FREESCALE: DISTRIBUTED TRAINING FOR SEQUENCE RECOMMENDATION MODELS WITH MINIMAL SCALING COST

Chenhao Feng [*1] **Haoli Zhang** [*1] **Shakhzod Ali-Zade** [1] **Yanli Zhao** [1] **Liang Luo** [1] **Jennifer Cao** [1] **Lisen Deng** [1] **Siqiao Chen** [1] **Chenyu Zhao** [1] **Tristan Rice** [1] **Daniel Johnson** [1] **Min Si** [1] **Tiantu Xu** [1] **Yi Zhang** [1] **Siqi Yan** [1] **Chuanhao Zhuge** [1] **Min Ni** [1] **Bi Xue** [1] **Qunshu Zhang** [1] **Shen Li** [1]

ABSTRACT

Modern industrial Deep Learning Recommendation Models typically extract user preferences through the analysis of sequential interaction histories, subsequently generating predictions based on these derived interests. The inherent heterogeneity in data characteristics frequently result in substantial under-utilization of computational resources during large-scale training, primarily due to computational bubbles caused by severe stragglers and slow blocking communications. This paper introduces FreeScale, a solution designed to (1) mitigate the straggler problem through meticulously load balanced input samples (2) minimize the blocking communication by overlapping prioritized embedding communications with computations (3) resolve the GPU resource competition during computation and communication overlapping by communicating through SM-Free techniques. Empirical evaluation demonstrates that FreeScale achieves up to 90.3% reduction in computational bubbles when applied to real-world workloads running on 256 H100 GPUs.

1 INTRODUCTION

Deep Learning Recommendation Models (DLRMs) (Naumov et al., 2019; Luo et al., 2024) have become the foundation for numerous industrial applications, including e-Commerce platforms, social media networks and digital advertising systems (Zhai et al., 2024a; Zeng et al., 2024; Steck et al., 2021; Zhao et al., 2021). These models are designed to predict user preferences by analyzing interaction patterns within historical data. The efficacy of these models is largely contingent upon their ability to accurately extract user interests from extensive historical interaction sequences.

The volume and composition of Users' Interaction History (UIH) exhibit significant heterogeneity across the user population. (Liu et al., 2023; Wang et al., 2021). This inherent variability introduces computational stragglers and inefficient communication patterns during model training scaling up. The stragglers emerge when workloads with substantially different execution times are allocated to different computational processes, causing processes with lighter workloads to remain idle while awaiting the completion of more computationally intensive ones. Meanwhile, the inefficienct communication patterns are due to blocking communication operations for embedding lookups that prevent concurrent computation, creating additional idle periods. In large-scale training involving hundreds of GPUs, these inefficiencies leads to extended training durations and significant financial waste. Addressing these inefficiencies has become critical to maintaining a high return on investment.

This paper introduces FreeScale, a holistic solution designed to systematically minimize the scaling cost in general distributed deep learning based recommender system training, which consists of:

- **Sequence Load Balancing**: FreeScale balances input samples by estimating their computational complexity to mitigate stragglers, ensuring more uniform execution times across all processes.
- **Prioritized Embedding Updates**: FreeScale identifies collision rows between consecutive training iterations to minimize blocking communications, allowing for greater overlapping with computation operations.
- **SM-Free Communication**: FreeScale communicates via CPU-RDMA, resolving GPU resource competition issue during communication and computation overlapping.

FreeScale is implemented on top of PyTorch (Paszke et al., 2019) and TorchRec (Ivchenko et al., 2022), with customized Triton (Tillet et al., 2019) kernels. Evaluation on

[*]Equal contribution [1]Meta Inc. Correspondence to: Shen Li <cs.shenli@gmail.com>.

Proceedings of the 9th MLSys Conference (Industry Track), Bellevue, WA, USA, 2026. Copyright 2026 by the author(s).

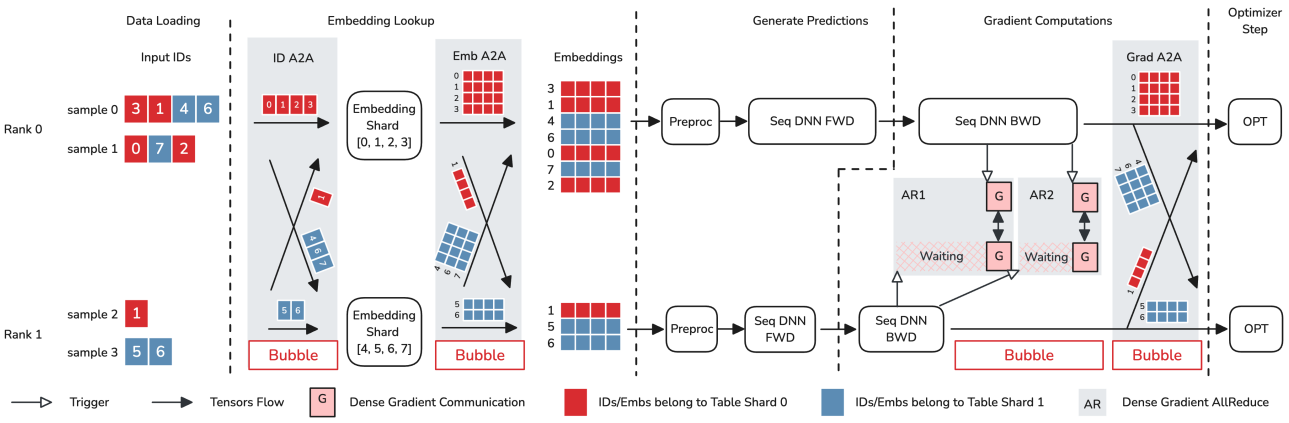

Figure 1. Simplified Sequence Model Training Iteration

real-world dataset with a cluster of 256 NVIDIA H100 GPUs demonstrates that FreeScale reduced exposed communications by 90.3% compared to vanilla TorchRec.

The remainder of this paper is organized as follows. Section 2 presents the background of efficiency challenges in recommendation systems. Section 3 details the underlying components of FreeScale. Implementations are discussed in Section 4. Section 5 presents our experimental setup and performance evaluation results. Section 6 illustrate more of FreeScale's enggierning analysis and SM Free communication. Section 7 summarizes related work, and Section 8 concludes the paper.

2 Background

This section briefly introduces sequence modeling in DLRMs, and highlights two unique efficiency concerns in distributed training for sequence models.

2.1 Sequence Modeling in Recommendation Systems

Depending on the problem, sequence has different meanings. In factorization-machine based recommendation (Zhang et al., 2024; Luo et al., 2024; Zhang et al., 2022), sequences represent abstract content IDs of past user interactions. The ID list is converted to a sequence of embeddings and a summarization architecture is applied to learn hidden information between user-item interations (Zeng et al., 2025; Employees, 2023). In generative recommendation, sequences serve as the flattened token list and the model outputs for the action tokens of whether the user interacted with the current item for the ID tokens, transducing the recommendation task into a next-token prediction task (Zhai et al., 2024b). Alternatively, as shown in (Wang et al., 2024), the sequence can represent organic content as well (e.g., a tokenized text post).

Since our proposed techniques apply to all these settings, in this paper, we do not our limit ourselves to specific problem formulations.

To provide a concrete example of sequence model training, Figure 1 illustrates a single distributed training iteration. Each computational process exclusively operates on a dedicated GPU, denoted by a unique rank. This example reflects how a typical model parameters distribution works in the recommendation systems, which consists of massive sparse embedding tables alongside substantially smaller dense Deep Neural Network (DNN) layers. Under such settings, the embedding tables are partitioned while the DNN layers are replicated across GPUs. Each rank independently retrieves inputs with UIH sequences of embedding IDs. During forward, these IDs are distributed to their corresponding partition through an `AllToAll` operation to perform embedding lookup, after which the resulting values are returned to their originating rank via a second `AllToAll` operation. During backward, the dense gradients are synchronized using `AllReduce` operations (Li et al., 2020), while embedding gradients are shuffled to their respective partitions through `AllToAll`.

Due to the nature of collective communications, the straggler rank forces other ranks to wait. In this example, rank 0 processes more IDs and hence becomes the straggler of this iteration, forcing rank 1 to wait for rank 0's operations. Also, blocking `AllToAll` communication on the critical path forces subsequent operations to wait. Scaling out to larger clusters will exacerbate these 2 issues.

2.2 Imbalanced Workload and Stragglers

Figure 2 presents empirical measurements of local batch sparsity and global straggler effects in DLRM training on real-world production workload. Sparsity, quantified ac-

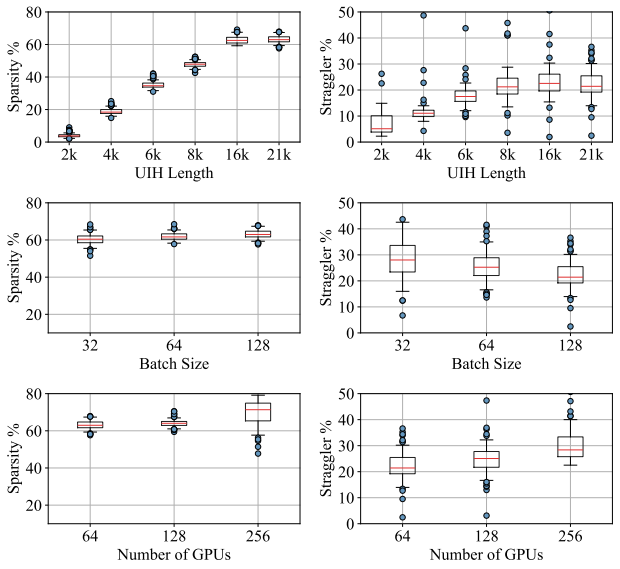

Figure 2. Sparsity and Straggler Percentage

cording to Equation 1, is calculated as the proportion of padded items required when normalizing all samples to the maximum UIH length within a given batch. The straggler percentage metric represents the ratio of the mean idle wait time across all ranks relative to the total duration of the iteration. Unless otherwise specified, all experiments employ 21,000 max UIH length, 128 batch size, and 64 H100 GPUs.

$$sparsity = 1 - \frac{\sum_{i=1}^{B} |uih_i|}{\max(|uih_i|, \forall i \in [1, B]) \times B} \quad (1)$$

The observed sparsity monotonically increases with the UIH length up to approximately 16,000 items, beyond which the metric stabilizes, as few samples come with exceptionally long interaction histories. Sparsity demonstrates modest increases with both the batch size and the cluster scale, suggesting that the distribution of UIH lengths maintains relative consistent distributions. The straggler-induced resource under utilization exceeds 20% when the maximum permitted UIH length surpasses 8,000 elements, regardless of variations in batch size and cluster scale parameters.

There're conventional approaches such as dense padding or sequence truncation to overcome stragglers. However, the former forces unnecessary computational overhead and the latter sacrifices potentially valuable behavioral data from highly engaged users. While large language model training utilises methodologies such as stratifying corpus elements by length into discrete training phases and implementing sequence or context parallel across computational nodes, these approaches are not applicably in recommendation contexts. The temporal ordering of training samples in recommendation systems significantly influence model quality, thereby prohibiting the solution of length-based sample clustering.

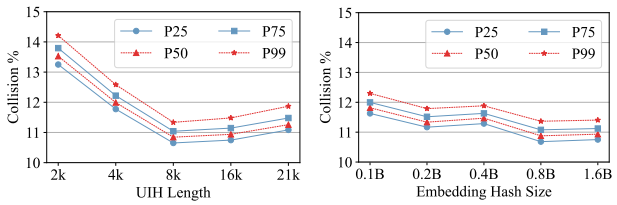

Figure 3. ID Collision Percentage

Furthermore, DLRMs are architecturally optimized for high-throughput processing of substantial traffic volumes, which usually contains relatively light computations where the additional communication overhead introduced by sequence or context parallelism would outweigh potential benefits. Evidently, DLRMs require innovations to efficiently manage workload imbalances arising from variable sequence lengths.

2.3 Embedding Communications and Row Collisions

Blocking embedding communications force subsequent computational operations to wait, leaving the computational resources under-utilized. A straightforward solution is to prefetch IDs and overlap the embedding lookup of the next iteration with the current computations. However, such design leads to the next iteration consuming stale embedding values, as embedding lookup takes place before embedding row updates. Although the ratio of collisions is relatively low, it will potentially result in unpredictable degradations. In production, even marginal regressions of 0.1% can translate to huge revenue losses, which is unacceptable.

Figure 3 visualizes the collision percentages derived from real production data, indicating that collision percentages consistently maintain relatively modest levels across all examined configurations. This suggests that an ideal solution would prioritize embedding updates, which only waits for collision rows while proactively prefetching all non-collision rows in advance.

3 Design

FreeScale minimizes the scaling cost by avoiding computational stragglers and prioritizing embedding updates and communicates via SM-Free techniques to avoid occupying SMs for communications overlapped with computation operations.

3.1 Sequence Model Load Balancing

We propose to balance model dense layer execution times across all ranks by redistributing input samples, which would in turn minimize the size of bubbles on computational resources caused by dense gradient synchronizations.

A naive solution would explore the feasibility of structuring data storage to achieve inherent balance prior to consump-

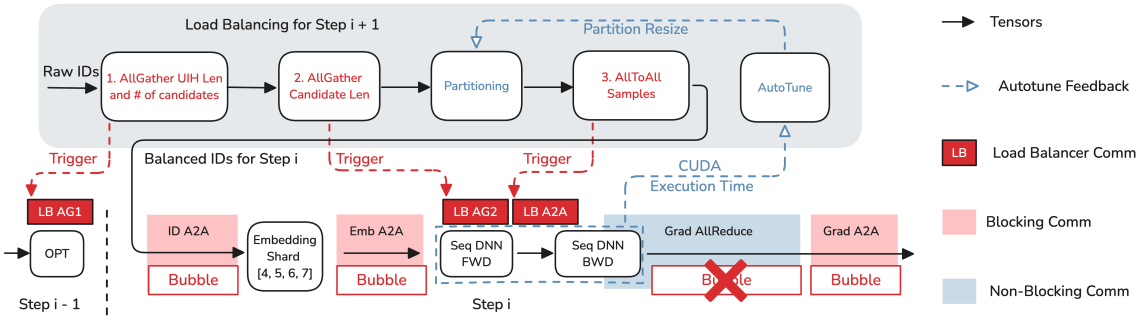

Figure 4. Load Balancing Example

tion by training processes. Unfortunately, this approach is impractical due to multiple constraints in production. First, a specific model may ingest data from heterogeneous sources, while simultaneously, data from a single source may serve multiple contexts. To minimize storage overhead, training process typically retrieve raw data from multiple sources and dynamically transform them into appropriate formats, rather than maintaining redundant data copies. Second, training processes are frequently deployed within dynamic resource allocation frameworks, wherein both the quantity and the type of GPUs vary according to incoming computational demands and the priority of the job. The dynamics in computational resources preclude static data partitioning strategies. Third, optimal load balancing decisions often require calibration based on actual model execution metrics, which are inherently unavailable until runtime. Therefore, effective load balancing mechanisms should be integrated proximally to the model training procedure.

Incorporating load balancing algorithms directly within the training process requires a judicious design that satisfies at minimum two requirements:

- **High Efficiency**: The load balancer must operate with minimal costs and avoid disrupting or slowing down any other trainer components, as even marginal overhead on GPU resources can result in significant operational cost at scale.
- **Non-Intrusive Integration**: Infrastructure components and models are typically developed by different teams, with the latter requiring accelerated development cycles and frequent iterations. For broad adoption, the load balancer must impose minimal constraints on model behavior and decouple API invocations from model implementations.

While standard DLRMs typically employ computationally intensive attention mechanisms and multi-layer perceptrons, the ID manipulation operations performed by the load balancer introduce relatively modest computational overhead, when properly implemented. The primary performance constraint stems from the inter-rank communication of IDs through collective operations. As discussed above, samples exhibit heterogeneity in multiple dimensions: variable-length UIH, variable candidate quantities, and variable content sizes per candidate. This heterogeneity forces a sequential communication pattern wherein each communication phase depends on the output of its predecessor, as outlined in Algorithm 1. The load balancing mechanism initially aggregates the UIH lengths and the number of candidates from all ranks. This information facilitates the allocation of appropriately sized tensors to accommodate subsequent `AllGather`-ed world candidate lengths. Then, the system employs a configurable, deterministic partitioning algorithm $\mathcal{P}$ that computes the balanced redistribution of samples. Finally, based on the partition result, all ranks join an `AllToAll` communication to shuffle samples to match the partitioning decision. Despite the sequential dependency

Algorithm 1 Load Balancing

Local input batch $\mathcal{B}$
Number of trainer processes WS
Runtime execution time info R
Partitioning function $\mathcal{P}$
Ensure balanced output batch $\mathcal{BB}$
$L_{uih} \leftarrow$ Tensor([WS, $\mathcal{B}$.size]) // world UIH lens
$N_{can} \leftarrow$ Tensor([WS, $\mathcal{B}$.size]) // world number candidates
Stage I: gather lengths
AllGather(L_{uih}, $\mathcal{B}$.uih_lens)
AllGather(N_{can}, $\mathcal{B}$.num_candidates)
$L_{can} \leftarrow$ Tensor([sum(N_{can})]) // world candidate lens
Stage II: gather composite candidate lengths
AllGather(L_{can}, $\mathcal{B}$.candidate_lens)
send_samples, recv_samples = $\mathcal{P}$($\mathcal{B}$, L_{uih}, N_{can}, L_{can}, R)
Stage III: shuffle samples
AllToAll(send_samples, recv_samples)
$\mathcal{BB} \leftarrow \mathcal{B}$.from_samples(recv_samples)
Return $\mathcal{BB}$

across the three communication stages, it remains feasible to hide the communication latency by overlapping with the

computation of DLRMs. This optimization is achievable because the training process typically maintains a prefetched buffer containing input batches for subsequent iterations.

Figure 4 provides a visual representation of this approach, with the three-phase communication sequences denoted by solid red rectangles. Through this design, the load balancing mechanism can effectively redistribute computational workloads across processing nodes without exposing the underlying communication overhead to the critical path. Consequently, this approach substantially mitigates computational bubbles in the computational resource that would otherwise result from imbalanced workloads. To fulfill the non-intrusive integration requirement, the load balancer implements its three-stage protocol as hooks, triggered at critical junctures within the training pipeline: during optimizer step execution, forward propagation, and backward propagation computations. These injection points are visually represented in Figure 4 by dashed red arrows. In popular machine learning frameworks such as PyTorch, these hooks can be easily installed into the execution graph via the established `nn.Module` APIs. Even in cases where such explicit hook interfaces are not directly exposed, the training pipeline orchestration typically falls under the purview of infrastructure teams, which are naturally decoupled from the modeling efforts.

The design of the load balancer deliberately stays agnostic to specific partitioning algorithms, and therefore the discussions are postponed to the implementation sections. Later, Section 4.1 introduces two paradigms: fixed batch size partitioning and variable batch size partitioning, which provide sufficient coverage for most use cases we have encountered in real-world deployment scenarios.

3.2 Prioritized Embedding Updates

As analyzed in Section 2.3, prefetching embedding values would theoretically eliminate these execution bubbles, but this approach introduces potential concerns regarding numerical stability and consequent regression of model quality. Based on this analysis, numerical parity with the synchronized baseline can be maintained by enforcing a strict write-read ordering on the collision rows, where communications on all other non-collision (*i.e.*, exclusive) rows can be performed in an asynchronous fashion. This section focuses on row-wise sharded embedding tables, omitting column-wise use cases, as the latter exhibits uniform collision patterns across all ranks, thus presenting fewer design challenges.

Firstly, it's crutial to differentiate between collision rows and exclusive rows, which need to be calculated under the appropriate distribution form. In synchronized DLRM training, IDs manifest in two distinct structures: batch-major and shard-major. The input batch is initially structured in batch-major form, wherein IDs within each batch are processed by

Algorithm 2 Prioritized Embedding Autograd Function

Persistent autograd context $\mathcal{C}$
Prefetched Embedding Indices, batch-major $\tilde{I}^{i+1}$
Current Embedding Gradients, batch-major $\tilde{G}^{i}$
Forward($\mathcal{C}$, $\tilde{I}^{i+1}$)
  with cuda.stream($\mathcal{C}$.stream)
    $G_{ex}^{i-1} \leftarrow$ AllToAll($\mathcal{C}.\tilde{G}_{ex}^{i-1}$) // to shard-major
    update_embedding($\mathcal{C}.I_{ex}^{i-1}$, G_{ex}^{i-1})
    $\mathcal{C}.I^{i+1} \leftarrow$ AllToAll($\tilde{I}^{i+1}$) // to shard-major
    // Compute collision and exclusive indices I_{co}, I_{ex}
    $\mathcal{C}.I_{co}^{i}, I_{ex}^{i}, I_{ex}^{i+1} \leftarrow$ compute_collision($\mathcal{C}.I^{i}$, $\mathcal{C}.I^{i+1}$)
    // Prepare exclusive embedding for step $i+1$
    $\mathcal{C}.\tilde{E}_{ex}^{i+1} \leftarrow$ AllToAll(Lookup(I_{ex}^{i+1})) // to batch-major
    $\mathcal{C}.\tilde{I}_{co}^{i} \leftarrow$ AllToAll($\mathcal{C}.I_{co}^{i}$)
    $\mathcal{C}.\tilde{I}_{ex}^{i} \leftarrow$ AllToAll(I_{ex}^{i+1})
  return merge($\mathcal{C}.\tilde{E}_{ex}^{i}$.wait(),$\mathcal{C}.\tilde{E}_{co}^{i}$.wait())
Backward($\mathcal{C}$, $\tilde{G}^{i}$)
  with cuda.stream($\mathcal{C}$.stream)
    $\tilde{G}_{co}^{i}, \mathcal{C}.\tilde{G}_{ex}^{i} \leftarrow$ split($\tilde{G}^{i}$, $\mathcal{C}.\tilde{I}_{co}^{i}$, $\mathcal{C}.\tilde{I}_{ex}^{i}$)
    $G_{co}^{i} \leftarrow$ AllToAll($\tilde{G}_{co}^{i}$) // to shard-major
    $E_{co}^{i+1} \leftarrow$ update_embedding($\mathcal{C}.I_{co}^{i}$, G_{co}^{i})
    $\mathcal{C}.\tilde{E}_{co}^{i+1} \leftarrow$ AllToAll(E_{co}^{i+1}) // to batch-major

the same dense model replica, while their corresponding embeddings are distributed across different embedding shards. After the ID `AllToAll` communication phase, these IDs undergo transformation into a shard-major form, where all IDs on a particular rank belong to the same embedding shard, potentially originating from different ranks. Given that embedding values are stored in a shard-major structure, collision detection must likewise be computed in this corresponding form to attain correctness. This constraint prevents pre-computing collisions, as the sharding strategy is determined at runtime based on available resources and configurations.

Since embedding tables are generally integrated within the model architecture as constituent components, the logic governing prioritized embedding updates should be encapsulated within an embedding module, which subsequently replaces the original embedding table instance in the model object through module-swapping techniques (Ivchenko et al., 2022). Under this paradigm, the algorithms can be carefully organized within the corresponding forward and backward methods of the `autograd.Function` (Paszke et al., 2017) of the embedding module. This design ensures that the autograd engine will automatically trigger these methods with the appropriate timing when embeddings and their gradients become available for processing. The detailed approach is presented in Algorithm 2.

For simplicity, Algorithm 2 deliberately excludes `Tensor`

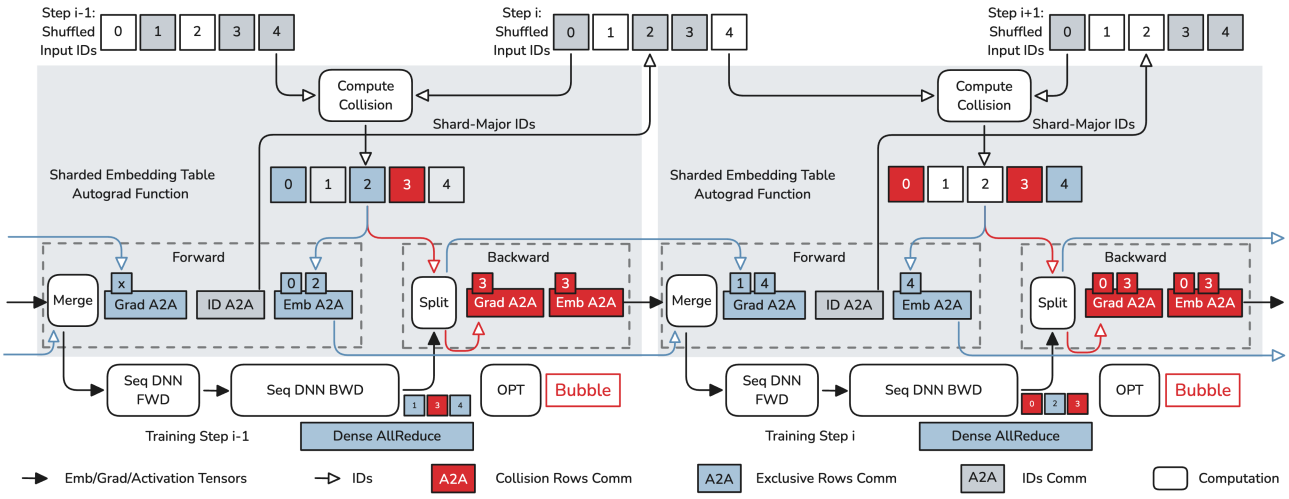

Figure 5. Prioritized Embedding Communication

size communications and focuses on index, embedding, and gradient communications, assuming that the `AllToAll` collective operation directly yields an appropriately dimensioned output `Tensor`. In our notation convention, tilde-adorned capital letters (*i.e.*, indices $\tilde{I}$, embeddings $\tilde{E}$, gradients $\tilde{G}$) denote `Tensors` in batch-major form, whereas their non-tilde counterparts represent the corresponding `Tensors` in shard-major form. The superscripts adjacent to capital letters indicate the iteration number, while the subscripts specify whether the `Tensor` contains collision or exclusive values. As an example, G^i_{co} represents embedding gradients from iteration i that collides with embeddings used in iteration $i+1$. The autograd context $\mathcal{C}$ persists across all iterations, which allows one iteration to carry values across iteration boundaries[1].

The `forward` function accepts the persistent autograd context and the prefetched indices designated for the subsequent iteration. Initially, it completes the exclusive gradient synchronization from the preceding iteration and updates the corresponding rows accordingly. After that, it transforms the prefetched indices into shard-major format and computes collisions with the current iteration. The algorithm then retrieves exclusive embeddings and transforms them into batch-major format, preparing them for the next iteration. Finally, it converts indices from the current iteration into batch-major form to distinguish collision and exclusive gradients during the backward propagation phase. All aforementioned operations are queued into a dedicated `CUDA` stream to overlap with the model computations. The `forward` function awaits the completion of prefetched exclusive embeddings and collision embeddings from the previous iteration, merging these two components into one complete input batch to serve the current iteration.

The `backward` function starts with splitting the gradients of the current iteration into collision and exclusive components. Collision gradients are expedited to finish their update process, and the resulting updated embedding values immediately transform to a batch-major format to facilitate consumption by the next iteration's `forward` function. Figure 5 illustrates this process across two consecutive iterations. Compared to the architecture depicted in Figure 1, the exposed blocking `AllToAll` communication operation is substantially reduced, now encompassing only collision gradients and embeddings. This optimization should produce significant performance enhancements according to the empirical measurements presented in Section 2.3.

3.3 SM-Free Communication

Load balancing and prioritized embedding communication either introduces additional overlappable communications or transforms exposed communications into overlapped operations, where the communications are either `AllToAll` or `AllGather` without any value reductions. While these communications can be effectively concealed through computational overlapping techniques, the communication operations nevertheless consume Streaming Multiprocessor (SM) resources, thereby creating potential resource contention with computation kernels.

To be noted that, even though NCCL uses GPU Direct RDMA that seems SM free, it's not autonomous (Hu et al., 2025). NCCL still subdivides every collective into communication channels. Each channel is launched as a separate CUDA block that runs on its own SM. Secondly, NCCL 2.28 introduced Copy Engine (CE) collectives to reduce SM utilization, but only within a single NVL domain (NVIDIA

[1]Garbage collection in the context is omitted for simplicity purposes

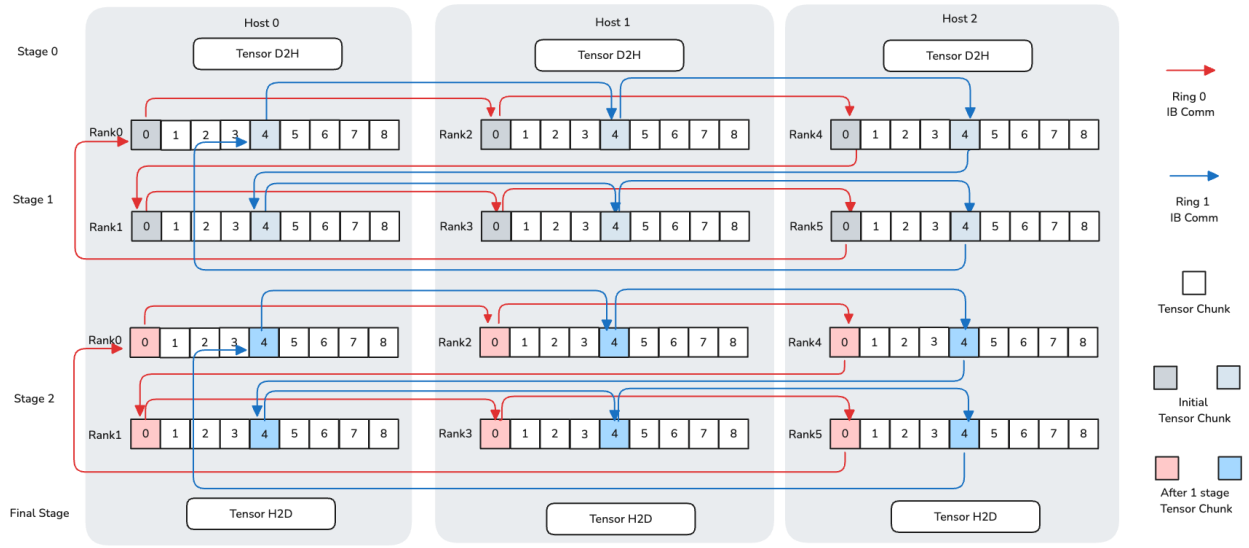

Figure 6. SM-Free Communication

Corporation, 2025). When the cluster size exceeds one NVL domain, NCCL falls back to implementation that uses SM. Additionally, it requires pre-registered memory (exposed via PyTorch's symmetric memory), which requires either careful memory planning or explicit buffer copies.

To circumvent occupying SMs, Figure 6 presents an `AllGather` communication example without SMs. In this configuration, rank 0 establishes an RDMA ring topology with ranks across hosts. Initially, each rank transfers a chunk of its local data to host memory via D2H operations. Subsequently, it transmits this host memory chunk through RDMA operations to its neighbor on the ring following the clockwise direction. In the subsequent stages, each rank propagates the chunk it received from the last stage on the RDMA ring to its clockwise neighbor. This process continues iteratively until all chunks have been successfully received by all participating ranks. Finally, the tensor on host memory will be moved back via H2D operations. In this way, non-reduction communications such as `AllGather` and `AllToAll` can be performed efficiently without consuming SM resources.

4 Implementation

This section presents the implementation details, beginning with two specific partitioning algorithms that underpin the load balancer, followed by Triton kernel optimizations for irregularly-shaped ID tensor manipulations. Finally, we explain how FreeScale integrated load balancing and prioritized embedding updates into the training pipeline while maintaining transparency for model engineers.

4.1 Partitioning Algorithms

FreeScale is compatible with arbitrary custom partitioning algorithms, as long as the algorithm maps global samples to ranks. We implemented two relatively generic approaches: Fixed Batch Size (FBS) and Variable Batch Size (VBS) partitioning methodologies. Both employ the UIH length as a computational complexity proxy to estimate execution time. The FBS algorithm performs a global sorting of samples based on UIH length, subsequently assigning samples to distinct ranks through a zig-zag strategy, i.e., rank `i` gets samples with indices `[i, 2n-i, 2n+i, 4n-i, ...]`, where `n` denotes the number of ranks in this distributed training job. This approach ensures that each rank receives an equivalent number of samples while simultaneously distributing samples with varying UIH lengths across ranks in a balanced manner. The VBS algorithm adopts a more nuanced approach, calculating sample weights using the formula L^α, where L represents the UIH length and α as a hyperparameter. The algorithm subsequently partitions globally sorted samples into `n` segments, with each segment summing up to approximately equivalent total weight. Under the VBS approach, individual ranks may receive disparate sample quantities, while all samples within a given batch maintain similar lengths. Since the initial value of weight might not accurately represent the model computation execution time, VBS also employs an autotune mechanism that increments or decrements local batch size depending on the difference between the local execution time and the global average execution time, which is illustrated in Figure 4 as the `AutoTune` module. Each of these two partitioning strategy presents distinct trade-offs:

- **FBS** offers versatility across models due to its inde-

pendence from precise computational execution time estimation. However, when `CUDA` kernels use an individual block to cover one entire sample, FBS may inadvertently generate intra-kernel load imbalances. Blocks assigned to long UIH samples can emerge as intra-kernel stragglers that compel certain SMs to remain in an idle state while awaiting task completion.

- **VBS** constructs batches with samples of similar UIH lengths, significantly reducing intra-batch sparsity. This optimization enables kernel blocks to receive more uniformly distributed workloads, thereby enhancing overall computational efficiency. However, the implementation of dynamic batch size introduces additional complexities that preclude universal applicability. The modification of local batch size engenders subtle interactions with gradient synchronization mechanisms. Practitioners might need to calibrate loss function formulations and gradient collective communications (*i.e.*, `AllReduce` and `AllToAll`) to ensure appropriate gradient scaling.

These two built-in partitioning algorithms are sufficient to address most use cases encountered in our deployment experiences. For specialized scenarios requiring special workload distribution strategies, practitioners can plug in custom partitioning logic by supplying user-defined Python functions following the parameter specifications as shown in Algorithm 1. This extensibility allows FreeScale to adapt to domain-specific workload characteristics without requiring modifications to the underlying distribution infrastructure.

4.2 Triton Kernel Optimizations

FBS and VBS partitioning strategies require advanced indexing and concatenation operations on irregularly shaped tensor structures (Organization, 2025). Implementing these operations using standard PyTorch primitives leads to iterative processing of individual samples followed by subsequent concatenations, which is inefficient.

To address these inefficiencies, we implemented a comprehensive suite of specialized Triton (Tillet et al., 2019) kernels optimized for jagged tensor operations. These operations include the indexed permute kernel that apply partitioning transformations to globally `AllGather`ed IDs, the ranged dispatch kernel and the ranged combine kernel that prepare input and output tensors for ID and embedding `AllToAll` operations, and the keyed transpose kernel that converts jagged tensor representations from the feature-major form to the batch-major form, among others. Due to space constraints, we only present the performance metrics for the ranged dispatch and ranged combine kernels. Figure 7 illustrates a performance analysis between vanilla PyTorch implementations and our specialized Triton kernels.

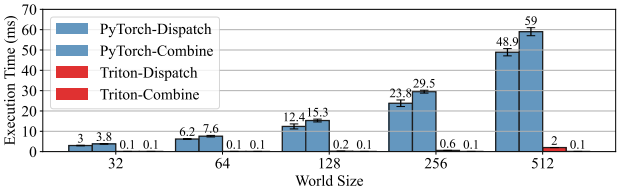

Figure 7. Kernel Efficiency

The execution latency of the PyTorch implementation increases linearly with the world size, becoming computationally prohibitive when the world size exceeds several hundred nodes. In contrast, the Triton implementation demonstrates superior performance, outperforming PyTorch eager implementations by a factor of 20 even at modest world sizes of 32, with efficiency differentials expanding to more than 600-fold at world size 512.

4.3 Staged Training Pipeline

The training framework usually faces an inherent dilemma. In one dimension, it must offer high flexibility to accommodate diverse model designs, which prevents the trainer from making specific assumptions of model architectures. Conversely, to achieve optimal computational efficiency, the training framework would benefit substantially from accessing detailed model architectural information. The theoretical optimum would involve obtaining an operator graph by tracing the full train step, allowing the training framework to systematically manipulate this graph to implement performance-enhancing transformations. However, imposing full graph traceability introduces significant constraints on model development practices. Such constraints would prohibit computational components including dynamic branching logic and third-party tensor operators, thereby impeding development flexibility.

For recommendation systems, the capability of fast model iteration represents a critical component for addressing evolving business requirements. Thus, FreeScale deliberately eschews full graph tracing, instead decomposing the training pipeline into five stages: data loading, forward propagation, backward propagation, optimizer step, and metrics computations. Furthermore, modern deep learning frameworks such as PyTorch provide module traversal APIs and a rich set of hook mechanisms, furnishing additional instrumentation capabilities for monitoring module performance and modifying module behaviors programmatically. This pragmatic approach enables FreeScale to achieve a balanced compromise between optimization opportunities and development flexibility, facilitating both performance enhancement and rapid model iterations essential for industrial recommendation systems.

FreeScale identifies embedding tables via the `named_modules()` interface, subsequently aggre-

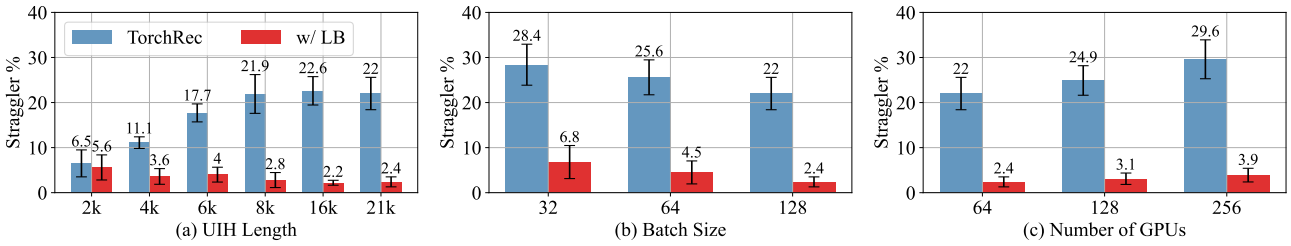

Figure 8. Straggler Reduction

gating `Embedding` objects with identical `dtype` and sharding strategies into unified sharded embedding table constructs. The three-stage communication protocol implemented by the load balancer is inserted after sharded embedding communications, forward propagation, and backward propagation accordingly through module hooks. The blocking communication of the sharded embedding table is only waited in the forward function of the next iteration, which facilitates natural overlap with optimizer step, metrics computation, and data preprocessing computations. Consequently, the exposed communication intervals are further reduced beyond the minimal requirement of collision gradient and embedding `AllToAll` collective operations.

5 Evaluation

We evaluated FreeScale using real production recommendation models and kernels running on up to 256 H100 GPUs, each with 94GB of High Bandwidth Memory (HBM). GPUs within one server are connected via $600GB/s$ NVSwitch ($4800Gb/s$) while servers are interconnected with $8 \times 200Gb/s$ InfiniBand (IB). Communications between HBM and CPU are facilitated by PCIe 5.0.

5.1 Straggler Overhead

Figure 8(a) demonstrates that the straggler intensifies in TorchRec with increasing permitted UIH length. In contrast, FreeScale attained superior performance with longer UIH sequence, achieving more than nine-fold straggler reduction at 21,000 UIH length. This behavior can be attributed to the distribution of engagement history in the dataset, where the majority of users have engaged with at least 2,000 items, thus truncation to this threshold effectively eliminates workload imbalances. Additionally, with reduced UIH lengths, the proportion of stragglers contributed by attention kernels could be overshadowed by the jitter from other system components (e.g., data loading subsystems). As a result, length-based load balancing becomes less effective in short-UIH scenarios.

Figure 8(b) illustrates that straggler percentage drops for both TorchRec and FreeScale as batch size increases, despite the corresponding increase in sparsity demonstrated in Figure 2. This counterintuitive result stems from the fact that the aggregate token count within a local batch constitutes the primary determinant of computation kernel execution time. Larger batch sizes inherently reduce cross-batch variance through statistical averaging effects, consistent with the law of large numbers. From an intuitive perspective, when the batch dimensions reach sufficient magnitude, all batches converge toward the intrinsic UIH length distribution of the source dataset. Across all evaluated configurations, FreeScale consistently outperformed TorchRec by factors ranging from four to nine. Figure 8(c) elucidates how scaling the computational cluster exacerbates the straggler effects. Straggler percentage increases for both TorchRec and FreeScale, as using more GPUs leads to higher probability of encountering outlier batches. All experiments presented employed FBS partitioning, because the evaluated model incorporates native variable-length UIH optimizations within custom Triton kernels, where the complexity introduced by VBS will not bring additional gains.

5.2 Exposed Embedding Communications

Experiments in this section employ synthetic datasets for two reasons. First, to isolate the impact of prioritized communication, we exclude all other optimization techniques introduced in this paper, while real data suffers from stragglers. Second, understanding the performance across varying embedding row collision ratios is essential for comprehensive evaluation.

As visualized in Figure 9, exposed communication is decomposed into three constituent components: IDs `AllToAll`, Embedding `AllToAll`, and Gradient `AllToAll`. Since FreeScale achieves complete overlap on IDs `AllToAll` through prefetching, this component is omitted from the figure. When analyzing the impact of increasing UIH length, both TorchRec and FreeScale experience proportionally increased exposed communication duration, attributable to concurrent increases in both communication volume and collision ratios. Nevertheless, FreeScale consistently demonstrates approximately a nine-fold reduction in exposed communication latency across these configurations. When examining performance across varying collision rates while

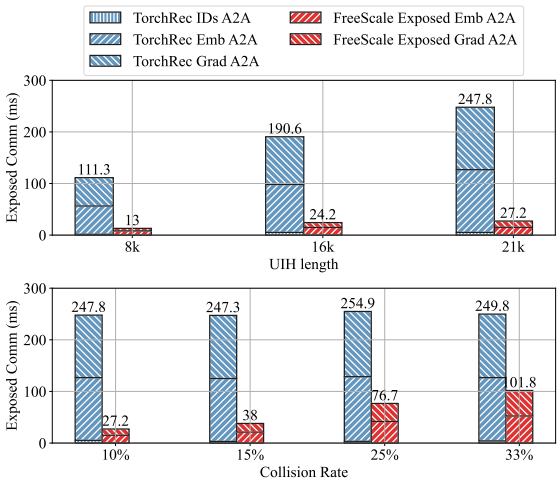

Figure 9. Isolated Exposed Communication

maintaining constant communication volume, the TorchRec exhibits relatively stable exposed communication latency. In contrast, FreeScale demonstrates a linear relationship between exposed communication duration and collision rate, confirming that the observed latency is dominated by communicating collision rows. This finding aligns with the theoretical design underlying the prioritization algorithm.

5.3 Communication and Computation Contention

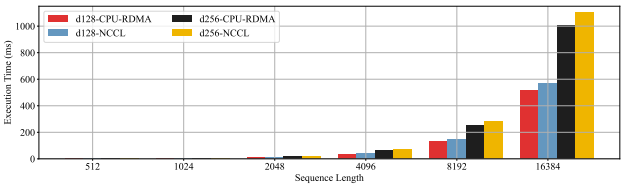

Figure 10. Execution Time Comparison

Figure 10 shows the kernel (which overlapped with communication) execution time benchmark on synthetic data. The execution time increases exponentially as input sequence lengths increases, while growing linearly when hidden dim increases. SM-Free communication, outperforms NCCL communication by 10% across all scenarios, shows its effectiveness in mitigating communication and cumputation contention. To be noted that, this consistent 10% speedup is also observed when varying the NCCL_MAX_NCHANNEL or NCCL_MIN_NCHANNEL. We hypothesis it because NCCL's internal mechanism that automatically tune the resource utilization, while these varied settings just serve as tuning hints during runtime which saturate the hardware network bandwidth.

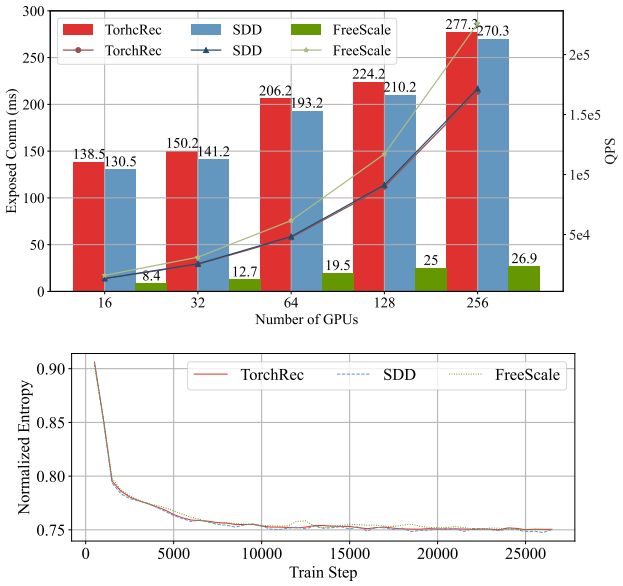

Figure 11. Overall Experiments

5.4 Overall Speedups

Figure 11 top summarizes the exposed communication latencies and corresponding throughput improvements on production data, when scaling up with larger clusters. The left vertical axis corresponds to the red, blue and green bars and the right vertical axis correlates with the lines.

TorchRec, the baseline, has its exposed communication latency monotonically increases with the cluster size, as straggler effects intensify in larger clusters and collective communication operations suffer from reduced efficiency with larger communication topologies.

SDD, a distributed training optimization that tries to overlap distributed metadata communication with model forward operations, helps reduce some exposed communications by 10ms. The communication reduction by SDD is roughly constant because the metadata communication only communicates how many elements should be transferred between GPUs and the time to transfer the number itself doesn't grow with the clusters.

In contrast, the FreeScale maintains consistently minimal exposed communication latency across cluster scales, achieving 90% reduction in exposed communication overhead relative to the baseline. FreeScale's exposed communication latency also become more expensive with larger clusters, despite experiments maintaining constant size for both local embedding shard and UIH length. This can be attributed to the multifaceted nature of communication overhead in distributed systems. The total communication latency comprises three distinct components: straggler synchronization delays, network topology traversal time, and data transmission duration. Although FreeScale effectively controls both

the straggler-induced waiting periods and data transmission volumes through advanced load balancing and prioritization techniques, the fundamental overhead associated with traversing increasingly complex interconnect topologies in larger clusters remains an unavoidable scaling constraint.

The three curves represents the QPS (number of items processed per second) during training. TorchRec and SDD yields a roughly same QPS, because SDD only addresses metadata communication, leaving the straggler issues and embedding communication untouched. While FreeScale shows the increasing impact on larger DLRM training jobs.

Note that the experiments are conducted on clusters with high communication bandwidth ($8 \times 200Gb/s$). For slower connections where communication latency dominates, speedups should be more significant.

Besides effciency improvements, an infrastructure optimization should also guarantee numeric parity with baseline. Figure 11 bottom compares the offline training normalized entropy (NE) curve of TorchRec, SDD and FreeScale on a critical task. The results confirms that the model converges to same NE level, confirms that FreeScale maintain numerical fidelity. Noted that, in online training settings, FreeScale can yield more topline metric wins compared to baseline. It is because in online settings, model has to complete training within a predefined time window and randomly drop out unfinished training examples to avoid stuck issues. FreeScale enables model to learn from more training examples and thus get better performance.

6 Discussion

In this section, we will discuss the memory overhead, runtime complexity, engineering effort and the failure modes. We will also illustrate more about the SM free communication and compare to NCCL in details.

6.1 Memory & Time Analysis

As FreeScale needs to prefetch future batch to do training batch balance and pre-fetch non-collision embeddings, it introduces 1 more batches of input data. However, since the input data only contains IDs and dense features, it usually occupies several GBs memory and much less than the memory consumption during model forward and backward computation, which usually contains 10+ transformer-like layers and each layer would create multiple tensors of similar size. So prefetching embeddings only contributes a small fraction of the overall HBM footprint. Additionally, FreeScale also balances the memory consumption across GPUs and reduce peak memory utilization which can also make up the memory overhead introduced by itself.

In terms of time complexity, FreeScale carefully implemented a multi-GPU stream pipeline that overlaps auxiliary computation kernels with main streams, minimizing overhead through better resource utilization and therefore yield great QPS wins.

6.2 Engineering Effort & Failure Mode

FreeScale comprises around 8,600 lines of core code which is 2.5% of the TorchRec codebase. During the implementation, preserving autograd context across iterations does introduce additional engineering complexity. We wrapped that in a relatively clean way into the autograd function and our train pipeline, which makes the complexity manageable at the trainer level.

FreeScale's failure patterns (collective timeouts, OOM, stream synchronization) are identical to existing RecSys infrastructure, which already runs multi-stream GPU workloads with communication/computation overlap at scale. We leverage the same observability stack (Perfetto traces, memory profilers) and debugging infrastructure. If the workload is already running at peak HBM capacity, the additional fraction of HBM footprint added by FreeScale could lead to OOM. However, in practice, this is usually a trade-off as HBM capacity can be restored by paying recomputation overhead through activation checkpointing. Another possible failure mode is where the model is using tiny embedding tables or very short sequences, then FreeScale won't bring much speedup.

6.3 More of SM Free Communcation

In theory, when we look at communication alone without any overlapping with computations or other operations, NCCL-based communication should always be faster than the SM free communication as it uses more GPU resources and fewer copies. However, the story is different when we take a holistic view where computations are also in the picture. More specifically, in FreeScale, load balancing and prioritized embedding communication either introduces additional overlapped communication or transforms exposed communications into overlapped operations. We would like to clarify that the CPU-RDMA is proposed to do these overlapped communications without consuming SM resources. As NCCL- based communication needs SMs, this causes SM contention when communication overlaps with computation. With CPU-RDMA communication, overlapped computation runs unaffected and finishes faster.

7 Related Work

7.1 Load Balancing in Distributed Training

LB-BSP (Moreno-Alvarez et al., 2020) proposes a semi-dynamic scheme to re-balances local batch size to mitigate

stragglers. Srifty (Luo et al., 2022) uses learned compute and communication performance models to statically load balance on heterogeneous, cloud fleets. Gandiva (Xiao et al., 2018) supports oversubscription, migration and packing to defragment cluster usage, improving fleet-wide utilization. NeuroShard (Eldeeb et al., 2022) better load balance placements of embedding tables across accelerators to minimize lookup performance.

7.2 Optimizing DLRM Efficiency

It's challenging to optimize DLRM, due to the large embedding tables (Lian et al., 2021), imbalanced workload and poorer efficiency on newer hardware (Luo et al., 2018; 2020). Wukong and DHEN (Zhang et al., 2024; 2022) use compressed dot products and GEMM-heavy linear compression operators for interaction mechanism. AutoInt (Song et al., 2019) adopts the heavily optimized transformer architecture. Mamba4Rec (Liu et al., 2024) uses state-space to learn sequence in linear time. DMT (Luo et al., 2024), decomposed AlltoAll communication into a multi-rail, multi-step version and leverages faster intra-host interconnect. PLink (Luo et al., 2020) constructs hierarchical reduction trees for efficient parameter synchronization. Hybrid sharding schemes that leverage intra-host locality are also proposed in FSDP and DHEN (Zhang et al., 2022; Zhao et al., 2023). Blink (Wang et al., 2018) constructs multiple reduction rings to optimally utilize resource. Zero++ (Wang et al., 2023) adopts quantized weight, gradient communication (Yang et al., 2020) and hierarchical weight partition. Specialized accelerators built for recommendation models are no longer uncommon (Firoozshahian et al., 2023; Maddury et al., 2024). Optimal collectives synthesis (Shah et al., 2025; Cowan et al., 2023; Cai et al., 2021; Shah et al., 2023) creates provably optimal collective schedules from communication primitives given a target setup.

8 Conclusion

This paper has addressed a critical efficiency challenge in distributed training of DLRMs. Our investigation identified that stragglers and blocking communications introduce substantial computational resource under-utilization which appears as computation bubbles. FreeScale represents a holistic solution that systematically eliminates these inefficiencies through load balancing and prioritized embedding communication and resolves the GPU resource competition during the overlapping communication and computation operations via SM-Free communication techniques. Empirical evaluation using production workloads on a cluster of 256 GPU demonstrates that FreeScale achieves a remarkable 90.3% reduction in computational bubbles. These results demonstrated FreeScale as an effective optimization framework for industrial-scale recommendation systems, enabling significantly improved hardware utilization without compromising model quality or numerical stability.

References

Cai, Z., Liu, Z., Maleki, S., Musuvathi, M., Mytkowicz, T., Nelson, J., and Saarikivi, O. Synthesizing optimal collective algorithms. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '21, pp. 62–75, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450382946.

Cowan, M., Maleki, S., Musuvathi, M., Saarikivi, O., and Xiong, Y. Mscclang: Microsoft collective communication language. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, pp. 502–514, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450399166.

Eldeeb, T., Chen, Z., Cidon, A., and Yang, J. Neuroshard: towards automatic multi-objective sharding with deep reinforcement learning. In *Proceedings of the Fifth International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, pp. 1–12, 2022.

Employees, M. Meta lattice, May 2023.

Firoozshahian, A., Coburn, J., Levenstein, R., Nattoji, R., Kamath, A., Wu, O., Grewal, G., Aepala, H., Jakka, B., Dreyer, B., et al. Mtia: First generation silicon targeting meta's recommendation systems. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pp. 1–13, 2023.

Hu, Z., Shen, S., Bonato, T., Jeaugey, S., Alexander, C., Spada, E., Dinan, J., Hammond, J., and Hoefler, T. Demystifying nccl: An in-depth analysis of gpu communication protocols and algorithms. In *2025 IEEE Symposium on High-Performance Interconnects (HOTI)*, pp. 48–59. IEEE, 2025.

Ivchenko, D., Van Der Staay, D., Taylor, C., Liu, X., Feng, W., Kindi, R., Sudarshan, A., and Sefati, S. Torchrec: a pytorch domain library for recommendation systems. In *Proceedings of the 16th ACM Conference on Recommender Systems*, RecSys '22, pp. 482–483, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392785.

Li, S., Zhao, Y., Varma, R., Salpekar, O., Noordhuis, P., Li, T., Paszke, A., Smith, J., Vaughan, B., Damania, P., et al. Pytorch distributed: Experiences on accelerating data parallel training. *Very Large Data Bases (VLDB) Conferences*, 2020.

Lian, X., Yuan, B., Zhu, X., Wang, Y., He, Y., Wu, H., Sun, L., Lyu, H., Liu, C., Dong, X., Liao, Y., Luo, M., Zhang, C., Xie, J., Li, H., Chen, L., Huang, R., Lin, J., Shu, C., Qiu, X., Liu, Z., Kong, D., Yuan, L., Yu, H., Yang, S., Zhang, C., and Liu, J. Persia: An open, hybrid system scaling deep learning-based recommenders up to 100 trillion parameters, 2021.

Liu, C., Lin, J., Wang, J., Liu, H., and Caverlee, J. Mamba4rec: Towards efficient sequential recommendation with selective state space models, 2024.

Liu, L., Cai, L., Zhang, C., Zhao, X., Gao, J., Wang, W., Lv, Y., Fan, W., Wang, Y., He, M., et al. Linrec: Linear attention mechanism for long-term sequential recommender systems. In *Proceedings of the 46th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 289–299, 2023.

Luo, L., Nelson, J., Ceze, L., Phanishayee, A., and Krishnamurthy, A. Parameter hub: a rack-scale parameter server for distributed deep neural network training. In *Proceedings of the ACM Symposium on Cloud Computing*, pp. 41–54, 2018.

Luo, L., West, P., Nelson, J., Krishnamurthy, A., and Ceze, L. Plink: Discovering and exploiting locality for accelerated distributed training on the public cloud. *Proceedings of Machine Learning and Systems*, 2:82–97, 2020.

Luo, L., West, P., Patel, P., Krishnamurthy, A., and Ceze, L. Srifty: Swift and thrifty distributed neural network training on the cloud. In Marculescu, D., Chi, Y., and Wu, C. (eds.), *Proceedings of Machine Learning and Systems*, volume 4, pp. 833–847, 2022.

Luo, L., Zhang, B., Tsang, M., Ma, Y., Chu, C.-H., Chen, Y., Li, S., Hao, Y., Zhao, Y., Lakshminarayanan, G., Wen, E. D., Park, J., Mudigere, D., and Naumov, M. Disaggregated multi-tower: Topology-aware modeling technique for efficient large-scale recommendation, 2024.

Maddury, M., Kansal, P., and Wu, O. Next gen mtia-recommendation inference accelerator. In *2024 IEEE Hot Chips 36 Symposium (HCS)*, pp. 1–27. IEEE, 2024.

Moreno-Alvarez, S., Haut, J. M., Paoletti, M. E., Rico-Gallego, J. A., Diaz-Martin, J. C., and Plaza, J. Training deep neural networks: a static load balancing approach. *The Journal of Supercomputing*, 76:9739–9754, 2020.

Naumov, M., Mudigere, D., Shi, H.-J. M., Huang, J., Sundaraman, N., Park, J., Wang, X., Gupta, U., Wu, C.-J., Azzolini, A. G., Dzhulgakov, D., Mallevich, A., Cherniavskii, I., Lu, Y., Krishnamoorthi, R., Yu, A., Kondratenko, V., Pereira, S., Chen, X., Chen, W., Rao, V., Jia, B., Xiong, L., and Smelyanskiy, M. Deep learning recommendation model for personalization and recommendation systems, 2019.

NVIDIA Corporation. NVIDIA Collective Communication Library (NCCL) Release 2.28.3, 2025. URL https://docs.nvidia.com. Accessed: [Insert Date Here].

Organization, P. Jagged tensor operators, 2025.

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. Automatic differentiation in pytorch. 2017.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pp. 8024–8035. Curran Associates, Inc., 2019.

Shah, A., Chidambaram, V., Cowan, M., Maleki, S., Musuvathi, M., Mytkowicz, T., Nelson, J., Saarikivi, O., and Singh, R. TACCL: Guiding collective algorithm synthesis using communication sketches. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pp. 593–612, Boston, MA, April 2023. USENIX Association. ISBN 978-1-939133-33-5.

Shah, A., Jangda, A., Li, B., Rocha, C., Hwang, C., Jose, J., Musuvathi, M., Saarikivi, O., Cheng, P., Zhou, Q., Dathathri, R., Maleki, S., and Yang, Z. Msccl++: Rethinking gpu communication abstractions for cutting-edge ai applications, 2025.

Song, W., Shi, C., Xiao, Z., Duan, Z., Xu, Y., Zhang, M., and Tang, J. Autoint: Automatic feature interaction learning via self-attentive neural networks. In *Proceedings of the 28th ACM international conference on information and knowledge management*, pp. 1161–1170, 2019.

Steck, H., Baltrunas, L., Elahi, E., Liang, D., Raimond, Y., and Basilico, J. Deep learning for recommender systems: A netflix case study. *AI magazine*, 42(3):7–18, 2021.

Tillet, P., Kung, H.-T., and Cox, D. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pp. 10–19, 2019.

Wang, G., Phanishayee, A., Venkataraman, S., and Stoicat, I. Blink: A fast nvlink-based collective communication library. In *Proc. Conf. Syst. Mach. Learn*, 2018.

Wang, G., Qin, H., Jacobs, S. A., Holmes, C., Rajbhandari, S., Ruwase, O., Yan, F., Yang, L., and He, Y. Zero++:

Extremely efficient collective communication for giant model training. *arXiv preprint arXiv:2306.10209*, 2023.

Wang, S., Cao, L., Wang, Y., Sheng, Q. Z., Orgun, M. A., and Lian, D. A survey on session-based recommender systems. *ACM Computing Surveys (CSUR)*, 54(7):1–38, 2021.

Wang, W., Lin, X., Feng, F., He, X., and Chua, T.-S. Generative recommendation: Towards next-generation recommender paradigm, 2024.

Xiao, W., Bhardwaj, R., Ramjee, R., Sivathanu, M., Kwatra, N., Han, Z., Patel, P., Peng, X., Zhao, H., Zhang, Q., Yang, F., and Zhou, L. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pp. 595–610, Carlsbad, CA, October 2018. USENIX Association. ISBN 978-1-939133-08-3.

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

Zeng, C., Liao, X., Cheng, X., Tian, H., Wan, X., Wang, H., and Chen, K. Accelerating neural recommendation training with embedding scheduling. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pp. 1141–1156, 2024.

Zeng, Z., Liu, X., Hang, M., Liu, X., Zhou, Q., Yang, C., Liu, Y., Ruan, Y., Chen, L., Chen, Y., Hao, Y., Xu, J., Nie, J., Liu, X., Zhang, B., Wen, W., Yuan, S., Wang, K., Chen, W.-Y., Han, Y., Li, H., Yang, C., Long, B., Yu, P. S., Tong, H., and Yang, J. Interformer: Towards effective heterogeneous interaction learning for click-through rate prediction, 2025.

Zhai, J., Liao, L., Liu, X., Wang, Y., Li, R., Cao, X., Gao, L., Gong, Z., Gu, F., He, M., Lu, Y., and Shi, Y. Actions speak louder than words: Trillion-parameter sequential transducers for generative recommendations, 2024a. [Accessed 25-02-2025].

Zhai, J., Liao, L., Liu, X., Wang, Y., Li, R., Cao, X., Gao, L., Gong, Z., Gu, F., He, M., Lu, Y., and Shi, Y. Actions speak louder than words: Trillion-parameter sequential transducers for generative recommendations, 2024b.

Zhang, B., Luo, L., Liu, X., Li, J., Chen, Z., Zhang, W., Wei, X., Hao, Y., Tsang, M., Wang, W., Liu, Y., Li, H., Badr, Y., Park, J., Yang, J., Mudigere, D., and Wen, E. Dhen: A deep and hierarchical ensemble network for large-scale click-through rate prediction, 2022.

Zhang, B., Luo, L., Chen, Y., Nie, J., Liu, X., Guo, D., Zhao, Y., Li, S., Hao, Y., Yao, Y., et al. Wukong: Towards a scaling law for large-scale recommendation. *arXiv preprint arXiv:2403.02545*, 2024.

Zhao, X., Gu, C., Zhang, H., Yang, X., Liu, X., Tang, J., and Liu, H. Dear: Deep reinforcement learning for online advertising impression in recommender systems. In *Proceedings of the AAAI conference on artificial intelligence*, volume 35, pp. 750–758, 2021.

Zhao, Y., Gu, A., Varma, R., Luo, L., Huang, C.-C., Xu, M., Wright, L., Shojanazeri, H., Ott, M., Shleifer, S., Desmaison, A., Balioglu, C., Damania, P., Nguyen, B., Chauhan, G., Hao, Y., Mathews, A., and Li, S. Pytorch fsdp: Experiences on scaling fully sharded data parallel, 2023.