MTGenRec: An Efficient Distributed Training System for Generative Recommendation Models in Meituan

Yuxiang Wang*
Wuhan University
School of Computer Science
Wuhan, China
nai.yxwang@whu.edu.cn

Chi Ma*
Meituan
Beijing, China
machi04@meituan.com

Xiao Yan†
Wuhan University
Institute for Math and AI
Wuhan, China
yanxiaosunny@whu.edu.cn

Mincong Huang
Xiaoguang Li
Meituan
Beijing, China
huangmincong@meituan.com
lixiaoguang12@meituan.com

Ruidong Han
Bin Yin
Meituan
Beijing, China
hanruidong@meituan.com
yinbin05@meituan.com

Shangyu Chen
Xiang Li
Meituan
Beijing, China
chenshangyu03@meituan.com
lixiang245@meituan.com

Fei Jiang
Lei Yu
Meituan
Beijing, China
jiangfei05@meituan.com
yulei37@meituan.com

Chuan Liu
Wei Lin
Meituan
Beijing, China
liuchuan11@meituan.com
linwei31@meituan.com

Haowei Han
Xiaokai Zhou
Wuhan University
School of Computer Science
Wuhan, China
haowei.han@whu.edu.cn
xiaokaizhou@whu.edu.cn

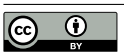
Bo Du
Wuhan University, School of Computer Science, National Engineering Research Center for Multimedia Software, Hubei Key Laboratory of Multimedia and Network Communication Engineering
Wuhan, China
dubo@whu.edu.cn

Jiawei Jiang†
Wuhan University
School of Computer Science
Wuhan, China
jiawei.jiang@whu.edu.cn

Abstract

Recommendation is crucial for both user experience and company revenue in Meituan as a leading lifestyle company, and generative recommendation models (GRMs) are shown to produce quality recommendations recently. However, existing systems are limited by insufficient functionality support and inefficient implementations for training GRMs in industrial scenarios. As such, we introduce **MTGenRec** as an efficient and scalable system for GRM training. Specifically, to handle real-time insertions/deletions of sparse embeddings, MTGenRec employs dynamic hash tables to replace static ones. To improve training efficiency, MTGenRec conducts dynamic sequence balancing to address the computation load imbalances among GPUs and adopts feature ID deduplication alongside automatic table merging to accelerate embedding lookup. Extensive experiments show that MTGenRec improves training throughput by 1.6× – 2.4× while achieving good scalability when running over 100 GPUs. MTGenRec has been deployed for many applications in Meituan and is now handling hundreds of millions of requests on a daily basis. On the delivery platform, we observe a 1.22% growth in user order volume and a 1.31% enhancement in online PV_CTR.

*Both authors contributed equally to this research.
†Corresponding authors.



KDD 2026, Jeju Island, Republic of Korea.

ACM ISBN 979-8-4007-2258-5/2026/08
https://doi.org/10.1145/3770854.3783925

CCS Concepts

• **Information systems** → **Recommender systems**; • **Theory of computation** → **Distributed computing models**.

Keywords

Recommendation Systems; Distributed Model Training

ACM Reference Format:

Yuxiang Wang, Chi Ma, Xiao Yan, Mincong Huang, Xiaoguang Li, Ruidong Han, Bin Yin, Shangyu Chen, Xiang Li, Fei Jiang, Lei Yu, Chuan Liu, Wei Lin, Haowei Han, Xiaokai Zhou, Bo Du, and Jiawei Jiang. 2026. MTGenRec: An Efficient Distributed Training System for Generative Recommendation Models in Meituan. In *Proceedings of the 32nd ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.1 (KDD 2026), August 9–13, 2025, Jeju Island, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3770854.3783925

1 Introduction

As a leading e-commerce platform for lifestyle services, Meituan spans multiple business verticals such as delivery, travel, and health. In 2024, Meituan served over 770 million transacting users and processed more than 98 million orders at the daily peak. As shown in Figure 1, recommendation systems [15, 27, 37, 46, 48] are at the core of our business by effectively filtering history logs and personalizing services for users. High-quality recommendations are crucial to sustain high customer retention rates and enhance company revenue. Currently, our recommendation model utilizes terabytes of user action data and item metadata for training on a daily basis and produces billions of predictions for various services.

Inspired by generative language models [1, 28, 34, 35], generative recommendation models (GRMs) [5, 10, 13, 38, 43–45] have gained significant attention from both academia and industry. Specifically, a GRM consists of two modules, i.e., *sparse embedding tables* that map discrete categorical features to continuous embedding vectors, and a *Transformer-based dense model* that predicts the next user action by modeling the action sequences of users. In contrast, earlier deep recommendation models (DRMs) [27, 37, 42, 48] utilize multilayer perceptrons to capture the interactions between users and items. As shown in Figure 2, GRM achieves superior accuracy compared with DRM, primarily due to its ability to model the entire action sequence of each user with self-attention. However, GRM has greater computation complexity due to its attention mechanism, which scales quadratically with sequence length, while DRM scales only linearly. Thus, to enjoy the good accuracy of GRMs and keep up with the high speed of training data generation, Meituan requires an efficient and scalable system for training GRMs.

TorchRec [12] is a state-of-the-art PyTorch-based system for training GRMs. It enables efficient computation of attention scores [4] and enhances scalability through hybrid parallel strategies, including data and model parallelisms [11, 26, 30, 32]. To leverage TorchRec's benefits, we further develop our MTGenRec based on it. However, we identify two fundamental limitations in TorchRec that hinder its application in industrial recommendation systems. In particular, ❶ when TorchRec handles the sparse embedding tables of GRMs, its built-in static embedding tables cannot process dynamic embedding entries and table capacity expansion. Specifically, additional embeddings cannot be allocated to new users and items (e.g., merchants updating menus) in real-time from fixed-size static tables. Although default embeddings can be used in this case, model accuracy will be degraded. Moreover, static tables typically require pre-allocating more capacity than necessary to prevent overflow, leading to memory underutilization. ❷ For the embedding

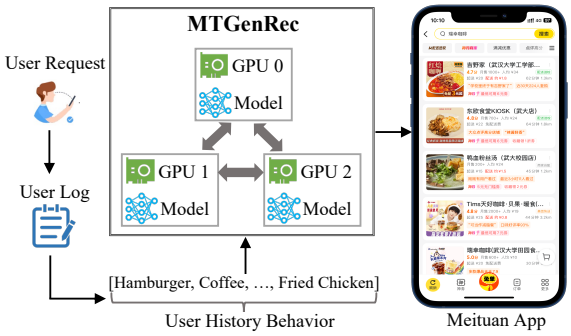


Figure 1: Recommendation workflow in Meituan.

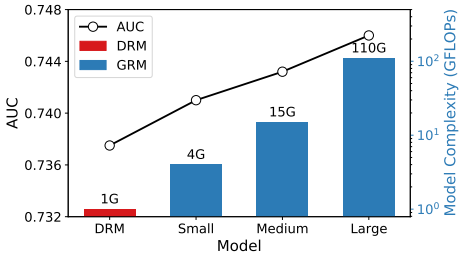


Figure 2: AUC (for accuracy, higher the better) and model complexity for DRM and GRM. Notably, an accuracy improvement of even 0.1% is crucial for practical recommendation.

lookup process, TorchRec transmits same embedding entries multiple times due to duplicate embedding IDs in the user sequences, leading to a long communication time. Moreover, TorchRec requires labor-intensive manual configuration to merge embedding tables, which is a common and effective operation for accelerating lookups. The two limitations severely constrain the practical deployment of GRMs and compromise training efficiency.

To address these challenges, we propose a *hash-based embedding table* for changeable IDs and *two-stage ID deduplication* for lookup acceleration. The hash-based embedding table eliminates fixed-capacity constraints by mapping arbitrary feature IDs to embedding indices via a hash function. We decouple the storage of keys (i.e., embedding index) and values (i.e., embedding vector). This design prioritizes the expansion of the smaller key list over the larger value list, thereby enhancing the efficiency of table expansion. Additionally, we adopt a two-stage ID deduplication operation to eliminate repetitive IDs before and after ID communication, thus avoiding the repetitive transmission of embedding vectors among devices. We also design a table configuration interface for automatic table merging, which combines multiple embedding tables into one according to defined patterns (e.g., the same embedding dimension). This technique reduces the number of embedding lookup operators, and thus accelerating the process. Besides the sparse embeddings, we observe that variable-length sequences cause significant load imbalances among GPUs. Since retaining complete sequences is crucial for GRM's accuracy, we cannot truncate or pad sequences as is done in LLMs. Instead, we propose a lightweight *dynamic*

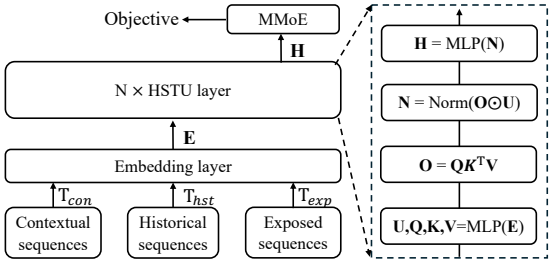


Figure 3: Model architecture of Meituan's GRM.

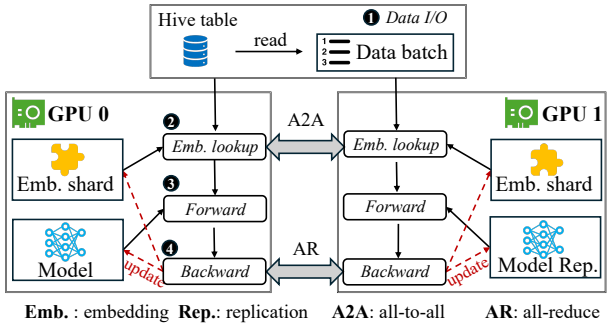


Figure 4: The workflow of MTGenRec for GRM training.

sequence batching technique that dynamically adjusts batch sizes to ensure a more balanced workload across GPUs.

We evaluate MTGenRec and its designs using 200 million real user sequences generated over a week at Meituan. Scalability studies show that MTGenRec achieves close to linear speedup in training throughput when using more GPUs. Compared with TorchRec, MTGenRec reduces training time by 2.44×. Ablation studies show that our designs are effective. In particular, dynamic sequence balancing achieves near-optimal load balance and improves system throughput by 1.75× compared to the baseline. ID deduplication significantly reduce network communication, yielding an improvement of 53% in throughput. Compared to the original DRM, GRM maintains similar training costs with MTGenRec, while achieving a 2.88pp increase in offline CTCVR GAUC, a 1.22% growth in take-away orders, and a 1.31% enhancement in PV_CTR.

To summarize, we make the following contributions.

- We present MTGenRec as an efficient and scalable system for training generative recommendation models (GRMs) at Meituan.
- To address the challenges caused by GRM's sparse and dense models, we introduce dynamic embedding tables and ID deduplication for changeable feature IDs and lookup acceleration, and dynamic sequence batching for computation load balancing.
- We conduct extensive experiments to evaluate MTGenRec. The results show that MTGenRec is efficient and scalable, and our designs are effective in improving system efficiency.

2 MTGenRec System Overview

Model Architecture. Figure 3 illustrates the model architecture of the GRM in Meituan, which consists of four core components: input sequence, embedding layer, hierarchical sequential transduction units [45] (HSTU) layer, and multi-gate mixture-of-experts [24] (MMoE) layer. In particular, the input sequence $\mathbf{T} = [\mathrm{T}_{con}, \mathrm{T}_{hst}, \mathrm{T}_{exp}]$ includes contextual sequences T_{con} (i.e., user features), historical sequences T_{hst} (e.g., click and purchase actions), and exposition sequences T_{exp} (i.e., candidate items). Since the model cannot directly process discrete category data, the embedding layer ϕ_{emb} is used to convert the feature IDs into continuous embedding vectors $\mathbf{E} = \phi_{emb}(\mathbf{T}) \in \mathbb{R}^{F \times d}$, where F and d denote the number of features and embedding dimension, respectively.

Subsequently, we employ multiple HSTU layers – a self-attention based Transformer [35] architecture – to learn user sequence embeddings from item interactions. As shown in Figure 3, each HSTU is equipped with four attention-based sub-layers for user feature extraction **U**, item feature extraction **O**, and feature interaction **H**. The computation process is expressed as follows:

$$\mathbf{U}, \mathbf{Q}, \mathbf{K}, \mathbf{V} = \mathrm{Split}(\phi_1(\mathrm{MLP}(\mathbf{E}))), \tag{1}$$

$$\mathbf{O} = \phi_2(\mathbf{Q}\mathbf{k}^T)\mathbf{V}, \tag{2}$$

$$\mathbf{H} = \mathrm{MLP}(\mathrm{Norm}(\mathbf{O} \odot \mathbf{U})), \tag{3}$$

where ϕ_1 and ϕ_2 are the SiLU [6] activation function. MMoE directs embeddings **H** from the HSTU layer to multiple expert models. Each expert model is equipped with a gating network that learns the model's weights. We finally aggregate the output embeddings of the top-k expert models. That is:

$$y = \sum_{i=1}^{k} g_i(\mathbf{H}) \cdot \mathrm{Expert}_i(\mathbf{H}). \tag{4}$$

We select the cross entropy loss as the training objective.

Distributed Training. MTGenRec facilitates the distributed training of GRMs across multiple devices. We follow TorchRec's design and employ a hybrid parallel strategy: model parallelism for sparse models and data parallelism for dense models. This design is driven by two considerations. First, data parallelism is impractical for the large embedding table, as no single device can store the entire embedding table. Second, dense models are relatively small, making data parallelism more suitable. MTGenRec is trained end-to-end with synchronous execution to ensure model quality and accuracy. To further enhance training efficiency, we introduce a pipeline that preprocesses the next data batch concurrently with the model computation. We detail the pipeline design in Appendix B.1. Next, We provide an introduction to the workflow of MTGenRec as follow.

- **Data I/O.** As shown in Figure 4, the training data is stored in partitioned Hive tables on HDFS, which utilizes a columnar storage format to optimize access speed and storage efficiency. Unlike DRM that pairs a user with a single item in a data sample, GRM uses a sequence-wise approach that pairs a user with multiple items. This design avoids the repeated computation of a user feature. We provide an explanation in Appendix A.
- **Embedding Lookup.** The embedding lookup uses All-to-all communication for cross-device embedding exchange. In particular, this process involves two data exchanges: First, devices transmit the required feature IDs and receive corresponding IDs

from peer devices. Subsequently, a collection operation facilitates the exchange of retrieved embeddings between devices.

- **Forward Computation.** After embedding lookup, we encode the input sequence through the dense model (i.e., HSTU and MMoE layers) and then calculate the loss function. Model parameters across all devices are consistently initialized by setting the same hyperparameter and random seed.
- **Backward Update.** MTGenRec employs a heterogeneous parameter update strategy in the backward propagation. Specifically, sparse embedding tables aggregate gradients through model parallelism, where each device updates its locally stored embedding shards. Dense parameters employ All-Reduce communication for gradient synchronization before parameter update.

3 Core Designs for MTGenRec

3.1 Dynamic Embedding Table

In a real production environment, the recommendation system must handle real-time insertions and deletions of sparse embedding entries (e.g., merchants adding or removing products). However, TorchRec cannot handle dynamic entries due to its static embedding tables (table capacity is fixed), and it typically uses default embeddings for feature IDs that exceed the table capacity. However, this method may lead to model accuracy degradation. Additionally, static tables require over-provisioning of initial capacity to anticipate potential expansion requirements, which results in inefficient memory utilization. These limitations motivate MTGenRec's implementation of hash-based embedding tables.

Storage Layout. MTGenRec employs a decoupled hash table architecture to separate keys and values into distinct structures. As shown in Figure 5 (a), the lightweight *key structure* maintains a mapping table of keys and pointers to embeddings, while the *embedding structure* stores embeddings and auxiliary data (e.g., timestamps). This design allows for efficient table expansion by prioritizing the expansion of the smaller key structure over the larger embedding structure. Additionally, we implement chunk-based allocation for the embedding structure. Specifically, multiple embeddings are stored within a single chunk, and insertions/deletions are managed on a per-chunk basis. This approach enhances efficiency, as it requires only a single chunk allocation or deletion operation, rather than multiple individual operations.

Hash Function and Collision Handling. MTGenRec selects MurmurHash3 [2] as the hash function to determine key indices due to its efficient hash value calculation. MurmurHash3 processes input IDs in 4-byte blocks through mixing operations, which maximize entropy and ensure avalanche effects from single-bit changes. The multi-stage mixing generates uniformly distributed hash values through iterative nonlinear transformations. Hash collisions occur when different keys are mapped to the same index by a hash function. Common techniques for handling collisions include chaining and open addressing. We employ open addressing due to its memory efficiency advantages over chaining. However, existing probing techniques are vulnerable to clustering under high load factors [3].

To address this issue, we propose a novel grouped parallel probing technique. This approach first ensures that the base step size

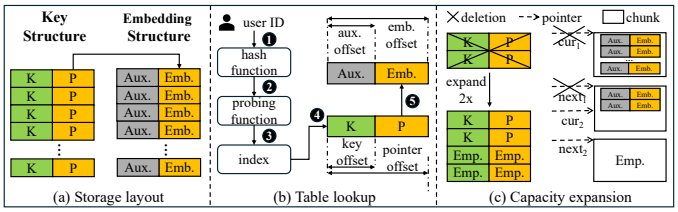


Figure 5: Dynamic embedding table in MTGenRec.

S is an odd number by performing a bitwise OR operation with 1, guaranteeing that all positions in the table can be accessed during probing. Next, we associate the step size with the key to prevent clustering caused by overlapping probing sequences of different keys. Finally, we multiply the step size by the thread group number, enabling distinct thread groups to probe different regions of the table. Formally, the computation is expressed as follows:

$$S = (k\%(M/\text{threads} - 1) + 1 \mid 1) * \text{threads}, \tag{5}$$

where k refers to the key's value, M represents the hash table size, threads indicates the number of thread groups, and $\mid$ denotes the bitwise OR operation. We use coprime conditions to prove that our grouped parallel probing technique can traverse all slots in the hash table. The proof is provided in Appendix B.2.

Theorem 3.1. *For a hash table of size $M = 2^n$ and an odd probing step S, the probe sequence $\{h_t\}_{t=0}^{M-1}, h_t = (h_0 + tS)\%M$ covers all M distinct slots if and only if S is odd. Formally:*

$$\forall i, j \in [0, M-1], i \neq j \Rightarrow h_i \neq h_j. \tag{6}$$

Table Lookup. With the hash function and probing method in place, we can now implement dynamic table lookup. As shown in Figure 5 (b), a hash value is computed for a given user ID, and probing identifies the index corresponding to this ID within the hash table (steps 1-3). Using the starting address of the key structure in memory, along with the index and pointer offset, we determine the embedding pointer p for the user ID (step 4):

$$p = st\_add + index * row\_offset + pointer\_offset, \tag{7}$$

where st_add denotes the start address of the key structure. Finally, we use this pointer, along with the embedding offset in the embedding structure, to extract the embedding vector (step 5).

Capacity Expansion. When the slots in the hash table approach full capacity, expansion becomes necessary. MTGenRec triggers this expansion when the slot occupancy reaches critical levels (i.e., load factor > 0.75). As shown in Figure 5 (c), the capacity increases in a power-of-two progression, doubling iteratively to meet the required capacity. The migration process transfers the original key structure to the expanded version, and the legacy memory is deallocated. Notably, our design prioritizes expanding the key structure while maintaining the chunk (i.e., embedding structure) capacity, thus optimizing expansion efficiency by avoiding bulk transfers of embedding data. To manage the expansion of the embedding structure, MTGenRec maintains two chunks: a current chunk and a next chunk. When the current chunk's remaining capacity is insufficient for new embeddings, they are stored in the next chunk.

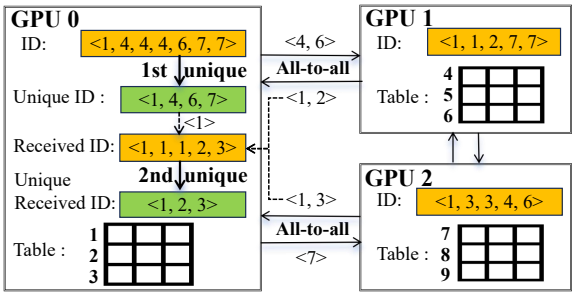


Figure 6: A running example of two-stage ID deduplication. We only visualize the ID deduplication process on GPU 0 and omit the embedding communication.

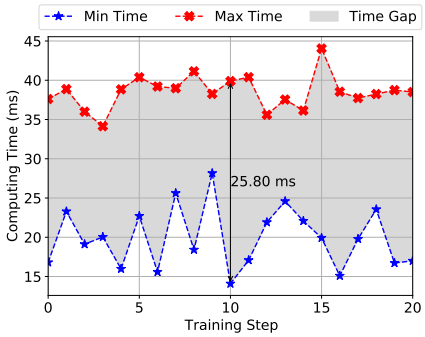


Figure 7: The visualization of imbalanced computational load. We train the MTGenRec on 8 GPUs and report the maximum and minimum GPU computation times across steps 0–20.

Concurrently, we retire the filled chunk and allocate a new chunk as the next chunk to preserve the dual-chunk configuration. This pre-allocation mechanism ensures immediate storage availability for new embeddings during the expansion process.

3.2 Embedding Table Merging

Industrial recommendation systems usually merge multiple embedding tables into a unified table, which improves efficiency by fusing multiple lookup operations into one [15, 46]. However, the merging process requires developers to manually configure each embedding table, which becomes labor-intensive for massive tables. To address this issue, we propose an automated table merging technique.

Automated Merging Table. We design a unified feature configuration interface, `FeatureConfig`, which enables automated table merging by defining parameters for each feature (e.g., *feature name, embedding dimensions, and lookup tables*). MTGenRec then generates merging strategies, such as combining tables with identical embedding dimensions. To support dynamic embedding table merging, we introduce a `HashTableCollection` that uses feature configurations and performs pooling computations as needed. Developers only need to specify the required features, and MTGenRec automatically registers them and executes the table merging without requiring manual intervention.

Previous Solution. Table merging can accelerate the lookup process, however, it introduces numerical overlap in the ID spaces of different tables. For example, user IDs and item IDs might share the same value in the merged table. TorchRec [12] addresses this issue using a row-wise offset mechanism for static table merging. In particular, each embedding table is assigned a unique row offset, which is calculated by summing the number of rows from all preceding tables. This offset is then added to the original ID to generate a globally unique ID, which is used to retrieve the corresponding embedding from the merged embedding table.

Our Solution. Dynamic table merging also faces ID overlap issues. However, the row counts of dynamic tables are unpredictable in advance, which precludes the use of fixed offsets. To resolve this, we dynamically calculate offsets based on the number of tables to prevent ID overflow. Specifically, we use bitwise operations to combine the original ID with a feature table identifier to create a globally unique ID. We first compute the number of bits needed for encoding feature table indices as $k = \lceil \log_2(m+1) \rceil$, using the high k bits of the 64-bit integer space as feature identifier bits. The highest bit is set to 0 to ensure the offset is a positive integer, while the remaining $(64 - 1 - k)$ bits are used to determine the maximum number of rows in the hash table. For example, if we use 2 identifier bits to distinguish between 3 tables (since $\lceil \log_2(3+1) \rceil = 2$), the remaining 61 bits (64-1-2) are used to calculate the maximum row capacity. Consequently, the offset values for "Table 2" and "Table 3" are configured as 2^{59} and 2^{60}, respectively, derived through successive halving of the maximum row capacity. The calculation formula of the ID x in the i-th feature table is expressed as follows:

$$\text{ID} = (i \ll (63 - k)) \mid x, \tag{8}$$

where $\ll$ denotes shift left operation, $\mid$ is bitwise OR. A detailed running case is provided in Appendix B.4 for better understanding.

3.3 Embedding Table Lookup Acceleration

Embedding tables are typically divided into shards distributed across devices. Each table lookup involves two all-to-all collective communications: one for ID exchange and another for embedding vector exchange. The communication cost is directly proportional to the number of features. However, a sequence batch may contain numerous duplicate features IDs, and wasted bandwidth can render the lookup communication as performance bottlenecks. To address this issue, we propose a two-stage ID deduplication technique aimed at reducing communication latency.

Two-stage ID Deduplication. As shown in Figure 6, the first stage involves deduplicating feature IDs on each device before exchanging them, thereby reducing ID communication latency. Despite this optimization, the ID exchange process can reintroduce duplicates due to different devices sending the same ID. Thus, the second deduplication stage is necessary to remove these newly generated duplicates. ID deduplication can significantly alleviate embedding communication latency. If feature IDs are redundant, devices will transmit multiple duplicate embeddings, as they must return the corresponding embedding vectors based on the received feature IDs. Notably, the primary objective of the two-stage ID deduplication is to reduce embedding communication latency, as ID communication has a relatively minor impact.

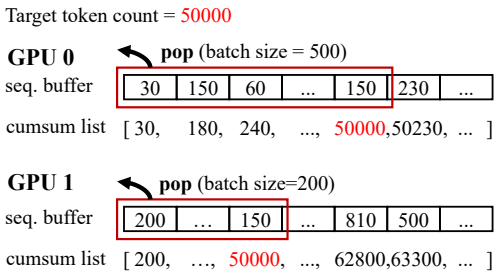


Figure 8: Running example of dynamic sequence batching. The number indicates the number of tokens in a sample.

3.4 Sequence Balancing

In addition to optimizing the sparse model of GRM, we observe that existing methods are still insufficient in optimizing the dense model. User sequences exhibit a long-tail distribution: a small subset of highly active users generate exceptionally long sequences, while the most produce short ones. Previous DRMs are constrained by architectural limitations that enforce sequence truncation and padding for length alignment. In contrast, GRMs require preserving complete user sequences to achieve superior accuracy. This advantage arises because truncation risks removing semantically critical tokens, even when separated by many intermediate interactions.

Initial Design. FlashAttention [4] processes sequences by dividing them into chunks, where each chunk independently computes local attention. The final results are iteratively merged to cover the full sequence. This chunk-based design inherently supports variable-length sequences, motivating our initial approach of directly computing complete user sequences. However, this straightforward batching approach results in significant efficiency challenges. Figure 7 illustrates the GPU computation times per training step. The shaded region in the figure reveals prolonged synchronization delays (up to 25.8 ms) across devices, resulting in resource underutilization and reduced training efficiency. The primary cause is the unequal distribution of sequence lengths, which leads to a load imbalance issue among devices.

Dynamic Sequence Batching. To address the problem, as shown in Figure 8, we introduce a dynamic sequence batching technique. Specifically, each GPU maintains a buffer for storing sequence samples retrieved from Hive table chunks. We define the target token count as the product of average sequence length (i.e., 600) and the batch size. We first compute token counts per sample in the buffer and calculate their cumulative sums. A binary search algorithm then identifies the optimal partition point where the cumulative sum most closely approaches the target token count. We finally obtain a balanced batch by outputting the first k sequences in the buffer. If the total number of tokens in the buffer falls below the target, the remaining samples are merged into the subsequent buffer. This process iterates until all Hive table chunk samples are consumed. A detailed algorithm is provided in Appendix B.5.

In data parallelism, gradients are typically averaged using All-Reduce operations. However, with dynamic sequence batching, directly averaging gradients may introduce biases, as each device computes gradients based on a varying number of samples. To address this issue, we implement All-to-all communication to synchronize batch sizes across devices, followed by performing a weighted average of gradients proportional to their respective batch sizes. This ensures numerical correctness in both loss computation and gradient update processes.

Discussions. Dynamic sequence batching provides two critical advantages: (1) improved device load balancing via token constraints, and (2) optimized hardware utilization via adaptive batch sizes. A fixed batch size strategy requires conservative configurations to prevent out-of-memory risks from clusters of extremely long sequences. In contrast, dynamic batching maximizes device memory utilization by loading batches near device memory limits. While more complex load balancing methods exist, such as sequence packing based on length distributions [18, 20] or dynamic programming-based parallelism strategies [21], we ultimately exclude these methods from our MTGenRec. This decision stems from the empirical observation that our lightweight dynamic sequence balancing achieves near-optimal workload balancing under current training conditions, rendering additional complexity unnecessary.

Besides the above optimizations for sparse and dense model, we incorporate a suite of additional techniques to implement a comprehensive GRM training system, including checkpoint resumption, mixed-precision training, gradient accumulation, and operator fusion. A detailed introduction is provided in Appendix C.

4 Experimental Evaluations

4.1 Experiment Settings

Model Setups. For GRM's dense model, we scale model architecture based on *computational complexity*, resulting in GRM 4G and GRM 110G variants, where "G" corresponds to Giga Floating-Point Operations (GFLOPs) required per forward pass. For sparse models, we use the *embedding dimension factor* to expand the embedding tables, including 1D, 8D, and 64D configurations. The baseline 1D configuration chooses the widely adopted embedding dimensions [27, 37, 42], typically ranging from 32 to 128 (with variations across different features). Accordingly, 64D represents a 64× expansion of all embedding table dimensions relative to this 1D baseline. The detailed model hyperparameters are provided in Appendix D.2.

Baselines. We compare MTGenRec with TorchRec [12], the state-of-the-art GRM training framework proposed by Meta. Actually, our initial design is developed based on TorchRec, and we further implement MTGenRec with the proposed optimizations. We conduct experimental comparisons against TorchRec regarding both model accuracy and efficiency. We exclude alternative baselines as TorchRec is the only PyTorch-native training framework specifically optimized for PyTorch-based GRMs. Although other training systems such as TensorFlow are widely used in the industry, their framework currently cannot support GRM training.

Dataset and Implementation. We conduct experiments on 10 days of user logs from Meituan, with daily log data exceeding 200 million sequences. The average sequence length is 600, reaching a maximum length of 3,000. The detailed dataset statistics are provided in Appendix D.1. All models are trained from scratch using

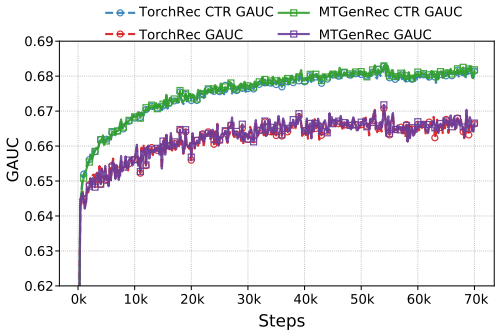


Figure 9: The GAUC results of CTR and CTCVR on GRM 4G 1D for TorchRec and MTGenRec, respectively.

Table 1: Comparison of offline and online performances on the Meituan Take-away application. We report the improved accuracy of GRM compared to DRM.

	Offline Metric diff		Online Metric diff	
	CTR GAUC	CTCVR GAUC	PV_CTR	UV_CTCVR
GRM 4G	+0.0036	+0.0154	+1.04%	+0.04%
GRM 110G	+0.0153	+0.0288	+1.90%	+1.02%

the same data. To optimize sparse and dense features, we employ the Adam [17] optimizer.

Environment. Our experiments are conducted on a training node equipped with 8 GPUs by default. For scaling experiments, the cluster configuration extends to a maximum of 16 GPU nodes, each containing 8 GPUs. All GPUs are NVIDIA A100 SXM4 models with 80GB of memory. GPUs within a node are interconnected via NVLink with a bandwidth of 600 GB/s, while nodes are connected using Infini-Band interconnects with a bandwidth of 200 GB/s.

Evaluation Metrics. We use *Group AUC* (GAUC) to evaluate the accuracy of the model on CTR and CTCVR tasks and system *throughput* to evaluate the training efficiency.

4.2 Comparisons to Baselines

Accuracy and Business Benefits. Figure 9 presents the CTR and CTCVR GAUC results obtained from training a GRM using MTGenRec and TorchRec. The results show that both MTGenRec and TorchRec ensures model correctness and training stability. To further validate the effectiveness of our GRM, we deploy MTGenRec on the Meituan Take-away application, conducting A/B testing with 2% of the traffic. The comparison baseline is the most advanced online DRM (i.e., DLRM [27]), which has been continuously learning for 2 years. As shown in Table 1, although the volume of training data is significantly lower, the offline and online metrics of GRM still greatly exceed those of the baseline.

Time Decomposition. Table 2 presents execution times for GRM 4G 1D and 50G 64D over 100 training steps, detailing embedding lookup, forward, and backward phases. The results show that MTGenRec achieves faster execution than TorchRec across all phases.

Table 2: Time (second) decomposition for GRM 4G 1D and GRM 110G 64D in TorchRec and MTGenRec, respectively.

Model	Method	Lookup	Forward	Backward
GRM 4G 1D	TorchRec	7.0 (s)	14.8 (s)	50.7 (s)
	MTGenRec	3.2 (s)	14.0 (s)	32.6 (s)
GRM 110G 64D	TorchRec	95.2 (s)	27.8 (s)	177.7 (s)
	MTGenRec	14.4 (s)	19.9 (s)	54.7 (s)

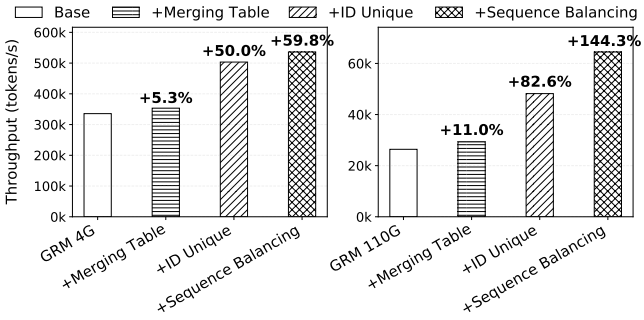


Figure 10: Ablation studies for GRM 4G 1D and GRM 110G 1D in MTGenRec. Compared with the baseline, MTGenRec achieves a 1.60 × to 2.44 × throughput improvement.

Table 3: We report the comparison results in batch size and average GPU memory utilization before and after enabling sequence balancing. Numbers in parentheses denote improvements in GPU memory utilization.

Model	Batch size	GPU memory utilization
GRM 4G 1D	480 → 496	86.3% → 95.7% (+9.4)
GRM 110G 1D	80 → 116	75.3% → 90.3% (+15.0)

The efficiency improvements from sequence balancing increase substantially with model complexity. Larger embedding dimensions induce significant increases in the lookup and backward phases due to communication overhead and sparse parameter update latency.

4.3 Evaluation of Individual Designs

Ablation Study. Figure 10 presents an ablation study under 4G and 10G complexity settings by incrementally integrating different techniques in MTGenRec, including merging table, two-stage ID deduplication and sequence balancing. The experimental results demonstrate all these techniques in MTGenRec can enhance system efficiency. Furthermore, we observe that performance gains are more pronounced with higher computational complexity. This trend reveals a positive correlation between MTGenRec 's efficiency improvements and computational complexity.

Sequence Balancing Evaluation. We evaluate sequence balancing on GRM 4G and 110G models, scaling from 8 to 64 GPUs. To ensure fairness, batch sizes are adjusted to maintain near-full GPU memory utilization under both enabled and disabled conditions. Figure 11 presents the throughput results, and table 3 reports batch

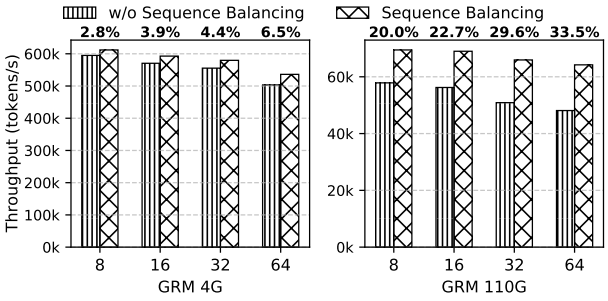


Figure 11: Throughput comparison for disabling and enabling sequence balancing on GRM 4G 1D and 110G 1D. We scale the system from 8 GPUs to 64 GPUs and report the gain values of sequence balancing at the top of figure.

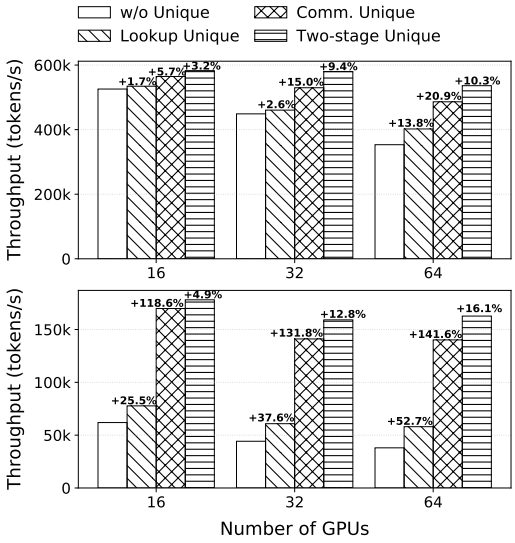


Figure 12: Throughput comparisons of two-stage ID deduplication for GRM 4G 1D (upper) and GRM 4G 64D (bottom). The improvements of the current strategy compared to the previous one are reported above the bars.

sizes and average GPU memory utilization, yielding four key observations. (1) Sequence balancing significantly improves throughput across all settings (e.g., an average improvements of 26.5% for GRM 110G, peaking at 33.5% on 64 GPUs). (2) Throughput gains scale with GPU count due to reduced synchronization delay from straggler devices. As the number of GPUs increases, the likelihood of encountering long sequences that induce computational delays also rises. (3) Throughput improvements correlate positively with computational complexity. Since sequence processing FLOPs scale quadratically with hidden embedding dimensions, higher computational complexity amplifies load imbalance, which makes sequence balancing increasingly critical. (4) Table 3 shows that sequence balancing maximizes GPU memory utilization, whereas fixed-size batching requires conservative sizing to avoid OOM errors.

Table 4: Throughput comparison for MCH and MTGenRec. *OOM* denotes out of memory and *Gain* denotes the throughput improvement. "-" indicates that the comparison cannot be made due to OOM problems.

Complexity	Dim. Factor	MCH	MTGenRec	Gain
4G	1D	392,731	579,649	47.59%
	8D	260,590	438,190	68.2%
	64D	80,857	155,832	92.7%
110G	1D	44,719	68,993	54.28%
	8D	28,329	55,929	97.4%
	64D	OOM	38,575	-

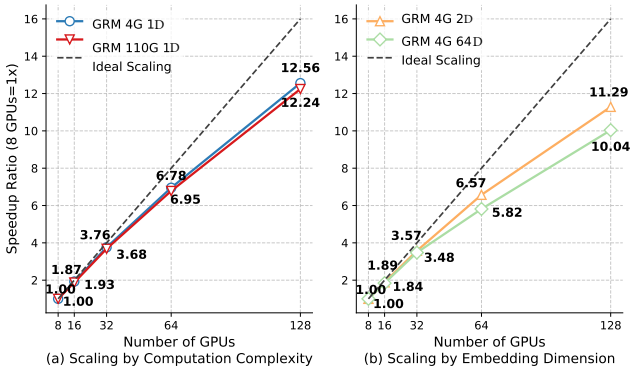


Figure 13: Comparison of scalability by computational complexity and embedding dimension. We use 8 GPUs configuration as the baseline for speedup ratio. The specific speedup ratio values are marked near the broken line.

The Effect of Two-stage ID Deduplication. To evaluate the impact of two-stage ID deduplication on system efficiency, we conduct experiments at 4 GFLOPs complexity with 1D and 64D embedding dimensions. We employ four strategies: (a) *w/o unique* (no deduplication), (b) *Comm. unique* (first-stage deduplication only), (c) *Lookup unique* (second-stage deduplication only), and (d) *Two-stage unique* (both stages). Starting with 16 GPUs, we incrementally increase the GPU count and report throughput results in Figure 12. We get the following observations. (1) The two-stage approach improves throughput by 1.1–3.7× over the baseline (a), highlighting its effectiveness in reducing ID communication and embedding communication. (2) "Comm. unique" outperforms "Lookup unique" because embedding communication is the primary source of latency, while hash table lookups are inherently fast. (3) The benefits of deduplication increase with embedding dimensions: higher-dimensional embeddings exacerbate network bandwidth waste, making ID deduplication more crucial for reducing communication latency.

The Effect of Dynamic Embedding Table. We evaluate the proposed dynamic embedding table's effectiveness with TorchRec's Managed Collision Handling [12] (MCH) as the baseline. MCH remaps changeable IDs into a continuous space via a fixed-size mapping table. It employs binary search for ID localization and

invokes eviction when a threshold is met. Experimental results are shown in Table 4. Our findings reveal that the dynamic embedding table achieves 1.47–2.22× throughput improvement over MCH. This improvement stems from the grouped parallel probing technique, which accelerates hash collision resolution via multi-threaded parallel probing. Additionally, MCH encounters OOM issues at larger embedding dimensions due to pre-allocated memory for all tables. In contrast, the dynamic embedding table requires substantially less initial memory and allocates additional resources only when thresholds are exceeded.

4.4 Scalability

We assess the system's scalability by examining deviation between actual speedup and ideal linear scaling for two factors: computational complexity and embedding dimensions. We fix either dimension or complexity while scaling the other, and use 8 GPUs as baseline up to 128 GPUs. The experimental results in Figure 13 reveal three key findings. (1) All systems exhibit sublinear scaling with increasing GPUs due to communication overhead. However, MTGenRec achieves 62.75%–78.5% of the ideal speedup at 128 GPUs, demonstrating its robust scalability. (2) Scaling computational complexity by 27.5× (4G 1D vs. 110G 1D) and embedding dimensions by 32× (4G 2D vs. 4G 64D) causes minor speedup degradation. The stability arises from sequence balancing, which mitigates load imbalance, and table merging and two-stage ID deduplication also reduce the lookup and communication latency. (3) Embedding dimensions impact speedup more significantly than computational complexity, as latency is dominated by sparse embedding communication, computation, and updates.

5 Related Work

Systems for DRM Training. Deep Recommendation Models (DRMs) combine sparse embeddings and dense neural networks to model each user-item individually. As DRM is widely used, many systems are developed to accelerate it [7–9, 15, 16, 19, 22, 25, 33, 39, 41, 47]. Specifically, Persia [22] decouples sparse embedding computation from dense model processing, training the embeddings asynchronously to improve throughput and the dense models synchronously to ensure accuracy. PICASSO [46] leverages the patterns of model architecture and data distribution to accelerate execution with packing, interleaving, and caching optimizations. XDL [15] compresses the pair-wise communication among GPUs into a tree structure to accelerate embedding lookup and update. ScaleFreeCTR [9] leverages host memory to store massive embedding tables and employs GPU-optimized synchronization to update the embeddings. Zion [33] and RecSpeed [19] mitigate the I/O bottlenecks for embedding communication by deploying additional NICs. However, these systems are typically designed based on specific assumptions about the access patterns of sparse embeddings (e.g., skewed towards frequently accessed embeddings), which limits their applicability compared to our dynamic embedding table. Additionally, these systems are implemented using TensorFlow, making them unsuitable for PyTorch-based GRM training.

Systems for Transformer. To train Transformer efficiently, the focuses are attention computation and parallel strategies [4, 11, 14, 21, 23, 29–32, 36, 40]. FlashAttention [4] reduces the accesses to slow GPU global memory with tiled computation and online recomputation fusion. FlexFlow [23] and OptCNN [14] automatically search the optimal parallel strategy for model training by distributing model computation and parameters among the GPUs in different ways. GPipe [11] employs pipeline parallelism by splitting each mini-batches into micro-batches to interleave the computation and communication across the micro-batches. Megatron-LM [32] and DeepSpeed [30] propose hybrid parallel strategies for large-scale model training by partitioning the model and data across the GPUs. Hydraulis [21] adopts dynamic parallelism for each iteration according to the sequence length distribution to achieve workload balance. MTGenRec enjoys existing attention kernel optimizations for efficient computation but simple data parallel suffices for GRMs since their dense models are usually small. Moreover, MTGenRec optimizes the processing of sparse embeddings, which are not present in most Transformer-based models.

6 Conclusion

We propose MTGenRec, a distributed training system designed to enhance scalability and efficiency of industrial generative recommendation models (GRMs). Our sparse model optimizations include dynamic embedding tables for changeable IDs, automated table merging, and two-stage ID deduplication. For dense models, dynamic sequence batching mitigates workload imbalances. The experiment results demonstrate substantially improved training efficiency without compromising accuracy, paving the way for broader adoption of GRMs in recommendation systems.

Acknowledgments

This work was sponsored by National Natural Science Foundation of China (62472327), Sichuan Clinical Research Center for Imaging Medicine (YXYX2402), Geological Hazard Prevention and Control Project of the Three Gorges Follow-up Work of the National Major Water Conservancy Project Construction Fund (0001212024CC60003), and the Innovative Research Group Project of Hubei Province (2024AFA017). This work was supported by Meituan through the Efficient Training and Inference System for Generative Recommendation Models project (MT20250112053P).

References

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[2] Austin Appleby. 2008. MurmurHash. https://github.com/aappleby/smhasher. Accessed: 2023-10-15.

[3] Michael A Bender, Bradley C Kuszmaul, and William Kuszmaul. 2022. Linear probing revisited: Tombstones mark the demise of primary clustering. In *2021 IEEE 62nd annual symposium on foundations of computer science (FOCS)*. IEEE, 1171–1182.

[4] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems* 35 (2022), 16344–16359.

[5] Jiaxin Deng, Shiyao Wang, Kuo Cai, Lejian Ren, Qigen Hu, Weifeng Ding, Qiang Luo, and Guorui Zhou. 2025. OneRec: Unifying Retrieve and Rank with Generative Recommender and Iterative Preference Alignment. *arXiv preprint arXiv:2502.18965* (2025).

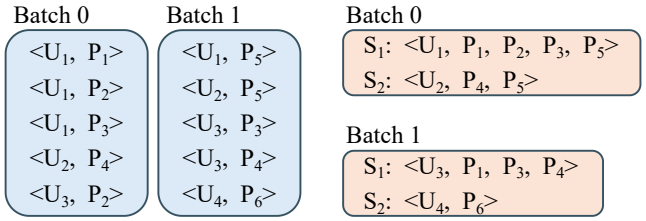
[6] Stefan Elfwing, Eiji Uchibe, and Kenji Doya. 2018. Sigmoid-weighted linear units for neural network function approximation in reinforcement learning. *Neural networks* 107 (2018), 3–11.

[7] Miao Fan, Jiacheng Guo, Shuai Zhu, Shuo Miao, Mingming Sun, and Ping Li. 2019. MOBIUS: Towards the next generation of query-ad matching in baidu's

sponsored search. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery and data mining*. 2509–2517.
[8] Yufei Feng, Binbin Hu, Fuyu Lv, Qingwen Liu, Zhiqiang Zhang, and Wenwu Ou. 2020. Atbrg: Adaptive target-behavior relational graph network for effective recommendation. In *Proceedings of the 43rd international ACM SIGIR conference on research and development in information retrieval*. 2231–2240.
[9] Huifeng Guo, Wei Guo, Yong Gao, Ruiming Tang, Xiuqiang He, and Wenzhi Liu. 2021. 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*. 1269–1278.
[10] Lei Huang, Hao Guo, Linzhi Peng, Long Zhang, Xiaoteng Wang, Daoyuan Wang, Shichao Wang, Jinpeng Wang, Lei Wang, and Sheng Chen. 2025. SessionRec: Next session prediction paradigm for generative sequential recommendation. *arXiv preprint arXiv:2502.10157* (2025).
[11] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. 2019. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems* 32 (2019).
[12] Dmytro Ivchenko, Dennis Van Der Staay, Colin Taylor, Xing Liu, Will Feng, Rahul Kindi, Anirudh Sudarshan, and Shahin Sefati. 2022. Torchrec: A pytorch domain library for recommendation systems. In *Proceedings of the 16th ACM conference on recommender systems*. 482–483.
[13] Jianchao Ji, Zelong Li, Shuyuan Xu, Wenyue Hua, Yingqiang Ge, Juntao Tan, and Yongfeng Zhang. 2024. Genrec: Large language model for generative recommendation. In *European conference on information retrieval*. Springer, 494–502.
[14] Zhihao Jia, Sina Lin, Charles R Qi, and Alex Aiken. 2018. Exploring hidden dimensions in parallelizing convolutional neural networks. In *Proceedings of the international conference on machine learning*, Vol. 2279. 2288.
[15] Biye Jiang, Chao Deng, Huimin Yi, Zelin Hu, Guorui Zhou, Yang Zheng, Sui Huang, Xinyang Guo, Dongyue Wang, Yue Song, et al. 2019. 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*. 1–9.
[16] Peng Jiang and Gagan Agrawal. 2018. A linear speedup analysis of distributed deep learning with sparse and quantized communication. *Advances in neural information processing systems* 31 (2018).
[17] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).
[18] Mario Michael Krell, Matej Kosec, Sergio P Perez, and Andrew Fitzgibbon. 2021. Efficient sequence packing without cross-contamination: Accelerating large language models without impacting performance. *arXiv preprint arXiv:2107.02027* (2021).
[19] Suresh Krishna and Ravi Krishna. 2020. Accelerating recommender systems via hardware" scale-in". *arXiv preprint arXiv:2009.05230* (2020).
[20] Achintya Kundu, Rhui Dih Lee, Laura Wynter, Raghu Kiran Ganti, and Mayank Mishra. 2024. Enhancing training efficiency using packing with flash attention. *arXiv preprint arXiv:2407.09105* (2024).
[21] Haoyang Li, Fangcheng Fu, Sheng Lin, Hao Ge, Xuanyu Wang, Jiawen Niu, Jie Jiang, and Bin Cui. 2024. Demystifying workload imbalances in large transformer model training over variable-length sequences. *arXiv preprint arXiv:2412.07894* (2024).
[22] Xiangru Lian, Binhang Yuan, Xuefeng Zhu, Yulong Wang, Yongjun He, Honghuan Wu, Lei Sun, Haodong Lyu, Chengjun Liu, Xing Dong, et al. 2022. Persia: An open, hybrid system scaling deep learning-based recommenders up to 100 trillion parameters. In *Proceedings of the 28th ACM SIGKDD conference on knowledge discovery and data mining*. 3288–3298.
[23] Wenyan Lu, Guihai Yan, Jiajun Li, Shijun Gong, Yinhe Han, and Xiaowei Li. 2017. Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks. In *2017 IEEE international symposium on high performance computer architecture (HPCA)*. IEEE, 553–564.
[24] Jiaqi Ma, Zhe Zhao, Xinyang Yi, Jilin Chen, Lichan Hong, and Ed H Chi. 2018. Modeling task relationships in multi-task learning with multi-gate mixture-of-experts. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery and data mining*. 1930–1939.
[25] Xupeng Miao, Hailin Zhang, Yining Shi, Xiaonan Nie, Zhi Yang, Yangyu Tao, and Bin Cui. 2021. HET: Scaling out huge embedding model training via cache-enabled distributed framework. *arXiv preprint arXiv:2112.07221* (2021).
[26] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. 2019. PipeDream: Generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM symposium on operating systems principles*. 1–15.
[27] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G Azzolini, et al. 2019. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091* (2019).
[28] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. 2018. Improving language understanding by generative pre-training. (2018).
[29] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. Zero: Memory optimizations toward training trillion parameter models. In *International conference for high performance computing, networking, storage and analysis*. IEEE, 1–16.
[30] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery and data mining*. 3505–3506.
[31] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. 2021. Zero-offload: Democratizing billion-scale model training. In *2021 USENIX annual technical conference*. 551–564.
[32] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053* (2019).
[33] Misha Smelyanskiy. 2019. Zion: Facebook next-generation large memory training platform. In *2019 31th IEEE hot chips symposium*. IEEE, 1–22.
[34] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).
[35] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).
[36] Guanhua Wang, Heyang Qin, Sam Ade Jacobs, Connor Holmes, Samyam Rajbhandari, Olatunji Ruwase, Feng Yan, Lei Yang, and Yuxiong He. 2023. Zero++: Extremely efficient collective communication for giant model training. *arXiv preprint arXiv:2306.10209* (2023).
[37] Ruoxi Wang, Bin Fu, Gang Fu, and Mingliang Wang. 2017. Deep & cross network for ad click predictions. In *Proceedings of the ADKDD'17*. 1–7.
[38] Wenjie Wang, Xinyu Lin, Fuli Feng, Xiangnan He, and Tat-Seng Chua. 2023. Generative recommendation: Towards next-generation recommender paradigm. *arXiv preprint arXiv:2304.03516* (2023).
[39] Zehuan Wang, Yingcan Wei, Minseok Lee, Matthias Langer, Fan Yu, Jie Liu, Shijie Liu, Daniel G Abel, Xu Guo, Jianbing Dong, et al. 2022. Merlin hugectr: Gpu-accelerated recommender system training and inference. In *Proceedings of the 16th ACM conference on recommender systems*. 534–537.
[40] Houming Wu, Ling Chen, and Wenjie Yu. 2024. BitPipe: Bidirectional interleaved pipeline parallelism for accelerating large models training. *arXiv preprint arXiv:2410.19367* (2024).
[41] Minhui Xie, Kai Ren, Youyou Lu, Guangxu Yang, Qingxing Xu, Bihai Wu, Jiazhen Lin, Hongbo Ao, Wanhong Xu, and Jiwu Shu. 2020. Kraken: Memory-efficient continual learning for large-scale real-time recommendations. In *International conference for high performance computing, networking, storage and analysis*. IEEE, 1–17.
[42] Jun Xu, Xiangnan He, and Hang Li. 2018. Deep learning for matching in search and recommendation. In *The 41st International ACM SIGIR conference on research and development in information retrieval*. 1365–1368.
[43] Songpei Xu, Shijia Wang, Da Guo, Xianwen Guo, Qiang Xiao, Fangjian Li, and Chuanjiang Luo. 2025. An efficient large recommendation model: Towards a resource-optimal scaling law. *arXiv preprint arXiv:2502.09888* (2025).
[44] Bencheng Yan, Shilei Liu, Zhiyuan Zeng, Zihao Wang, Yizhen Zhang, Yujin Yuan, Langming Liu, Jiaqi Liu, Di Wang, Wenbo Su, et al. 2025. Unlocking scaling law in industrial recommendation systems with a three-step paradigm based large user model. *arXiv preprint arXiv:2502.08309* (2025).
[45] Jiaqi Zhai, Lucy Liao, Xing Liu, Yueming Wang, Rui Li, Xuan Cao, Leon Gao, Zhaojie Gong, Fangda Gu, Michael He, et al. 2024. Actions speak louder than words: Trillion-parameter sequential transducers for generative recommendations. *arXiv preprint arXiv:2402.17152* (2024).
[46] Yuanxing Zhang, Langshi Chen, Siran Yang, Man Yuan, Huimin Yi, Jie Zhang, Jiamang Wang, Jianbo Dong, Yunlong Xu, Yue Song, et al. 2022. PICASSO: Unleashing the potential of GPU-centric training for wide-and-deep recommender systems. In *2022 IEEE 38th international conference on data engineering*. IEEE, 3453–3466.
[47] Weijie Zhao, Deping Xie, Ronglai Jia, Yulei Qian, Ruiquan Ding, Mingming Sun, and Ping Li. 2020. Distributed hierarchical gpu parameter server for massive scale deep learning ads systems. *Proceedings of machine learning and systems* 2 (2020), 412–428.
[48] Guorui Zhou, Xiaoqiang Zhu, Chenru Song, Ying Fan, Han Zhu, Xiao Ma, Yanghui Yan, Junqi Jin, Han Li, and Kun Gai. 2018. Deep interest network for click-through rate prediction. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery and data mining*. 1059–1068.

A The Differences between GRM and DRM

Our model builds upon GR [45] by introducing a novel prediction head enhanced through MMoE. This design replaces GR's MLP



(a) Batch in DRM (b) Batch in GRM

Figure 14: Batch structure of DRM and GRM. "*U*" denotes users, "*P*" denotes products. DRM and GRM use a pairwise and sequence-wise batch construction method, respectively.

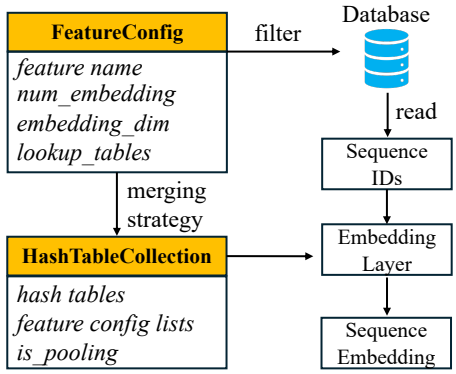


Figure 15: We achieve automatic table merging by two embedding configuration interfaces.

with MMoE, which employs task-specific gate networks to dynamically balance multiple objectives, improving accuracy in multi-task scenarios. Compared to MLP's parameter-intensive tuning, MMoE reduces computational overhead by selectively activating only the top-k expert outputs. Additionally, our MTGenRec diverges from DRM in batch construction. As illustrated in Figure 14, DLRM's pairwise batches contain redundant user computations (e.g., "User 1" features are processed twice). In contrast, MTGenRec employs a sequence-wise approach that consolidates multiple user interactions into a single sample, which ensures each user feature is computed once. Notably, user features contain orders of magnitude more tokens than item features (e.g., 10,000 vs. 100), enabling MTGenRec to significantly accelerate training and improve scalability.

B Additionally Details for MTGenRec

B.1 Pipeline.

We maximize parallelism through pipeline technology using three streams: *copy*, *dispatch*, and *compute*. Specifically, the copy stream loads sequences from CPU to GPU, the dispatch stream performs table lookups based on IDs, and the compute stream handles forward computation and backward updates.

B.2 Proof

We use coprime conditions and the characteristics of moduli to prove that our grouped parallel probing technique can traverse all slots in the hash table.

Lemma B.1. *If $M = 2^n$ and S is odd, then* $\gcd(S, M) = 1$. gcd *denotes the greatest common divisor.*

Proof. S contains no factor of 2 (since S is odd), while M has prime factorization 2^n. Thus, S and M share no common prime factors, proving $\gcd(S, M) = 1$. □

Proof. Assume there exist $t_1, t_2 \in [0, M-1]$, $t_1 \neq t_2$ such that:

$$h_{t_1} \equiv h_{t_2}, \tag{9}$$

This implies:

$$(t_1 - t_2) \cdot S \equiv 0. \tag{10}$$

By Lemma 1, $\gcd(S, M) = 1$. According to Bézout's identity, S has a multiplicative inverse modulo M. Multiplying both sides by this inverse yields:

$$t_1 - t_2 \equiv 0. \tag{11}$$

Since $t_1, t_2 \in [0, M-1]$, the only solution is $t_1 = t_2$, contradicting the initial assumption. Hence, all probe positions are unique.

□

Proof. By the pigeonhole principle, M probes generate M unique positions, exactly covering all slots:

$$\{h_t\}_{t=0}^{M-1} = \{0, 1, 2, ..., M-1\}. \tag{12}$$

□

B.3 Interface for Automated Table Merging

Figure 15 illustrates our unified feature configuration interface, `FeatureConfig`, which enables automatic table merging by defining a configuration entry for each feature based on its *name*, *embedding vocabulary size*, *embedding dimension*, and *lookup table*. Subsequently, MTGenRec reads data from the database according to feature names and generates a table merging strategy (e.g., merging tables with identical embedding dimensions). To support the merging of dynamic embedding tables, we design `HashtableCollection`, which generates a collection of hash tables based on the embedding configurations (e.g., *feature name, embedding dimension*) within the `FeatureConfig` list and performs pooling operations if required. Finally, MTGenRec retrieves the corresponding embeddings for sequence IDs from the embedding layer.

B.4 Running Example for Dynamic Offset

As shown in Figure 16 (a), a unique row offset value is assigned to each embedding table (i.e., offset 100 for table 2), calculated by summing the number of rows from all preceding embedding tables. In contrast, as illustrated in Figure 16 (b), we use 2 identifier bits to distinguish between three tables, while calculating the maximum row capacity of hash table as 2^{61} through utilization of the remaining 61 bits.

B.5 Algorithm for Sequence Balancing

Algorithm 1 depicts MTGenRec's dynamic sequence batching. Each GPU maintains a sequence buffer Q populated from hive table chunks $C_i \in C$. The target token count N equals the batch size multiplied by the average sequence length. Token counts are aggregated into a cumulative sum S over samples in Q. A binary search finds the partition index k where $S(k)$ best approximates N. The

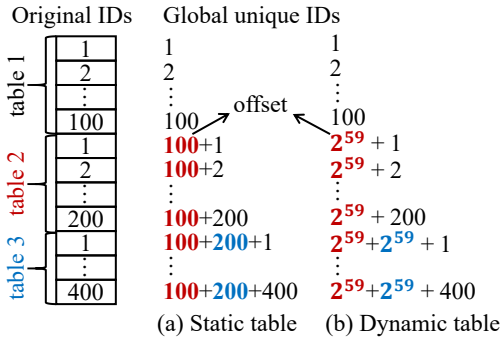


Figure 16: An example of offsets in static table and hash table.

Algorithm 1: Dynamic Sequence Batching

Input: Target token count N, input chunks C
$Q \leftarrow \emptyset$, $C_i \in C$, $i = 0$
while $C_i \neq \emptyset$ **do**
  if $\texttt{sum}(Q) < N$ **then**
    $Q \leftarrow$ add all sequence in C_i
    $i++$
  $S \leftarrow$ cumulative sum of sequence token count in Q
  $k \leftarrow$ binary search in S for closest value to N
  if $k < \texttt{len}(Q)$ **then**
    $\mathcal{B} \leftarrow \texttt{pop}(Q[:k])$
    yield $\mathcal{B}$
Output: Balanced batch $\mathcal{B}$

first k sequences in Q form the output batch $\mathcal{B}$. Under-filled buffers accumulate samples until processing exhausts all hive chunk data.

C Implementation Details for MTGenRec

Checkpoint Resuming. The core objective of checkpoint is to achieve persistent model preservation during training, enabling rapid recovery to previous training states. MTGenRec not only supports conventional checkpoint resuming but also facilitates preservation and loading in dynamic distributed environments (e.g., saving on 8 GPUs but loading on 16 GPUs). Existing systems address this challenge by saving model parameters from all devices in a single checkpoint, then redistributing parameters based on new device quantities during loading. However, this approach suffers from a critical flaw: each device must scan the entire checkpoint, resulting in redundant read operations. In contrast, MTGenRec implements a novel approach where each device independently preserves its own checkpoint. During loading, new devices locate required checkpoint files through modulo operations. For instance, when loading checkpoints saved from 8 GPUs onto 16 GPUs, both GPU 0 and GPU 8 load parameters from the checkpoint saved on the original GPU 0. This design is grounded in the insight that distributed cluster scaling typically follows powers of two.

Mixed Precision Training. During training, we reduce computational resource consumption by converting dense models from FP32 to FP16 precision. For sparse embeddings, we dynamically partition hot feature sets based on access frequency. Specifically, for high-frequency accessed feature embeddings, we preserve embedding vectors in FP32 format to avoid quantization accumulation errors caused by frequent gradient updates. Conversely, low-frequency features employ FP16 storage and computation, significantly reducing memory footprint while accelerating table lookup operations.

Gradient Accumulation. MTGenRec enhances training stability by accumulating gradients across multiple training batches to mitigate large gradient variances caused by small batch sizes. For sparse embedding parameters, we first record activated embedding IDs and their corresponding gradient values within each batch. These gradients from identical IDs across multiple batches are accumulated and then updated collectively. Furthermore, we avoid full parameter updates for sparse embeddings, instead selectively updating only activated parts. This design simultaneously reduces memory waste and improves update efficiency. For smaller dense models, we also implement gradient accumulation followed by full parameter updates.

Operator Fusion. Inspired by Flash Attention, we implement customized operator fusion for the critical HSTU module in forward computation. Specifically, we partition the U, Q, K, and V matrices in Figure 3 into tiles and process them sequentially in SRAM. Crucially, we use casual mask vectors to reduce unnecessary calculations by dynamically determining token skipping.

Table 5: Dataset Statistics

Dataset	#Users	#Items	#Exposure	#Click	#Purchases
Train	0.21 billion	4,302,391	23.74 billion	1.08 billion	0.18 billion
Test	3,021,198	3,141,997	76,855,608	4,545,386	769,534

Table 6: Model hyperparameters in GRM. Complexity denotes computational complexity, and the unit is FLOPs.

	Complexity	# Emb. dim.	# HSTU block	# HSTU head
Small	4G	512	3	2
Large	110G	1024	22	4

D Experiment Details

D.1 Datasets

We construct a training dataset based on logs from the real industrial-scale recommendation system in Meituan. The detailed dataset statistics are reported in Table 5. Unlike public datasets, our real dataset contains a richer cross features set and longer user behavior sequences. In addition, the volume of our dataset is large, allowing complex models to achieve more adequate convergence during training. For the offline experiments, we collect data over a 10-day period. For the online experiments, in order to compare with the DLRM baseline that has been trained for over 2 years, we constructed a longer-term dataset for the experiments, using data spanning more than 6 months.

D.2 Model Hyperparameters

We use two models: small and large GRM for all the experimental evaluations. The detailed model hyperparameters are provided in Table 6.