Accelerating Communication in Deep Learning Recommendation Model Training with Dual-Level Adaptive Lossy Compression

Hao Feng*, Boyuan Zhang*, Fanjiang Ye*, Min Si†, Ching-Hsiang Chu†, Jiannan Tian*, Chunxing Yin†, Summer Deng†, Yuchen Hao†, Pavan Balaji†, Tong Geng‡, Dingwen Tao§

* Indiana University, Bloomington, IN, USA; {haofeng, bozhan, fanjye, jti1}@iu.edu
† Meta, Menlo Park, CA, USA; {msi, chchu, cyin9, summerdeng, haoyc, pavanbalaji}@meta.com
‡ University of Rochester, Rochester, NY, USA; tgeng@ur.rochester.edu
§ SKLP, Institute of Computing Technology, Chinese Academy of Sciences, China; taodingwen@ict.ac.cn

***Abstract*—DLRM is a state-of-the-art recommendation system model that has gained widespread adoption across various industry applications. The large size of DLRM models, however, necessitates the use of multiple devices/GPUs for efficient training. A significant bottleneck in this process is the time-consuming all-to-all communication required to collect embedding data from all devices. To mitigate this, we introduce a method that employs error-bounded lossy compression to reduce the communication data size and accelerate DLRM training. We develop a novel error-bounded lossy compression algorithm, informed by an in-depth analysis of embedding data features, to achieve high compression ratios. Moreover, we introduce a dual-level adaptive strategy for error-bound adjustment, spanning both table-wise and iteration-wise aspects, to balance the compression benefits with the potential impacts on accuracy. We further optimize our compressor for PyTorch tensors on GPUs, minimizing compression overhead. Evaluation shows that our method achieves a 1.38× training speedup with a minimal accuracy impact.**

I. Introduction

Deep Learning Recommendation Models (DLRMs) have significantly risen to prominence in both research and industry sectors in recent years. These models integrate sparse input embedding learning with neural network architectures, marking a notable advance over traditional collaborative filtering-based recommendation systems [1]. DLRMs have been successfully implemented in various industry applications, including product recommendations system by Amazon [2], personalized advertising by Google [3], and e-commerce service by Alibaba [4]. As a result, they constitute a significant portion of deep learning applications across multiple industries.

DLRMs are uniquely designed to process high-dimensional categorical features, typically represented by one- or multi-hot vectors matching the size of the category, which leads to significant data sparsity. To efficiently manage this, DLRMs utilize embedding tables that transform these high-dimensional sparse vectors into lower-dimensional, dense vector representations. In a typical DLRM architecture, dense features are processed through a multi-layer perceptron (MLP), combined with sparse embedding lookups in a feature interaction module, and then fed into the top MLP. This process culminates in generating a click-through rate (CTR) prediction. Such a structure elegantly combines sparse and dense data processing, underscoring the complexity and challenges associated with the efficient implementation and scaling of DLRMs.

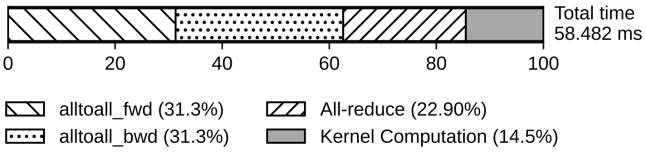


Fig. 1: Performance profiling of DLRM training with 32 GPUs.

A critical challenge in deploying large-scale DLRMs lies in managing the massive size of embedding tables, which can extend to terabytes, far exceeding the memory capacity of a single GPU. To address this issue, hybrid-parallel distributed training systems are widely employed. In these systems, MLP layers are replicated across multiple GPUs for data-parallel training, while embedding tables are partitioned and distributed for model-parallel training. This setup necessitates the use of collective communication primitives for synchronization across all GPUs. Specifically, the partitioning of sparse embedding tables requires nodes to aggregate sparse embedding lookups during forward passes and their corresponding gradients during backward passes. Consequently, **all-to-all** communication is utilized in both forward and backward passes for synchronizing sparse lookups and gradients, while all-reduce is employed for synchronizing dense/MLP gradients during the backward pass.

Communication for synchronizing embedding lookups and gradients across all GPUs during each mini-batch iteration significantly contributes to the overall training time. For example, Figure 1 shows that all-to-all communication accounts for more than 60% of the total training time for DLRM on an 8-node, 32 A100 GPUs cluster (connected through a Slingshot 10 interconnect [5]). Consequently, various studies have been conducted to address these communication challenges.

One method involves the application of low-bit quantization (e.g., FP16, FP8) to represent embedding tables [6]. However, quantization has two primary limitations: ❶ Its capacity for data reduction (e.g., 2×) is relatively limited. ❷ While

Hao Feng and Boyuan Zhang contribute equally in this work.
Jiannan Tian's participation in this work ended on 03/26/2024.
Dingwen Tao is the corresponding author.

SC24, November 17–22, 2024, Atlanta, GA, USA
979-8-3503-5291-7/24/$31.00 ©2024 IEEE

quantization is viable for inference, training with a quantized embedding table often results in significant accuracy losses [7]. Another approach is the use of lossless compression to compress embedding lookups just before the all-to-all communication [8]. However, this method also faces challenges due to the sparse and random nature of embedding lookups and the mantissa of floating-point data, which limits the achievable compression ratio.

Unlike quantization and lossless compression approaches, error-bounded lossy compression achieves a significantly higher data reduction ratio while maintaining strict error control in the reconstructed data. However, effectively employing lossy compression in DLRM training necessitates addressing several key challenges: ❶ **Low Compression Ratio:** Existing error-bounded lossy compression methods, such as SZ [9] and ZFP [10], also face the challenge of achieving a high compression ratio on embedding lookups (which will be explained in Section III). Thus, it is essential to develop a lossy compression algorithm optimized for embedding lookups. ❷ **High Compression Overhead:** Compression for every all-to-all communication at each iteration introduces a high compression overhead; thus, implementing an efficient compression algorithm on GPUs and seamlessly integrating it into both the communication and DLRM computation workflows is vital, ensuring minimal performance overhead. ❸ **Error Propagation:** Lossy compression introduces errors to the reconstructed embedding lookups after all-to-all communication. Thus, developing a strategy for adaptively controlling error bounds across different embedding tables and training iterations is critical to ensure an acceptable impact on accuracy.

To address these challenges, we introduce a highly efficient approach to accelerate communication in DLRM training through the use of error-bounded lossy compression, deeply optimizing and adaptively applying it.

Our key contributions include:

- We introduce a novel hybrid compression method for embedding lookups/vectors, consisting of two algorithms: a newly developed LZ compression algorithm for embedding vectors and an optimized entropy-based Huffman compression algorithm for vector elements.
- We develop a two-level adaptive strategy for error-bound adjustment for different embedding tables and training iterations, aiming to maintain relatively large error bounds (for higher compression ratios) while minimizing the impact on accuracy.
- We optimize our compression method on modern GPUs, enabling parallel compression of multiple tensors into a single compressed tensor, effectively minimizing data movements and kernel launches.
- We evaluate our method using three widely used DLRM datasets with up to 32 GPUs and demonstrate that our method significantly accelerates all-to-all communication in DLRM training by 8.6×, with an accuracy loss of less than 0.02%—well within the tolerable level.

II. Background and Problem Statement

A. Deep Learning Recommendation Model (DLRM)

DLRM is a widely used recommendation model, which is designed to utilize both categorical and numerical inputs for personalized recommendations. We will discuss the architecture, pipeline, and large-scale training of DLRM.

DLRM Architecture Generally, DLRM comprises three components: Embedding tables, Interaction Module, and MLP. The architecture is shown in Figure 2. Embedding tables in DLRM process categorical data by looking up each categorical feature and mapping it into an embedding vector representation as the output vector. The Interaction Module will apply input vectors from the embedding tables and the Bottom MLP, and perform interactions on them to generate a new output vector. There are two MLPs in DLRM: the Bottom MLP and the Top MLP. The Bottom MLP transforms dense features to match the length of embedding vectors. The Top MLP applies the concatenated data as input and calculates the Click-Through Rate (CTR) as output.

DLRM Training Parallelisms and Bottlenecks Training DLRM involves both model and data parallelism to manage its diverse computational needs efficiently. Model parallelism is crucial for handling the large Embedding Tables (EMBs) distributed across different devices due to their size, while data parallelism is applied to the Multilayer Perceptrons (MLPs), which, despite requiring full access every epoch, consume a relatively small amount of memory. This dual approach allows for the division of a global batch of embedding vectors into smaller local batches, facilitating interaction operations and subsequent processing by the Top MLP through an all-to-all communication pattern. This setup is particularly effective for implementing various second-order interaction methods, including dot products between pairs of embedding vectors and dense features, which are then concatenated with dense features for input to the Top MLP, ultimately enabling classification. During backward propagation, a symmetrical all-to-all communication redistributes gradients back to their respective devices for updating the Embedding tables and Bottom MLP, reflecting the forward phase's operations.

The bottlenecks across DLRM's components vary: embedding layers are bandwidth-dominated due to high-bandwidth memory access requirements for lookup operations, MLP layers are computation-dominated, and the feature interaction layer is communication-dominated. Our profiling indicates that DLRM training is notably communication-intensive, underscoring the necessity of optimizing these parallelism strategies for large-scale training efficiency (see details in Section IV).

B. Floating-point Data Compression

There are two principal classes of data compression: lossless and lossy. While lossless compression preserves data integrity perfectly, lossy compression achieves significantly higher compression ratios at the cost of acceptable accuracy loss. Traditional lossy compressors, such as JPEG [11] for images and MPEG [12] for videos, are designed with human

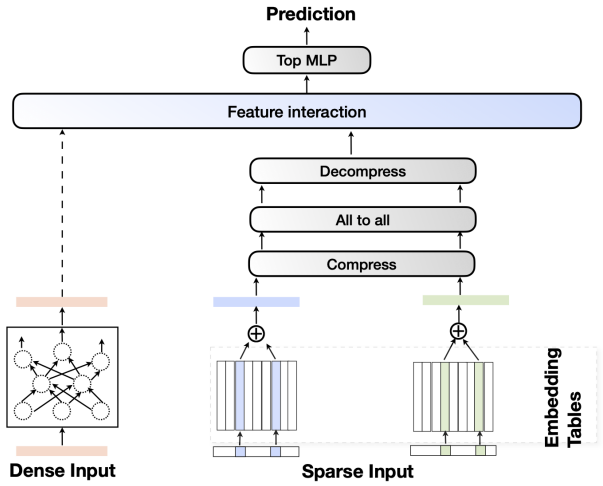


Fig. 2: Illustration of DLRM architecture.

perception in mind, lacking precise error-control mechanisms for scientific post-analysis.

A new generation of lossy compression techniques for scientific data, particularly floating-point data, has emerged, exemplified by SZ [9, 13, 14], ZFP [10], and TTHRESH [15]. Unlike their counterparts for media, these scientific data compressors offer strict error-controlling schemes, enabling users to manage accuracy loss both in the reconstructed data and during post-analysis.

With the proliferation of GPU-based HPC systems and applications, compressors like SZ and ZFP have introduced GPU-optimized versions (i.e., cuSZ [16] and cuZFP [10]), delivering significantly enhanced throughput compared to their CPU-based implementations. ZFP, a transform-based compressor, allows users to set a desired bitrate, while SZ, a prediction-based compressor, enables the specification of a maximum tolerable error. ZFP in fixed-rate mode tends to offer consistently higher throughput, whereas SZ in error-bounded mode achieves superior compression ratios.

C. Problem Statement

Dominance of Communication Overhead. As illustrated in Figure 1, all-to-all communication accounts for over 60% of DLRM's total training time, establishing communication as the bottleneck rather than computation. This bottleneck is exacerbated when training DLRM with datasets of various sizes. For instance, the Criteo Kaggle dataset, with sparse feature lengths of 32, generates more than 121 GB of lookup data per epoch. This figure escalates dramatically with larger datasets, such as the Criteo Terabyte dataset, which can accumulate up to terabytes of lookup data per epoch. The scale increases further with industry-level recommendation models, often exceeding multiple terabytes, necessitating larger volumes of training data and distributed systems of larger scales for parameter storage. This significantly increases communication data across devices, highlighting the urgent need to reduce communication data volume in DLRM training.

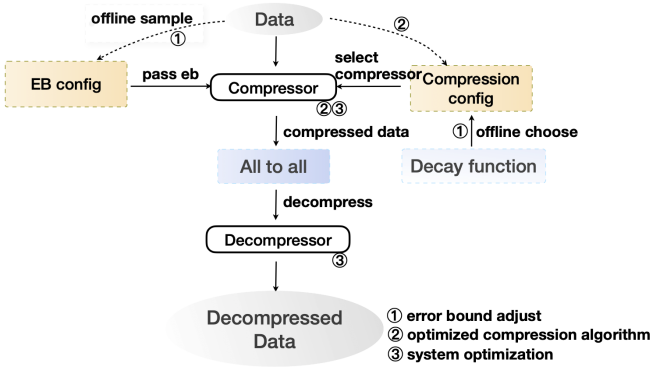


Fig. 3: Overview of proposed DLRM training framework

However, as aforementioned, there are several challenges to addressing this issue due to the limitations of directly applying current error-bounded lossy compression techniques. To effectively tackle these hurdles, this paper focuses on the following key research questions: ❶ **Error Bound Configuration:** It's essential to determine the optimal error bounds for various embedding tables across different training iterations to maintain training accuracy while enhancing compression ratios. ❷ **DLRM-specific Lossy Compression Algorithm:** There's a critical need to devise a compression algorithm uniquely suited for DLRM's embedding tables, aiming for elevated compression ratios without substantially compromising data integrity. ❸ **GPU Compression Performance Optimization:** Optimizing the GPU execution of our specialized compression and ensuring its smooth incorporation within DLRM training is crucial for improving overall training performance.

III. System Design

In this section, we discuss our approach to accelerating DLRM training, divided into four main parts. First, we provide an overview of our proposed training pipeline that incorporates lossy compression in Section III-A. Second, we share our observations of DLRM data features in Section III-B. Following this, we explain how we dynamically adjust our error bound during training. Lastly, we introduce our optimized compression algorithm designed to enhance performance.

A. Overview of DLRM Training Pipeline with Compression

First, we present the complete framework, showcasing the overview pipeline in Figure 3. This pipeline can be divided into two main components: an offline analysis process and a training pipeline that incorporates lossy compression.

Offline Analysis. The purpose of this step is to obtain an optimized configuration by sampling and analyzing some iterations from the original training process. There are two tasks involved in offline analysis: Compressor Selection and Embedding Table Classification. In the Compressor Selection task, we evaluate various compressors using sampled data to select the best one for the current system. In the Embedding Table Classification task, we analyze the characteristics of embedding tables using sampled data and classify them according to different error-bound adjustment strategies.

Training Pipeline with Compression. Our proposed training process incorporates lossy compression into all-to-all communications. Unlike fix-rate compression, error-bounded compression does not maintain a consistent compression ratio, making our pipeline distinctive by accommodating variable-size all-to-all communication. The pipeline is organized into four primary stages: ① Compressing data on each device; ② Sending metadata through all-to-all communication; ③ Transmitting compressed data via all-to-all communication; and ④ Decompressing data on each device for training. Stages ① and ④ introduce additional steps for compression and decompression, respectively, where we employ our online error bound adjustment strategy to dynamically tune the error bound. Stage ② addresses the challenges of executing variable-size all-to-all by managing metadata, including the size of compressed data and compressor specifications.

B. Observation and In-Depth Analysis of Embedding Vector

❶ **False Prediction**. Prediction is a crucial technique in lossy compression algorithms like SZ [9, 13], leveraging spatial correlations among data points to estimate the value of a point based on its neighbors, as seen with the Lorenzo predictor we mentioned. This approach is effective in many scientific datasets where floating-point numbers represent real-world phenomena, thanks to substantial spatial correlation. However, DLRM embedding vectors are markedly different from scientific data. In a batch of embedding vectors, the spatial correlation is minimal, both within individual vectors and among neighboring ones. It is attributed to the independence of data points across dimensions within an embedding vector and the random order of the vectors. In contrast, the use of prediction can even result in *False Prediction* (illustrated in Figure 4), a phenomenon we will elaborate on subsequently.

❷ **Vector Homogenization**. This phenomenon, stemming from quantization and precision loss, significantly impacts data representation. Note that repeated vectors occur not only in original EMB vectors but also increase after quantization, as depicted in Figure 4. Within a lossy compression algorithm, two distinct floating-point values within an error-bound range can be considered identical, leading to two EMB vectors being treated as the same if their values at each dimension are sufficiently similar. We term this occurrence *Vector Homogenization*, where similar vectors are transformed into more repetitive ones. Our findings indicate that this phenomenon is more pronounced in certain tables compared to others, attributed to the unique data characteristics of those tables.

❸ **Gaussian Distribution of Data Values**. In our analysis of the distribution of embedding vectors across different embedding tables, we observe that the distributions tend to vary between Gaussian and uniform, contingent upon the specific table. Embedding tables characterized by significantly unbalanced query frequencies are more inclined to demonstrate a Gaussian distribution. This is attributed to repeated vectors, which result in certain values appearing more frequently, hence deviating the distribution from a uniform to a Gaussian pattern.

EMB Table ID	1	3	4
False Prediction	✓	✓	✓
Violently Vector Homogenization	✓	×	×
Gaussian Distribution	✓	✓	×

TABLE I: Characteristics of their representative EMB tables from Criteo Kaggle dataset.

C. Adaptive Fine-Grain Error-Bound Adjust Strategy

Next, we discuss our two-level adaptive strategy for selecting the error bound to ensure the high accuracy of the trained model. As the error bound increases, the compression ratio also increases, but the precision of the decompressed data decreases. There exists a trade-off between reducing communication data size and preserving original information. To address this, we introduce an adaptive strategy for choosing the error bound along two dimensions.

❶ **Iteration-wise Configuration**. This approach involves gradually decreasing the error bound over iterations, akin to adjusting the learning rate during model training. Two well-recognized insights support this strategy: first, applying different learning rate schedules can yield diverse convergence outcomes for the same model. Second, optimizers do not guarantee an optimal direction for gradient optimization; instead, they aim for a sub-optimal approach, meaning they can tolerate noise on the ideal gradient. Viewing the effects of lossy compression as introducing noise to intermediate data, this noise affects calculations, impacting the gradient during backward propagation. Given that controllable noise does not result in model non-convergence, the key consideration is the acceptable noise amplitude. Similarly to how the learning rate determines the optimization step size, the error bound can be adjusted over time. In the early stages of training, a larger error bound does not hinder convergence. However, as training advances and optimization steps require greater precision, tightening the error bound becomes necessary to limit the noise's impact.

Specifically, we divide the training period into two phases: an **initial phase**, characterized by rapid loss reduction, and

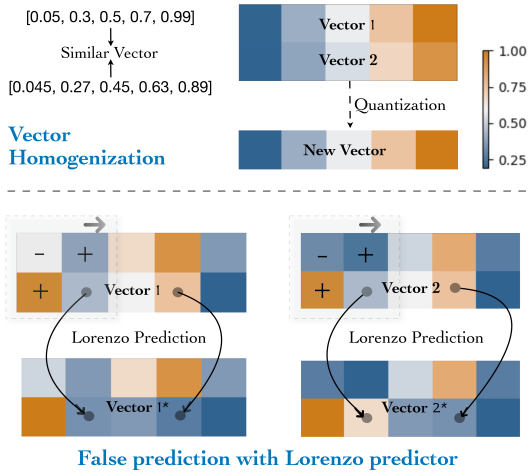
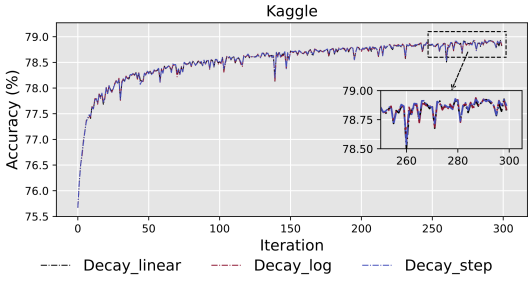
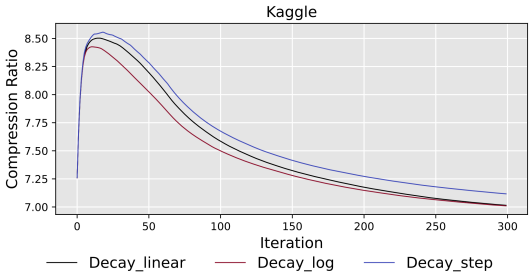


Fig. 4: Illustration of observed Vector Homogenization and false prediction with Lorenzo predictor.



(a) Accuracy



(b) Compression Ratio

Fig. 5: Accuracy and CR with different decay functions.

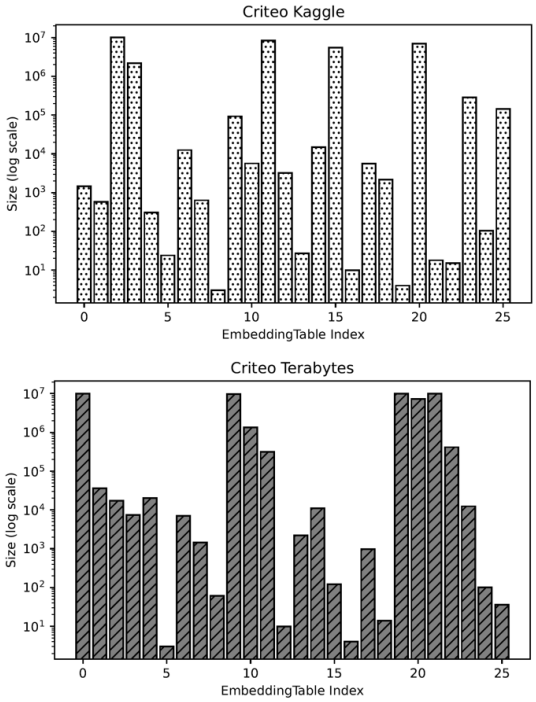


Fig. 6: EMB table sizes in Criteo Kaggle and Terabytes datasets.

a **later phase**, where the loss tends to stabilize. During the initial phase, we gradually decrease the error bound via a predefined decay function (e.g., logarithmic, stepwise) to minimize deviation from the original data, facilitating swift model convergence. In the later phase, we maintain a consistent error bound to ensure the model converges effectively. According to experiment results in Figure 5, it demonstrated that a step-wise (staircase descent) decay function offers the greatest compression benefits while ensuring model convergence. In that case, we select step-wise decay as the default decay function.

❷ **Table-wise Configuration**. Different embedding tables necessitate distinct error bounds, a principle stemming from the inherent properties of the lossy compression algorithm: data quality varies with different characteristics even under identical compression ratios. Embedding vectors within tables symbolize items with diverse semantic meanings. Given the wide variation in embedding table sizes—ranging from fewer than ten to over a million, as depicted in Figure 6—the data characteristics among embedding tables significantly diverge.

Specifically, the phenomenon of Vector Homogenization is pivotal for setting error bounds tailored to each embedding table, accounting for their unique semantic representations. This necessitates assigning distinct error bounds to ensure uniform quality across tables. Hence, we introduce the Homogenization Index (written in Homo Index in the following text), a metric to assess the quality of embedding tables. This index spans from 0 to 1, where 0 indicates no homogenization and 1 denotes complete vector homogenization into a singular vector. The Homogenization Index is calculated as follows:

$$\eta = \frac{N_{\text{original}} - N_{\text{compressed}}}{N_{\text{original}}}. \quad (1)$$

Subsequently, embedding tables are categorized into three groups based on their Homo Index, corresponding to three levels of error bounds: Large, Medium, and Small.

Distinguishing Error Bound and Homo Index. While the error bound for DLRM embedding vectors primarily addresses point-wise error, functioning as a black-box metric, the Homogenization Index (Homo Index) sheds light on the compressed embedding vectors' quality, facilitating adaptive error bound adjustments according to specific requirements. Note that Error Bound and Homo Index serve different purposes and operate on distinct levels: the former at the *point-wise level* and the latter at the *vector-wise level*.

Overall, we detail our proposed adaptive error bound adjustment strategy, as outlined in Algorithm 1. In pseudo-code, line 1 defines the error-bound parameter to adjust. Lines 2 to 7 define hyper-parameter to adjust the error bound. Homo index calculation in line 11 refers to Equation (1).

D. Optimized Compression Algorithm for DLRM Data

After identifying the optimal error bound for training, the next step involves fine-tuning the compressor within that error bound. Compressors typically aim to balance achieving a high compression ratio with maintaining high compression speed. The ideal compressor for our purposes should satisfy three criteria: ❶ Operate on the GPU to avoid data transfers between the device and host since embedding vectors reside on the GPU. ❷ Offer high compression throughput to minimize compression overhead and, consequently, accelerate overall DLRM training. ❸ Achieve a high compression ratio to maximize the benefits of reduced data volume during communication. To meet these requirements, we propose an optimized hybrid error-bounded lossy compression algorithm tailored specifically for DLRM data.

```
Global EBConf = {}
Global LargeEB ← GlobalEB × α
Global SmallEB ← GlobalEB ÷ β
Global MediumEB ← GlobalEB
Global L_EMB_hindex, S_EMB_hindex
Global DecayPhase
Global func DecayFunc

func OfflineAnalysis():
    for t in EMB_Tables:
        sd ← sampleData(t)
        hIndex ← homoIndexCal(sd)
        c ← EMBClassification(hIndex)
        if c is 'large': EBConf[t] ← LargeEB
        if c is 'small': EBConf[t] ← SmallEB
        else EBConf[t] ← MediumEB

func OnlineDecay(iter: i):
    for t in EMB_Tables:
        if i ∈ DecayPhase:
            EMBConf[t] ← DecayFunc(i) * EMBConf[t]
func EMBClassification(hindex):
    if hindex > S_EMB_hindex:
        return 'small'
    elif hindex < L_EMB_hindex:
        return 'large'
    else: return 'medium'
```

Algorithm 1: Proposed Error Bound Adjustment Strategy

Our algorithm comprises two main components: a quantization encoder and a lossless encoder. Initially, the quantization encoder converts floating-point numbers into discrete bins, representing them as integers. These integers are then compressed using a hybrid method that incorporates two types of lossless encoders: LZ encoder [17][18] and Entropy encoder such as Huffman encoder [19].

Vector-based LZ Encoding. The phenomenon of **unbalanced queries**, as illustrated in numerous studies [20], is pivotal in DLRM training. The imbalance in query frequency implies that recognizing frequently recurring queries can dramatically boost the compression ratio. A distinct feature of repetitive patterns in DLRM applications is the consistency of bytes within an embedding vector for repeated queries, independent of the vector's size. This indicates that the length of the repeating pattern is predetermined and constant. While traditional LZ algorithms are designed to identify repeating patterns of varying lengths, we propose to refine the LZ compression algorithm specifically for DLRM by introducing vector-based LZ compression. This innovation significantly diminishes data volume and amplifies the compression ratio for certain embedding tables through effective pattern recognition.

Optimized Entropy Encoding. Our design of the optimized compression algorithm is informed by two critical observations. The first, observation ❸, underscores the effectiveness of an entropy-based compressor, such as Huffman encoding, given the high entropy typically exhibited by such data. The second, observation ❶, highlights that identical vectors within the same batch might be surrounded by different neighboring vectors, which can lead to divergent predictions for vectors that are initially identical. This phenomenon not only risks misrepresentation and loss of identical embedding vectors but also elevates the data's entropy. An example depicted in Figure 4 (right part) illustrates how employing a 2x2 Lorenzo predictor [21] on embedding vectors can transform identical vectors into distinct ones. These insights lead to the conclusion that traditional prediction techniques are ill-suited for our DLRM training-specific compression algorithm.

Selection Between Two Encoders. During the offline analysis phase, we sample data and evaluate the two encoders to identify the most effective one. Given the complexity of many systems, it is not justifiable to compare compressors solely based on compression ratio or throughput. In our proposed compressor, we utilize a sample-based speed-up approximation to determine the optimal compressor. Equation (2) illustrates the method for estimating theoretical speed-up. In this equation, CR denotes the compression ratio, B represents network bandwidth, and T_c and T_d refer to the compression and decompression throughputs, respectively.

$$speedup = \frac{1}{\frac{1}{CR} + B \times (\frac{1}{T_c} + \frac{1}{T_d})} \tag{2}$$

Overall, in our hybrid compression framework, we use this formula to pinpoint the most efficient compressor that maximizes speed-up for the given system configuration. We present the detail in Algorithm 2. In this pseudo code, Line 1 defines the compressor selection parameter. Line 2 defines the hyper-parameter for alternative compressors. The speedup calculation in line 6 refers to Equation (2). Although theoretically any compression algorithm could be included in our selection pool, for simplicity and effectiveness, we limit our final design to these two encoders.

```
Global TableCompressorConfig = {}
Global Compressors = {}
func OfflineCompConfig(e):
    for t in EMB_Ta\ttfamilybles:
        d ← sampleEMBData(t)
        speedups ← SpeedUpCompute(d, Compressors)
        maxSup, bestCompressor = Max(speedups)
        TableCompressorConfig[t] ← bestCompressor
```

Algorithm 2: Compressor Selection

E. System Implementation and Performance Optimization

Finally, we detail our system implementation and performance optimization, focusing on fine-tuning the vector-based LZ compressor and GPU compression kernels.

Compressor Fine-tuning. First, we refine our vector-based LZ compression algorithm, distinguishing it from the original LZ approach in two significant ways. ❶ *Extended window size*: Traditional LZ algorithms typically opt for a relatively small window size, like 4KB/8KB. However, given the longer repeated patterns observed in DLRM applications, with lengths ranging from 128-256 bytes (considering an embedding vector length of 32/64 with 32-bit floating point data type), we extend the window size to accommodate these patterns. ❷ *Fixed pattern length*: To minimize memory access and byte comparison time, we introduce fixed-length pattern matching. In contrast to the standard LZ process, which advances the

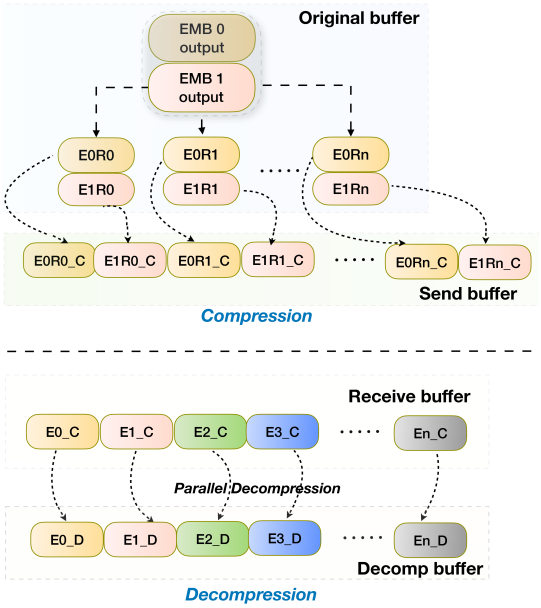


Fig. 7: Proposed buffer optimization resulting in single-kernel compression (top) and parallel decompression (bottom).

pointer to the next byte in the absence of a match, our knowledge of the repeated pattern length allows us to leap forward many bytes in search of the next match, thereby avoiding fruitless comparisons and pointer movements. If the initial bytes of two embedding vectors differ, further comparison is unnecessary. This optimization also forces the compressor to match longer patterns rather than shorter patterns. Compared to the current state-of-the-art (SOTA) GPU lossless compressor nvCOMP-LZ4, our approach achieves 2.72× and 4.88× higher compression ratios on two datasets, respectively.

Buffer Optimization. Next, we implement multi-threading to minimize memory copy overhead and speed up decompression. Typically, compressors output compressed data to a memory chunk and return a pointer, a versatile but overhead-inducing solution. Given the all-to-all collective communication in DLRM training, where each data chunk must be compressed separately for transmission to each rank, this method introduces unnecessary memory copying since compressed data may not be stored contiguously. Besides, launching kernel multiple times introduces overhead of a lot of kernel launching. To address these, we optimize our compressor that not only introduces one kernel launching but also writes directly to the sending buffer. Figure 7 shows the workflow of compression and decompression in buffer optimization. In the compression process, we involve synchronization and Atomic Add to get the writing offset of each chunk. Furthermore, we leverage multi-threading for compression and decompression, partitioning the compressed data into multiple chunks for simultaneous processing. Although GPU compressors typically achieve high resource utilization, some operations cannot fully leverage GPU resources. Thus, executing multiple decompression kernels in parallel is faster than serial decompression.

IV. Experimental Evaluation

A. Experimental Setup

Platform, Software, and Dataset. Our experimental platforms include a workstation with two NVIDIA A4000 GPUs (16GB memory each) and an HPC cluster comprising 8 GPU nodes. Each node of the cluster is equipped with 256 GB of memory, a 64-core 2.0 GHz 225-watt AMD EPYC 7713 processor, and four NVIDIA A100 GPUs (40GB memory each). Experiments were conducted using PyTorch 2.0.1, CUDA 11.7, and NCCL 2.14.3. We employed the Criteo Ad Kaggle dataset [22] and the Criteo Terabyte dataset [23] for our experiments. They feature 13 continuous and 26 categorical features, totaling about 45 million samples over 7 days.

Baselines. To evaluate our design, we compare it against three baselines that focus on reducing communication data volume: ❶ The open-source release version of DLRM [1], serving as the original DLRM training baseline. ❷ The low-precision approach, is a straightforward method to reduce communication data volume. Various works [24, 25] have demonstrated the feasibility of DLRM training with FP8 data type, making this approach a SOTA solution for reducing communication overhead. ❸ We also use nvCOMP [26], a software developed by Nvidia that integrates several SOTA lossy and lossless compressors, as a baseline. We compare the compression ratio and throughput between nvCOMP and our optimized compressor.

EMB Tables Classification. We apply our proposed Homogenization Index to classify embedding tables into three categories, as detailed in Table II. These categories correspond to large, medium, and small error bounds, denoted as L, M, and S, respectively. In Table III and Table IV, we select some representative EMB Table to show how Homo Index ranking achieves EMB Table Classification.

B. Evaluation of Error Bound Adjustment Strategy

To assess the effects of lossy compression on model accuracy, we evaluate the model prediction accuracy of our training method compared with original DLRM training using both FP32 and FP16 precisions, alongside a SOTA low precision approach employing 8-bit quantization. Current standards deem an accuracy loss within 0.02% as acceptable in production models [27].

Figure 8a and figure 8b display a comparison of accuracy convergence curves and accuracy delta curves across these methods, respectively. Through extensive experimentation, we determine and apply a suitable fixed global error bound (i.e., 0.02) for each model to guarantee convergence. On the two datasets we utilized, the average prediction accuracy losses are 0.0031% and 0.0042%, respectively.

Table-wise Error Bound Adjustment. Next, to evaluate the effectiveness of our table-wise error bound adjustment strategy, we evaluate the model prediction accuracy and compression ratio throughout the training process, comparing the use of a fixed global error bound against tailored error bounds for different embedding tables.

TABLE II: Classification of EMB tables on test datasets.

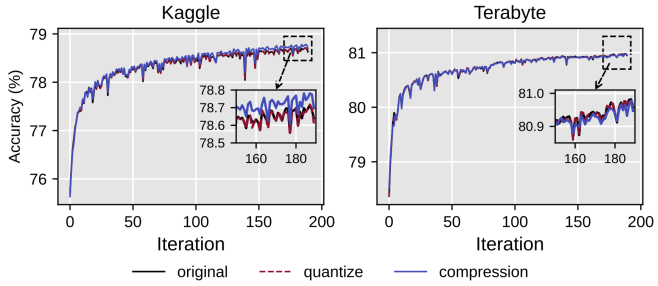
EMB ID	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
Kaggle	M	M	S	S	M	M	M	M	L	S	M	S	M	M	M	S	L	M	M	L	S	L	L	S	L	S
Terabytes	S	M	M	M	M	L	M	M	L	S	S	M	L	M	M	L	L	L	L	S	S	S	S	M	L	L

TABLE III: Ranked Homo Index on Criteo Kaggle dataset.

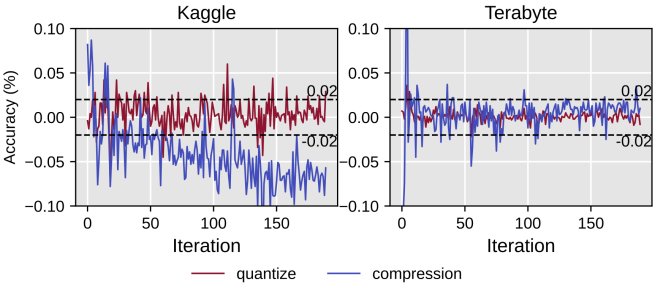
TAB. ID	EB.	# Ori. Patterns	# Quant. Patterns	Batch Size	Homo Index
20	0.01	110	68	128	0.618182
11	0.01	110	69	128	0.627273
2	0.01	110	73	128	0.663636
15	0.01	108	76	128	0.703704
3	0.01	103	86	128	0.834951
23	0.01	84	77	128	0.916667
25	0.01	67	63	128	0.940299
0	0.01	19	19	128	1
1	0.01	61	61	128	1

TABLE IV: Ranked Homo Index on Criteo Terabytes dataset.

TAB. ID	EB.	# Ori. Patterns	# Quant. Patterns	Batch Size	Homo Index
0	0.005	1055	484	2048	0.458768
19	0.005	1072	576	2048	0.537313
21	0.005	1042	623	2048	0.597889
9	0.005	1025	621	2048	0.605854
20	0.005	937	795	2048	0.848453
1	0.005	983	983	2048	1
2	0.005	1302	1302	2048	1



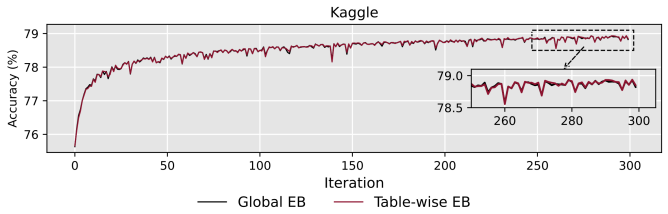
(a) Accuracy



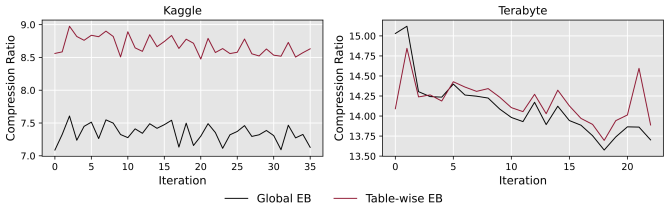
(b) Delta accuracy

Fig. 8: (a) Accuracy and (b) delta accuracy (versus baseline) with different compression methods.

As illustrated in Table II, embedding tables are classified into three categories according to their Homogenization Index scores, with assigned error bounds of 0.01, 0.03, and 0.05, respectively. The accuracy evaluation, detailed in Figure 9a, reveals that our approach, which applies specific error bounds to different tables rather than a uniform global error bound, maintains the model's accuracy intact. Additionally, this method achieves a higher compression ratio, up to 1.21× on the Criteo Kaggle dataset, compared to the fixed global error bound strategy.



(a) Accuracy



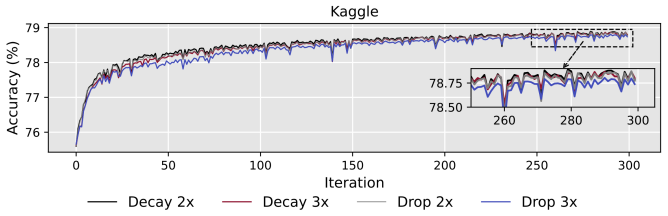
(b) Compression Ratio

Fig. 9: Accuracy and compression ratio of our method with proposed table-wise EB configuration strategy on different datasets. e.g. subgraph(b) represents compression ratio on embedding table 0

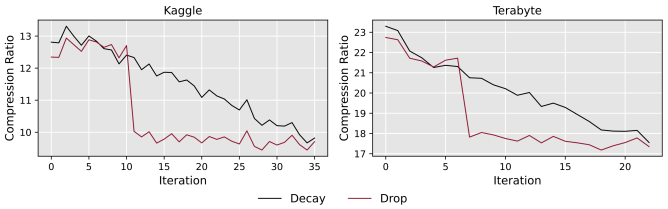
Error Bound Decay Strategy. Furthermore, to evaluate the effectiveness of our error bound decay strategy, we compare model prediction accuracy and compression ratio throughout the training process using two distinct day approaches: a more aggressive method that abruptly reduces the error bound at a predetermined moment and a gradual approach that decreases the error bound according to a decay function.

As we discussed in III-C. This experiment's results of different compressors led us to employ the step-wise function for comparison against the aggressive adjustment approach. Comparative experiments illustrated in Figure 10 reveal how the model training and compression ratios are affected when the error bound is reduced from twice and three times the conservative error bound down to the conservative error bound.

Our analysis indicates that while starting with a larger error bound at the beginning of training and aggressively reducing it can hinder model convergence, a gradual decrease promotes convergence. The comparison of compression ratios clearly shows that, compared to a sharp reduction, our error bound decay strategy allows for starting from a much larger error



(a) Accuracy



(b) Compression Ratio

Fig. 10: Accuracy and compression ratio of our method with two decay methods on different datasets. Decay_2x represents the error bound decayed from 2 times base error bound to base error bound; Drop_2x represents the error bound dropped from 2 times base error to base error bound at the end of the initial phase. E.g. subgraph (b) represents the compression ratio on embedding table 1.

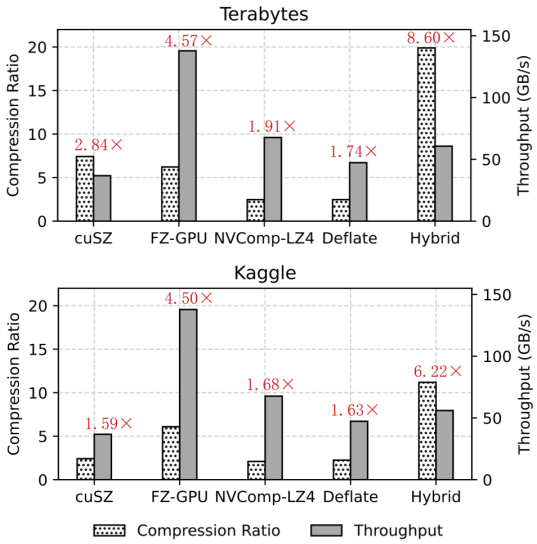


Fig. 11: Compression ratio, throughput, and communication speedup of different compression methods on different datasets. Batch size = 128 (Kaggle), 2048 (Terabytes).

bound, which is then gradually reduced over time. As a result, this approach yields further 1.09× and 1.03× higher compression ratios (i.e., 1.32× and 1.06× over the fixed global error bound solution) on Criteo Kaggle and Criteo Terabytes datasets, respectively, delivering more significant benefits.

Based on our evaluations, we choose *LargeEB: 0.05, MediumEB: 0.03, SmallEB: 0.01, Decay Func: stepwise* as the optimal error bound configuration for subsequent evaluations.

C. Evaluation on Compression Performance

In this section, we evaluate the overall compression performance including compression ratio and throughput of different compressors on different DLRM datasets and models.

Overall Compression and Communication Performance. Figure 11 illustrates the average compression ratio and throughput during DLRM training. Each sub-graph's left bars depict the average compression ratio of each compressor for a given dataset, while the right bars display each compressor's throughput of compression and decompression. As aforementioned, our hybrid compressor includes two compression algorithms: our vector-based LZ compression algorithm and our optimized entropy compression algorithm.

The results indicate that, for the datasets utilized in DLRM training, our hybrid compressor outperforms other compressors in terms of compression ratio and achieves exceptionally high throughput. It reaches an overall 11.2× and 19.9× compression ratio on Criteo Kaggle and Criteo Terabytes, respectively. Our two proposed compressors, vector-based LZ can reach 40.5GB/s in compression, and 205.4GB/s in decompression, while the optimized entropy compressor can reach 78.4GB/s in compression, and 38.9GB/s in decompression. Compared to the SOTA lossless compressors nvCOMP LZ4 our compressor achieves a compression ratio 5.3× and 8.1× higher on two datasets respectively. The other SOTA nvCOMP Deflate achieves a similar compression ratio to nvCOMP LZ4 but compression and decompression throughput are 30.1GB/s and 109.7GB/s. The SOTA lossy compressor FZ-GPU [28] has the highest throughput which is over 136GB/s in both compression and decompression, since it relies on a very fast encoder (i.e., bitshuffle and sparse encoding) [28]. However, its compression ratio is significantly lower than our hybrid compressor attained, leading to a greater overall speedup in end-to-end communication.

As shown in Figure 11, our proposed hybrid compressor achieves a 6.22× and 8.6× speedup for two different datasets in all-to-all communication, surpassing all other approaches when all-to-all communication throughput is 4GB/s.

DLRM End-to-end Performance Speed-up Figure 12 shows the breakdown of DLRM training with lossy compression on our cluster with 32 A100 GPUs. Our compression accelerates all-to-all communication in forward propagation which takes 31.3% proportion of the whole training time. On the Criteo Kaggle dataset, our proposed compressor achieves 6.22× overall speed-up in communication and 1.30× in end-to-end training, by reducing all-to-all in forward propagation cost to 5.03%. On the Criteo Terabytes dataset, our compressor achieves 8.6× and 1.38× in all-to-all communication and end-to-end training, respectively.

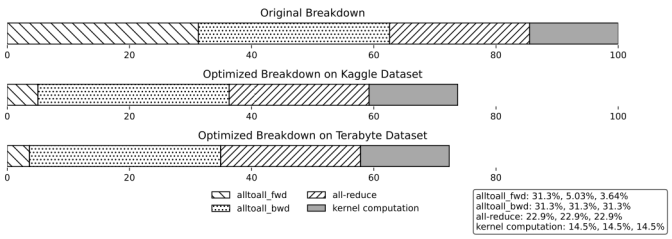


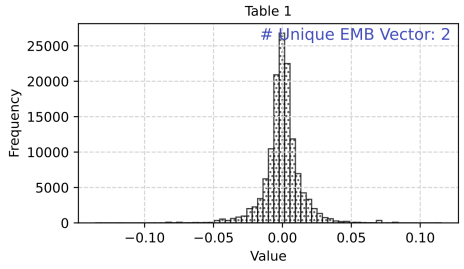
Fig. 12: Breakdown of optimized end-to-end DLRM training time on different DLRM datasets.

TABLE V: Compression ratio of different compressors on the two datasets (left: Criteo Kaggle, right: Terabytes). Bolded numbers indicate the highest compression ratios among compressors.

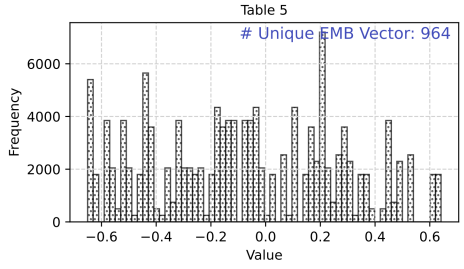
EMB Table ID	cuSZ		FZ-GPU		Ours-Vector GPULZ		Ours-Huffman		Baseline NVComp-LZ4		Baseline Deflate		Huffman+ GPULZ hybrid	
0	2.40 /	6.40	5.84 /	5.14	7.58 /	**14.94**	**8.74** /	**10.94**	5.87 /	1.62	6.32 /	1.67	8.74 /	14.94
1	2.40 /	9.90	6.45 /	7.70	3.96 /	5.93	**11.20** /	**21.44**	1.90 /	1.30	2.05 /	1.34	11.20 /	21.44
2	2.34 /	8.67	5.37 /	7.28	4.46 /	4.47	**9.44** /	**17.18**	1.11 /	1.11	1.17 /	1.18	9.44 /	17.18
3	2.30 /	6.53	5.33 /	5.87	4.01 /	**16.44**	**8.78** /	**10.13**	1.19 /	3.69	1.27 /	3.44	8.78 /	16.44
4	2.53 /	6.64	6.22 /	5.98	**12.96** /	**10.49**	8.68 /	**10.65**	10.23 /	2.46	10.65 /	2.42	12.96 /	10.65
5	2.39 /	12.34	6.05 /	9.87	**9.90** /	**915.47**	9.81 /	11.03	12.86 /	67.84	13.71 /	41.44	9.90 /	915.47
6	2.50 /	8.38	6.99 /	7.15	3.88 /	4.87	**14.03** /	**16.02**	1.08 /	1.20	1.16 /	1.25	14.03 /	16.02
7	2.45 /	6.77	6.13 /	6.07	**9.96** /	**9.38**	8.67 /	**11.30**	8.16 /	2.17	8.61 /	2.07	9.96 /	11.30
8	2.81 /	8.33	8.92 /	6.84	**41.92** /	**124.77**	10.05 /	13.46	37.55 /	16.81	34.06 /	11.68	41.92 /	124.77
9	2.23 /	6.22	4.93 /	5.09	4.07 /	**11.77**	**7.55** /	**10.31**	1.32 /	1.63	1.42 /	1.68	7.55 /	11.77
10	2.48 /	6.07	6.84 /	5.18	3.87 /	9.34	**13.28** /	**9.51**	1.15 /	1.98	1.24 /	1.97	13.28 /	9.51
11	2.34 /	9.14	5.30 /	6.82	4.55 /	8.03	**9.28** /	**18.69**	1.12 /	1.31	1.19 /	1.38	9.28 /	18.69
12	2.48 /	7.32	6.73 /	6.37	3.87 /	**188.52**	**13.47** /	**11.50**	1.19 /	20.84	1.28 /	13.59	13.47 /	188.52
13	2.38 /	7.40	5.75 /	6.33	**9.74** /	**25.54**	9.51 /	10.67	9.68 /	5.92	10.29 /	5.83	9.74 /	25.54
14	2.56 /	8.75	7.33 /	7.33	3.88 /	6.48	**15.27** /	**17.94**	1.14 /	1.55	1.22 /	1.54	15.27 /	17.94
15	2.35 /	7.83	5.48 /	6.51	4.23 /	**70.26**	**9.70** /	**12.16**	1.14 /	12.76	1.20 /	9.99	9.70 /	70.26
16	2.41 /	7.52	6.38 /	6.35	8.09 /	**338.78**	**10.49** /	**11.26**	10.80 /	36.04	11.53 /	25.06	10.49 /	338.78
17	2.47 /	8.08	6.66 /	6.47	3.88 /	**41.66**	**12.77** /	**12.44**	1.34 /	8.43	1.44 /	7.21	12.77 /	41.66
18	2.44 /	7.81	6.25 /	6.49	8.31 /	**136.59**	**9.40** /	**11.67**	4.63 /	18.32	4.94 /	12.53	9.40 /	136.59
19	2.46 /	6.35	6.51 /	5.22	**13.72** /	**13.72**	10.62 /	11.05	18.62 /	1.61	19.93 /	1.66	13.72 /	13.72
20	2.37 /	6.00	5.42 /	5.18	4.51 /	9.00	**9.93** /	**9.58**	1.13/	1.69	1.20 /	1.73	9.93 /	9.58
21	2.66 /	6.31	6.68 /	5.11	**21.50** /	**11.81**	10.87 /	10.67	19.23 /	1.62	19.13 /	1.67	21.50 /	11.81
22	2.46 /	6.24	6.48 /	5.15	**8.40** /	**12.59**	11.81 /	9.15	10.40 /	2.73	11.11 /	2.78	8.40 /	12.59
23	2.21 /	8.19	5.08 /	7.03	3.99 /	4.75	**12.59** /	**15.25**	1.43 /	1.18	1.53 /	1.28	12.59 /	15.25
24	2.49 /	7.78	6.95 /	6.58	7.21 /	**110.97**	**12.03** /	**12.63**	7.32 /	16.30	7.87 /	11.67	12.03 /	110.97
25	2.25 /	7.59	5.09 /	6.46	5.27 /	**80.88**	**12.08** /	**12.69**	2.09 /	13.09	2.23 /	9.61	12.08 /	80.88
avg.	2.42 /	7.42	6.09 /	6.22	5.73 /	12.50	10.45 /	12.06	2.10 /	2.47	2.25 /	2.47	**11.19** /	**19.89**

Compression Performance across Embedding Tables. Furthermore, we outline the compression ratios our compressor achieved across various embedding tables and delve into the data characteristics that enable our compressor to secure maximum compression benefits. Table V presents the compression ratios for various compressors across different embedding tables. We can draw three key observations from this result. Firstly, the compression ratios for all compressors vary significantly across embedding tables, underscoring the importance of choosing a compressor that's well-suited for each specific table. Secondly, our optimized vector-based LZ algorithm excels with certain embedding tables, while its performance on others is less impressive. Lastly, the performance trends of our optimized entropy-based compressor and the vector-based LZ algorithm appear to be in stark contrast.

We employ data sampling to shed light on the substantial variance in compression ratios among different embedding tables. Figure 13 illustrates the matched pattern number and data distribution for two representative tables from the Terabytes dataset. The data histogram reveals that *EMB Table 1* exhibits a highly concentrated Gaussian distribution, whereas *EMB Table 5* displays a broad dispersion of data with similar frequencies. This distinction in data entropy explains the higher compression ratio achieved with the Huffman encoder for *EMB Table 1*. Due to the limited unique embedding vectors in *EMB Table 5*, the likelihood of LZ encoder matching patterns is significantly high, resulting in a superior compression ratio. The marked difference in the number of matched patterns elucidates why the LZ encoder outperforms embedding tables like *EMB Table 5*.



(a) Sampled Batch of EMB Table 1



(b) Sampled Batch of EMB Table 5

Fig. 13: Data features of two representative EMB tables.

Compression Performance across Training Phases. Based on Figures 9b and 10b, we observe that our compressor performs effectively throughout all training phases, maintaining a consistently high compression ratio. Implementing an error-bound decay to enhance the quality of compressed data results in only a slight decrease in the compression ratio. This stable compression ratio can be attributed to two main factors. Firstly, the Vector-based LZ encoder, which relies on pattern matching, maintains its effectiveness due to the

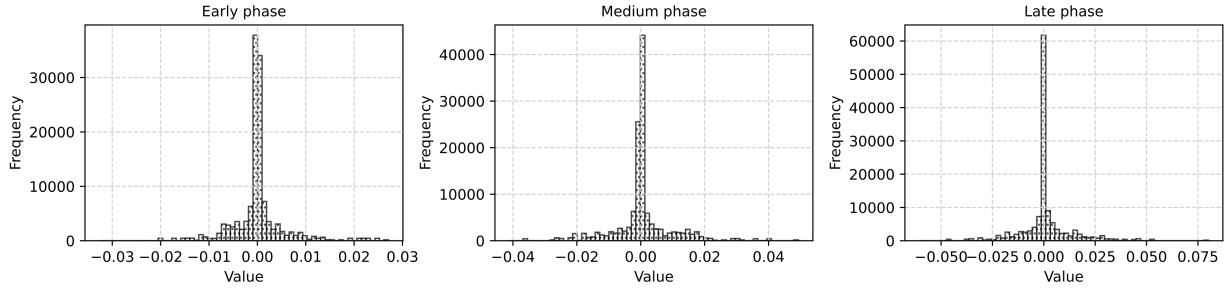


Fig. 14: Data distribution of representative EMB tables in different phases on Criteo Terabyte dataset.

TABLE VI: Compression ratio improvement of fine-tuned LZ encoder with different window sizes.

Window Size	32	64	128	255
Criteo Kaggle	1×	2.21×	3.89×	5.23×
Criteo Terabytes	1×	1.47×	1.52×	1.54×

presence of identifiable patterns in each batch, a factor that remains constant regardless of the training phase. Secondly, the uniformity in data distribution across the training, as depicted in Figure 14, ensures consistent compression performance.

D. Evaluation of Compression Optimization

Finally, we evaluate the performance of our optimized compression compared to the non-optimized solution.

LZ Fine-tuning. First, we evaluate our optimized compressor with different LZ fine-tuning window sizes. Table VI shows the compression ratio and throughput on DLRM data with varying window sizes. Generally, larger window sizes result in more pattern matches. For the Criteo Terabyte dataset, we observe that the overall compression ratio with the window sizes of 128 and 255 are 3.9× and 5.2× higher, respectively, compared to the baseline window size of 32. On the Criteo Kaggle dataset, the difference between the window sizes of 128 (1.52×) and 255 (1.54×) is negligible. In instances of small batch sizes, our vector-based LZ is less beneficial with the increase in window size. This is due to the proportion between the sliding window size and the volume of data. In the model of the Criteo Kaggle dataset, for example, $BatchSize$ is 128 by default and can be fully covered by one sliding window. The scaled compression ratios of the Criteo Terabyte dataset demonstrate that increasing the window size does not linearly increase the compression ratio. This phenomenon is attributed to the unbalanced frequency of queries. EMB vectors with very high frequency can be matched with an appropriate window size, whereas EMB vectors with low frequency require a window size that exceeds hardware limitations and is inefficient.

Buffer Optimization. Second, we evaluate our buffer optimization in both compression and decompression processes. We split the EMB vectors into chunks, with the number of chunks equal to the RANK in distributed training, which reflects the scalability of the training process. Figure 15 illustrates the compression speedup across different EMB vector sizes, with the number of chunks ranging from 2 to 16. The term 'single_comp' denotes our solution. The results

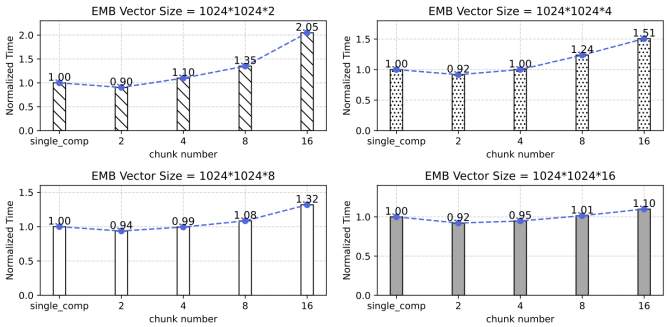


Fig. 15: Normalized time of our work with and without buffer optimization for different EMB vector sizes. 'single_comp' denotes our solution, while the chunk number indicates how many chunks the original EMB vector is equally partitioned and compressed.

indicate that our design achieves higher speedups with an increased number of chunks. According to our evaluation, our optimizations achieve a maximum speedup of 2.04×.

Our proposed buffer optimization technique performed as expected. When evaluating speed-up across different data chunk sizes, we observed that with 8MB data blocks, the performance is 1.86× better than with 64MB blocks. As discussed previously, for limited data sizes, the volume of an individual chunk is too small to achieve optimal GPU utilization. The bottleneck in compression for these small chunks arises from frequent kernel launches rather than from the compression process or memory copying itself. Conversely, for larger data volumes, the advantage of buffer optimization becomes less significant because each chunk's larger volume allows the GPU to attain higher utilization, even when compressing chunks sequentially.

V. Related Work

A variety of research efforts have sought to expedite DLRM training, generally falling into three categories: embedding compression, embedding caching, and low-precision training.

Embedding Compression. Yin *et al.*'s TT-Rec [29] and Wang *et al.*'s EL-Rec [20] utilize Tensor Train decomposition to reduce memory consumption in Embedding Tables, aiming to lessen resource usage in constrained environments. Despite their advantages, these model-compression techniques face challenges: firstly, they may not always ensure convergence or maintain high model accuracy; secondly, they can introduce significant computational overheads. Specifically, the recovery of embedding vectors for each batch necessitates extra matrix

multiplications. Given the often long and narrow shape of these matrices, such operations can prove highly inefficient on GPUs. Furthermore, it is important to highlight that these techniques are complementary to our approach. Essentially, our method could be combined with model compression in DLRM training to further enhance performance in scenarios with limited resources.

Embedding Caching. Strategies like Pattern-Aware Sparse Communication by He *et al.* [30], cDLRM by Balasubramanian *et al.* [31], and a heterogeneous SmartNIC system by Guo *et al.* [32] aim to alleviate caching overhead. Unlike these approaches that may require additional memory for data copies or hardware support for caching, our compression technique avoids extra storage demands for heterogeneous embedding table access. Moreover, maintaining cache involves computational costs, such as updating cache entries and ensuring cache coherence, which our method does not incur.

Low-precision Training. Exploring low-precision data types, as proposed by Rouhani *et al.* with the new datatype MX[27] and mixed-precision strategies by Yang *et al.*[33], is another direction of research. These methods, though direct, offer limited compression benefits due to their fixed compression ratio and lack of fine-tuning capabilities, providing a coarse granularity control over data compression. Our strategy, in contrast, allows for a smooth adjustment of error bounds, offering a more flexible and efficient solution to communication reduction in DLRM training.

General Compression-accelerated Communication. In addition, several studies have explored leveraging compression to boost communication speeds more generally. Zhou *et al.* have contributed a series of works [34], [35] that emphasize enhancing collective communication efficiency through compression. Similarly, Ramesh *et al.* introduced Efficient Pipelined Communication Schemes [36], aimed at minimizing blocking time and maximizing bandwidth utilization.

Our approach, however, stands out from these efforts in a couple of key ways. Firstly, unlike the aforementioned studies, our method is tailor-made for DLRM applications, incorporating adaptive error-bound adjustments that are absent in general compression techniques. These general approaches often provide a low-level interface that may prove challenging to finely tune in real-world scenarios. Secondly, our compression algorithm is specifically optimized for DLRM training, offering both higher compression throughput and ratios compared to existing GPU compressors—let alone CPU compressors, which generally deliver significantly lower throughput. Overall, our method provides an integrated end-to-end solution specifically for enhancing DLRM training communication, unlike the general methods that only offer a compression-accelerated communication library or tool without tailoring to the specific needs of the application.

VI. Conclusion and Future Work

In this paper, we introduce a method that employs error-bounded lossy compression to reduce the communication data size and accelerate DLRM training. We develop a novel error-bounded lossy compression algorithm to achieve hybrid by hybridizing our optimized LZ encoder and entropy encoder on GPU. Moreover, we introduce a dual-level adaptive strategy for error-bound adjustment, spanning both table-wise and iteration-wise aspects, to balance the compression benefits with the potential impacts on accuracy. We further optimize our compressor for PyTorch tensors on GPUs, minimizing compression overhead. The evaluation shows that our method achieves an 8.6× all-to-all communication speedup and 1.38× end-to-end training speedup with a minimal accuracy impact.

In the future, we plan to further refine our system to reduce compression overhead by employing strategies like kernel fusion on GPUs and seamlessly integrating (de)compression processes with communication libraries such as NCCL. Additionally, we aim to develop a more advanced and automated approach for offline selection of a fixed global error-bound and for online error-bound adjustments.

Acknowledgment

The work is supported by the Meta Research Award for "AI System Hardware/Software Codesign." Hao Feng, Boyuan Zhang, Fanjiang Ye, and Dingwen Tao's work on this project was supported by the National Science Foundation (Grant Nos. 2312673, 2247080, 2303064, 2326494, and 2326495). Dingwen Tao was also supported by the National Natural Science Foundation of China (Grant Nos. 62032023 and T2125013) and the Innovation Funding of ICT, CAS (Grant No. E461050).

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] D. D. Sensi, S. D. Girolamo, K. H. McMahon, D. Roweth, and T. Hoefler, "An in-depth analysis of the slingshot interconnect," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*, C. Cuicchi, I. Qualters, and W. T. Kramer, Eds., IEEE/ACM, 2020, p. 35. DOI: 10.1109/SC41405.2020.00039. [Online]. Available: https://doi.org/10.1109/SC41405.2020.00039.

[6] J. A. Yang, J. Huang, J. Park, P. T. P. Tang, and A. Tulloch, "Mixed-precision embedding using a cache," *CoRR*, vol. abs/2010.11305, 2020. arXiv: 2010.11305. [Online]. Available: https://arxiv.org/abs/2010.11305.

[7] 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.

[8] S. Pumma and A. Vishnu, "Semantic-aware lossless data compression for deep learning recommendation model (DLRM)," in *IEEE/ACM Workshop on Machine Learning in High Performance Computing Environments, MLHPC@SC 2021, St. Louis, MO, USA, November 15, 2021*, IEEE, 2021, pp. 1–8. DOI: 10.1109/MLHPC54614.2021.00006. [Online]. Available: https://doi.org/10.1109/MLHPC54614.2021.00006.

[9] S. Di and F. Cappello, "Fast error-bounded lossy hpc data compression with sz," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016, pp. 730–739. DOI: 10.1109/IPDPS.2016.11.

[10] P. Lindstrom, "Fixed-rate compressed floating-point arrays," *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, Aug. 2014. DOI: 10.1109/TVCG.2014.2346458.

[11] G. Wallace, "The jpeg still picture compression standard," *IEEE Transactions on Consumer Electronics*, vol. 38, no. 1, pp. xviii–xxxiv, 1992. DOI: 10.1109/30.125072.

[12] D. Le Gall, "The mpeg video compression standard," in *COMPCON Spring '91 Digest of Papers*, 1991, pp. 334–335. DOI: 10.1109/CMPCON.1991.128827.

[13] D. Tao, S. Di, Z. Chen, and F. Cappello, "Significantly improving lossy compression for scientific data sets based on multidimensional prediction and error-controlled quantization," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2017, pp. 1129–1139. DOI: 10.1109/IPDPS.2017.115.

[14] X. Liang, K. Zhao, S. Di, S. Li, R. Underwood, A. M. Gok, J. Tian, J. Deng, J. C. Calhoun, D. Tao, Z. Chen, and F. Cappello, "Sz3: A modular framework for composing prediction-based error-bounded lossy compressors," *IEEE Transactions on Big Data*, vol. 9, no. 2, pp. 485–498, 2023. DOI: 10.1109/TBDATA.2022.3201176.

[15] R. Ballester-Ripoll, P. Lindstrom, and R. Pajarola, "Tthresh: Tensor compression for multidimensional visual data," *IEEE Transaction on Visualization and Computer Graphics*, vol. 26, pp. 2891–2903, 9 2019.

[16] J. Tian, S. Di, K. Zhao, C. Rivera, M. H. Fulp, R. Underwood, S. Jin, X. Liang, J. Calhoun, D. Tao, and F. Cappello, "Cusz: An efficient gpu-based error-bounded lossy compression framework for scientific data," in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '20, Virtual Event, GA, USA: Association for Computing Machinery, 2020, 3–15, ISBN: 9781450380751. DOI: 10.1145/3410463.3414624. [Online]. Available: https://doi.org/10.1145/3410463.3414624.

[17] B. Zhang, J. Tian, S. Di, X. Yu, M. Swany, D. Tao, and F. Cappello, "Gpulz: Optimizing lzss lossless compression for multi-byte data on modern gpus," in *Proceedings of the 37th International Conference on Supercomputing*, 2023, pp. 348–359.

[18] *Https://lz4.org/*.

[19] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952. DOI: 10.1109/JRPROC.1952.273898.

[20] Z. Wang, Y. Wang, B. Feng, D. Mudigere, B. Muthiah, and Y. Ding, "El-rec: Efficient large-scale recommendation model training via tensor-train embedding table," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '22, Dallas, Texas: IEEE Press, 2022, ISBN: 9784665454445.

[21] L. Ibarria, P. Lindstrom, J. Rossignac, and A. Szymczak, "Out-of-core compression and decompression of large n-dimensional scalar fields," in *Computer Graphics Forum*, Wiley Online Library, vol. 22, 2003, pp. 343–348.

[22] *Https://ailab.criteo.com/ressources/*.

[23] *Https://www.kaggle.com/c/criteo-display-ad-challenge*.

[24] P. Micikevicius, D. Stosic, N. Burgess, M. Cornea, P. Dubey, R. Grisenthwaite, S. Ha, A. Heinecke, P. Judd, J. Kamalu, N. Mellempudi, S. Oberman, M. Shoeybi, M. Siu, and H. Wu, *Fp8 formats for deep learning*, 2022. arXiv: 2209.05433 [cs.LG].

[25] H. Shen, N. Mellempudi, X. He, Q. Gao, C. Wang, and M. Wang, *Efficient post-training quantization with fp8 formats*, 2023. arXiv: 2309.14592 [cs.LG].

[26] Nvidia, *Https://developer.nvidia.com/nvcomp*, 2024.

[27] B. Darvish Rouhani, R. Zhao, V. Elango, R. Shafipour, M. Hall, M. Mesmakhosroshahi, A. More, L. Melnick, M. Golub, G. Varatkar, L. Shao, G. Kolhe, D. Melts, J. Klar, R. L'Heureux, M. Perry, D. Burger, E. Chung, Z. S. Deng, S. Naghshineh, J. Park, and M. Naumov, "With shared microexponents, a little shifting goes a long way," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ser. ISCA '23, Orlando, FL, USA: Association for Computing Machinery, 2023, ISBN: 9798400700958. DOI: 10.1145/3579371.3589351. [Online]. Available: https://doi.org/10.1145/3579371.3589351.

[28] B. Zhang, J. Tian, S. Di, X. Yu, Y. Feng, X. Liang, D. Tao, and F. Cappello, "Fz-gpu: A fast and high-ratio lossy compressor for scientific computing applications on gpus," in *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing*, 2023, pp. 129–142.

[29] C. Yin, B. Acun, X. Liu, and C.-J. Wu, *Tt-rec: Tensor train compression for deep learning recommendation models*, 2021. arXiv: 2101.11714 [cs.LG].

[30] J. He, S. Chen, and J. Zhai, "Poster: Pattern-aware sparse communication for scalable recommendation model training," in *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, 2024, pp. 466–468.

[31] K. Balasubramanian, A. Alshabanah, J. D. Choe, and M. Annavaram, "Cdlrm: Look ahead caching for scalable training of recommendation models," in *Proceedings of the 15th ACM Conference on Recommender Systems*, ser. RecSys '21, Amsterdam, Netherlands: Association for Computing Machinery, 2021, 263–272, ISBN: 9781450384582. DOI: 10.1145/3460231.3474246. [Online]. Available: https://doi.org/10.1145/3460231.3474246.

[32] A. Guo, Y. Hao, C. Wu, P. Haghi, Z. Pan, M. Si, D. Tao, A. Li, M. Herbordt, and T. Geng, "Software-hardware co-design of heterogeneous smartnic system for recommendation models inference and training," in *Proceedings of the 37th International Conference on Supercomputing*, 2023, pp. 336–347.

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

[34] Q. Zhou, Q. Anthony, L. Xu, A. Shafi, M. Abduljabbar, H. Subramoni, and D. K. Panda, "Accelerating distributed deep learning training with compression assisted allgather and reduce-scatter communication," in *IEEE International Parallel and Distributed Processing Symposium, IPDPS 2023, St. Petersburg, FL, USA, May 15-19, 2023*, IEEE, 2023, pp. 134–144. DOI: 10.1109/IPDPS54959.2023.00023. [Online]. Available: https://doi.org/10.1109/IPDPS54959.2023.00023.

[35] Q. Zhou, Q. Anthony, A. Shafi, H. Subramoni, and D. K. Panda, "Accelerating broadcast communication with GPU compression for deep learning workloads," in *29th IEEE International Conference on High Performance Computing, Data, and Analytics, HiPC 2022, Bengaluru, India, December 18-21, 2022*, IEEE, 2022, pp. 22–31. DOI: 10.1109/HIPC56025.2022.00016. [Online]. Available: https://doi.org/10.1109/HiPC56025.2022.00016.

[36] B. Ramesh, Q. Zhou, A. Shafi, M. Abduljabbar, H. Subramoni, and D. K. Panda, "Designing efficient pipelined communication schemes using compression in MPI libraries," in *29th IEEE International Conference on High Performance Computing, Data, and Analytics, HiPC 2022, Bengaluru, India, December 18-21, 2022*, IEEE, 2022, pp. 95–99. DOI: 10.1109/HIPC56025.2022.00024. [Online]. Available: https://doi.org/10.1109/HiPC56025.2022.00024.

Appendix: Artifact Description/Artifact Evaluation

Artifact Description (AD)

I. Overview of Contributions and Artifacts

A. Paper's Main Contributions

C_1 We develop a two-level adaptive strategy for error-bound adjustment for different embedding tables and training iterations. We aim to maintain relatively large error bounds (for higher compression ratios) while minimizing accuracy loss.

C_2 We develop a new hybrid compression method for DLRM embedding lookups/vectors, including a new LZ encoder and entropy-based Huffman encoder, achieving a high compression ratio on specific data domains.

C_3 We optimize our compression method on GPUs, including parallel compression and decompression of multiple tensors and compression algorithm parameter fine-tuning.

B. Computational Artifacts

$A_1, A_2$10.5281zenodo.13119688

Artifact ID	Contributions Supported	Related Paper Elements
A_1	C_1	Tables 1-2, 5 Figure 4, 8
A_2	C_2	Tables 3-4 Figures 5-6, 9-11
A_2	C_3	Tables 6 Figures 15

II. Artifact Identification

A. Computational Artifact A_1

Relation To Contributions

See Table I-B for the relation among artifacts, contributions, and related paper figures and tables.

Expected Results

This artifact will output DLRM training accuracy results with lossy compression enabled. It will also output embedding vectors in DLRM training.

Expected Reproduction Time (in Minutes)

Without data pre-processing, the end-to-end training times of Criteo Kaggle and Criteo Terabytes datasets are 21 hours and 70 hours, respectively. With data pre-processing, Criteo Kaggle and Criteo Terabytes datasets need an additional 2 days and 7 days, respectively.

Artifact Setup (incl. Inputs)

Hardware: Storage: 50GB for the Criteo Kaggle dataset, 1.5TB for the Criteo Terabytes dataset.

CPU: no specific requirement.

RAM: 128GB for training, up to 1TB for faster pre-processing.

GPU: 1 Nvidia A100(40GB) for Criteo Kaggle dataset, 2 Nvidia A100(40GB) for Criteo Terabytes dataset.

Software: Python@3.9.18, PyTorch@1.10.2, CUDA@11.7, nvCOMP@3.02, nvcc@11.3.58, NCCL@2.10.3

Datasets / Inputs: There are two links for downloading two open-source dataset we used. Criteo Kaggle, Criteo Terabyte.

Installation and Deployment: We prepared two installation methods, a Singularity image and a build from source instruction. To use the Singularity image, reviewers should install Singularity first, then download the pre-built image, and run this image. To build from source, reviewers should install Python, PyTorch, and CUDA first. Then they should build the compressors we mentioned above.

Artifact Execution

A workflow may consist of two tasks: T_1, T_2 and T_3. The task T_1 will training DLRM with two-level adaptive error bound lossy compression enabled. The task T_2 will dump embedding vectors during training to disk storage. The task T_3 will profile the training process.

Artifact Analysis (incl. Outputs)

Task T_1 will output accuracy convergence curves with lossy compression enabled. Task T_2 will output original DLRM embedding vector data in training. Task T_3 will output the overhead breakdown in DLRM training.

B. Computational Artifact A_2

Expected Results

This artifact will output compression evaluation results, data features analysis results in DLRM embedding tables and vectors, and optimized compressor evaluation results.

Expected Reproduction Time (in Minutes)

It takes about 5 minutes to produce per compression evaluation result with a parameter combination. It takes about 1 minute to produce the Homo Index result of one checkpoint. It takes about 1 minute to produce data distribution of sampled embedding tables. It takes about 5 minutes to produce the results of compressor parameter fine-tuning and system optimization. The total time is 50 minutes including compression evaluation(30 minutes in total), Homo Index calculation(10 minutes in total), data distribution analysis(5 minutes in total), and optimization on compressor(5 minutes).

Artifact Setup (incl. Inputs)

Inputs should dump embedding vector data from artifact A_1.

Hardware: CPU: no specific requirement. GPU: 1 Nvidia A100(40GB) Storage: 5 GB

Software: Besides software we used in A_1, we also need compressors below: GPULZ(GPULZ), cuSZ(cuSZ), FZ-GPU(FZ-GPU)

Datasets / Inputs: We applied dumped embedding vector data in A_1 as input in this artifact.

Installation and Deployment: Same as A_1.

Artifact Execution

A workflow may consist of three tasks: T_1, T_2, and T_3. The task T_1 will apply compressors on input data to calculate compression ratio, compression throughput, and speed-up. Task T_1 will also make compressor selection based on speed-up. The task T_2 will calculate various data features including distribution and Homo Index based on the input embedding data. Task T_2 will also do embedding table classification based on these data features. The task T_3 will apply different compression parameters on the optimized compressor and evaluate the performance of the compressor.

Artifact Analysis (incl. Outputs)

Task T_1 will output the compression ratio and throughput on embedding data with all compressors we used. Task T_2 will output data feature results, and embedding table classification results. Task T_3 will output performance improvement of fine-tuning and optimized compressor.

Artifact Evaluation (AE)

A. Computational Artifact A_1

Artifact Setup (incl. Inputs)

We provide an image includes needed software of artifact evaluation. Please install Singularity and run the image. To get the dataset, please download two dataset via these two url, Kaggle, Criteo-Terabytes.

Artifact Execution

There are two main tasks, the first one is to train DLRM with adaptive lossy compression, the second one is to draw accuracy curve with logs in first task.

Please use command to run singularity.

```
$ sudo singularity build sc24_dlrm.sif dlrm_image
$ sudo singularity shell --writable --nv dlrm_image
```

To train DLRM. Execute these commands.

```
$ bash SC_ADAE_scripts/kaggle_run.sh
$ bash SC_ADAE_scripts/tb_run.sh
```

To draw accuracy curve. Execute these commands.

```
$ python accuracy_parser.py
$ python accuracy_curve.py
```

Artifact Analysis (incl. Outputs)

The training log should be like.

```
Finished training it 1/3 of epoch 0, -1.00 ms/it, loss 0.451893, accuracy 0.000%
Finished training it 2/3 of epoch 0, -1.00 ms/it, loss 0.402002, accuracy 0.000%
Finished training it 3/3 of epoch 0, -1.00 ms/it, loss 0.275460, accuracy 0.000%
```

B. Computational Artifact A_2

Artifact Setup (incl. Inputs)

Same as A_1. Should be executed after A_1.

Artifact Execution

We use dumped data to do compression evaluation. First, please run quantization script to introduce lossy error.

```
$ python python quantization.py EMB_file_path decay_stag
```

Then, please run lossless encoder script to apply different lossless encoder based on lossy input.

```
$ python lossless_encoder.py
```

Finally, please run parser script to extract compression ratio and throughput from compressors' output log in previous step. In

```
$ python huffman_parser.py
$ python nvcomp_parser.py
$ python gpulz_parser.py
```

We also provide a jupyter-notebook file to generate raw compression ratio. **parser.ipynb** will parse and generate raw information.

Artifact Analysis (incl. Outputs)

The first step will generate quantized binary files. For example, **EMB_{emb_table}_iter_{iter}.bin.quan** The second step will generate compression logs with different compressor as log files. The third step will generate averaged or raw compression ratio and throughput as text files.