RecShard: Statistical Feature-Based Memory Optimization for Industry-Scale Neural Recommendation

Geet Sethi
Stanford University and Meta
Stanford, California, USA
geet@cs.stanford.edu

Bilge Acun
Meta
Menlo Park, California, USA
acun@fb.com

Niket Agarwal
Meta
Menlo Park, California, USA
niketa@fb.com

Christos Kozyrakis
Stanford University
Stanford, California, USA
kozyraki@stanford.edu

Caroline Trippel
Stanford University
Stanford, California, USA
trippel@stanford.edu

Carole-Jean Wu
Meta
Cambridge, Massachusetts, USA
carolejeanwu@fb.com

ABSTRACT

We propose RecShard, a fine-grained embedding table (EMB) partitioning and placement technique for deep learning recommendation models (DLRMs). RecShard is designed based on two key observations. First, not all EMBs are equal, nor all rows within an EMB are equal in terms of access patterns. EMBs exhibit distinct memory characteristics, providing performance optimization opportunities for intelligent EMB partitioning and placement across a tiered memory hierarchy. Second, in modern DLRMs, EMBs function as hash tables. As a result, EMBs display interesting phenomena, such as the birthday paradox, leaving EMBs severely under-utilized. RecShard determines an optimal EMB sharding strategy for a set of EMBs based on training data distributions and model characteristics, along with the bandwidth characteristics of the underlying tiered memory hierarchy. In doing so, RecShard achieves over 6 times higher EMB training throughput on average for capacity constrained DLRMs. The throughput increase comes from improved EMB load balance by over 12 times and from the reduced access to the slower memory by over 87 times.

CCS CONCEPTS

• **Information systems** → **Recommender systems**; • **Computer systems organization** → **Neural networks**.

KEYWORDS

Deep learning recommendation models, AI training systems, Memory optimization, Neural networks

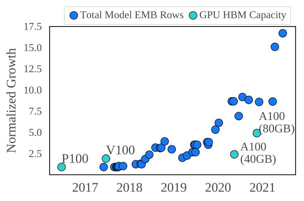
ACM Reference Format:
Geet Sethi, Bilge Acun, Niket Agarwal, Christos Kozyrakis, Caroline Trippel, and Carole-Jean Wu. 2022. RecShard: Statistical Feature-Based Memory Optimization for Industry-Scale Neural Recommendation. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22), February 28 – March 4, 2022, Lausanne, Switzerland.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3503222.3507777

1 INTRODUCTION

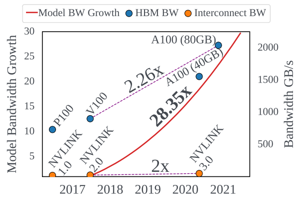
Deep learning (DL) is pervasive, supporting a wide variety of application domains [5, 6, 14, 15, 21, 32, 37, 40]. A significant fraction of deep learning compute cycles in industry-scale data centers can be attributed to deep learning recommendation models (DLRMs) [3, 4, 9, 11, 20, 34, 43, 47, 48]. For example, at Facebook, DLRMs account for more than 50% of training demand [32] and more than 80% of inference demand [11]. Moreover, Google's search engine relies on its recommender system, such as RankBrain, for search query processing [36].

DLRMs DLRMs exhibit distinct systems implications compared to more traditional neural network architectures [10, 16, 23, 24, 38]. This is due to their use of *embedding layers* which demand orders-of-magnitude higher memory capacity and exhibit significantly lower compute-intensity [11, 26, 33]. Embedding layers, comprised of *embedding tables* (EMBs), support the transformation of categorical (i.e., sparse) features into dense representations. Categorical features are typically represented as one-hot or multi-hot binary vectors, where entries represent feature categories. Activated categories (binary value of 1) in a feature vector then induce a set of look-ups to the feature's corresponding EMB to extract dense latent vectors.

System Requirement Characteristics for DLRMs The large feature space for industry-scale DLRMs demands significant compute throughput (PF/s), memory capacity (10s of TBs), and memory bandwidth (100s of TB/s) [31]. Figure 1 illustrates that the *memory capacity and bandwidth demands for DLRMs have been growing super-linearly, exceeding the memory capacities available on training hardware.* Figure 1a shows that between 2017-2021, the memory capacity requirements of DLRMs have grown by 16 times. EMB memory footprints are on the order of terabytes (TB) [26, 46] and account for over 99% of the total model capacity [11]. The growth in the number and sizes of EMBs stems from the increase in the number of features and feature categories represented, in order to improve the overall DLRM prediction quality. Figure 1b shows that, within the same four-year period, per-sample DLRM memory bandwidth demand, determined by the amount of EMB rows accessed in a single training data sample, has increased by almost 30 times,



(a) DLRM memory requirements have grown by 16x, while memory capacity on GPU accelerators has improved by less than 6x.



(b) DLRM bandwidth demands have grown by 30x, far outpacing the bandwidth growth of accelerator memories and interconnects.

Figure 1: DLRM system requirement growth trend.

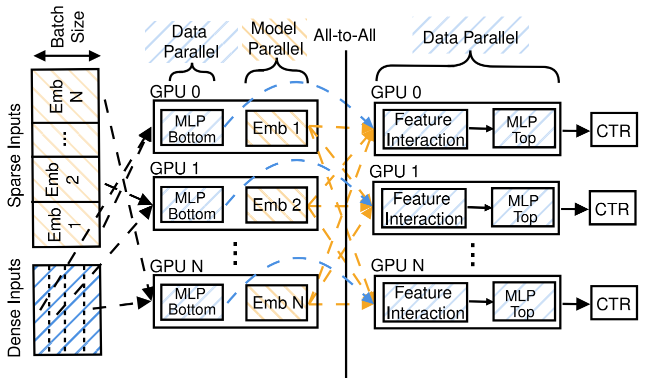


Figure 2: Generalized hybrid-parallel DLRM architecture. Data parallel modules (MLPs) are shaded blue while model parallel EMBs are shaded orange.

outpacing the growth and availability of memory bandwidth on state-of-the-art training hardware.

Hierarchical Memory in Training Systems The widening gap between the DLRM memory needs and the memory specifications of modern training system hardware motivates new memory optimization techniques to effectively scale training throughput. While the exact training system architectures differ, hierarchical memory systems, e.g. tiered hierarchies composed of GPU HBM, CPU DRAM, and SSD [46], are becoming increasingly common for DLRM training. Since not all EMBs can fit entirely in GPU HBMs, this scenario gives rise to optimization strategies to address the first challenge – deciding *where EMBs should be placed in the hierarchical memory system* to maximize training throughput. Strategically placing EMBs brings up the second challenge – ensuring *efficient utilization of all available memory capacity and bandwidth.*

Characterizing EMB Access Patterns for DLRMs In this paper, we make two key observations regarding the memory access behaviors of EMBs that motivate more performant and efficient EMB partitioning and placement schemes.

First, not all EMBs are equal, nor are all rows within an EMB equal in terms of access behaviors. For example, the *frequency distribution* of a sparse feature's categorical values often follows a power law distribution. Therefore, a relatively small fraction of EMB rows will source the majority of all EMB accesses. Furthermore, as illustrated in Figure 3, sparse features, and thus EMBs, exhibit varying bandwidth demands due to varying *pooling factors* – the number of activated categories on average in a particular sparse feature sample – and *coverage* – the fraction of training samples in which a particular feature appears. Second, in modern DLRMs, EMBs function as *hash tables.* As a result, EMBs display interesting phenomena, such as the birthday paradox, which leaves a significant portion of EMBs unused due to hash collisions. Unused EMB space is further increased with increasing *hash sizes.*

Building on the in-depth sparse feature characterization of production scale DLRMs (Section 3), we propose *RecShard* – a new approach to improve DLRM training throughput using a *data-driven and system-aware EMB partitioning and placement strategy.* RecShard's EMB *sharding* strategy is informed by per-feature training data distributions—categorical value frequency distributions (Figure 5), pooling factor statistics (Figure 6a) as well as coverage distributions of all sparse features (Figure 6b). RecShard also considers EMB design settings—hash functions and table sizes (Figure 7) as well as characteristics of the underlying tiered memory. RecShard considers the training system design parameters simultaneously through the use of a mixed integer linear program (MILP) to produce an optimal EMB sharding strategy. Overall, the key contributions of this paper are as follows:

- **Fine-grained, data-driven EMB sharding:** We demonstrate that EMB access patterns during DLRM training vary within and across EMBs. As a result, DLRM training throughput stands to improve with fine-grained EMB sharding. Further, EMB access patterns can be estimated by deriving statistics from less than 1% of training data (categorical value frequency distribution, pooling factor, and coverage) and the target DLRM architecture (hash function and hash size). Thus, intelligent EMB sharding schemes can be instituted prior to training time.
- **RecShard:** We propose RecShard – a new approach for fine-grained sharding of EMBs with respect to a multi-level memory hierarchy consisting of GPU HBM and CPU DRAM. RecShard optimizes EMB partitioning and placement globally based on the estimated sparse feature characteristics and DLRM architecture.
- **Real system evaluation:** To demonstrate its efficacy, we implement and evaluate RecShard in the context of a production scale DLRM. We demonstrate that RecShard can on average improve the performance and load balance of DLRM EMB training by over 5x and over 11x, respectively, compared to the state-of-the-art industry sharding strategies [1, 26, 31].

2 BACKGROUND

Figure 2 gives an overview of the canonical Deep Learning Recommendation Model (DLRM) architecture [33]. In this section, we provide background on DRLMs and the training systems.

DLRMs process user-content pairs to predict the probability that a user will interact with a particular piece of content, commonly referred to as the click-through-rate (CTR). To produce such a prediction, DLRMs consume two types of features: *dense* and *sparse.*

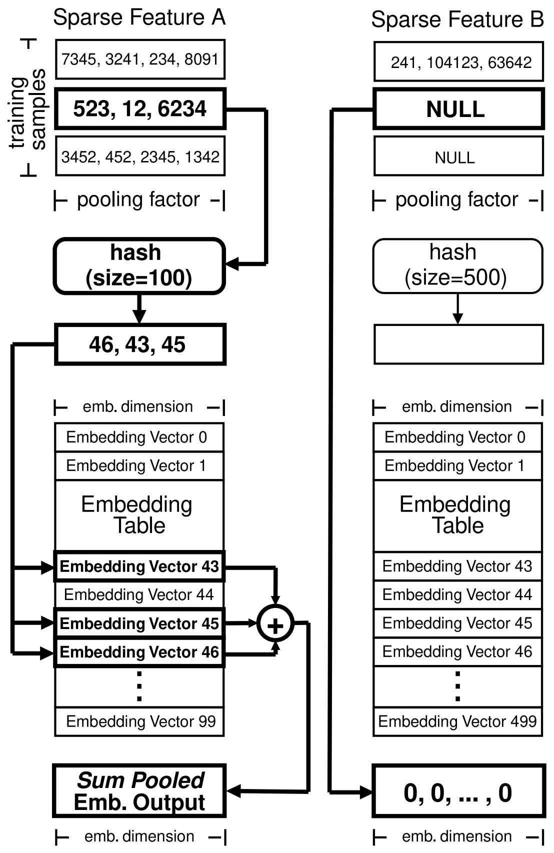


Figure 3: Example illustrating the pooling factor and coverage statistics, along with the embedding lookup and (sum) pooling operation. In this example there are two sparse features, *A* and *B*, with two corresponding embedding tables, and a training dataset composed of three training data samples. The average pooling factors of sparse features *A* and *B* over the dataset are 3.66 and 3, respectively, while the coverages are 1.0 and .33, respectively. The example shows the embedding lookup and pooling operation for the second training data sample (highlighted in bold). For sparse feature *A*, the raw input IDs are *hashed* with an output size of 100 (which corresponds to the number of rows in *A's* EMB), generating the corresponding embedding lookup indices. These embedding rows, each containing *embedding dimension* number of values, are then read and combined, i.e. *pooled*, via element-wise summation to produce the output vector of the lookup operation. For sparse feature *B*, the second training data sample is *NULL*, signifying that *B* contains no feature data for that particular data sample. This results in the stages which sparse feature *A* went through being bypassed and a 0-vector being produced as the output.

Dense features represent continuous data, such as a user's age or the time of day, while sparse features represent categorical data, such as domain names or recent web pages viewed by a user. To encode this categorical data, sparse features are represented as one-hot or

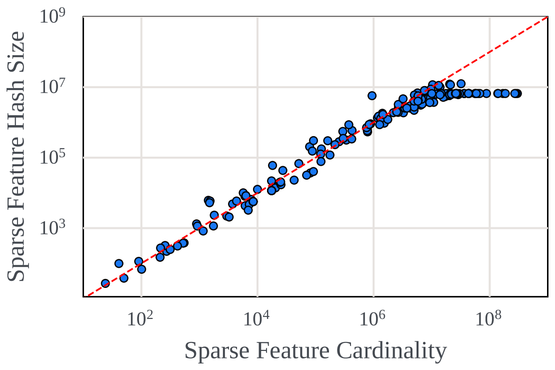


Figure 4: Sparse feature cardinality (categorical space; x-axis) versus chosen feature hash size (EMB size; y-axis) for 200 sparse features used in a large production-scale model. Hash size equal to cardinality is shown by the red-dotted line.

multi-hot binary vectors which are only activated for a small subset of relevant categories (hence the term *sparse*). Sparse features used in DLRMs can have cardinalities in the billions [22, 46].

At a high level, the primary components of DLRMs are Multi-Layer Perceptrons (MLPs) and Embedding Tables (EMBs). EMBs are commonly-used to transform sparse features from the high-dimensional, sparse input space to low-dimensional, dense embedding vectors. EMBs perform this operation by functioning as large lookup tables, where, in theory, each rows acts as a latent vector encoding of a particular sparse feature value (i.e., category). The activated, or hot, indices of the sparse inputs then act as indices into the EMBs, gathering one or more embedding vectors.

In practice, however, the binary-encoded sparse feature inputs are hashed prior to EMB look-up. Hashing serves two purposes. First, hashing allows the bounding of a sparse feature's EMB to a pre-determined, fixed size. Second, hashing permits the handling of unseen inputs at runtime [1, 22]. Once gathered, the embedding vectors are aggregated on a per-EMB basis using a *feature pooling* operation, such as summation or concatenation. The pooled vectors, along with the outputs of the bottom MLP layers (which process dense inputs), are then combined using a *feature interaction* layer, before proceeding through the top MLP layers and producing a prediction for the estimated engagement for the user-content pair.

Training Systems for DLRMs DLRMs present significant infrastructure challenges. While the MLP layers are compute-intensive and exhibit (relatively) small memory footprints, single EMBs of production-scale DLRMs can be on the order of 100s of gigabytes each, with the total memory capacity on the multi-TB scale [22, 31, 46]. Furthermore, EMBs exhibit irregular memory access patterns [41], and the concurrent vector accesses per-EMB and across EMBs require substantial memory bandwidth [1, 23]. This has led to a hybrid data- and model-parallel training approach (Figure 2). MLP layers (both top and bottom) are replicated across all trainers (GPUs in figure) in a data-parallel manner, while EMBs are sharded across trainers to exploit model-parallelism [17, 18, 31, 44].

The ever-increasing memory capacity and bandwidth demands of DLRM training has also led to the emergence of training systems with tiered hierarchical memories (such as hierarchies with HBM, DRAM, and SSD tiers). The large collection of EMBs are partitioned and/or cached across the various tiers [31, 46]. One class of partitioning approaches leverages *unified virtual memory* (UVM) [13]. This places both host DRAM and accelerator HBM in a shared virtual address space, allowing transparent access of host DRAM on a GPU accelerator without explicit host-device transfers [27, 30]. UVM can greatly expand the usable memory capacity of a GPU node with ease. For example, a server with 8x 32GB HBM GPUs can have 2TB of DRAM [1].

However, for memory-bound workloads, such as DLRMs, using UVM naïvely can come with significant performance cost. While the latest GPUs contain HBMs with bandwidth capacity approaching 2TB/s, the interconnects used can have bandwidth capacity an order of magnitude less. Single direction throughput of PCIe 4.0x16, for example, is just 32 GB/s. This places particular importance on the DLRM EMB sharding scheme—*hundreds of EMBs with heterogeneous memory characteristics have to be placed across potentially hundreds of trainers.*

To address the performance needs of production-scale DLRM training in the presence of rapidly-growing memory capacity and bandwidth demands, this paper focuses on the *partitioning and placement problem*—determining the optimal placement of EMBs on a tiered memory system with fixed memory capacity and bandwidth constraints.

3 CHARACTERIZATION OF DLRM SPARSE FEATURES

The goal of a *DLRM sharder* is to partition a model's EMBs across a training system's hardware topology, in order to fully exploit model parallelism and thereby maximize training throughput. This requires an EMB placement across an increasingly tiered memory hierarchy that balances training load across all trainers (GPUs). To achieve such load balancing, an effective EMB sharder must be able to accurately estimate the memory demands of each EMB. RecShard addresses this problem through a data-driven approach.

This section presents our in-depth memory characterization of sparse features used in industry-scale DLRMs. The characterization study captures the statistical nature of recommendation training data, and sheds light on five key characteristics of DLRM sparse features which RecShard exploits to improve the EMB training throughput performance. Notably, we find that a sparse feature's *value distribution* enables us to determine the portion of an EMB that will exhibit high temporal locality during training, the feature's *average pooling factor* provides a proxy for its memory bandwidth cost, and the feature's *coverage* allows us to rank the placement priorities across EMBs. Furthermore, these statistics are *distinct and vary over time* for each sparse feature.

3.1 Skewed Categorical Distribution Presents Unique EMB Locality Characteristics

A small subset of categories can constitute the majority of accesses to an EMB.

Sparse features represent categorical data, with each sparse feature's data sample containing a variable length list of categorical values from its sparse feature space. As the size of this categorical feature space can be arbitrarily large, it is natural to ask if a subset of values appear more often than others, and in fact they do [8, 19, 25, 42]. For example, the country a user is located in is a common feature for recommendation use cases. If we were to measure the distribution of this feature, we would see the feature follows a skewed power-law distribution, as the world population by country itself follows a power-law distribution with a long tail. Production-scale DLRMs often consist of hundreds of features that exhibit similar categorical frequency distributions [1, 31].

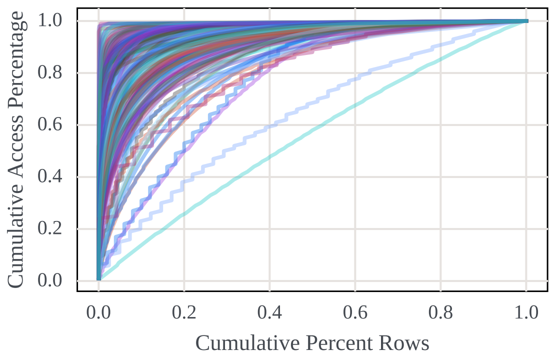


Figure 5: Hashed Value Frequency CDFs of 200 sparse features used in a production DLRM. The CDFs are generated from over two billion randomly-selected training samples over ten days of data, *post-hash.*

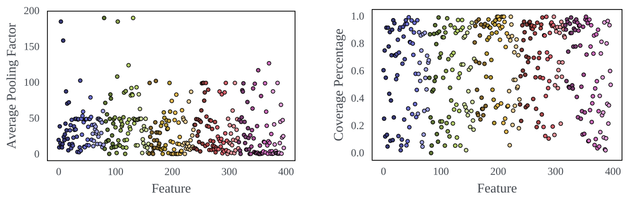
Figure 5 illustrates the cumulative distribution function (CDF) of 200 representative categorical features of a production DLRM. While a handful of features exhibit more uniform value distributions, the vast majority display a power-law distribution over the categorical values. In other words, for the majority of features, a small subset of categories appear much more frequently than the rest. This implies that a small set of EMB rows comprise the majority of EMB accesses. It is important to also highlight that *the strength of the distribution varies from one feature to another, requiring consideration of the distribution on a per-feature basis.*

Overall, the locality characteristics unique to each feature give rise to an optimization opportunity – EMB entries *within a table* can be placed across a tiered memory hierarchy based on expected access patterns. We refer to this optimization as fine-grained EMB partitioning.

3.2 Pooling Factors Determine Memory Bandwidth Demand

Within a training data sample, each EMB exhibits its own bandwidth demand due to varying pooling factor distributions.

Activated indices in a sparse feature's input effectively correspond to the rows in the feature's EMB that should be accessed to acquire latent vector representations of the categories. This results in a scatter-gather memory access pattern, where one embedding vector is accessed for each activated index. The *n* EMB rows accessed by a sparse feature's input is its sample *pooling factor*, whereas the interaction of the corresponding *n* latent embedding



(a) *Average pooling factor*: the number of 'hot' indices in an average sparse feature's input sample. **(b) *Coverage percentage*: the probability a sparse feature is present in a random training data sample.**

Figure 6: Average pooling factor and coverage vary widely from feature to feature. Collectively, they serve as a proxy for the per-sample bandwidth demand of a feature.

vectors via *pooling* determines the feature sample's representation. The distribution of the pooling factors – n – of a sparse feature across the training data models the feature's memory bandwidth consumption.

Furthermore, the pooling factor distribution can vary from feature to feature, resulting in memory bandwidth needs that are feature-specific (i.e., EMB-specific). This is due to variability in the information each feature represents. While the feature representing the location of a user may always be of length one, a feature representing the pages recently viewed by a user will likely have length greater than one. Figure 6a depicts the average pooling factor distribution for hundreds of sparse features which varies widely. Some sparse features exhibit high pooling factors of approximately two hundred on average, while the average pooling factors of others are on the order of a few tens; the result is an order of magnitude difference in the memory bandwidth demand.

As with sparse feature value distributions, the pooling distributions for sparse features are also skewed with a long tail; however unlike the value distributions, they cannot be broadly classified as being power-laws with varying degrees of strengths. We experimented with an assortment of summary statistics, such as the median and mean, to determine which provides the best estimate for the 'average' case across all features; resulting in the choice of mean as the estimate for the *average pooling factor* of a sparse feature. This choice was made as we observed that the mean generally tends to over-estimate an EMB's bandwidth demand, which we find preferable to under-estimating and potentially resulting in a sub-optimal EMB placement.

In summary, pooling factor diversity across features motivates optimizations that consider per-feature *average pooling factors* to approximate the unique memory bandwidth consumption characteristics for EMBs.

3.3 Varying Degrees of Coverage for Sparse Features Determines EMB Placement Priority

Sparse features exhibit varying degrees of coverage, with some EMBs being used much more often than others.

Not all sparse features of a DLRM are referenced in each training data sample. There are a variety of reasons for this, such as a particular feature being phased in or out, or a user simply not having the content interaction or metadata necessary for the feature to be instantiated. Regardless of the reason, there is variability in the presence of sparse features across training inputs, which provides us with additional empirical information for system performance optimizations.

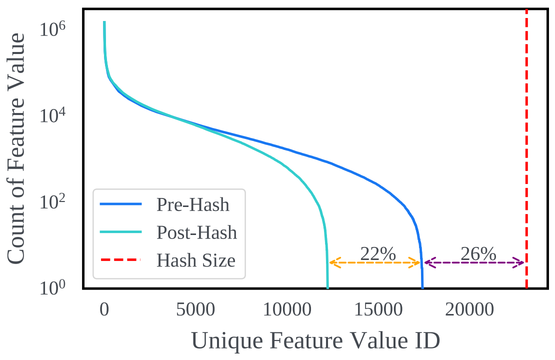


Figure 7: The impact of hashing on the feature value frequency distribution. Even using a hash size greater than the number of unique values, hashing causes the compression of the raw value distribution, leaving considerable EMB underutilization.

Figure 6b depicts the feature access probabilities (y-axis) across hundreds of sparse features sampled from a number of industry-scale DLRMs (x-axis). The probability that a sparse feature is present in a training sample is referred to as its *coverage*. Similar to the pooling factor distribution (Section 3.2), the coverage of individual sparse features varies widely from feature to feature – ranging from less than 1% on the low-end to 100% on the high-end. This observation demonstrates the importance of considering per-feature coverage characteristics in EMB placement decisions. Thus, a feature's *coverage* gives rise to system optimizations based on the prioritization of EMBs according to their frequency of use.

3.4 Embedding Hashing Leads to Sub-optimal System Memory Utilization

While a simple technique, embedding hashing is inefficient from the perspective of system memory utilization.

The cardinality of a given sparse feature can be on the order of billions. Thus, constructing an EMB representing the entirety of such a sparse feature would be prohibitively expensive in terms of the memory capacity requirement. Furthermore, it would not generalize to unseen feature values when new categories emerge. Thus, it is unrealistic to construct a one-to-one mapping between every sparse feature value and EMB rows. Instead, the EMBs of industry-scale DLRMs typically employ hashing [1, 22, 39], using a random hash function to map arbitrary feature values to output values constrained by a specified hash size. The hash size therefore dictates the size of the EMB.

A consequence of using random hashing to map a feature's inputs to corresponding EMB entries is *hash collisions*—where the hash function maps two unique input values to the same output value. The existence of hash collisions can be demonstrated via the

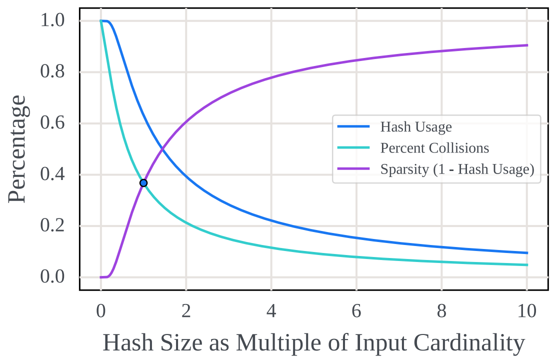


Figure 8: Increasing the hash size to accommodate the tail leaves an increasing percentage of the hash space unused, which RecShard can reclaim. The blue dot denotes the point at which hash size is equal to input cardinality.

pigeonhole principle, as mapping $H + 1$ unique values with a hash size of H requires at least two input value overlap. What is less obvious however, is whether or not, and to what degree, collisions occurs when the hash size is equal to or even slightly greater than the number of unique input values seen. Commonly known as the birthday paradox, when hashing N unique input values with a hash size of $H = N$, one will observe that approximately $\frac{1}{e}$ input values will collide. And, as $N = H$, this results in $\frac{1}{e}$ hash entries being unused.

Figure 7 depicts the birthday paradox phenomenon by illustrating the pre- and post-hash distributions for a specific feature of a production DLRM. The pre-hash distribution (dark blue line) depicts the input feature value space, whereas the post-hash distribution (light blue line) depicts the distribution of accesses to the corresponding EMB. The red-dotted vertical line denotes the specified hash size and therefore the number of unique embedding vectors that can be captured by this EMB. Although the hash size is greater than the number of unique pre-hash values observed (the red dotted line is to the *right* of the dark blue line), the post-hash embedding space compresses the pre-hash categorical feature space (the light blue line terminates to the *left* of the dark blue line). Furthermore, Figure 7 highlights the under-utilization of EMBs due to training data sparsity by 26% and hash collisions by another 22%.

Increasing the hash size to accommodate the tail of the power-law distribution – a technique which can improve model performance [46] – leaves an increasing percentage of the hash space under-utilized, which RecShard can reclaim. Figure 8 illustrates that, as the hash size is increased to accommodate the tail of the input sparse feature distribution (Section 3.1), an increasing percentage of the hash space is unused by training samples (sparsity increases).

Given the observations above, hashing gives additional insight into designing an intelligent partitioning strategy for EMBs. Due to the birthday paradox and the desire to choose a hash size which can retain as much of the tail as possible, a non-trivial percentage of embedding rows will not be accessed at all during training. This enables us to move the under-utilized portions of EMBs to a slower memory tier (or potentially avoid allocation altogether) without visible impact on the training time performance.

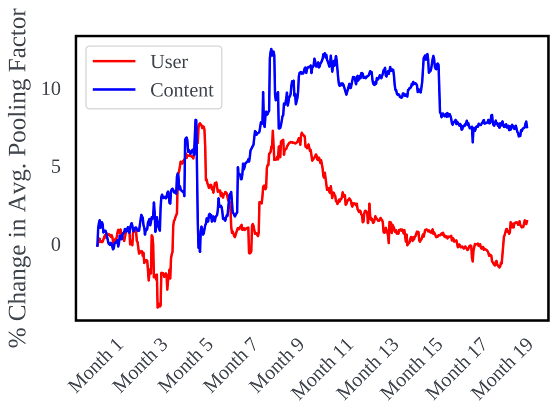


Figure 9: Sparse features are grouped into two general categories, users and content. Both feature types exhibit dynamic memory demand over time. We show memory demand for a large production model (~400 features) over a 20-month period. Data represent averages over all relevant features.

3.5 Sparse Feature Memory Patterns Evolve over Time

Sparse features exhibit distinct, dynamic memory demands over time.

Sections 3.1-3.4 provide insights into *how* memory characteristics specific to DLRM sparse features and EMB design can be used to optimize the EMB performance of DLRMs through an intelligent data-driven sharding strategy. It is, however, also important to know *how often* EMB sharding should be performed. Once deployed, industry-scale production models may be continuously retrained on new data for potentially many weeks [14] at a time.

Figure 9 illustrates how average feature lengths evolve over a 20-month time period for two distinct types of sparse features: *content* features and *user* features. Based on the time-varying nature of sparse feature statistics, ideally the benefit of re-sharding would be evaluated regularly throughout training as new data arrives, due to the potentially large impact that data distribution shifts can have on training throughput. Although this benefit can be approximated quickly by RecShard (Section 4), it must be dynamically weighed against the cost of carrying out the re-sharding on the given training stack and topology.

4 RECSHARD

Building on the EMB memory access characterization results in Section 3, we design, implement, and evaluate an intelligent EMB sharding strategy – *RecShard*. RecShard is a data-driven EMB sharding framework that optimizes embedding table partitioning and placement across a tiered memory hierarchy. Figure 10 provides the design overview for RecShard, which is comprised of three primary phases: Training Data Profiling (Section 4.1), Embedding Table Partitioning and Placement (Section 4.2), and Remapping (Section 4.3).

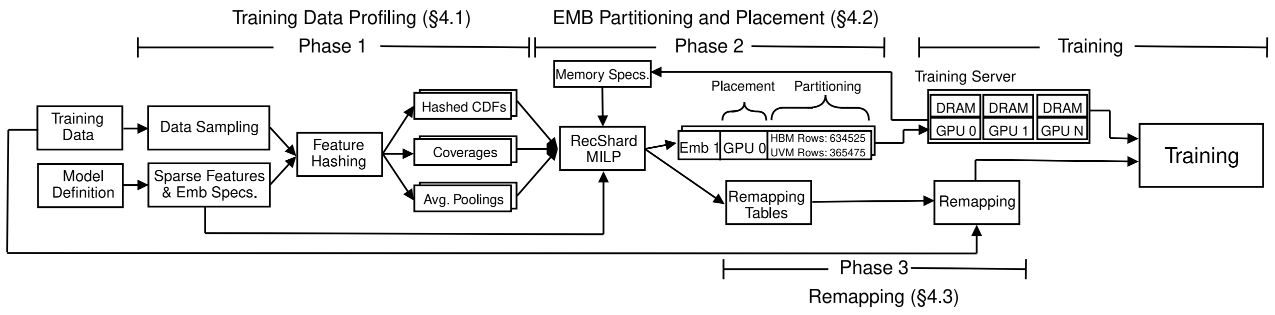


Figure 10: Overview flow diagram of the RecShard pipeline.

RecShard leverages a MILP along with the latest training data distributions and EMB design characteristics to produce an optimal EMB sharding strategy *each time a given DLRM is trained.*

4.1 Training Data Profiling

The first stage of the RecShard pipeline is model-based training data profiling, which approximates the aforementioned memory characteristics in Section 3. In this stage, RecShard first samples and hashes a random subset of the input training dataset based on the DLRM architecture specification. The purpose of this sampling is to estimate three per-EMB statistics: (1) the value frequency CDF over the EMB entries, (2) the average pooling factor of accesses for each EMB, and (3) each EMB's coverage over the training dataset.

We observe empirically that sampling 1% or less of large training data stores achieves statistical significance to accurately facilitate high-performance EMB partitioning decisions. This is largely because increasing the sampling rate primarily serves to capture more of the tail of a sparse feature's skewed distribution. With respect to the value frequency CDF, these extra "tail values," when hashed, will either map to their own EMB entry with minimal access count, or will collide with other previously-seen feature values. And viewing more of the tail has little to no impact on the average pooling factor and coverage of an EMB. In all cases, not capturing the full tail is sufficient from the perspective of memory pattern profiling.

In the training data profiling phase, RecShard constructs the value frequency and pooling factor statistics as well as the coverage of each sparse feature for use in sharding.

4.2 Embedding Table Partitioning and Placement

RecShard uses the generated per-feature statistics to produce an efficient, load-balanced EMB partitioning decision. In order to perform partitioning and sharding across multiple compute nodes with a tiered memory hierarchy, RecShard formulates the partitioning problem as a mixed integer linear program (MILP). By solving the MILP [12], RecShard can globally minimize per-GPU cost, a proxy for EMB training latency, simultaneously, while ensuring that neither GPU on-device nor per-node host memory limits are violated. The remainder of this section outlines our MILP formulation, which considers the problem of sharding EMBs across a two-tier memory hierarchy consisting of GPU HBM and host DRAM accessed via UVM. We refer to the latter as UVM for the rest of this paper. Table 1 summarizes parameters used by the MILP formulation.

Table 1: Description of Parameters used in the RecShard MILP.

Parameter	Description
M	Number of GPUs
$\mathcal{J}$	Number of EMBs
B	Batch size
Cap_D	Per-GPU HBM Capacity
Cap_H	Per-GPU Host DRAM Capacity
BW_{HBM}	GPU HBM Bandwidth
BW_{UVM}	UVM Transfer Bandwidth
$ICDF_j$	Inverse Value Frequency CDF of EMB j
avg_pool_j	Average Pooling Factor of EMB j
$coverage_j$	Coverage of EMB j
$hash\_size_j$	Hash Size of EMB j
dim_j	Embedding Dimension of EMB j
$bytes_j$	Size of data-type of EMB j

MILP Formulation As the training throughput is determined by the embedding operator performance of the slowest trainer, we formulate the MILP as a minimization problem to:

$$\begin{aligned} &\textit{minimize} && C \\ &\textit{subject to} && c_m \le C \qquad \forall m \in M \end{aligned} \tag{1}$$

M is the set of GPUs available for training (each GPU is represented by an integer ID m ranging from 0 to $M-1$), c_m is the total EMB cost for GPU m, and C is the maximum single GPU cost to minimize, subject to Constraint 1.

In order to estimate the total EMB cost per GPU, RecShard incrementally incorporates the per-EMB memory statistics to construct constraints which effectively describe the space of all possible EMB partition and placement combinations for the underlying tiered memory hierarchy.

To construct a search space of candidate placements, the first constraint specified by RecShard is the mapping of each EMB to a single GPU. An EMB can either be located fully in a GPU's HBM, fully in UVM, or split across both in a fine-grained manner. If an EMB is placed entirely in HBM, the corresponding GPU will be the sole accessor of the entire EMB. If an EMB is placed entirely in UVM, it must be assigned a GPU that will issue memory accesses to it. When an EMB is located in both HBM and UVM, we map both partitions to the GPU whose HBM is utilized. This constraint is formulated as follows:

$$\sum_{m} p_{mj} = 1 \qquad \forall j \in J \tag{2}$$

$$p_{mj} \in \{0, 1\} \qquad \forall m \in M \quad \forall j \in J \tag{3}$$

p_{mj} is a binary variable indicating whether EMB j is assigned to GPU m, and Constraint 2 ensures that each EMB is assigned to exactly one GPU.

When determining the EMB-to-GPU mappings, RecShard must also decide how many, or if any at all, of each EMB's rows should be placed in HBM. To do so, RecShard uses each EMB's post-hash value frequency CDF to estimate the trade-off between the number of rows placed in HBM and the corresponding percentage of EMB accesses covered. To use the CDF within the MILP, RecShard first converts the CDF to its inverse, or ICDF, so that it can map the percentage of accesses covered to the corresponding number of EMBs rows. RecShard then produces a piece-wise linear approximation of the ICDF – as the ICDF is a non-linear function, it cannot be used directly within the MILP. To do so, 100 steps are uniformly selected with respect to the ICDF's x values, where each step i corresponds to a cumulative access percentage between 0 and 100%. To capture both the x and y values of the ICDF, the constraints are formulated as follows:

$$\sum_{i} x_{ij} * ICDF_j(i) * dim_j * bytes_j = mem_j \qquad \forall j \in J \tag{4}$$

$$\sum_{i} x_{ij} * \frac{i}{100} = pct_j \qquad \forall j \in J \tag{5}$$

$$\sum_{i} x_{ij} = 1 \qquad \forall j \in J \tag{6}$$

$$x_{ij} \in \{0, 1\} \qquad i = 0, ..., 100 \qquad \forall j \in J \tag{7}$$

x_{ij} is a binary variable indicating whether step i was chosen for EMB j. Constraint 6 ensures that one and only one step from the ICDF can be selected per EMB (i.e. there is a single split point separating the EMB rows mapped to HBM from those mapped to UVM). Constraint 5 converts the chosen step value for each EMB into the corresponding percentage – the ICDF's corresponding x value. For each EMB, this percentage represents the cumulative percentage of accesses covered by the chosen split, and its value is stored as pct_j. Finally, Constraint 4 translates each EMB's chosen split into the number of bytes needed to store its rows, mem_j – the per-EMB HBM usage.

Given the constraints for encoding per-EMB HBM usage, constraints are added to guarantee per-GPU memory capacity limits are not violated.

$$hash\_size_j * dim_j * bytes_j = EMB_j \qquad \forall j \in J \tag{8}$$

$$\sum_{j} p_{mj} * mem_j \leq Cap_D \qquad \forall m \in M \tag{9}$$

$$\sum_{j} p_{mj} * (EMB_j - mem_j) \leq Cap_H \qquad \forall m \in M \tag{10}$$

Constraint 9 accomplishes this for per-GPU HBM by summing the memory capacity requirements of all EMB portions assigned to each GPU m and ensuring that ensuring that no GPU exceeds its HBM capacity of Cap_D. Constraint 10 accomplishes this similarly for per-GPU host DRAM capacity limits, Cap_H.

With the EMB partitioning and placement assignments properly constrained, RecShard can formulate the estimated per-GPU EMB cost.

$$(avg\_pool_j * dim_j * bytes_j * B)* \\ ((pct_j * \frac{1}{BW_{HBM}}) + ((1 - pct_j) * \frac{1}{BW_{UVM}})) \\ = c_j \qquad \forall j \in J \tag{11}$$

$$\sum_{j} p_{mj} * coverage_j * c_j = c_m \qquad \forall m \in M \tag{12}$$

Constraint 11 estimates the cost of each EMB during a single forward pass of DLRM training. This is achieved by first calculating each EMB's approximate *per-training step* memory demand using the EMB's average pooling factor, embedding vector dimension, size (in bytes) of its embedding vector entries, and batch size. Per-step demand is then multiplied by: (1) the percentage of EMB rows that are estimated to be sourced from HBM (pct_j) along with a bandwidth based scaling factor ($\frac{1}{BW_{HBM}}$); and (2) the percentage that are estimated to be sourced from UVM ($1 - pct_j$) along with its scaling factor ($\frac{1}{BW_{UVM}}$). The two products are summed to form an estimate of an EMB's cost to perform a single step lookup on average.

Constraint 12 formulates the per-GPU cost for the MILP's objective function. Instead of simply summing the per-EMB costs assigned to a GPU, we sum the product of the per-EMB cost and the corresponding EMB's *coverage*. This is because the per-EMB CDF presents a normalized view of accesses over *a particular EMB*, and the average pooling factor estimates the EMB's memory performance requirement *over the samples it is present in*. Therefore, to provide a global view of bandwidth requirements *across all EMBs*, RecShard weights each EMB's cost by its coverage.

With the constraints in place to formulate the per-GPU EMB cost, c_m, the MILP solver considers all possible combinations of EMB partitioning and placement decisions based on RecShard's EMB statistics and the bandwidth characteristics of the underlying memory hierarchy (supplied via the BW parameters in Constraint 11). In doing so, the MILP solver can compute an optimal sharding strategy that minimizes the model's largest single GPU EMB cost.

Key Properties of RecShard's MILP We address a number of key properties pertaining to RecShard's MILP formulation. First, when constructing its placements, RecShard's MILP begins by assigning each EMB to a single GPU. This design decision, and others which follow from it (such as per-GPU host DRAM capacity limits, Cap_H), is done to simplify the handling of sharding across many GPUs and nodes. By splitting resources uniformly and constructing assignments on an abstract per-GPU basis, the resulting sharding assignment will not be tied to a specific system GPU (i.e. GPU 3 in the MILP can be mapped to any GPU in the system).

Second, RecShard's MILP features *summation* of the expected HBM and UVM lookup times to form a cost. Another operator, such as *max*, may be used, depending on the target system architecture. We use *summation* in our implementation of RecShard because, when accessed within the same kernel, the memory latency of performing mixed reads from HBM and UVM on current GPUs is approximately equal to the time to perform each read separately. However, if the target system architecture supports concurrent reads fully from mixed memories, the estimated EMB cost can be approximated using *max*.

Third, while RecShard performs partitioning and placement for DLRM training, it only estimates embedding operation latencies of the forward pass in the MILP. This is because the timing performance of the backward pass is roughly proportional to its forward pass performance. By doing so, it simplifies the MILP formulation and lowers the solver time.

In our experiments in Section 6, RecShard's MILP features 47,276 variables, and is solved in 21 seconds when UVM is not needed (RM1), and in 42 seconds when used (RM2/RM3), with a state-of-the-art solver (Gurobi [12]). It is important to note that solving time *is not impacted* by model size, but instead in terms of the number of trainers (e.g. GPUs), and the steps used to approximate the ICDF. In our experiments solving time tended to scale approximately linearly with number of trainers and steps.

4.3 Remapping Layer

Once the MILP solver produces a sharding strategy, RecShard determines the number of rows to be placed in HBM for each EMB via the activated x_{ij} variable and the corresponding location on the EMB's ICDF, $ICDF_j(i)$.

These selected rows cannot be placed directly in HBM and must go through a remapping stage. This step is necessary as EMBs are typically allocated contiguously in memory, with an EMB index also serving as the memory offset to access the underlying storage directly. As the EMB rows selected by the MILP to be placed in HBM are chosen based on their access frequency, they can be located at arbitrary positions within the EMB and thus be non-contiguous. To address this, RecShard creates a per-EMB *remapping table*, which maps each EMB index to its corresponding location in either HBM or UVM.

4.4 Expansion Beyond Two-Tiers

While the RecShard implementation is modeled after a two-tier memory hierarchy consisting of GPU HBM and host DRAM accessed via UVM, RecShard can be easily expanded to support a multi-tier memory hierarchy. At its core, each additional tier represents a new point on each EMB's CDF, potentially producing an additional split of EMB rows to be placed on the new memory tier. As each memory tier has its own bandwidth specifications from the view of the executing device (e.g. the GPU), the RecShard MILP solver will automatically order the memory tiers via the bandwidth scaling factors.

5 EXPERIMENTAL METHODOLOGY

Baselines: To evaluate the efficacy of RecShard, we compare the performance of EMB operators under RecShard's throughput optimized sharding strategy with sharding schemes from prior work on production DLRM training systems [1, 26, 31]. State-of-the art sharding schemes typically follow a two-step approach. First, they assign a fixed cost to each EMB based on a specific cost function. Second, they apply a heuristic algorithm to incrementally assign EMBs to GPUs while attempting to minimize the maximum cost across all GPUs (a measure of load balancing).

Table 2: DLRM Specifications

Model	# Sparse Features	Total Hash Size	Emb. Dim.	Size
RM1	397	1,331,656,544	64	318 GB
RM2	397	2,661,369,917	64	635 GB
RM3	397	5,320,796,628	64	1270 GB

Step I–Cost Functions: We implement the following three cost functions – two representing the state-of-the-art and a third derived from the first two – and compare their impact on EMB training throughput with RecShard:

- **Size** [1, 26]**:** An EMB's cost is the product of its hash size and its embedding dimension (latent vector length).
- **Lookup** [1, 26]**:** An EMB's cost is the product of its average pooling factor and its embedding dimension.
- **Size-and-Lookup:** An EMB's cost is the product of its lookup based cost (above) and the log of its hash size – $log_{10}(hash\_size_{EMB})$ – adding a non-linear function that attemps to capture potential caching effects of smaller EMBs.

In comparison, **RecShard** considers EMB access distributions, average pooling factor, coverage, hash function, hash size, and the memory bandwidth characteristics of the target system.

Step II–Heuristic Sharding Algorithms: To shard EMBs once assigned a cost, we implement a **greedy heuristic algorithm** [31] that works as follows. After receiving the list of EMBs to shard along with their associated costs, the greedy heuristic first sorts EMBs in descending cost order. It then descends the list, starting with the highest-cost EMB, and iteratively assigns EMBs to GPUs one-by-one. The heuristic continues down the sorted list of EMBs, placing each successive EMB on the GPU with the current lowest sum of costs. When GPU HBM has been saturated, the heuristic then allocates the remaining EMBs in UVM. In comparison, **RecShard** considers cost on a per-*EMB entry* basis and optimizes the placement of all EMB rows *simultaneously, in one shot.*

5.1 DLRM Specification

We evaluate the performance of the different sharding strategies on a system running a modified version of open-source DLRM [7, 33]. The implementation is modified to support the use of multi-hot encoded training data samples and the open-source implementation of the high-performance embedding operator in the PyTorch FBGEMM library[1].

We implement the RecShard remapping layer as a custom PyTorch C++ operator which is executed as a transform during the data loading stage. This allows remapping to be performed in parallel with training iterations, thus removing it from the critical path of model execution.

We evaluate RecShard using three production-scale DLRMs: RM1, RM2, and RM3, summarized in Table 2. All three RMs feature the same underlying DLRM architecture, implementing a large number

[1]https://github.com/pytorch/FBGEMM/tree/master/fbgemm_gpu

of sparse features (397) and spanning a breadth of feature characteristics: categorical value distributions, pooling factors, and coverage; which collectively determine the locality characteristics of embedding accesses (Section 3). The difference between the RMs is the approximate doubling of the hash size for each EMB from RM1 to RM2, and furthermore from RM2 to RM3.

We generate different workloads by having a large, constant number of features and scaling the hash sizes for two key reasons. First, the complexity of the sharding problem directly scales with the number of features to be sharded and their characteristics; thus, a large number of features maximizes sharding complexity. Second, as has been observed internally at our company, and in prior evaluations of industry-scale DLRMs [1, 22, 46], increasing the hash size of an embedding table and thereby reducing collisions between sparse feature values is a simple, yet effective method of realizing accuracy improvements.

Based on the system specification discussed in the next section, RM1 requires 14 GPUs to completely fit all EMBs in reserved HBM, while RM2 requires 27 GPUs, and RM3 requires 53 GPUs.

5.2 Training System Specification

We evaluate the timing performance for all three sharding strategies on a two-socket server-node. Each socket features an Intel Xeon Platinum 8339HC CPU, 376GB of DDR4 DRAM, and 8x NVIDIA A100 (40GB) GPUs, connected to host DRAM via PCIe 3.0x16 for UVM support. As the scale of the RMs exceeds that of the memory capacity of the training nodes, during benchmarking we run each model-parallel section separately and extract the EMB performance metrics.

When implementing the training sharding strategies from prior work [1, 26] (our baselines for comparison), we use a batch size of 16,384 and limit each sharding strategy to use at most: (1) 24GB of HBM per GPU as the reserved memory for EMBs; (2) 128GB of host DRAM for usage per GPU for UVM allocated EMBs. The remaining HBM/DRAM capacity reserved for other model parameters, computation, and training overheads.

Performance Profiling: As the goal of RecShard is to improve per-iteration EMB latency, due to the large percentage of total run-times they constitute for many types of DLRMs [1, 11, 46], we evaluate execution time by tracing each GPUs execution and extracting all kernel run times related to the embedding operator. We do this by using the integrated PyTorch profiler, `torch.profiler`, which allows for tracing to begin after a specified waiting and warm-up period. We specify a waiting period of 10 iterations, a warm-up period of 5 iterations, and trace for 20 iterations.

6 EVALUATION RESULTS AND ANALYSIS

Overall, RecShard achieves an average of 5x EMB training iteration time improvement across RM1, RM2, and RM3 in the 16-GPU setting, covering a wide range of memory demands. Figure 11 illustrates that RecShard improves the EMB training iteration time by 2.58x, 5.26x, and 7.41x for RM1, RM2, and RM3, respectively, over the next fastest sharding strategy. Table 3 summarizes the timing results for RecShard and the state-of-the-art schemes.

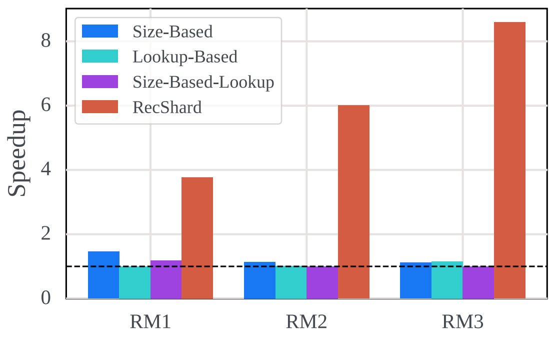


Figure 11: EMB training performance improvement of different sharding strategies normalized to slowest strategy in group. RM1, RM2, and RM3 evaluated using 16 GPUs.

6.1 RecShard Workload Balance Analysis

A major factor contributing to RecShard's significant performance improvement comes from its ability to achieve better load balance across the GPU trainers. In particular, the EMB memory footprint of RM1 is approximately 318GB, allowing all EMBs to fit entirely in HBM when distributed among the 16 available GPUs. RecShard improves EMB training throughput for RM1 by over 2.5x with respect to the next fastest sharding strategy (**Size**). It does so with an almost 9 times improvement in the standard-deviation of the iteration time across all GPUs, providing a much more uniform distribution of work (Table 3).

RecShard's ability to better load balance comes from two key aspects of its design. First, RecShard's hash analysis allows it to effectively determine *which* portion of each EMB is unused or sparsely used during training. The sparse regions are effectively assigned a cost of zero and thus can be allocated last. Second, RecShard's formulation of the EMB sharding problem as a MILP allows it to globally balance EMB operations across all GPUs *simultaneously, in one shot*. Since RM1 does not require UVM for EMB placement, the sharding cost formulation reduces to a function that is similar to the **Lookup** cost function of Section 5.1. However, when used with the greedy heuristic, the **Lookup** sharding strategy performs 46% worse than the **Size** strategy (the best baseline RM1 strategy). This result highlights the performance improvements that stem from RecShard's fine-grained, data-driven MILP approach to embedding vector placement.

6.2 RecShard Embedding Placement Analysis

As DLRM sizes grow beyond the capacity of available GPU HBM, as is the case for RM2 and RM3, sharding pressure moves beyond simply load balancing across HBM and into load balancing across HBM and UVM. With orders of magnitude difference in the memory performance of HBM and UVM, incorrect EMB placements on UVM come with severe performance penalties. In this scenario, the state-of-the-art sharding strategies can significantly under-perform RecShard. This is exemplified with RM2's and RM3's results.

RecShard uses feature and EMB statistics to dynamically estimate EMB cost at the row granularity, enabling it to intelligently break

Table 3: Min/Max/Mean/StdDev EMB training iteration time (in ms) across all GPUs based on per GPU averages for all sharding strategies on 16 GPUs. Training performance is bound by the slowest (i.e. *max*) EMB time, therefore lowest max iteration time is better. Load balanced is embodied by the standard deviation, with lower standard deviation signifying more balanced execution.

Model	Size-Based	Lookup-Based	Size-Based-Lookup	RecShard
RM1	7.12/21.23/13.06/4.01	5.08/30.97/12.99/5.59	5.55/26.03/12.91/4.72	6.53/**8.21**/7.48/**0.45**
RM2	20.52/49.65/33.82/7.37	10.40/55.85/32.47/9.87	7.47/56.66/32.95/10.26	6.52/**9.44**/7.75/**0.78**
RM3	40.43/76.15/56.45/10.86	3.37/73.30/55.27/18.53	5.10/85.01/56.04/20.39	6.83/**9.90**/8.31/**0.69**

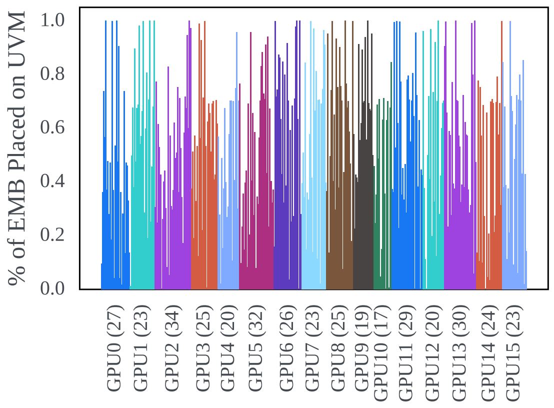


Figure 12: Partitions and Placements made by RecShard for RM2 on 16 GPUs. Each bar represents a single EMB. Bar height is the percentage of a specific EMB that RecShard placed on UVM. EMBs are grouped by the GPU they were assigned to (shown as colors). The number of EMBs assigned to each GPU is shown in parentheses. As expected, the number of EMBs assigned to each GPU is variable and the height of each bar is unique to each EMB.

apart an EMB into non-contiguous memory blocks and place each block independently across different tiers of the memory hierarchy. By doing so, RecShard determines and places the least performance-critical embedding portions of large DLRMs (i.e. RM2 and RM3) onto UVM.

For RM2, RecShard places an average of 53.4% of rows per EMB and a total of 61% of all EMB rows on UVM. For RM3, it places an average of 64.8% of rows per EMB and a total of 61% of all EMB rows on UVM. Figure 12 illustrates the partitioning decisions for RM2 using RecShard.

To further understand the difference in decision making between the baseline strategies and RecShard, we compare the EMB assignments and expected access counts for all strategies across RM2 and RM3. First, we explore how the individual EMB assignment by the **Size**, **Lookup**, and **Size-and-Lookup** strategies differ from RecShard's placement. That is, if an EMB was assigned to HBM, we examine the degree of overlap for the rows placed on UVM between the state-of-the-art strategy and RecShard. Table 4 summarizes this analysis. The rows labeled 'UVM->HBM' quantify the difference in the percentage of EMB rows placed in UVM for RM2 and RM3 by the state-of-the-art strategies versus RecShard. RecShard's ability to place more performance-critical, frequently-accessed embedding vectors onto HBM across all EMBs is the primary reason for its significantly higher performance.

Table 4: Percent of EMB rows allocated in UVM (resp. HBM) by each baseline strategy which RecShard places in HBM (resp. UVM). RM2 and RM3 require UVM for training on 16 GPUs, whereas RM1 does not. LB and SBL assign the same EMBs to HBM and UVM, but their exact GPU assignments differ. SB, LB, and SBL stand for Size-Based, Lookup-Based, and Size-Based-Lookup, respectively.

Model	Disparity	SB	LB	SBL
RM1	UVM->HBM	N/A	N/A	N/A
	HBM->UVM	N/A	N/A	N/A
RM2	UVM->HBM	28.67%	28.26%	28.26%
	HBM->UVM	39.93%	39.99%	39.99%
RM3	UVM->HBM	23.29%	23.21%	23.21%
	HBM->UVM	58.34%	59.36%	59.36%

Table 5: Average number of HBM and UVM accesses per-GPU, per-iteration for each sharding strategy on RM2 and RM3 (batch-size of 16384 on 16 GPUs). RM1 does not not require UVM. Baseline strategies source on average 20.3% (RM2) and 36.3% (RM3) of EMB accesses from UVM. RecShard sources 0.2% (RM2) and 0.5% (RM3) of EMB accesses from UVM. LB and SBL assign the same EMBs to HBM and UVM, but their exact GPU assignments differ. SB, LB, and SBL stand for Size-Based, Lookup-Based, and Size-Based-Lookup, respectively.

Model	Location	SB	LB	SBL	RecShard
RM1	HBM	88.74M	88.74M	88.74M	88.74M
	UVM	0	0	0	0
RM2	HBM	70.32M	70.90M	70.90M	88.48M
	UVM	18.42M	17.84M	17.84M	259K
RM3	HBM	55.82M	56.85M	56.85M	88.29M
	UVM	32.92M	31.89M	31.89M	450K

6.3 RecShard Scalability Analysis

As model sizes increase, as expected, RecShard sees little performance degradation. This comes from the asymmetric impact on memory access statistics and memory usage that hash size scaling causes. The state-of-the-art strategies experience an average of 3.07

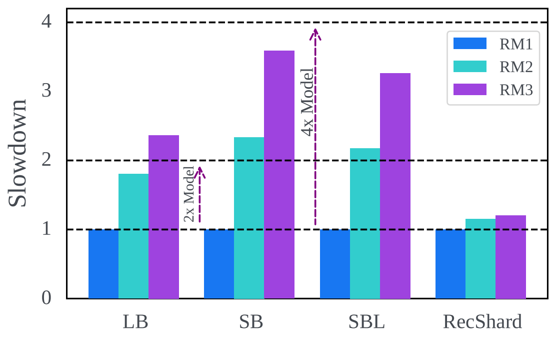


Figure 13: Slowdown of each sharding strategy on max EMB iteration time as model sizes scale 2x and 4x from RM1 to RM2, and RM1 to RM3, respectively. While heuristic based fixed-cost strategies suffer over a 3x slowdown on average from RM1 to RM3, RecShard is less sensitive to performance degradation from model size scaling and only experiences a 1.2x slowdown.

times performance slowdown in the EMB training iteration time between the largest DLRM (RM3) and RM1. However, RecShard only observes a 20.6% increase in the EMB training iteration time over the same model size growth (Figure 13).

We explored this sparsity in access count by analysing the number of HBM and UVM accesses made by the EMBs in each of the sharding strategies in our training traces. We found (Table 5) that when doubling the hash size from RM1 to RM2, the baseline sharding strategies sourced on average 20.3% of their accesses per-GPU per-iteration from UVM, while RecShard only sourced 0.2% – *over a 100x reduction*. When hash size is quadrupled from RM1 to RM3 and sharding pressure doubles from RM2, the baseline sharding strategies sourced on average 36.3% of their accesses per-GPU per-iteration from UVM, while RecShard only sourced 0.5%. As HBM capacity is already exceeded in RM2, the additional model capacity (in bytes) of RM3 *must* be allocated in UVM. While the percentage of accesses sourced from UVM for RecShard more than doubles from RM2 to RM3, this value is still only 0.5% of the total accesses (and *over 70x less* than the baseline strategies). This result highlights the sparsity of memory access to the new memory regions allocated by increased hash size.

6.4 End-to-End Training Time Improvement

While embedding operations can represent a significant portion of many industry-scale DLRMs [11, 46], the actual percentage of runtime varies based on model composition. RecShard improves end-to-end training performance in proportion to the time spent on embedding operations in the critical path of model execution (which in the canonical DLRM architecture consists of all embedding operations).

Knowing the runtime breakdown, the expected end-to-end DLRM training performance improvement can be approximated using Amdahl's law. With p being the percentage of total execution time spent on critical path embedding operations, and s being the speedup in embedding operation latency via improved sharding, the estimated end-to-end speedup is $\frac{1}{(1-p)+\frac{p}{s}}$.

As a concrete example, for memory-intensive models whose timing composition consists of 35-75% embedding operations [11, 23] (with the remaining time being largely dominated by dense MLP layers and communication), and for which RecShard improves embedding performance by 2.5x, the expected end-to-end performance benefit of RecShard is 1.27x to 1.82x. While the performance improvements afforded by RecShard are less pronounced for more MLP-dominated DLRMs, the position of embedding operations on the critical path of model execution and the scale of industry-DLRM training time (on the order of days [1]) indicates the importance of their acceleration.

Table 6: RecShard Ablation. Average number of HBM and UVM accesses per-GPU on RM3 (across 16 GPUs) over more than 200 million training data samples for different RecShard formulations. *CDF only* is the use of only the per-sparse feature value CDF in the MILP (i.e. average pooling factor and coverage are set 1). *CDF + Coverage* is the use of both the CDF and coverage in the MILP; while *CDF + Pooling* is the use of both the CDF and average pooling factor in the MILP. *RecShard (Full)* is the access counts when all per-EMB statistics are used simultaneously in the MILP.

Formulation	HBM	UVM
RecShard (Full)	69.07B	353M
CDF + Pooling	68.82B	604M
CDF + Coverage	68.54B	881M
CDF Only	67.79B	1.63B

6.5 RecShard Ablation

To better understand the impact the various sparse feature characteristics used within RecShard have on the performance of the generated sharding, we performed an ablation study to measure their significance on the number of HBM and UVM accesses made by each GPU. We evaluate four different formulation of RecShard, each differing by which per-EMB statistics are enabled for use within the MILP. The results of this ablation on RM3 (with 16 GPUs) over more than 200 million training data samples is shown in Table 6. The four formulations of RecShard evaluated are:

- *CDF only*: Only the sparse feature value CDF is used in the MILP and the average pooling factor and coverage for each EMB are set to 1.
- *CDF + Coverage*: Both the CDF and the per-EMB coverage are used in the MILP.
- *CDF + Pooling*: Both the CDF and the per-EMB average pooling factor are used in the MILP.
- *Full*: All of the per-EMB statistics are used in the MILP simultaneously.

Similar to the results in Section 6.3, we observe that approximately 0.5% of accesses on average in the full formulation of RecShard are sourced from UVM, while the simplest RecShard formulation, *CDF only*, sources approximately 2.4% of its accesses from UVM. While this is still significantly less than the baseline sharding strategies, this nearly 5x increase over the full formulation is due to the CDF providing no information about *how often* each EMB will be accessed in a training data sample. Thus when evaluating different potential partitioning and placement decisions, the MILP in the *CDF only* formulation has no information which it can exploit to accurately load balance EMBs across the GPUs based on their expected number of accesses. Adding one piece of per-sample EMB access information via the coverage almost halves the average UVM sourced access percentage to approximately 1.3%, while using the average pooling factor instead provides an even greater reduction to approximately 0.9%.

6.6 RecShard Overhead

For all models studied in this work, the Gurobi solver [12] was able to solve the placement and partitioning MILP in under 1 minute. After which, generating the remapping tables takes approximately 20 seconds for each GPU and has a storage cost of 4 bytes per row remapped (as the sign of the remapped index can be used to denote whether the corresponding table is the HBM or UVM partition). For the largest DLRM–RM3, this is a total storage overhead of ~20GB for over 5-billion rows remapped. In the scope of model training time (many hours to potentially days depending on model and data size), and model size (hundreds of GBs to multiple TBs), this overhead is minimal, especially due to the performance improvements RecShard provides.

Additionally RecShard incurs some overhead from training data profiling due to the consumption of feature level statistics. However, besides only needing to sample a small portion (~1%) of large training data stores to achieve statistical significance, as the statistics are based on raw training data values and corresponding hash sizes (which are generally constant across models within a size tier), they can be shared across models and also updated in real-time as training data arrives, amortizing the cost.

7 RELATED WORK

Power-law distributions are a well-known phenomenon of features related to recommender systems [2, 8, 22, 42, 45]. This sparsity characteristic is an important feature for a variety of DL system performance optimizations. However, maintaining the long tail is important because of the statistically significant accuracy impact [46]. This has led to recent works attempting to balance the trade-off between EMB sizes and model accuracy. One such category of work explores scaling the dimension of an EMB, that is the number of parameters used to encode an EMB row, based on the frequency of accesses to individual rows—more frequently accessed rows are given more space through increased embedding vector dimensions [8]. Another work explores the impact of hashing, ranging from the use of multiple hash functions alongside a 1:1 mapping for frequent categorical values [45], to entirely replacing the hashing plus embedding table structure with its own neural network [22]. In addition, other prior work proposes to prioritize frequently-accessed embedding rows for model parameter checkpointing [28], in order to improve failure tolerance of DLRM training. While prior work also tackles the problem of ever-increasing EMB sizes, their primary focus is the size of EMB itself, rather than on training throughput improvement.

Recent work has also explored the performance of splitting EMBs based on their frequency characteristics [2]. While similar in motivation, the type of training data and the scale of DLRMs explored in this paper are fundamentally different from the open-source datasets used in the related work. Our DLRMs read *multi-hot encoded sparse features* resulting in order-of-magnitude higher memory bandwidth needs, and EMB sizes demanding model-parallel training. In Criteo Terabyte (the largest of the open-source datasets), all of the features are 1-hot encoded (meaning their pooling factor is always 1), the number of features present is 26, and the total number of un-hashed embedding table rows is approximately 266 million. Thus, for each of these properties, the scale of open-source datasets/DLRMs [29, 35, 42] is an order of magnitude (or more) less than our evaluated datasets/DLRMs. Furthermore, all open-source datasets that we are aware of can fit entirely within a single GPU, making sharding and model-parallel training unnecessary.

8 CONCLUSION

Deep learning recommendation systems are the backbone of a wide variety of cloud services and products. Unlike other neural networks with primarily convolution or fully-connected layers, recommendation model embedding tables demand orders-of-magnitude higher memory capacity (>99% of the model capacity) and bandwidth, and exhibit significantly lower compute-intensity. In this paper, we perform an in-depth memory characterization analysis and we identify five important memory characteristics for sparse features of DLRMs. Building on the analysis, we propose RecShard, which formulates the embedding table partitioning and placement problem for training systems with tiered memories. RecShard uses a MILP to reach a partitioning and placement decision that minimizes embedding access time under constrained memory capacities. We implement and evaluate RecShard by training a modified version of open-source DLRM with production data. RecShard can achieve an average of over 5 times speedup for the embedding kernels of three representative industry-scale recommendation models. We hope our findings will lead to further memory optimization insights in this important category of deep learning use cases.

ACKNOWLEDGMENTS

We would like to thank Jade Nie, Jianyu Huang, Jongsoo Park, Andrew Tulloch, Xing Liu, Benny Chen, Ying Liu, Liu Ke, Udit Gupta, Newsha Ardalani, Hsien-Hsin S. Lee, and Kim Hazelwood at Meta for their valuable feedback and various discussions on this work, as well as Fan Yang and the anonymous reviewers for their constructive feedback. This work was supported in part by the Stanford Platform Lab and its affiliates for Geet Sethi and Christos Kozyrakis.

REFERENCES

[1] Bilge Acun, Matthew Murphy, Xiaodong Wang, Jade Nie, Carole-Jean Wu, and Kim Hazelwood. 2021. Understanding Training Efficiency of Deep Learning Recommendation Models at Scale. In *2021 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. https://doi.org/10.1145/3466752.3480127

[2] Muhammad Adnan, Yassaman Ebrahimzadeh Maboud, Divya Mahajan, and Prashant J. Nair. 2021. High-Performance Training by Exploiting Hot-Embeddings in Recommendation Systems. *CoRR* (2021). https://arxiv.org/abs/2103.00686

[3] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, Rohan

Anil, Zakaria Haque, Lichan Hong, Vihan Jain, Xiaobing Liu, and Hemal Shah. 2016. Wide & Deep Learning for Recommender Systems. In *Workshop on Deep Learning for Recommender Systems*. https://doi.org/10.1145/2988450.2988454

[4] Paul Covington, Jay Adams, and Emre Sargin. 2016. Deep Neural Networks for YouTube Recommendations. In *ACM Recommender Systems Conference*. https://doi.org/10.1145/2959100.2959190

[5] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 248–255. https://doi.org/10.1109/CVPR.2009.5206848

[6] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423

[7] Facebook Research. 2021. An implementation of a deep learning recommendation model (DLRM). https://github.com/facebookresearch/dlrm.

[8] Antonio Ginart, Maxim Naumov, Dheevatsa Mudigere, Jiyan Yang, and James Zou. 2019. Mixed Dimension Embeddings with Application to Memory-Efficient Recommendation Systems. *arXiv preprint arXiv:1909.11810* (2019). https://arxiv.org/abs/1909.11810

[9] Carlos A. Gomez-Uribe and Neil Hunt. 2016. The Netflix Recommender System: Algorithms, Business Value, and Innovation. *ACM Trans. Manage. Inf. Syst.* 6, 4, Article 13 (Dec. 2016), 19 pages. https://doi.org/10.1145/2843948

[10] Udit Gupta, Samuel Hsia, Vikram Saraph, Xiaodong Wang, Brandon Reagen, Gu-Yeon Wei, Hsien-Hsin S. Lee, David Brooks, and Carole-Jean Wu. 2020. DeepRecSys: A System for Optimizing End-To-End At-Scale Neural Recommendation Inference. In *Proceedings of the ACM/IEEE Annual International Symposium on Computer Architecture*. https://doi.org/10.1109/ISCA45697.2020.00084

[11] Udit Gupta, Carole-Jean Wu, Xiaodong Wang, Maxim Naumov, Brandon Reagen, David Brooks, Bradford Cottel, Kim Hazelwood, Mark Hempstead, Bill Jia, Hsien-Hsin S. Lee, Andrey Malevich, Dheevatsa Mudigere, Mikhail Smelyanskiy, Liang Xiong, and Xuan Zhang. 2020. The Architectural Implications of Facebook's DNN-Based Personalized Recommendation. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. https://doi.org/10.1109/HPCA47549.2020.00047

[12] Gurobi Optimization, LLC. 2021. Gurobi Optimizer Reference Manual. (2021). https://www.gurobi.com

[13] Mark Harris. 2013. Unified Memory in CUDA 6. https://developer.nvidia.com/blog/unified-memory-in-cuda-6/.

[14] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, James Law, Kevin Lee, Jason Lu, Pieter Noordhuis, Misha Smelyanskiy, Liang Xiong, and Xiaodong Wang. 2018. Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. https://doi.org/10.1109/HPCA.2018.00059

[15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778. https://doi.org/10.1109/CVPR.2016.90

[16] S. Hsia, U. Gupta, M. Wilkening, C. Wu, G. Wei, and D. Brooks. 2020. Cross-Stack Workload Characterization of Deep Recommendation Systems. In *IEEE International Symposium on Workload Characterization (IISWC)*. IEEE Computer Society. https://doi.org/10.1109/IISWC50251.2020.00024

[17] Biye Jiang, Chao Deng, Huimin Yi, Zelin Hu, Guorui Zhou, Yang Zheng, Sui Huang, Xinyang Guo, Dongyue Wang, Yue Song, Liqin Zhao, Zhi Wang, Peng Sun, Yu Zhang, Di Zhang, Jinhui Li, Jian Xu, Xiaoqiang Zhu, and Kun Gai. 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* (Anchorage, Alaska) *(DLP-KDD '19)*. Association for Computing Machinery, New York, NY, USA, Article 6, 9 pages. https://doi.org/10.1145/3326937.3341255

[18] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. 2020. A Unified Architecture for Accelerating Distributed DNN Training in Heterogeneous GPU/CPU Clusters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 463–479. https://www.usenix.org/conference/osdi20/presentation/jiang

[19] Manas R. Joglekar, Cong Li, Mei Chen, Taibai Xu, Xiaoming Wang, Jay K. Adams, Pranav Khaitan, Jiahui Liu, and Quoc V. Le. 2020. Neural Input Search for Large Scale Recommendation Models. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Virtual Event, CA, USA) *(KDD '20)*. Association for Computing Machinery, New York, NY, USA, 2387–2397. https://doi.org/10.1145/3394486.3403288

[20] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the ACM/IEEE 44th Annual International Symposium on Computer Architecture*. https://doi.org/10.1145/3079856.3080246

[21] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, Alex Bridgland, Clemens Meyer, Simon A. A. Kohl, Andrew J. Ballard, Andrew Cowie, Bernardino Romera-Paredes, Stanislav Nikolov, Rishub Jain, Jonas Adler, Trevor Back, Stig Petersen, David Reiman, Ellen Clancy, Michal Zielinski, Martin Steinegger, Michalina Pacholska, Tamas Berghammer, Sebastian Bodenstein, David Silver, Oriol Vinyals, Andrew W. Senior, Koray Kavukcuoglu, Pushmeet Kohli, and Demis Hassabis. 2021. Highly accurate protein structure prediction with AlphaFold. *Nature* (2021). https://doi.org/10.1038/s41586-021-03819-2

[22] Wang-Cheng Kang, Derek Zhiyuan Cheng, Tiansheng Yao, Xinyang Yi, Ting Chen, Lichan Hong, and Ed H. Chi. 2021. Learning to Embed Categorical Features without Embedding Tables for Recommendation. *CoRR* (2021). https://arxiv.org/abs/2010.10784

[23] Liu Ke, Udit Gupta, Benjamin Youngjae Cho, David Brooks, Vikas Chandra, Utku Diril, Amin Firoozshahian, Kim M. Hazelwood, Bill Jia, Hsien-Hsin S. Lee, Meng Li, Bert Maher, Dheevatsa Mudigere, Maxim Naumov, Martin Schatz, Mikhail Smelyanskiy, Xiaodong Wang, Brandon Reagen, Carole-Jean Wu, Mark Hempstead, and Xuan Zhang. 2020. RecNMP: Accelerating Personalized Recommendation with Near-Memory Processing. In *47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Valencia, Spain, May 30 - June 3, 2020*. IEEE, 790–803. https://doi.org/10.1109/ISCA45697.2020.00070

[24] Sameer Kumar, James Bradbury, Cliff Young, Yu Emma Wang, Anselm Levskaya, Blake Hechtman, Dehao Chen, HyoukJoong Lee, Mehmet Deveci, Naveen Kumar, Pankaj Kanwar, Shibo Wang, Skye Wanderman-Milne, Steve Lacy, Tao Wang, Tayo Oguntebi, Yazhou Zu, Yuanzhong Xu, and Andy Swing. 2021. Exploring the limits of Concurrency in ML Training on Google TPUs. https://arxiv.org/abs/2011.03641

[25] Haochen Liu, Xiangyu Zhao, Chong Wang, Xiaobing Liu, and Jiliang Tang. 2020. Automated Embedding Size Search in Deep Recommender Systems. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval* (Virtual Event, China) *(SIGIR '20)*. Association for Computing Machinery, New York, NY, USA, 2307–2316. https://doi.org/10.1145/3397271.3401436

[26] Michael Lui, Yavuz Yetim, Özgür Özkan, Zhuoran Zhao, Shin-Yeh Tsai, Carole-Jean Wu, and Mark Hempstead. 2021. Understanding Capacity-Driven Scale-Out Neural Recommendation Inference. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. https://doi.org/10.1109/ISPASS51385.2021.00033

[27] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2020. Pump Up the Volume: Processing Large Data on GPUs with Fast Interconnects. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1633–1649. https://doi.org/10.1145/3318464.3389705

[28] Kiwan Maeng, Shivam Bharuka, Isabel Gao, Mark Jeffrey, Vikram Saraph, Bor-Yiing Su, Caroline Trippel, Jiyan Yang, Mike Rabbat, Brandon Lucia, and Carole-Jean Wu. 2021. Understanding and Improving Failure Tolerant Training for Deep Learning Recommendation with Partial Recovery. In *Proceedings of Machine Learning and Systems*.

[29] Peter Mattson, Christine Cheng, Gregory Diamos, Cody Coleman, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, David Brooks, Dehao Chen, Debo Dutta, Udit Gupta, Kim Hazelwood, Andy Hock, Xinyuan Huang, Daniel Kang, David Kanter, Naveen Kumar, Jeffery Liao, Deepak Narayanan, Tayo Oguntebi, Gennady Pekhimenko, Lillian Pentecost, Vijay Janapa Reddi, Taylor Robie, Tom St John, Carole-Jean Wu, Lingjie Xu, Cliff Young, and Matei Zaharia. 2020. MLPerf Training Benchmark. In *Proceedings of Machine Learning and Systems*.

[30] Seung Won Min, Vikram Sharma Mailthody, Zaid Qureshi, Jinjun Xiong, Eiman Ebrahimi, and Wen-mei Hwu. 2020. EMOGI: Efficient Memory-Access for out-of-Memory Graph-Traversal in GPUs. *Proc. VLDB Endow.* 14, 2 (Oct. 2020), 114–127. https://doi.org/10.14778/3425879.3425883

[31] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, Liang Luo, Jie Amy Yang, Leon Gao, Dmytro Ivchenko, Aarti Basant, Yuxi Hu, Jiyan Yang, Ehsan K.

Ardestani, Xiaodong Wang, Rakesh Komuravelli, Ching-Hsiang Chu, Serhat Yilmaz, Huayu Li, Jiyuan Qian, Zhuobo Feng, Yinbin Ma, Junjie Yang, Ellie Wen, Hong Li, Lin Yang, Chonglin Sun, Whitney Zhao, Dimitry Melts, Krishna Dhulipala, KR Kishore, Tyler Graf, Assaf Eisenman, Kiran Kumar Matam, Adi Gangidi, Guoqiang Jerry Chen, Manoj Krishnan, Avinash Nayak, Krishnakumar Nair, Bharath Muthiah, Mahmoud khorashadi, Pallab Bhattacharya, Petr Lapukhov, Maxim Naumov, Lin Qiao, Mikhail Smelyanskiy, Bill Jia, and Vijay Rao. 2021. High-performance, Distributed Training of Large-scale Deep Learning Recommendation Models. *CoRR* (2021). https://arxiv.org/abs/2104.05158

[32] Maxim Naumov, John Kim, Dheevatsa Mudigere, Srinivas Sridharan, Xiaodong Wang, Whitney Zhao, Serhat Yilmaz, Changkyu Kim, Hector Yuen, Mustafa Ozdal, Krishnakumar Nair, Isabel Gao, Bor-Yiing Su, Jiyan Yang, and Mikhail Smelyanskiy. 2020. Deep Learning Training in Facebook Data Centers: Design of Scale-up and Scale-out Systems. *CoRR* (2020). https://arxiv.org/abs/2003.09518

[33] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. 2019. Deep Learning Recommendation Model for Personalization and Recommendation Systems. (2019). https://arxiv.org/abs/1906.00091

[34] Yves Raimond. 2018. Deep Learning for Recommender Systems. https://www.slideshare.net/moustaki/deep-learning-for-recommender-systems-86752234.

[35] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, Ramesh Chukka, Cody Coleman, Sam Davis, Pan Deng, Greg Diamos, Jared Duke, Dave Fick, J. Scott Gardner, Itay Hubara, Sachin Idgunji, Thomas B. Jablin, Jeff Jiao, Tom St. John, Pankaj Kanwar, David Lee, Jeffery Liao, Anton Lokhmotov, Francisco Massa, Peng Meng, Paulius Micikevicius, Colin Osborne, Gennady Pekhimenko, Arun Tejusve Raghunath Rajan, Dilip Sequeira, Ashish Sirasao, Fei Sun, Hanlin Tang, Michael Thomson, Frank Wei, Ephrem Wu, Lingjie Xu, Koichi Yamada, Bing Yu, George Yuan, Aaron Zhong, Peizhao Zhang, and Yuchen Zhou. 2020. MLPerf Inference Benchmark. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. https://doi.org/10.1109/ISCA45697.2020.00045

[36] Danny Sullivan. 2016. Google uses RankBrain for every search, impacts rankings of "lots" of them. https://searchengineland.com/google-loves-rankbrain-uses-for-every-search-252526.

[37] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, undefinedukasz Kaiser, and Illia Polosukhin. 2017. Attention is All You Need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 6000–6010. https://doi.org/10.5555/3295222.3295349

[38] Yu Emma Wang, Carole-Jean Wu, Xiaodong Wang, Kim Hazelwood, and David Brooks. 2021. Exploiting Parallelism Opportunities with Deep Learning Frameworks. *ACM Transactions on Architecture and Code Optimization* 18, 1 (2021). https://doi.org/10.1145/3431388

[39] Kilian Weinberger, Anirban Dasgupta, John Langford, Alex Smola, and Josh Attenberg. 2009. Feature Hashing for Large Scale Multitask Learning. In *Proceedings of the 26th Annual International Conference on Machine Learning* (Montreal, Quebec, Canada) *(ICML '09)*. Association for Computing Machinery, New York, NY, USA, 1113–1120. https://doi.org/10.1145/1553374.1553516

[40] Jonathan A. Weyn, Dale R. Durran, and Rich Caruana. 2020. Improving Data-Driven Global Weather Prediction Using Deep Convolutional Neural Networks on a Cubed Sphere. *Journal of Advances in Modeling Earth Systems* 12, 9 (Sep 2020). https://doi.org/10.1029/2020ms002109

[41] Mark Wilkening, Udit Gupta, Samuel Hsia, Caroline Trippel, Carole-Jean Wu, David Brooks, and Gu-Yeon Wei. 2021. RecSSD: Near Data Processing for Solid State Drive Based Recommendation Inference. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. https://doi.org/10.1145/3445814.3446763

[42] Carole-Jean Wu, Robin Burke, Ed Chi, Joseph A. Konstan, Julian J. McAuley, Yves Raimond, and Hao Zhang. 2020. Developing a Recommendation Benchmark for MLPerf Training and Inference. *CoRR* abs/2003.07336 (2020). https://arxiv.org/abs/2003.07336

[43] Carole-Jean Wu, Ramya Raghavendra, Udit Gupta, Bilge Acun, Newsha Ardalani, Kiwan Maeng, Gloria Chang, Fiona Aga Behram, James Huang, Charles Bai, Michael Gschwind, Anurag Gupta, Myle Ott, Anastasia Melnikov, Salvatore Candido, David Brooks, Geeta Chauhan, Benjamin Lee, Hsien-Hsin S. Lee, Bugra Akyildiz, Maximilian Balandat, Joe Spisak, Ravi Jain, Mike Rabbat, and Kim Hazelwood. 2021. Sustainable AI: Environmental Implications, Challenges and Opportunities. *CoRR* abs/2111.00364 (2021). https://arxiv.org/abs/2111.00364

[44] Chunxing Yin, Bilge Acun, Xing Liu, and Carole-Jean Wu. 2021. TT-Rec: Tensor Train Compression for Deep Learning Recommendation Models. *CoRR* abs/2101.11714 (2021). https://arxiv.org/abs/2101.11714

[45] Caojin Zhang, Yicun Liu, Yuanpu Xie, Sofia Ira Ktena, Alykhan Tejani, Akshay Gupta, Pranay Kumar Myana, Deepak Dilipkumar, Suvadip Paul, Ikuhiro Ihara, Prasang Upadhyaya, Ferenc Huszar, and Wenzhe Shi. 2020. Model Size Reduction Using Frequency Based Double Hashing for Recommender Systems. In *Fourteenth ACM Conference on Recommender Systems* (Virtual Event, Brazil) *(RecSys '20)*. Association for Computing Machinery, New York, NY, USA, 521–526. https://doi.org/10.1145/3383313.3412227

[46] 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. In *Proceedings of Machine Learning and Systems*.

[47] Zhe Zhao, Lichan Hong, Li Wei, Jilin Chen, Aniruddh Nath, Shawn Andrews, Aditee Kumthekar, Maheswaran Sathiamoorthy, Xinyang Yi, and Ed Chi. 2019. Recommending What Video to Watch next: A Multitask Ranking System. In *Proceedings of the 13th ACM Conference on Recommender Systems* (Copenhagen, Denmark) *(RecSys '19)*. Association for Computing Machinery, New York, NY, USA, 43–51. https://doi.org/10.1145/3298689.3346997

[48] Guorui Zhou, Na Mou, Ying Fan, Qi Pi, Weijie Bian, Chang Zhou, Xiaoqiang Zhu, and Kun Gai. 2019. Deep interest evolution network for click-through rate prediction. In *AAAI conference on artificial intelligence*, Vol. 33. 5941–5948. https://doi.org/10.1145/3219819.3219823