style, while the memory-intensive embedding tables are sharded across different GPUs and trained in a model-parallel style. Yet, due to the huge footprint of embedding tables and the limited GPU HBM capacity, a large number of GPUs are often required to handle an industry-scale DLRM, leading to a very expensive and energy-consuming system design choice.

Towards efficient DLRM training, embedding table compression is a promising approach to reduce the demand for large HBM capacity. Two major directions of effort have been devoted to compressing the embedding tables. The first direction leverages low-bit quantization to represent the embedding table [6], [19]. The quantization is feasible for inference, but training with a quantized embedding table often yields significant accuracy losses [19]. The second direction employs a factorization method for memory storage saving like TT-Rec [20]. It leverages Tensor-train (TT) decomposition to replace large embedding tables in a DLRM with a sequence of small matrix multiplications. These methods often give negligible accuracy loss, but they fail to provide efficient DLRM computation primitives built on top of the compressed embedding tables and thus introduce significant computation overhead. For example, the embedding lookup operation of TT-Rec is $2.3\times$ slower than Pytorch embedding API on uncompressed embedding tables. In addition, DLRMs often come with a diverse set of embedding tables (e.g., size, accessing frequency), while TT-Rec only employs a homogeneous compression scheme to process all tables without taking into account the distinct index distribution pattern of the DLRM training input.

Another direction to build an efficient DLRM training system design [21], [22] capitalizes on the host memory for maintaining the embedding parameters of large sizes, such

SC22, November 13-18, 2022, Dallas, Texas, USA
978-1-6654-5444-5/22/$31.00 ©2022 IEEE

as Facebook DLRM [23] and FAE [24]. These distributed DLRM training systems are often built on the Parameter Server (PS) architecture [25], [26]. Specifically, the servers maintain embedding table parameters in the host memory and conduct the highly sparse embedding table operations (e.g., embedding lookup) on CPUs. While workers hold dense parts of DLRM (MLP layers) in the GPU HBM and perform the forward and backward computations. A worker needs to pull its corresponding parameters from the server at the forward phase and push the gradient back to the server after the backward phase. Such PS-based DLRM training framework usually suffers from these two bottlenecks [27]: (1) parameter communication latency between the servers and the workers; (2) CPU side embedding table computation and synchronization latency. This motivates further system-level optimization based on the PS architecture to address the performance bottlenecks.

To this end, we propose EL-Rec (Figure 1), a more flexible and economical DLRM training system design, which could democratize the training of the large-scale DLRMs with limited GPU resources and lower the training cost of the industry-scale DLRMs. EL-Rec is also the first framework that systematically integrates the benefits from both embedding table compression and utilization of host memory for DLRM training. In particular, to address the major challenges from previous individual work [20]–[22], [24], EL-Rec offers a algorithm-input-system co-design. **1) At the algorithm level**, we propose *Efficient TT (Eff-TT) table*, a compressed representation of embedding table, which simultaneously leverages the computation patterns of both tensorization and other embedding table related primitives. This new design allows a much smaller footprint while keeping high embedding lookup throughput. Our Eff-TT table can be easily integrated into various DLRM frameworks by directly replacing Pytorch `nn.EmbeddingBag()` API with our Eff-TT table API. **2) At the input level**, EL-Rec introduces an *Index Reordering* technique which not only takes advantage of the highly skewed access pattern of the embedding table (global information) but also employs the index relationship within each batch (local information) to help TT tables achieve better performance. **3) At the system level**, EL-Rec utilizes the host memory to further extend the memory capacity. A three-stage training pipeline is designed to largely hide the communication overhead between CPU and GPUs. We also implement a low-cost embedding cache to resolve the read-after-write conflict in pipelined DLRM training. The comparison between EL-Rec with the most relevant DLRM frameworks are summarized in Table I.

Overall, we make the following contributions in this paper:

- We optimize TT decomposition based on the key computation pattern of embedding tables and propose Eff-TT table, a GPU-friendly embedding table representation that achieves a significantly smaller footprint and maintains low lookup latency.
- We leverage both local and global information of training data and propose an index reordering technique that maximizes the performance of Eff-TT tables.
- A pipeline training paradigm is designed to eliminate the communication overhead between different memory hierarchies and a low overhead embedding cache is incorporated to solve the read-after-write conflict.
- Comprehensive experiments show that our model can handle the largest open-source recommendation model dataset with limited GPU resources, and achieves 3× speedup compared to the state-of-the-art DLRM systems.

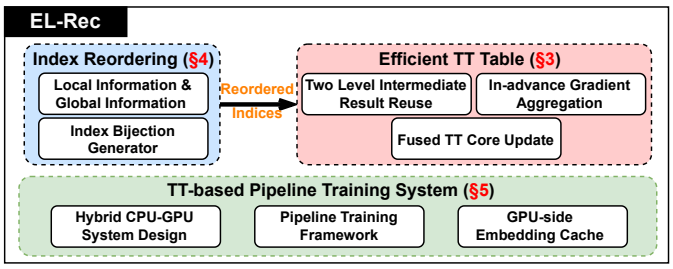


Fig. 1. EL-Rec Overview.

TABLE I
COMPARISON BETWEEN DIFFERENT FRAMEWORKS.

Framework	Host Memory	Embedding Compression	CPU-GPU Comm. Latency	Compression Overhead
DLRM [23]	✓	✗	High	N/A
FAE [24]	✓	✗	Moderate	N/A
TT-Rec [20]	✗	✓	N/A	High
EL-Rec	✓	✓	Low	Low

II. BACKGROUND

In this section, we will introduce the basics of the DLRM and tensor-train decomposition.

A. Deep Learning Recommendation Model

Figure 2 shows the workflow of a DLRM. There are two types of inputs in the recommendation model, *dense input* and *sparse input*. Dense inputs typically represent continuous data (e.g. user's age, login time) which are processed by a neural network (Bottom MLP). Sparse inputs are discrete data (e.g. user's rating for an item) that are represented as one-hot or multi-hot binary vectors. The sparse inputs will be converted to embeddings by looking up corresponding rows from the embedding tables. After input processing, both dense inputs and sparse inputs are converted to features with the same number of dimensions. Then the embedding layer output will combine with the output from Bottom MLP via feature interaction layer which computes dot products of all feature pairs. The results of the dot product are concatenated with the original embeddings and fed into another neural network (Top MLP) to get the prediction of users' click through rate (CTR), e.g., the probability of a user clicking a recommended item.

In contrast to more traditional DNN layers like Conv and MLP, the embedding table that demands significant memory capacity is the key challenge in DLRM training [14], [16]. Many frameworks capitalize on the host memory for maintaining the embedding parameters of large size [21]–[24]. To mitigate the communication overhead between host memory and GPU, pre-fetch embedding parameters for the next few

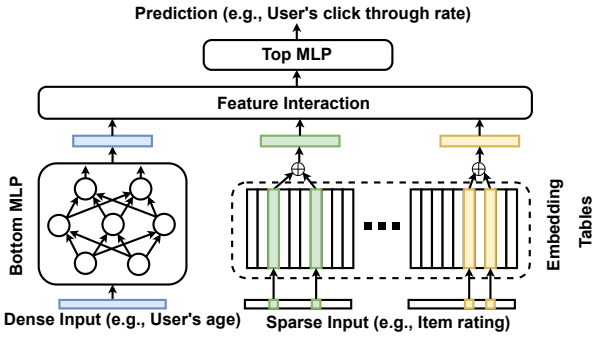


Fig. 2. Workflow of deep learning recommendation model.

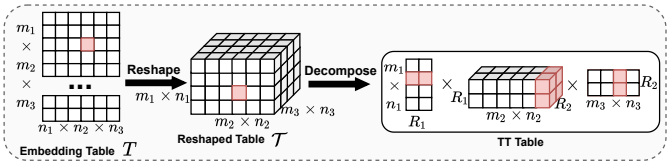


Fig. 3. The process of representing an embedding table into a TT table. m_i and n_i are factorized embedding table size, R_i is TT rank.

training batches is a promising way. However, embedding tables are trainable parameters that should be updated after every training iteration. Simply pre-fetch will incur data consistency issues caused by read-after-write conflict and slow down the model convergence [27]. This motivates us to design a dedicated software-managed embedding cache to synchronize embedding parameters and guarantee data consistency.

B. Tensor-train Decomposition

Tensor-train (TT) decomposition is widely adapted to compress the DNN models and shows a high compression ratio with low accuracy loss [20], [28], [29]. TT decomposition decomposes a d-dimension tensor $\mathcal{T} \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d}$ to a multiplication of d lower dimension tensors [30]–[32]. The elements in $\mathcal{T}$ can be computed by:

$$\mathcal{T}(i_1, i_2, \cdots, i_d) = \sum_{r_1=1}^{R_1} \sum_{r_2=1}^{R_2} \cdots \sum_{r_{d-1}=1}^{R_{d-1}} \mathcal{C}^{(1)}(i_1, r_1) \mathcal{C}^{(2)}(r_1, i_2, r_2) \cdots \mathcal{C}^{(d)}(r_{d-1}, i_d) \quad (1)$$

where the low dimension tensor $\mathcal{C}^{(k)} \in \mathbb{R}^{R_{k-1} \times n_k \times R_k}$ is called **TT core**, R_k is so-called TT ranks which are hyperparameters and $R_0 = R_d = 1$ by definition.

The TT decomposition can also be generalized to compress the embedding table [33]. We assume that embedding table $T \in \mathbb{R}^{M \times N}$. The size M and N can be factorized to $M = m_1 \times m_2 \times \cdots \times m_d$ and $N = n_1 \times n_2 \times \cdots \times n_d$. Then embedding table T can be converted to a d-dimensional tensor $\mathcal{T} \in \mathbb{R}^{(m_1 \times n_1) \times (m_2 \times n_2) \times \cdots \times (m_d \times n_d)}$. Now we can apply TT decomposition to the converted embedding table $\mathcal{T}$. The element in $\mathcal{T}$ which is indexed by $[(i_1 \cdot j_1), (i_2 \cdot j_2), \cdots, (i_d \cdot j_d)]$ can be computed following the equation below:

$$\mathcal{T}[(i_1 \cdot j_1), (i_2 \cdot j_2), \cdots, (i_d \cdot j_d)] = \mathcal{C}^{(1)}[(i_1 \cdot j_1), :] \mathcal{C}^{(2)}[:, (i_2 \cdot j_2), :] \cdots \mathcal{C}^{(d)}[:, (i_d \cdot j_d)] \quad (2)$$

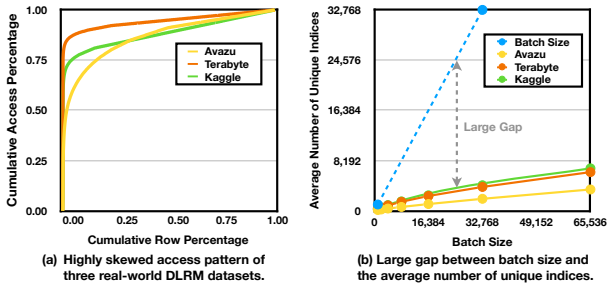


Fig. 4. Characteristics of DLRM training data: (a) Highly skewed access pattern of three real-world DLRM datasets (embedding rows are ordered by access frequency); (b) Large gap between batch size and the average number of unique indices within each batch.

We name the decomposed embedding table as **TT table**. Figure 3 shows the process of converting an embedding table into a TT table. The key challenge of applying TT tables in DLRM training is the high computation overhead. TT table lookup requires several tensor multiplication to get one element of the original embedding table which makes it much slower than Pytorch embedding API. This encourages us to implement a more efficient TT table to make it practical in large-scale DLRM training.

C. Characteristics of DLRM Training Data

In addition to the model architecture, the training data of DLRM also exhibits distinct characteristics, which provide opportunities for optimizing TT table operators. First, the distribution of activated sparse indices follows a "power-law" distribution which leads to a highly skewed access pattern of embedding tables. In Figure 4(a), we illustrate the cumulative access percentage of three real-world DLRM datasets. The access pattern indicates that a small proportion of embeddings accounts for the majority of embedding access. This observation motivates us to reuse the intermediate result of these popular embeddings to reduce the additional computation of the TT table lookup (detailed in Section III-A)

Second, the number of unique indices in a batch is much smaller than the batch size. As shown in Figure 4(b), there is a large gap between batch size and the average number of unique indices within each batch. This implies that many embedding entries are accessed more than once in a batch which will bring redundant gradient computation when back propagating the gradient of TT cores. Such a large gap provides us optimization opportunities to aggregate the embedding gradient in advance to reduce the computation amount in the TT table backward (detailed in Section III-B).

III. Efficient TT Table Design

In this section, we will detail our Eff-TT table design which includes forward phase intermediate result reuse and backward phase in-advance gradient aggregation.

A. Two-level Intermediate Result Reuse

The footprint of the TT table is smaller, but it incurs extra computation when lookup embeddings in the forward phase and computing gradient in the backward phase. The

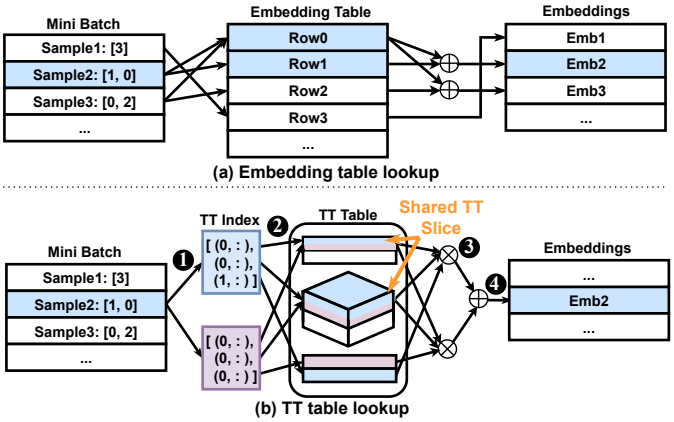


Fig. 5. The difference between embedding table lookup and TT table lookup: (a) The process of embedding table lookup. (b) The process of TT table lookup. Different colors indicate different TT indices and the shared TT slices provide the opportunity for intermediate result reuse.

additional computation overhead is non-trivial, resulting in a significant increase in training time. In this work, we analyze the computation pattern of the TT table in-depth and explore the data reuse opportunity in the TT table computation. To make full use of the reusable intermediate result in the TT table computation, the main challenge is how to determine which intermediate result can be reused. To this end, we explore the intermediate result reuse opportunity at two different levels:

Sample-level Reuse: Figure 5(a) shows the process of embedding table lookup. An input sample in a mini-batch consists of several indices. Taking $Sample_2$ as an example, first, we need to get Row_0 and Row_1 from the embedding table, and then apply element-wise addition to Row_0 and Row_1 to get the embedding of $Sample_2$: $Emb_2 = Row_0 + Row_1$. For TT table lookup, the process is more complicated. It requires several steps to lookup $Sample_2$ from the TT table. As shown in Figure 5(b): Step ❶ is to *convert indices to TT indices*: each TT table has several TT cores each needs an index, so we need first convert the 1-D embedding table index into a higher dimensional TT index. Assuming that the embedding table size is $M \times N$ (where $M = m_1 \times m_2 \times \cdots \times m_d$ and $N = n_1 \times n_2 \times \cdots \times n_d$) and the original index is i_{emb}, the TT index of the k-th TT core can be obtained by:

$$\begin{aligned} i_k &= \frac{i_{emb}}{\prod_{i=k+1}^{d} m_i} \ mod \ m_k \\ j_k &= 0, 1, \cdots, n_{k-1} \end{aligned} \tag{3}$$

In Figure 5(b) with $M = 2 \times 2 \times 2$, the original index $[1, 0]$ is converted to $t_1 = [(0, :), (0, :), (1, :)]$ and $t_2 = [(0, :), (0, :), (0, :)]$. Step ❷ is to *get TT slices from a TT table*: we need to get the corresponding TT slices from the TT table according to the TT index for later computation. Here t_1 and t_2 have the same indices in the first two dimensions, so they share the same TT slices from the first two TT cores. Step ❸ is to *compute TT slice multiplication*: the embedding row from the original embedding table can be computed by multiplying all TT slices following Equation 2. We define $Slice_{[p,q]}$ as the TT slice indexed by t_p from the q-th TT core, then $Row_1 = Slice_{[0,0]} \times Slice_{[0,1]} \times Slice_{[0,2]}$ and $Row_0 = Slice_{[1,0]} \times Slice_{[1,1]} \times Slice_{[1,2]}$. Step ❹ is to *sum up all embedding rows*: to get the final embedding, we obtain Emb_2 by adding Row_0 and Row_1:

$$\begin{aligned} Emb_2 &= Row_0 + Row_1 \\ &= (Slice_{[0,0]} \times Slice_{[0,1]} \times Slice_{[0,2]}) + \\ &\quad (Slice_{[1,0]} \times Slice_{[1,1]} \times Slice_{[1,2]}) \end{aligned} \tag{4}$$

To reduce the computation complexity of TT table lookup, we seek the data reuse opportunity. Taking $Sample_2$ in Figure 5(b) as an example, the TT indices of $Sample_2$ are $t_1 = [(0, :), (0, :), (1, :)]$ and $t_2 = [(0, :), (0, :), (0, :)]$. The first two indices of t_1 and t_2 are the same, so that $Slice_{[0,0]} = Slice_{[1,0]}$ and $Slice_{[0,1]} = Slice_{[1,1]}$, and the intermediate result of $Slice_{[:,0]} \times Slice_{[:,1]}$ can be reused. Then, the computation of Emb_2 turns into:

$$\begin{aligned} Emb_2 = Slice_{[0,0]} &\times Slice_{[0,1]} \\ &\times \left(Slice_{[0,2]} + Slice_{[1,2]}\right) \end{aligned} \tag{5}$$

In Equation 5, the times of TT slices multiplication have been reduced from four to two. Assuming that we represent a $M \times N$ embedding table into three TT cores. And we assume that each input sample has k indices, the average TT rank is R, and the average size of the reshaped embedding table $\mathcal{T}$ in each dimension is $(m \times n)$ which means $M = m^3$ and $N = n^3$. Then the computation complexity of lookup a sample from the embedding table is $\mathcal{O}_{emb} = \mathcal{O}((k-1)N) = \mathcal{O}(kn^3)$. However, the computation complexity of lookup a sample from the TT table is $\mathcal{O}_{TT} = \mathcal{O}(k(2R-1)n^2(n+R)+(k-1)N) = \mathcal{O}(kn^2R^2)$. In practice, $R \gg n$, so that $\mathcal{O}_{TT}$ is much larger than $\mathcal{O}_{emb}$. Ideally, if all TT indices in a sample are partially equal in some dimensions, we can reuse the intermediate results from every embedding row. Then the computation complexity can be reduced to $\mathcal{O}_{eff\_TT} = \mathcal{O}(n^2R^2)$. This result demonstrates that reusing sample-level intermediate results will significantly reduce the computation amount of TT table lookup.

Batch-level Reuse: In practice, training data is usually processed in batches, which brings more data reuse opportunities. We generalize the sample level intermediate result reuse to the batch level. Our key insight is that if the samples within a batch have partially identical TT indices, then the intermediate result of the identical part can be reused. The main challenge is how to identify the reusable intermediate result within an input batch. To address this challenge, we design a *Reuse buffer* to maintain the reusable intermediate result of the first two TT cores and propose an parallel pointer preparation kernel to identify the inevitable computation and prepare the pointers for the batched-GEMM kernel (e.g. *cublasGemmBatchedEx()*) to compute multiple matrix multiplications simultaneously.

We summarize the process in Algorithm 1. The goal of Algorithm 1 is to prepare the pointer list Ptr_a, Ptr_b, and Ptr_c for the batched-GEMM kernel. Ptr_a, Ptr_b store the address of the first two TT cores ($TT\_cores[0]$

Algorithm 1: Parallel Pointer Preparation.

input : Batched input: $Batch\_idx$, Reuse buffer: Buf
output: Tensor address pointers Ptr_a, Ptr_b, Ptr_c.
/* Initialize some auxiliary variables. */
1 $Buf\_len = 0$, $Buf\_flag = [0]$
2 **for each** $Idx \in Batch\_idx$ **do in parallel**
  /* Compute buffer index of the row. */
3 $Buf\_idx = Idx \ / \ length_3$;
  /* Check availability of buffer at Buf_idx. */
4 **if** $(atomicCAS(Buf\_flag[Buf\_idx], 0, 1) == 0)$ **then**
    /* Update buffer length. */
5 $cur\_offset = atomicAdd(Buf\_len, 1)$;
    /* Compute TT index0 and TT index1. */
6 $TT\_idx1 = Buf\_idx \ / \ length_2$;
7 $TT\_idx0 = Buf\_idx \ \% \ length_2$;
    /* Update Ptr_a, Ptr_b, Ptr_c. */
8 $Ptr\_a[Buf\_idx] = \&TT\_cores[1] + TT\_idx1 * shapes[1]$;
9 $Ptr\_b[Buf\_idx] = \&TT\_cores[0] + TT\_idx0 * shapes[0]$;
10 $Ptr\_c[Buf\_idx] = \&Buf + cur\_offset * shapes[2]$;
11 **end**
12 **end**

and $TT\_cores[1]$), and Ptr_c should be pointing to some address in the *Reuse Buffer* that stores the intermediate result of $TT\_cores[0]$ multiplying $TT\_cores[1]$. The number of threads is equal to the number of indices in the batch. Each thread processes one embedding table index in the batch in parallel (Line 2). It will first compute the corresponding address in *Reuse Buffer* by dividing the length of the last TT core $length_3$ (Line 3). Then each thread will check whether its computation can be skipped (Line 4). This information is stored in Buf_flag. If $Buf\_flag[Buf\_idx] == 1$, it means the intermediate result of the first two TT cores will be computed by other threads. If not, the Ptr_a, Ptr_b, and Ptr_c should be assigned with corresponding addresses (Line 8 to 10) for subsequent GEMM computation. Once the pointer preparation is done, we can call the batched-GEMM kernel with the address pointers Ptr_a, Ptr_b, and Ptr_c to simultaneously compute the intermediate results of the first two TT cores and store them into the *Reuse Buffer*. Algorithm 1 helps Eff-TT table determine which computation is inevitable and minimizes the computation amount of the Eff-TT table lookup.

B. TT-centric Backward Optimization

For TT table backward, the main task is to calculate the gradient of TT cores and update the parameters of TT cores. We assume that a TT table has d TT cores, then an embedding row can be computed by: $e = \mathcal{C}^{(1)}[i_1, :]\mathcal{C}^{(2)}[:, i_2, :] \cdots \mathcal{C}^{(d)}[:, i_d]$. During the backward process, we first obtain the gradient of embedding $\frac{\partial L}{\partial e}$. Next, we follow the chain rule and compute the gradient of the k-th TT cores $\frac{\partial L}{\partial \mathcal{C}^{(k)}[:, i_k, :]}$:

$$\frac{\partial L}{\partial \mathcal{C}^{(k)}[:, i_k, :]} = \left(\mathcal{C}^{(1)}[i_1, :]\mathcal{C}^{(2)}[:, i_2, :] \cdots \mathcal{C}^{(k-1)}[:, i_{k-1}, :]\right)^T \cdot \frac{\partial L}{\partial e} \cdot \left(\mathcal{C}^{(k+1)}[:, i_{k+1}, :] \cdots \mathcal{C}^{(d)}[:, i_d]\right)^T \quad (6)$$

Equation 6 indicates that $(d-1)$ times of tensor multiplication are required to compute one TT core's gradients. Since there

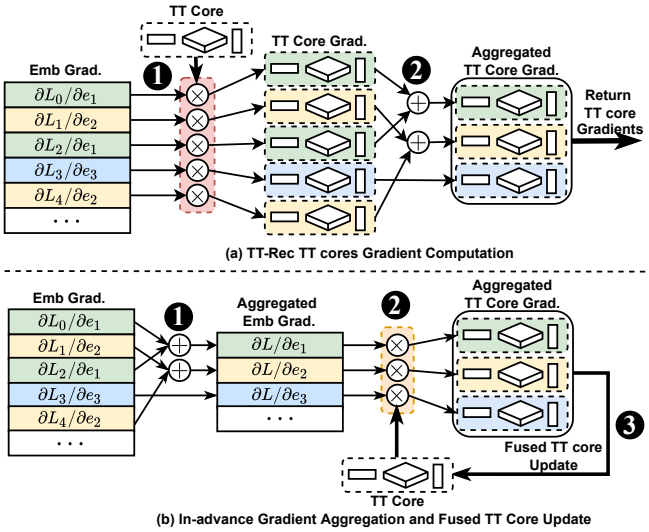


Fig. 6. The process of TT table backward: (a) TT-Rec TT table backward; (b) Eff-TT table backward with in-advance gradient aggregation. The in-advance gradient aggregation method reduces a large amount of tensor multiplication. Different color indicates the gradients corresponding to different embeddings.

are d TT cores in total, the computation complexity is d times larger than TT table lookup. The process of computing TT core gradients has been shown in Figure 6(a), for each embedding row, the corresponding gradients of TT core can be obtained by multiplying embedding gradients $\frac{\partial L}{\partial e}$ with TT cores following Equation 6, as illustrated in Step ❶. Then we need to aggregate the obtained gradients to the corresponding TT cores as illustrated in Step ❷. Finally, the aggregated TT core gradients will be returned and later used for TT core update. A large amount of tensor multiplication at Step ❶ is the bottleneck of TT table backward. To minimize redundant computation and improve the efficiency of TT table backward, we propose two methods:

In-advance Gradient Aggregation: We follow the insight that the access pattern of embedding tables is highly skewed and some popular embeddings may be accessed more than once in a batch. As shown in Figure 6(b), the gradients corresponding to the same embedding have the same color, and some embeddings occurred multiple times in a batch. We first compute the unique embeddings within a batch, and then aggregate the gradients of embeddings rows into corresponding unique embedding gradients (Step ❶ in Figure 6(b)). The next step is to multiply the aggregated embedding gradients with TT cores to get the aggregated TT core gradients (Step ❷ in Figure 6(b)). As shown in Figure 6, our in-advance gradient aggregation method significantly reduces the computation amount of tensor multiplication and also cuts down the memory consumption for intermediate gradients since we do not need to compute and store a large amount of non-aggregated TT core gradients.

Fused TT core Update: In the TT table backward phase, the gradients of TT cores will be returned to the GPU global memory and then used by the optimizer. However, the

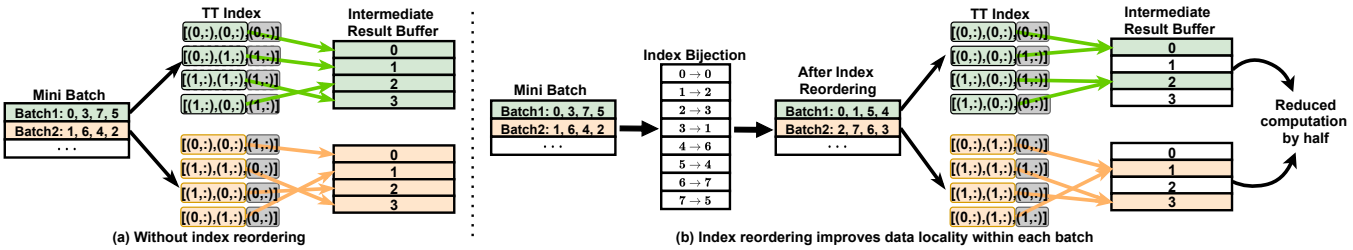


Fig. 7. (a) Eff-TT table lookup without reordering. (b) Eff-TT table lookup with index reordering. The index reordering improves data locality and provides more intermediate result reuse opportunities for the Eff-TT table.

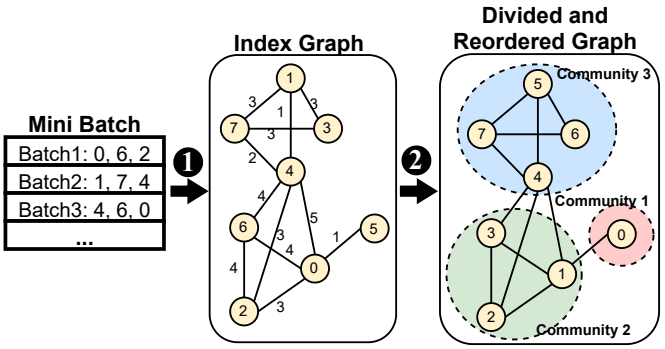


Fig. 8. The workflow of generating an index bijection: 1) Generating an index graph based on batched input data. 2) Detecting community in graph and assigning new index.

Algorithm 2: Generate Index Graph.

input : Batched Indices: $Batch\_list[\,]$,
Frequency Based Ordering: $Fre\_order[\,]$
output: Edge List of Index Graph: $Edge\_list[\,]$
/* Compute hot embedding threshold. */
$Hot\_thre = Table\_length * Hot\_ratio$;
/* Iterate through all batches. */
for each $Batch$ **in** $Batch\_list$ **do**
  /* Global Information: get frequency based index. */
  $Fre\_batch = Fre\_order\ [Batch]$;
  /* Keep Hot embeddings' index. */
  $Fre\_batch.clamp(min = Hot\_thre)$ - Hot_thre;
  /* Local Information: generate edges for $Batch$. */
  $Batch\_edges = Fre\_batch.self\_combinations()$;
  /* Append to $Edge\_list[\,]$. */
  $Edge\_list.append(Batch\_edges)$;
end

gradients of TT cores are the endpoint of backpropagation, which makes it possible to fuse the TT core parameters update operation into the TT core backward kernel. TT-Rec [20] also employs kernel fusion in its backward kernel, but TT-Rec requires additional TT core gradient aggregation before the parameter updating which incurs additional data copy. In contrast, our in-advance gradient aggregation method directly computes the aggregated gradients so that the TT cores can be updated by the aggregated gradients without additional data copy (Step ❸ in Figure 6(b)). The fused TT core update mitigates the CUDA kernel launch overhead and reduces the data loading amount from GPU global memory.

IV. Locality-based Index Reordering

In this section, we will first discuss the global and local information of embedding tables, then introduce an index reordering method that improves the performance of Eff-TT tables.

A. Global and Local Information

Global information represents the data access pattern of the DLRM embedding tables. For DLRM trained on real-world data, some rows of embedding tables are significantly more popular than others, and the access distribution of embedding tables is highly skewed which generally follows the "power-law" distribution. Such global information has been widely leveraged in various recommendation model frameworks [3], [4], [24]. However, few frameworks take advantage of the local information of the training data. Local information, unlike the global information which represents the distribution of the entire dataset, reflects the indices relationship within each mini-batch. We are driven by the insight that, the local information within input batches represents user behaviors during different periods of time (e.g. users may view more work-related information during the day and more entertainment information at night), and such information can be leveraged to improve the performance of DLRM training.

B. Locality-based Index Reordering

The performance of the Eff-TT table depends on the data distribution within each data batch. Based on Equation 3, if the indices within a batch are closer, then more TT indices will be partially equal which will provides more opportunities for intermediate result reuse. And higher data locality is also beneficial for improving the GPU L1/L2 cache hit rate.

An Eff-TT table lookup process without index reordering has been illustrated in Figure 7(a). There is no intermediate result that can be reused since indices within a batch are not closely distributed. Such poor data locality limits the performance of our TT-based optimization. To improve data locality within each batch, one solution is to physically reorder the rows of embedding tables in some order. However, it will incur a large amount of data movement because of the large footprint of embedding tables. A better way to achieve our goal is to reorder the indices. Embedding tables are trainable parameters that are initialized randomly before training so that all embedding rows in the embedding table are equivalent before the training starts. Based on the above observation, we

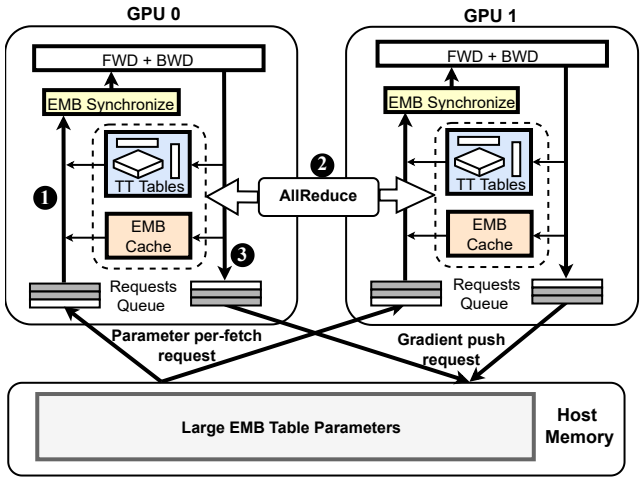


Fig. 9. Illustration of TT-based pipeline training system.

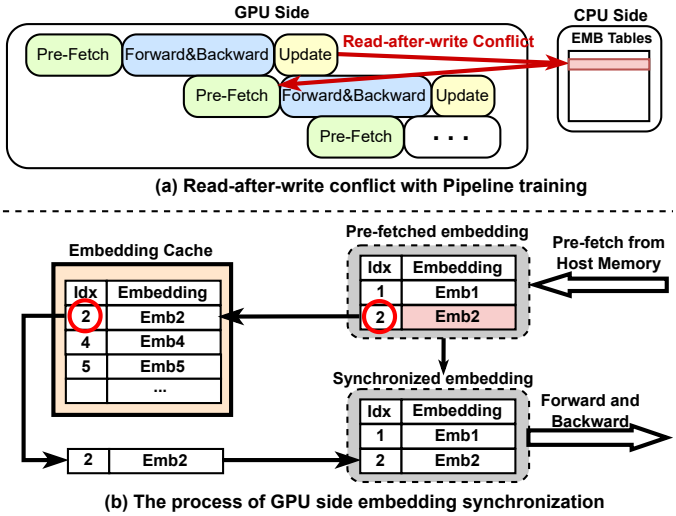


Fig. 10. Using GPU side embedding cache design to solve the read-after-write conflict: (a) Causes of the read-after-write conflict; (b) The process of GPU side embedding synchronization.

propose a locality-based index reordering method to increase the data locality within each batch and help improve the performance of our Eff-TT table design. The lookup process with index reordering has been shown in Figure 7(b), each batch is a set of indices $Batch\_i = \{i_1, i_2, \cdots, i_n\}$, all indices will first be transformed into new indices based on the index bijection function (detailed in §IV-C) $New\_Batch\_i = \{\tilde{i}_j | \tilde{i}_j = f_{index}(i_j), i_j \in Batch\_i\}$. Then the New_Batch_i will be used for embedding lookup. The index reordering significantly improves the data locality of each batch and provides more opportunities to reuse the intermediate result.

C. Index Bijection Generator

In order to facilitate an effective index reordering, we need to generate an index bijection that improves data locality in each batch. To achieve this, we propose an index bijection generator that leverages both global and local information. As shown in Figure 8, the workflow of generating an index bijection has two main steps:

First, we convert the batched indices into an index graph to capture the local information of the training data. As described in Algorithm 2, we first sort the indices by the access frequency in a descending order. Then a hyperparameter Hot_ratio is chosen to define the "hot embeddings" which means frequently-accessed embeddings, and they will not be reordered later. Here the global information is leveraged to gather hot embeddings together. Other non-hot indices are used to generate the index graph. We treat each index as a vertex in the graph, if two indices appear in the same batch simultaneously, then we will add an edge between these two vertices. Following Algorithm 2, an index graph can be generated (Step ❶ in Figure 8) which represents the locality information of the indices within each batch.

The next step is to reorder these indices based on the index graph. We leverage the modularity-based community detection algorithm [34], [35]. The modularity [36] is defined as $Q = \frac{1}{2m}\sum_i [e_{ii} - \frac{k_i^2}{2m}]$, where e_{ij} is the total number of edges between community i and j, k_i is the total degree of vertices in community i, and m is the total number of edges in the whole graph. Our goal is to find a graph partition with high modularity Q. Higher modularity means denser intra-community edges and sparser inter-community edges. After dividing the graph into different communities, a new index bijection can be get by assigning continuous indices to the vertices in the same community (Step ❷ in Figure 8). Such index bijection is able to take advantage of both global information and local information and help the TT table gain better performance. And it is worth noting that the generation of indexed bijections can be done offline, so it will not bring additional overhead to the training process at runtime.

V. TT-Based Pipeline Training System

In this section, we will first detail our TT-based pipeline training system design, then we will introduce how to solve the read-after-write conflict in pipeline DLRM training with a GPU side embedding cache design.

A. TT-based Training System Design

In addition to the TT-based algorithmic innovations, we propose a hierarchical memory system design to utilize the host memory and additionally extend the scalability of EL-Rec. Although TT decomposition can greatly compress the embedding table, the TT tables may still be too large to fit into the GPU HBM when training an industry-scale DLRM model. In this case, we need to rely on the host memory to maintain some embedding table parameters. Such a hierarchical memory design often suffers from inferior performance due to the high communication overhead between CPU and GPUs. To this end, we propose a TT-based pipeline training system to mitigate the communication latency.

As shown in Figure 9, our system design follows a Parameter Server (PS) architecture. The MLP layers (including bottom MLP and top MLP) are replicated to all GPUs and trained in a data-parallel style. The embedding layers are divided into two parts, most embedding tables are represented as TT tables

which are also replicated to all GPUs while the remaining embedding parameters that can not be fit into GPU HBM are placed in host memory.

The CPU side serves as a parameter server, it will pre-fetch the embedding parameters for the next a few batches from the host memory to the *Pre-fetch Queue* of different GPUs. And the GPUs act as workers that conduct forward and backward computation. The workers will first concatenate the embeddings from the TT tables and the pre-fetch queue to collect all embedding parameters for the working batch. And then the workers will synchronize the concatenated embeddings with the embedding cache (details in §V-B) to make sure that all embeddings are updated (Step ❶). The TT tables are trained in a data-parallel style. So that after forward and backward computation, we must AllReduce the gradients of TT tables and embedding cache before updating the parameters (Step ❷). Finally, the gradients of embedding parameters from host memory will be pushed into the *Gradient Queue* and then the server will pull the gradients and update the embedding parameters in the host memory (Step ❸).

B. Solving RAW Conflict with Embedding Cache

Pipelining is a commonly used technique to accelerate the training of deep learning on hierarchical memory architecture. The DLRM workload can also leverage pipeline training by overlapping the CPU side embedding parameter gathering and GPU side MLP forward and backward. However, such pipeline training design will incur a *Read-After-Write (RAW)* conflict. As shown in Figure 10(a), when MLP is processing data batch i, the embeddings of data batch $i+1$ will be pre-fetch at the same time. However, the update stage of batch i have not finished yet, so the pre-fetched embeddings may contain some "stale" parameters that have not been updated by the gradients of batch i.

To tackle this RAW conflict, we design an *Embedding Cache* to track the embeddings that will be used for training in the next few batches and keep the GPU side embeddings up to date. The process of GPU side embedding synchronization has been shown in Figure 10(b). The pre-fetched embeddings batch contains the indices and the embedding parameters. When a new pre-fetched embedding batch comes, the embedding cache will search the indices from the index table. If the index is found in the cache, it means that this embedding has been used by several previous batches and needs to synchronize like the Emb2 in Figure 10(b). After embedding synchronization, all embeddings are up to date and ready to be trained.

To minimize the size of the embedding cache, we use a "life cycle (LC)" system to manage the embeddings in the cache. Once the training process of an input batch is finished, the embeddings will be pushed into the cache and be assigned with LC values equal to the maximum length of the Requests Queue (pre-fetch queue and gradient queue). If the CPU pulls a batch from the gradient queue, the LC value of the corresponding embeddings will be decremented by 1. And once the LC value of an embedding is equal to 0, the embedding will be evicted from the cache. With such a life cycle management system, we are able to only keep the required embeddings into the embedding cache, and minimize its memory demand.

TABLE II
DATASETS FOR EVALUATION.

Dataset	#Input Features		Embedding Tables		
	Dense	Sparse	#Rows	Dim.	Size
Avazu [37]	1	20	8.9M	16	0.55 GB
Criteo Terabyte [38]	13	26	242.5M	64	59.2 GB
Criteo Kaggle [39]	13	26	30.8M	16	1.9 GB

TABLE III
COMPARISON OF EMBEDDING TABLE FOOTPRINT.

Dataset	*DLRM*	*EL-Rec*	Compression Ratio
Avazu	0.55GB	87.6MB	6.22×
Terabyte	59.2GB	797.9MB	74.19×
Kaggle	1.9GB	258.2MB	7.29×

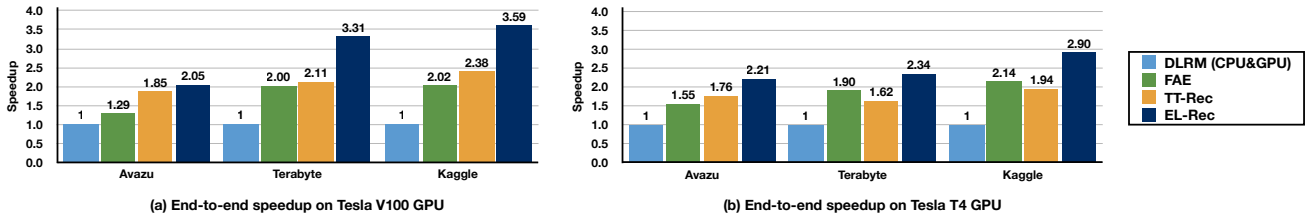
VI. EVALUATION

A. Experimental Setup

Implementation To demonstrate the advantages of our design, we built *EL-Rec* based on *DLRM* [23], an open-source recommendation model training system proposed by Facebook. The Eff-TT table is implemented with C++/CUDA and integrated with Pytorch framework by using Pytorch Wrapper. The Eff-TT table is also highly portable to other Pytorch-based frameworks by directly replacing the Pytorch nn.EmbeddingBag() API in with the Eff-TT table API.

Benchmark and Datasets We use three widely adopted real-world datasets. **Avazu** [37] is a dataset for click through rate prediction which consists of 11 days worth of Avazu data. Each sample in Avazu has 1 numerical feature and 20 categorical features. **Criteo Terabyte** [38] is the largest publicly available dataset for DLRM. It contains over four billion samples spanning 24 days. Each record contains 39 features: 13 numerical features, and 26 categorical features. **Criteo Kaggle** [39] is a subset of Criteo Terabyte which is used for Criteo Kaggle Display Advertising Challenge. It contains a portion of Criteo's traffic over a period of 7 days and the data format is the same as Criteo Terabyte. Table II shows the details of these datasets. It is worth noting that the footprint of Criteo Terabytes's embedding tables is about 59.2 GB, which has exceeded the storage capacity of most GPUs. In industry, the training data is larger than these public datasets [15] which highlights the importance of *EL-Rec*.

Baseline Besides the *DLRM* framework, we choose four state-of-the-art DLRM frameworks that are optimized at the algorithm or system level. **1) TT-Rec** [20] is proposed by Facebook. It highlights an algorithm-level optimization that leverages TT decomposition to compress large embedding tables. For a fair comparison, we integrate compressed embedding table API of *TT-Rec* into the model of *DLRM*. In the end-to-end training comparison, both *EL-Rec* and *TT-Rec* decompose the embedding tables that have more than 1 million rows while keeping other small embedding tables uncompressed. **2) FAE** [24] focuses on system-level design.



(a) End-to-end speedup on Tesla V100 GPU

(b) End-to-end speedup on Tesla T4 GPU

Fig. 11. End-to-end training speedup (×) with limited GPU resources (single GPU).

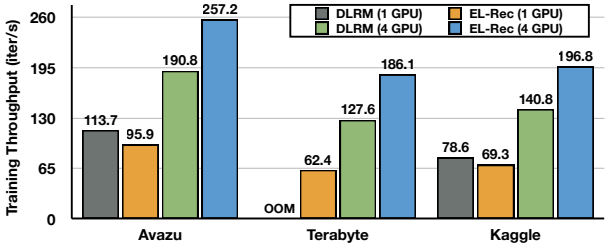


Fig. 12. Training throughput under multi-GPU setting.

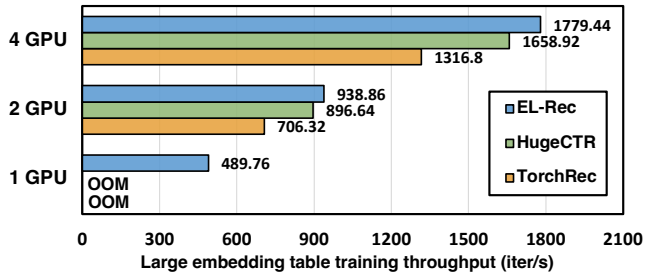


Fig. 13. Comparison of single large embedding table training throughput with HugeCTR and TorchRec.

FAE also utilizes the host memory to maintain the large embedding tables. It offloads the frequently-accessed embedding to GPU so that most training batches can be trained purely on GPU which eliminates a large proportion of communication overhead between CPU and GPUs. **3) HugeCTR** [18] is a highly-optimized industry-level recommendation training framework proposed by Nvidia. It provides distributed training with model-parallel embedding tables and data-parallel MLP across multiple GPUs and nodes for maximum performance. For the large embedding tables, HugeCTR scales the number of GPUs to fit the entire embedding table within GPUs' HBM. **4) TorchRec** [40] is recently released by Facebook. It provides the implementation of "4D Parallelism" [16] that combines table-wise, row-wise, column-wise, and data parallelism for training massive embedding operators in DLRMs. Large embedding tables will be split into multiple pieces by either row-wise or column-wise sharding and be placed on different GPUs for model-parallel training.

Platform and Tools Our major evaluation platform is a single AWS p3.8xlarge instance with one Intel Xeon CPU@2.30GHz, 239 GB CPU memory, and 4× Nvidia Tesla V100 GPU. We also evaluate end-to-end performance on AWS g4dn.12xlarge instance which has Intel Xeon CPU@2.50GHz, 192 GB CPU memory and 4× Nvidia Tesla T4 GPU. On the software side, we employ Nvidia NVTabular [41] for data preprocessing. Nvidia NVTabular also provides a high-performance dataloader for loading DLRM training data from the disk. For a fair comparison, we replace the default PyTorch dataloader with the Nvidia NVTabular dataloader in the baseline frameworks. Our efficient TT table uses cuBLAS [42] library for batched-GEMM computation in Eff-TT table.

B. Compared with other Frameworks

In this section, we demonstrate the training throughput of EL-Rec and compare with different baseline models. We design two main evaluation settings to show the advantages of *EL-Rec* in different aspects: 1) DLRM training with limited GPU resources, especially under the single GPU setting; 2) DLRM training with multiple GPUs.

DLRM training with limited GPU resources: To simulate a limited GPU resource environment, we use one GPU on both platforms. *DLRM* and *FAE* place embedding table parameters in host memory, while *TT-Rec* and *EL-Rec* use a compressed format of embedding table and store embedding table parameters in GPU HBM. The batch size we choose is 4K, and the setting of TT rank is 128 on Tesla V100 GPU and 64 on Tesla T4 GPU. We compare the end-to-end training time of *EL-Rec* to all baseline models, the result has been shown in Figure 11. *EL-Rec* consistently shows the best end-to-end performance on different platforms and datasets. We observe that *EL-Rec* achieves 3× speedup on average over *DLRM* on Tesla V100 GPU since the small footprint of Eff-TT table makes it possible to fit all parameters into a single GPU and avoid the communication overhead between CPU and GPU. Compared to *FAE*, *EL-Rec* achieves 1.5× speedup on average. *FAE* caches frequently-accessed embeddings on GPU but the batches that contain non-frequently-accessed embeddings (about 25% in our profiling) still have to be trained on CPU which prevents *FAE* from getting better performance. And we also observe that *EL-Rec* outperforms *TT-Rec* with 1.4× speedup on average, the reason is that *EL-Rec* leverages multiple optimization techniques to reduce the computation of TT table and improve the performance of TT table lookup and backward.

DLRM training on multi-GPU platform: With more GPU resources, we would like to keep all embedding table parameters in GPUs. We trained *EL-Rec* and *DLRM* on AWS p3.8xlarge platform with different numbers of GPUs (1 GPU and 4 GPU), and compare the training throughput of *EL-Rec* and *DLRM*. As shown in Figure 12, *EL-Rec (4 GPU)* achieves significant improvement (up to 1.4× on average) in training throughput compared with *DLRM (4 GPU)*. This is

TABLE IV
COMPARISON OF PREDICTION ACCURACY.

Model \ Dataset	Avazu	Criteo Terabyte	Criteo Kaggle
DLRM	83.53	81.96	78.53
TT-Rec	83.51	81.86	78.51
FAE	83.53	81.94	78.52
EL-Rec	**83.51**	**81.90**	**78.50**

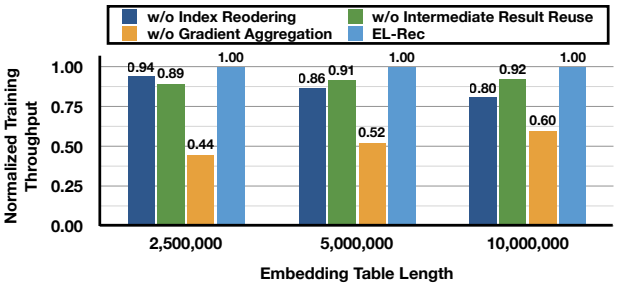


Fig. 14. Eff-TT table optimization breakdown.

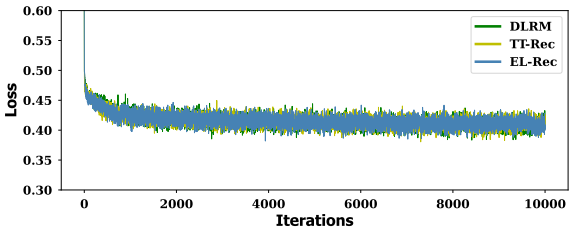


Fig. 15. Loss convergence of *EL-Rec*.

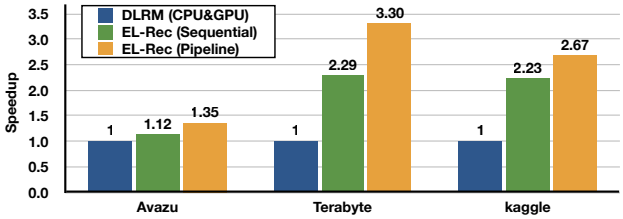


Fig. 16. Pipeline training speedup (×) over sequential training.

because the much smaller footprint of the Eff-TT table makes it possible to replicate embedding parameters to all GPUs and train the embedding layer in a data-parallel style. While the embedding tables of *DLRM* are stored on different GPUs and trained in a model-parallel style, which incurs intensive peer-to-peer communication between GPUs. In the single GPU setting, the training throughput of *DLRM (1 GPU)* is slightly higher than *EL-Rec*. This is because the tensorization method of *EL-Rec* inevitably introduced some additional computation which is minor compared to the significant memory saving (Table III). The evaluation result demonstrates the scalability of *EL-Rec* and reveals the advantages of *EL-Rec* in distributed large-scale DLRM training.

Large embedding table training: To highlight the advantage of *EL-Rec* when dealing with huge embedding tables, we construct a very large embedding table that has 40 million rows and its feature dimension is set to 128. The footprint of this large embedding table is about 19 GBs which exceeds the single GPU HBM capacity (16 GBs) that we use. We trained this large embedding table using *EL-Rec*, *HugeCTR*, and *TorchRec* with different numbers of GPUs and compare the training throughput of this very large embedding table. As shown in Figure 13, *EL-Rec* outperforms *TorchRec* and *HugeCTR* with $1.35\times$ and $1.07\times$ speedup, respectively. Due to the limited single GPU HBM capacity, *HugeCTR* distributes the embedding table parameters to multiple GPUs and *TorchRec* uses column-wise sharding to split the embedding tables into multiple smaller embedding shardings for model parallel. Both solutions introduce intensive inter-GPU communication since the model parallel embedding table training requires embedding synchronization in the forward phase and gradient synchronization in the backward phase. Instead, *EL-Rec* leverages the Eff-TT table to reduce the footprint and makes it possible to train such a large embedding table with a single GPU. Meanwhile, *EL-Rec* trains the large embedding table in a data-parallel style that eliminates embedding synchronization in the forward phase. The only communication is the gradient all-reduce in the backward phase.

Accuracy and loss convergence: To demonstrate the impact of tensorization method on model accuracy, we evaluate the prediction accuracy of *EL-Rec* on different datasets. We trained all models on Kaggle and Avazu datasets for 5 epochs, and on the Terabyte dataset for 100K iterations. As shown in Table IV, *EL-Rec* matches the accuracy of its corresponding baseline models on all three datasets. The accuracy of *DLRM* is slightly higher than *EL-Rec* since *EL-Rec* compresses large embedding tables into Eff-TT tables, the tensorization method slightly affects the accuracy, but the degree is very low (below 0.1%) which is acceptable compared to its high compression ratio. Additionally, we show the loss convergence curve of *DLRM*, *TT-Rec*, and *EL-Rec* during the training process on the Terabyte dataset in Figure 15. Although the computation pattern of the TT table is more complicated than the embedding table, the convergence curve of the *EL-Rec* is almost the same as the *DLRM* baseline. It means that utilizing the Eff-TT table will not slow down convergence and we do not need extra iterations to get *EL-Rec* converged.

C. Optimization Analysis

Optimization breakdown study: To analyze the impact of different optimizations separately, we trained three embedding tables with different rows, ranging from 2.5 million to 10 million, and compared the training throughput of the embedding table when disabling one of the optimizations. The result has been shown in Figure 14, the in-advance gradient aggregation shows the most significant influence on training throughput. Without the in-advance gradient aggregation, the training throughput decreases by about 52%, this is because the TT table backward is much more computationally intensive than TT table forward. Meanwhile, the index reordering technique and the intermediate result reuse method also show notable impact on the training throughput. Disabling the index reordering and the intermediate result reuse causes 13% and 10% performance decrease on average respectively. With the embedding table length increasing, the impact of index

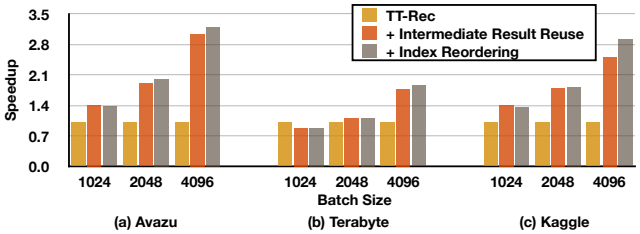


Fig. 17. Speedup (×) breakdown of the TT table *lookup* optimization.

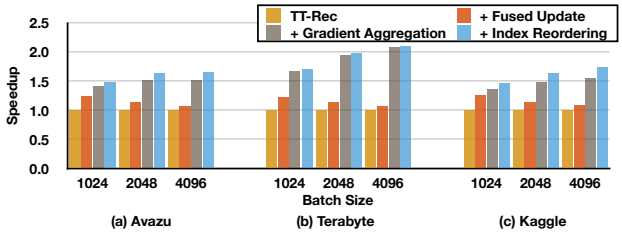


Fig. 18. Speedup (×) breakdown of the TT table *backward* optimization.

reordering steadily increases. This is because a larger table length will lead to more scattered input indices which make more room for index reordering optimization.

Pipeline training system: To demonstrate the advantage of our TT-based pipeline training system design, we compress the largest embedding table into Eff-TT table and put it into GPU HBM. The remaining embedding tables are kept in the host memory. We compare the training throughput of *EL-Rec* with the setting that disables pipeline training by setting the length of pre-fetch queue to 1. As shown in Figure 16, *EL-Rec (Pipeline)* achieves 2.44× speedup on average over *DLRM*. This is because our pipeline training design overlaps CPU-side embedding lookup and parameter updating with GPU-side MLP layer training. The communication overhead between CPU and GPU is also mitigated by an embedding pre-fetch queue and a gradient queue. And compared with *EL-Rec (Sequential)*, we observe that *EL-Rec (Pipeline)* achieves 1.30× speedup on average. In *EL-Rec (Sequential)*, the length of pre-fetch queue is set to 1, so the pipeline training will degrade to a sequential one. In this case, the GPU-side training workers can only wait for the CPU side finishing parameter update and also can not pre-fetch data for the next batch. This result demonstrates the effectiveness of our embedding cache design and shows the advantage of pipeline training.

Eff-TT table lookup optimization: We then demonstrate the effectiveness of our algorithm-level optimization for Eff-TT table lookup (§III-A) and input-level index reordering optimization (§IV). We measure the latency of Eff-TT table lookup operation with different batch sizes, the result has been shown in Figure 17. Overall, we achieve 1.83× speedup on average over the *TT-Rec* implementation of TT table. The speedup increases as batch size increases since a larger batch size provides more opportunity to reuse intermediate results at the batch level. On individual optimizations, we observed 1.75× speedup from intermediate result reuse and 1.05× speedup from the locality-aware index reordering.

Eff-TT Table Backward Optimization We further show the benefits of our algorithm-level and input-level optimization on TT table backward(§III-B and §IV). We evaluate our optimizations with different batch sizes. As shown in Figure 18, we observe 1.70× speedup on average (from 1.47× to 2.10×) over the *TT-Rec* implementation of TT table. As for the individual optimization, we observe 1.15× from fused TT core update (§III-B), 1.40× speedup from the in-advance gradient aggregation (§III-B), and 1.06× speedup from the input-level index reordering (§IV). It is worth noting that, although index reordering does not directly reduce the computation amount of the TT table backward, it helps improve GPU L1/L2 cache hit rate since reordering improves data locality. We evaluate the cache hit rate of the batched-GEMM kernel during training and observe that the L1 hit rate improves 1.27× and the L2 hit rate improves 1.32×. These results prove the effectiveness of our optimizations.

VII. Related Work

System Designs for DLRM: The DLRM is one of the most critical deep learning applications in the industry [43]–[45]. The training and inference of the recommendation model come with various challenges due to the compute-intensive MLP layer and the memory-intensive embedding tables. Work in [46] accelerates the inference of recommendation model by leveraging fused embedding lookup, DeepRecSys [4] proposed an inference scheduler that maximizes latency-bounded throughput. As for DLRM training, NEO [16] and HugeCTR [18] distribute large embedding tables to different GPUs and train the embedding table in a model-parallel style which incurs non-trivial data communication between GPUs. XDL [47] and FAE [24] leverage CPU memory to handle the large embedding tables while keeping the MLP layer in GPU, the non-optimized hybrid CPU-GPU system design makes CPU side computation becomes a bottleneck. Overall, all of these techniques have not properly solved the issues of high data communication overhead.

Embedding Table Compression: The embedding tables of the DLRMs have large memory footprint [6], [15], [20], [48], [49]. To cut down memory consumption, there have been many efforts on embedding table compression. ALBERT [48] leverages factorized embedding parameterization and cross-layer parameter sharing to reduce the footprint of parameters. Kilian et al. propose feature hashing [49] which maps multiple items to the same embedding vector. Jie et al. [6] and Hui et al. [19] use fewer bits to represent the embedding vectors. The above techniques decrease the footprint of embedding tables but often incur an accuracy tradeoff. Compressed data direct computing [50]–[54] is a novel processing method that can also be applied in embedding table compression. TT-Rec [20] applied Tensor-train (TT) factorization on embedding tables which achieves a considerable reduction of embedding table size while keeping a high model accuracy. However, the TT factorization brings additional computation and increases training time. This motivates us to optimize the TT-based

solution with system-level designs to capitalize on the benefits of TT-based embedding compression.

VIII. Conclusion

In this work, we propose EL-Rec, a more flexible and economical DLRM training system design, which could democratize the training of large-scale DLRMs with limited GPU resources and lower the training cost of the industry-scale DLRM training. Specifically, EL-Rec provides an efficient TT table design which is a compressed representation of embedding table. It achieves a much smaller footprint and maintains low lookup and backward latency. EL-Rec also incorporates an index reordering strategy based on both global and local information to help TT-based embedding computation achieve better performance. Additionally, EL-Rec introduces a TT-based pipeline training paradigm to largely hide the data communication overhead between CPU and GPUs. Comprehensive experiments demonstrate that EL-Rec can handle the largest publicly available DLRM dataset with limiting GPU HBM. And EL-Rec achieves significant speedup over the state-of-the-art DLRM training framework.

IX. Acknowledgment

We would like to thank anonymous SC paper reviewers for their valuable suggestions on our paper writing and SC artifact reviewers for helping us improve our software artifact functionality and reusability to benefit future research. This work was supported in part by NSF 2124039. Also, we would like to thank Meta for their grant in support of UCSB IEE for energy efficiency research.

References

[1] S. Zhang, L. Yao, A. Sun, and Y. Tay, "Deep learning based recommender system: A survey and new perspectives," *ACM Computing Surveys (CSUR)*, vol. 52, no. 1, pp. 1–38, 2019.

[2] B. Acun, M. Murphy, X. Wang, J. Nie, C.-J. Wu, and K. Hazelwood, "Understanding training efficiency of deep learning recommendation models at scale," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 802–814.

[3] H. Kal, S. Lee, G. Ko, and W. W. Ro, "Space: locality-aware processing in heterogeneous memory for personalized recommendations," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 679–691.

[4] U. Gupta, S. Hsia, V. Saraph, X. Wang, B. Reagen, G.-Y. Wei, H.-H. S. Lee, D. Brooks, and C.-J. Wu, "Deeprecsys: A system for optimizing end-to-end at-scale neural recommendation inference," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 982–995.

[5] Z. Huang, Y. Liu, C. Zhan, C. Lin, W. Cai, and Y. Chen, "A novel group recommendation model with two-stage deep learning," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 2021.

[6] J. A. Yang, J. Huang, J. Park, P. T. P. Tang, and A. Tulloch, "Mixed-precision embedding using a cache," *arXiv preprint arXiv:2010.11305*, 2020.

[7] S. Liu, C. Gao, Y. Chen, D. Jin, and Y. Li, "Learnable embedding sizes for recommender systems," *arXiv preprint arXiv:2101.07577*, 2021.

[8] J. B. Schafer, D. Frankowski, J. Herlocker, and S. Sen, "Collaborative filtering recommender systems," in *The adaptive web*. Springer, 2007, pp. 291–324.

[9] Y. Ma, B. Narayanaswamy, H. Lin, and H. Ding, "Temporal-contextual recommendation in real-time," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2020, pp. 2291–2299.

[10] H.-T. Cheng, L. Koc, J. Harmsen, T. Shaked, T. Chandra, H. Aradhye, G. Anderson, G. Corrado, W. Chai, M. Ispir *et al.*, "Wide & deep learning for recommender systems," in *Proceedings of the 1st workshop on deep learning for recommender systems*, 2016, pp. 7–10.

[11] J. Wang, P. Huang, H. Zhao, Z. Zhang, B. Zhao, and D. L. Lee, "Billion-scale commodity embedding for e-commerce recommendation in alibaba," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2018, pp. 839–848.

[12] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[13] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[14] M. Lui, Y. Yetim, Ö. Özkan, Z. Zhao, S.-Y. Tsai, C.-J. Wu, and M. Hempstead, "Understanding capacity-driven scale-out neural recommendation inference," in *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2021, pp. 162–171.

[15] W. Zhao, D. Xie, R. Jia, Y. Qian, R. Ding, M. Sun, and P. Li, "Distributed hierarchical gpu parameter server for massive scale deep learning ads systems," *arXiv preprint arXiv:2003.05622*, 2020.

[16] D. Mudigere, Y. Hao, J. Huang, Z. Jia, A. Tulloch, S. Sridharan, X. Liu, M. Ozdal, J. Nie, J. Park *et al.*, "Software-hardware co-design for fast and scalable training of deep learning recommendation models," *arXiv preprint arXiv:2104.05158*, 2021.

[17] J. Choquette and W. Gandhi, "Nvidia a100 gpu: Performance & innovation for gpu computing," in *2020 IEEE Hot Chips 32 Symposium (HCS)*. IEEE Computer Society, 2020, pp. 1–43.

[18] "Nvidia merlin hugectr." [Online]. Available: https://developer.nvidia.com/nvidia-merlin/hugectr

[19] H. Guan, A. Malevich, J. Yang, J. Park, and H. Yuen, "Post-training 4-bit quantization on embedding tables," *arXiv preprint arXiv:1911.02079*, 2019.

[20] C. Yin, B. Acun, C.-J. Wu, and X. Liu, "Tt-rec: Tensor train compression for deep learning recommendation models," *Proceedings of Machine Learning and Systems*, vol. 3, 2021.

[21] R. Hwang, T. Kim, Y. Kwon, and M. Rhu, "Centaur: A chiplet-based, hybrid sparse-dense accelerator for personalized recommendations," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 968–981.

[22] W. Zhao, J. Zhang, D. Xie, Y. Qian, R. Jia, and P. Li, "Aibox: Ctr prediction model training on a single node," in *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, 2019, pp. 319–328.

[23] M. Naumov, D. Mudigere, H.-J. M. Shi, J. Huang, N. Sundaraman, J. Park, X. Wang, U. Gupta, C.-J. Wu, A. G. Azzolini *et al.*, "Deep learning recommendation model for personalization and recommendation systems," *arXiv preprint arXiv:1906.00091*, 2019.

[24] Y. Ebrahimzadeh Maboud, "Accelerating recommendation system training by leveraging popular choices," Ph.D. dissertation, University of British Columbia, 2021.

[25] S. Kim, G.-I. Yu, H. Park, S. Cho, E. Jeong, H. Ha, S. Lee, J. S. Jeong, and B.-G. Chun, "Parallax: Sparsity-aware data parallel training of deep neural networks," in *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019, pp. 1–15.

[26] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014, pp. 583–598.

[27] H. Guo, W. Guo, Y. Gao, R. Tang, X. He, and W. 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*, 2021, pp. 1269–1278.

[28] A. Tjandra, S. Sakti, and S. Nakamura, "Compressing recurrent neural network with tensor train," in *2017 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2017, pp. 4451–4458.

[29] Z. Qu, L. Deng, B. Wang, H. Chen, J. Lin, L. Liang, G. Li, Z. Zhang, and Y. Xie, "Hardware-enabled efficient data processing with tensor-train decomposition," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021.

[30] I. V. Oseledets, "Tensor-train decomposition," *SIAM Journal on Scientific Computing*, vol. 33, no. 5, pp. 2295–2317, 2011.

[31] X. Wang, L. T. Yang, Y. Wang, L. Ren, and M. J. Deen, "Adtt: a highly efficient distributed tensor-train decomposition method for iiot big data," *IEEE Transactions on Industrial Informatics*, vol. 17, no. 3, pp. 1573–1582, 2020.

[32] A. Novikov, P. Izmailov, V. Khrulkov, M. Figurnov, and I. V. Oseledets, "Tensor train decomposition on tensorflow (t3f)." *J. Mach. Learn. Res.*, vol. 21, no. 30, pp. 1–7, 2020.

[33] O. Hrinchuk, V. Khrulkov, L. Mirvakhabova, E. Orlova, and I. Oseledets, "Tensorized embedding layers," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: Findings*, 2020, pp. 4847–4860.

[34] J. Arai, H. Shiokawa, T. Yamamuro, M. Onizuka, and S. Iwamura, "Rabbit order: Just-in-time parallel reordering for fast graph analysis," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2016, pp. 22–31.

[35] X. Zhou, K. Yang, Y. Xie, C. Yang, and T. Huang, "A novel modularity-based discrete state transition algorithm for community detection in networks," *Neurocomputing*, vol. 334, pp. 89–99, 2019.

[36] H. Shiokawa, Y. Fujiwara, and M. Onizuka, "Fast algorithm for modularity-based graph clustering," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 27, no. 1, 2013.

[37] "Avazu mobile ads ctr." [Online]. Available: https://www.kaggle.com/c/avazu-ctr-prediction

[38] "Terabyte click logs." [Online]. Available: https://labs.criteo.com/2013/12/downloadterabyte-click-logs

[39] "Criteo display ad challenge." [Online]. Available: https://www.kaggle.com/c/criteodisplay-ad-challenge

[40] "Torchrec," 2022. [Online]. Available: github.com/pytorch/torchrec

[41] "Nvidia merlin nvtabular." [Online]. Available: https://developer.nvidia.com/nvidia-merlin/nvtabular

[42] Nvidia, "Dense linear algebra on gpus," developer.nvidia.com/cublas. [Online]. Available: developer.nvidia.com/cublas

[43] P. Covington, J. Adams, and E. Sargin, "Deep neural networks for youtube recommendations," in *Proceedings of the 10th ACM conference on recommender systems*, 2016, pp. 191–198.

[44] S. Okura, Y. Tagami, S. Ono, and A. Tajima, "Embedding-based news recommendation for millions of users," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2017, pp. 1933–1942.

[45] C. A. Gomez-Uribe and N. Hunt, "The netflix recommender system: Algorithms, business value, and innovation," *ACM Transactions on Management Information Systems (TMIS)*, vol. 6, no. 4, pp. 1–19, 2015.

[46] "Accelerating wide & deep recommender inference on gpus." [Online]. Available: https://developer.nvidia.com/blog/accelerating-wide-deep-recommender-inference-on-gpus/

[47] B. Jiang, C. Deng, H. Yi, Z. Hu, G. Zhou, Y. Zheng, S. Huang, X. Guo, D. Wang, Y. Song *et al.*, "Xdl: an industrial deep learning framework for high-dimensional sparse data," in *Proceedings of the 1st International Workshop on Deep Learning Practice for High-Dimensional Sparse Data*, 2019, pp. 1–9.

[48] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, and R. Soricut, "Albert: A lite bert for self-supervised learning of language representations," *arXiv preprint arXiv:1909.11942*, 2019.

[49] K. Weinberger, A. Dasgupta, J. Langford, A. Smola, and J. Attenberg, "Feature hashing for large scale multitask learning," in *Proceedings of the 26th annual international conference on machine learning*, 2009, pp. 1113–1120.

[50] F. Zhang, J. Zhai, X. Shen, O. Mutlu, and X. Du, "POCLib: A high-performance framework for enabling near orthogonal processing on compression," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 2, pp. 459–475, 2022.

[51] F. Zhang, J. Zhai, X. Shen, D. Wang, Z. Chen, O. Mutlu, W. Chen, and X. Du, "TADOC: Text analytics directly on compression," *The VLDB Journal*, vol. 30, no. 2, pp. 163–188, 2021.

[52] F. Zhang, J. Zhai, X. Shen, O. Mutlu, and W. Chen, "Efficient document analytics on compressed data: Method, challenges, algorithms, insights," *Proceedings of the VLDB Endowment*, vol. 11, no. 11, pp. 1522–1535, 2018.

[53] Z. Pan, F. Zhang, Y. Zhou, J. Zhai, X. Shen, O. Mutlu, and X. Du, "Exploring data analytics without decompression on embedded GPU systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 7, pp. 1553–1568, 2021.

[54] F. Zhang, W. Wan, C. Zhang, J. Zhai, Y. Chai, H. Li, and X. Du, "CompressDB: Enabling Efficient Compressed Data Direct Processing for Various Databases," in *Proceedings of the 2022 International Conference on Management of Data*, 2022, pp. 1655–1669.