cDLRM: Look Ahead Caching for Scalable Training of Recommendation Models

Keshav Balasubramanian
keshavba@usc.edu
University of Southern California
USA

Abdulla Alshabanah
aalshaba@usc.edu
University of Southern California
USA

Joshua Choe
choejd@usc.edu
University of Southern California
USA

Murali Annavaram
annavara@usc.edu
University of Southern California
USA

ABSTRACT

Deep learning recommendation models (DLRMs) are typically composed of two sets of parameters: large embedding tables to handle sparse categorical inputs, and neural networks such as multi-layer perceptrons (MLPs) to handle dense non-categorical inputs. Current DLRM training practices keep both these parameters in GPU memory. But as the size of the embedding tables grow, this practice of storing model parameters in GPU memory requires dozens or even hundreds of GPUs. This is an unsustainable trend with severe environmental consequences. Furthermore, such a design forces only a few conglomerates to be the gate keepers of model training. In this work, we propose cDLRM which democratizes recommendation model training by allowing a user to train on a single GPU regardless of the size of embedding tables by storing all embedding tables in CPU memory. A CPU based pre-processor analyzes training batches to prefetch embedding table slices accessed by those batches and caches them in GPU memory just-in-time. An associated caching protocol on the GPU enables efficiently updating the cached embedding table parameters. cDLRM decouples the embedding table size demands from the number of GPUs needed for compute. We first demonstrate that with cDLRM it is possible to train a large recommendation model using a single GPU regardless of model size. We then demonstrate that with its unique caching strategy, cDLRM enables pure data parallel training. We use two publicly available datasets to show that a cDLRM achieves identical model accuracy compared to a baseline trained completely on GPUs, while benefiting from large reduction in GPU demand.

CCS CONCEPTS

• **Information systems** → **Recommender systems**.

KEYWORDS

Recommendation models, efficient training, distributed data parallel training, caching, prefetching

RecSys '21, September 27-October 1, 2021, Amsterdam, Netherlands

ACM ISBN 978-1-4503-8458-2/21/09.
https://doi.org/10.1145/3460231.3474246

ACM Reference Format:
Keshav Balasubramanian, Abdulla Alshabanah, Joshua Choe, and Murali Annavaram. 2021. cDLRM: Look Ahead Caching for Scalable Training of Recommendation Models. In *Fifteenth ACM Conference on Recommender Systems (RecSys '21), September 27-October 1, 2021, Amsterdam, Netherlands.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3460231.3474246

1 INTRODUCTION

Recommendation systems have become an integral tool for driving user engagement in many technology platforms. From personalized news recommendation [8] to job candidate recommendation [5], these systems play an important role in improving user experience and maximizing platform viability. A variety of different machine learning techniques have been successfully used in constructing recommendation systems. These include statistical learning techniques such as Bayesian modelling [4, 9], kernel methods [16], and matrix-factorization based methods [15]. The success of deep learning frameworks such as deep neural networks has created new avenues to improve the accuracy of recommendation models [2, 3].

Deep learning based recommendation models are significantly larger than traditional DNNs [11]. For instance, the open sourced Facebook recommendation model (DLRM) [12] has a model size that can exceed several hundreds of GB, which we detail in the next section. Much of the DLRM parameters constitute dozens of embedding tables that are learned using terabytes of training data. Since the embedding tables are extremely large, accesses to these tables are extremely sparse. Out of the millions of table entries, only a few hundred to a few thousand entries are updated on each training batch. While DLRM could be trained on a CPU, it also has a high computational demand in the fully connected dense layers where substantial parallelism exists. To exploit the computation parallelism current recommendation model training practices rely on GPUs to train the models. But to train the model on a GPU requires storing the large embedding tables across multiple GPUs. Our preliminary analysis showed that recommendation model training exploits the memory distributed across GPUs but is rarely able to fully utilize the available GPU hardware parallelism. In particular, for large models we observe that there is a point beyond which multiple GPUs are used merely for their memory, even though the computational load of training does not require these GPUs. On the other extreme, if the training is entirely done on CPUs, where there is substantially more and relatively cheap memory,

the performance also suffers due to the lack of sufficient parallel hardware.

Inspired by these observations, in this work we present cDLRM, a novel approach to decouple the memory demands of recommendation models from the computational demands. cDLRM places the embedding tables on the CPU while allowing the training process to run on a GPU. However, instead of slowing down the training to access the CPU-resident embedding tables, cDLRM proactively analyzes training batches ahead of time to precisely identify the necessary subset of embedding table entries that are needed for a batch. Based on this analysis, it caches the embedding table entries that are necessary to train an upcoming batch into the GPU memory just-in-time before the start of that training batch, thereby avoiding the need to access embedding tables on the CPU. cDLRM relies on CPU threads to identify the embedding entries for caching while enabling training of large recommendation models using even just a single GPU. By enabling training on a single GPU our approach makes recommendation model training affordable to even small businesses. We then show how cDLRM can scale the training speed with the availability of additional GPUs by supporting data parallel training. By leveraging model caching and data parallel training we provide a resource efficient solution that gracefully scales performance and cost.

The primary contributions of this work are as follows:
(1) We propose a system, that we call *cDLRM* in which all embedding tables are kept in CPU DRAM while only a small cache of each table is stored in GPU memory. cDLRM is built on the concept of *lookahead caching* where a CPU thread pre-processes training batches and caches the necessary embedding table entries to GPU memory. Training is contained entirely on the GPUs without ever letting gradients flow back to the CPU. cDLRM is based on the insight that only a subset of embedding table rows are required to train for a set of samples.
(2) We demonstrate the effectiveness of cDLRM by training large models on just a single GPU and also demonstrate how cDLRM can scale out in a purely data parallel manner when additional GPUs are available. To the best of our knowledge this is the first work to demonstrate distributed data parallel training of large recommendation models using table embedding caching as the foundation.
(3) We quantify the impact of cDLRM by training on 2 publicly available datasets and demonstrate that with little ($< 0.02\%$) to no loss in accuracy, we can train the DLRM in a highly cost efficient manner.

2 BACKGROUND AND MOTIVATION

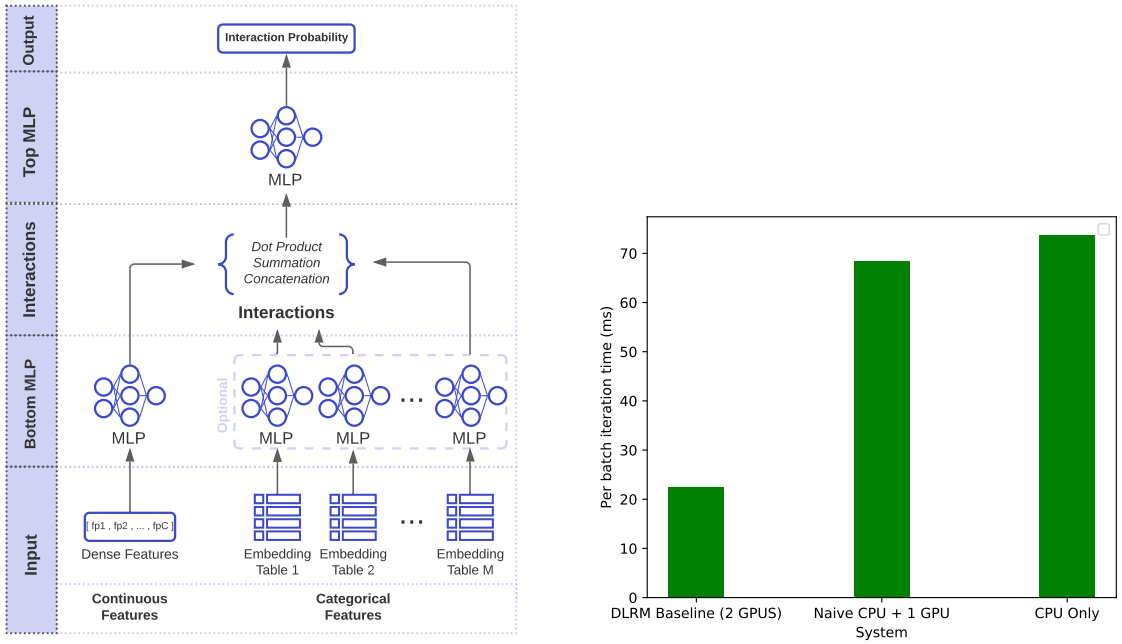
Deep Learning Recommendation Model Architecture: Our work builds on the DLRM model proposed in [12], which is a recommendation system deployed at Facebook. The DLRM model used in this study is based on the architecture shown in Figure 1a. The model learns on data that is composed of continuous and categorical features. Typically, the continuous features represent user information, such as age, gender, and the categorical features represent interaction object information. Interaction objects are usually links that a user can click on. For example, they could be links to product ads, news feeds etc.,. The user and interaction object relations are learned using embedding tables that map them to dense representations, and continuous features are learned using through Multi-layer Perceptrons (MLPs).

Training large models: Training the DLRM is both a memory-intensive task due to large embedding tables, and a compute-intensive task due to MLPs. While both the MLPs and the embedding tables make up the model's parameter set, the total memory footprint of the MLPs is much smaller than that of the embedding tables. The total memory footprint of the embedding tables can be hundreds of gigabytes or even terabytes, while MLPs only consume hundreds of megabytes. Current training systems store all model parameters in GPU memory. Since the embedding tables can exceed even the largest GPU memory size, typically tens or even hundreds of GPUs are employed to distribute the model parameters [11]. Thus, recommendation models like DLRM are implemented using a partial data-parallel and a partial model-parallel hybrid system, in which data parallelism is used to improve the performance of the MLPs and model parallelism is used to accommodate the memory requirements of the embedding tables. To achieve data parallelism, the MLPs are replicated on all the GPUs to be used in the training, whereas to achieve model parallelism the embedding tables are split between the GPUs using different allocation strategies. For instance, DLRM [12] uses round robin allocation, where embedding table are allocated to different GPUs in a round-robin manner.

Drawbacks of the current system: The number of GPUs needed for training is determined primarily by the total memory needed to store the embedding tables. The main drawback of this system is that when the embedding tables become larger, the number of GPUs that are needed to accommodate the embedding tables increases. GPUs with large integrated memory are extremely expensive compared to the much cheaper discreet DRAM DIMMs on CPUs. Hence, increasing GPU count to accommodate large models will substantially reduce the ability of small businesses to train large models. Even on the computing front there is a point, which we call the *compute saturation point*, beyond which increasing the number of GPUs used no longer provides any performance benefit during the data parallel computation of the MLPs. Compute saturation is reached when it is no longer feasibly to fully utilize available parallel GPU resources due to limited available parallelism in the computation. The result is that this system scales poorly in terms of the number of GPUs used, only using them for model storage beyond a point. Hence this system can require very expensive compute infrastructure to be able to train the model.

2.1 Basic approaches to addressing memory footprint and their shortcomings

CPU-only training - The simplest approach is to train the entire model on the CPU. The benefits of CPU-only training are that CPU DRAM is much cheaper and all communication overhead is eliminated. The downside to this approach is that computation now becomes the bottleneck since the MLPs and second order interactions are much slower on the CPU than they are on GPUs. It is also possible to distribute training on multiple CPU sockets



(a) (b)

Figure 1: (a) DLRM Model Architecture (b) Performance comparison of simple approaches to reduce memory

but they suffer potentially more communication bottleneck since inter-CPU bandwidth is typically less than inter-GPU bandwidth.

Naive CPU + GPU training - A strategy that improves upon CPU-only training is to use GPUs for MLP and second order interaction computation while still keeping all embedding tables in CPU DRAM. This involves frequent communication between the CPU and the GPU, with two major communication points - (1) during forward propagation, embeddings are fetched from the CPU and transferred to the GPU(s) and (2) during back propagation, gradients flow back into the embedding tables on the CPU. While this system speeds up MLP computations and second order interactions by performing them on the GPU, CPU to GPU data transfer is an expensive overhead. Thus, the major drawback here is the frequency of data transfer between the CPU and GPU. Figure 1b shows the comparison between training using 2 GPUs with the default DLRM based hybrid parallel training (first bar), Naive CPU + GPU training (second bar), CPU-only training (third bar). There is nearly 3X training delay when using CPU only or a naive CPU+GPU training approaches.

Hashing - Another approach that seeks to keep all the embedding tables on the GPU while reducing the amount of memory required by each table is based on hashing. This is a prevalent model size reduction strategy. The idea is to reduce the number of rows in each embedding table and map all the indices into the reduced table using a hash function. Naturally, this leads to entanglement of embeddings and our experiments show that the final model accuracy can decrease substantially as a result. This phenomenon has also been observed in prior work [17]. This solution is often the least used in industry since even a **0.1%** decrease in accuracy can lead to a significant reduction in revenue [19].

3 CDLRM PRELIMINARIES, TERMINOLOGY AND NOTATION

3.1 Caching Preliminaries

In computer systems, caching is used to reduce the latency of accessing main memory. Caches are small amounts of on-chip memory that is used for storing data that is frequently accessed. A copy of this frequently accessed data is fetched from main memory and is kept resident in the cache to speed up future accesses to the same data. Caches by design are much smaller in capacity than that of main memory, and hence the cache management system has to decide what parts of the main memory data to store in the cache and which data to remove from the cache.

Modern processors use a set-associative cache. Set-associative caches are organized into bins, which are called *sets*. A memory address is first mapped to a given set. For instance, with 1K sets in a cache a 64-bit memory address is hashed to generate a 10-bit set index. Each set is composed of multiple *ways*. Ways allows multiple memory locations to be cached in the same set. Thus, after a memory location is mapped to a particular cache set, it is cached in a way in the selected set based on a way-selection algorithm. Since the memory is much larger than main memory the hashing process may map many memory locations to the same set. When a set is fully occupied a way replacement may find an existing cache way and replace that with the new data. Some example selection

and replacement heuristics are the first available way, random way, or least recently used way.

Since multiple memory locations can map to the same set there is a need to identify which memory locations are currently stored in that set. For this purpose caches use a structure called a cache tags. To access the cache, each memory location is first hashed to find the set index. Then the cache access logic searches the cache tags of all the ways in that set to determine if that memory location is in fact present in that set. If it is present it is called a cache hit, otherwise it is treated as a miss and that location is fetched from its original memory location.

3.2 Embedding table cache structure

To reduce the overall memory footprint of the embedding tables in the GPU, cDLRM uses an embedding table cache structure that occupies a much smaller memory footprint in GPU DRAM. The embedding table cache structure in the GPU is organized much like a CPU cache described above. For a recommendation model with embedding table set $\mathbf{E} = \{e_1, ..., e_m\}$, in which table $e_i \in \mathcal{R}^{r_i \times d}$, where r_i is the number of rows in table e_i, and d is the dimensionality of the embeddings, the embedding cache is the set of smaller embedding tables $\mathbf{C} = \{c_1, ..., c_m\}$, $c_i \in \mathcal{R}^{s_i \times w \times d}$. Here s_i is the number of *sets* and w the number of *ways* in cache c_i, in line with standard terminology from memory caches. The number of sets is chosen based on a hyperparameter L. If $r_i \leq L$ then the entire table is stored on GPU, i.e. $s_i = r_i$, otherwise we use a simple hashing function to map the embedding table entry to a cache set as follows: $s_i = NextPrime(L)$ [1]. Note that the reason for using the NextPrime hashing function is to reduce potential collisions when two embedding table entries may map to the same set. The number of ways w is also a hyperparameter that is empirically selected. $\mathbf{E}$ resides in CPU DRAM, while $\mathbf{C}$ resides in GPU DRAM.

3.3 Lookahead Windows

A *lookahead window* of size n is a set of n consecutive batches that appear in the training set. For training set D_{train} and batch size b, the number of lookahead windows of size n that D_{train} can be split into is $\lceil \frac{|D_{train}|}{bn} \rceil$. Each lookahead window is made up of bn training examples. We refer to this as the *span* of the lookahead window. The span is the upperbound on the number of unique indices that are being looked up in any table in any given lookahead window, and is much smaller than the number of rows in the table. This observation is especially true when embedding tables are large, which is the case for many of the production recommendation models. In the training datasets that we have studied, we find in practice that the number of unique indices accessed in a lookahead window tends to be even smaller than the span.

Based on this observation, we conclude that maintaining an entire table on the GPU, when only a part of it is being accessed and updated over the next n batches, is wasteful. If we can find a way to fetch the embeddings for the indices that are going to be looked up over the next n batches from $\mathbf{E}$ and move these embeddings into $\mathbf{C}$ just-in-time as training on the next n batches starts, we can significantly reduce the memory usage on the GPU. The prefetching, caching+training, and eviction processes, described next, work in unison to achieve this goal and enable training of a recommendation model with minimal overhead.

[1] $NextPrime(x)$ computes the smallest prime number $\geq x$ efficiently by leveraging Bertrand's postulate.

4 CDLRM OVERVIEW

The block diagram in figure 2 illustrates the different components of the cDLRM working together. cDLRM consists of three processes: prefetching, caching+training, and eviction. Each process relies on a set of queues to move embeddings between the CPU and GPU. At the start of training, the prefetching process reads a set of n training batches, where n is the size of the lookahead window. The selection of this lookahead window parameter is based on multiple factors as will be discussed. The goal of the prefetching process is to extract all the unique indices accessed in the lookahead window for each table. It then fetches the embeddings associated with these unique indices and pushes them into a queue labeled $Q_{lookahead}$. Concurrently, the caching+training process reads the embedding entries stored in the $Q_{lookahead}$ queue and uses a hash algorithm to place each of these embeddings into the table caches resident in GPU DRAM. This process handles the complexities associated with cache management. After placing the entries from $Q_{lookahead}$ into the table caches, the caching+training process performs forward and backward propagation. This training is completely contained in the GPU, with gradients never flowing back into the primary embedding tables on the CPU. The last process, the eviction process, is responsible for writing the updated embedding vectors back to the CPU embedding tables.

The prefetching process must stay ahead of the caching+training process so that the caching+training process is not stalled waiting for embeddings. The selection of the lookahead parameter is primarily determined by this consideration. Since the speed of the prefetching and caching+training processes can vary based on the underlying hardware configuration, we use a profiling tool to measure the training speed for a given model and batch size on the selected GPU configuration. We also measure the speed of the prefetching process in identifying and extracting unique indices of a single batch from the CPU-resident embedding tables. The ratio of the training speed over the prefetching guides the choice of lookahead window size. If these profiled measurements change dynamically due to bandwidth and other runtime bottlenecks, training may occasionally stall. If such a stall occurs, our approach can be adapted to dynamically increase the lookahead window. In our experiments we did not have to change the lookahead window since the speed ratios are quite stable.

5 TRAINING CDLRM ON A SINGLE GPU

A unique feature of cDLRM is that is that it only ever needs a single GPU to accommodate the model. Since only caches of the embedding tables are stored in GPU memory, we can always choose a cache size to fit within the limits of a single GPU's memory regardless of the actual size of the whole embedding tables. We first describe the details of the various processes and how they cooperate to achieve training cDLRM on a single GPU before presenting how cDLRM scales out to multiple GPUs in a purely data parallel fashion.

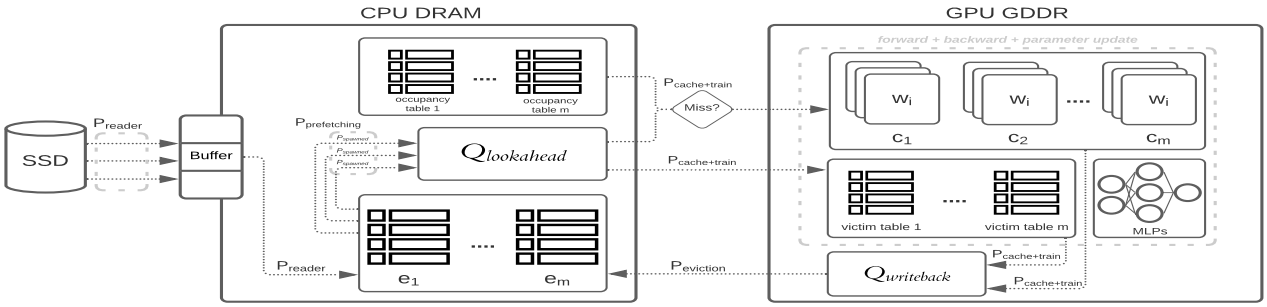


Figure 2: cDLRM Block Diagram

Algorithm 1 Prefetching

Input: training set: $\mathbf{D_{train}}$, lookahead window size: **n**, training batch size: **b**, embedding tables: $\mathbf{E} = \{e_1, ..., e_m\}$ shared lookahead window queue: $\mathbf{Q_{lookahead}}$

1: **while** $LW \leftarrow NextLookaheadWindow(D_{train}, b, n)$ **do**
2: $unique\_embeds \leftarrow []$
3: **for** $i \leftarrow 1$ **to** m **do**
4: $U_i \leftarrow ComputeUniqueIndices(LW, i)$
5: $P_i \leftarrow e_i[U_i]$
6: $unique\_embeds$.append(P_i)
7: **end for**
8: $Q_{lookahead}$.push($unique\_idx\_embeds$)
9: **end while**

5.1 Prefetching Process

The fundamental role of the prefetching process is to identify the embedding vectors needed to train on the next window of batches and feed these embeddings to the caching+training process. It does so by pre-processing a set of training batches that are read from a training dataset. The training dataset may be present in a remote storage, and a separate reader process may fetch these batches into a buffer and forward them to the prefetching process (figure 2 shows the buffering of the reader process). For every embedding table, the prefetching process computes the unique indices in a lookahead window of batches and fetches the embeddings corresponding to these unique indices from **E**. The fetched embeddings are buffered in $Q_{lookahead}$, which the caching+training process will pull from. Algorithm 1 outlines this procedure. In practice, the outer while loop is parallelized using a number of processes that are spawned from a process pool by the prefetching process. Each of the processes spawned from the pool is responsible for executing the body of the loop on a single lookahead window. This ensures that the caching+training process that consumes the entries from $Q_{lookahead}$, is never stalled as a result of a prefetching bottleneck. Popular machine learning frameworks such as PyTorch [13] and TensorFlow [1] use a similar approach in their dataloaders.

Note that it is possible that some embedding vectors corresponding to a subset of unique indices may already have been transferred to GPU cache in a prior training window. But the prefetching process does not try to prevent loading such vectors into the $Q_{lookahead}$ queue. Instead, we leave this responsibility to the caching+training process to provide a consistent view as we describe next.

5.2 Caching+Training Process

The computation performed by the caching+training process is split up into three phases [2]: (1) preloading / caching, (2) forward propagation and (3) backpropagation and parameter update.

5.2.1 Preloading. In the preloading phase, the process pulls the embeddings corresponding to the unique indices in the next n batches from the $Q_{lookahead}$ queue and caches them in **C**. The set index for original index j in table e_i is computed using the modulo operator (Specifically $j_{set} = j \% s_i$).

Handling coherence. Before placing an embedding vector into GPU cache c_i, the caching+training process must make sure that this index was not already cached in a prior caching step. If the index is already in c_i, then the prefetched embedding vector from CPU DRAM is stale. Rather than search c_i for a cache hit, which takes up GDDR bandwidth, we maintain a separate data structure on the CPU called the *occupancy table*. There are m occupancy tables, each one corresponding to one c_i. An occupancy table stores which embedding table entries are currently resident in GPU. As such, the occupancy table is essentially a tag structure for the embedding table caches and is stored in CPU DRAM to enable the CPU to manage the caching process.

To find out if the embedding corresponding to index j in table e_i is resident in c_i, the caching+Training process needs to compute the set index of j and lookup the occupancy table for c_i. Thus, the set of unique indices for table e_i that has been pulled from $Q_{lookahead}$ can be split up into two disjoint subsets: J^i_{hit}, the set of indices whose embeddings are already in c_i, and J^i_{miss}, the set of indices whose embeddings need to be moved into c_i. Any set+way that stores an index $\in J^i_{hit}$ is considered a pinned entry so it will not be evicted until it is used by the training pipeline.

Caching missing indices. The second step is to cache embeddings for indices in J^i_{miss}. For each $j \in J^i_{miss}$ we compute its set index S as described above. The way within the set S into which its embedding will be moved is chosen uniformly at random from

[2]For simplicity, we describe the phases by focusing on a single embedding table e_i. The description generalizes for all embedding tables.

Algorithm 2 Single GPU Training

Input: training set: $\mathbf{D_{train}}$, lookahead window size: **n**, training batch size: **b**, embedding table caches: $\mathbf{C} = \{c_1, ..., c_m\}$, victim tables $\mathbf{V} = \{v_1, ..., v_m\}$, DLRM MLPs: M^{bot}, M^{top}, shared lookahead window queue: $\mathbf{Q_{lookahead}}$, eviction queue: $\mathbf{Q_{writeback}}$, number of epochs: **num_epochs**,

for *epochs* ← 1 **to** *num_epochs* **do**
  while $b_k \leftarrow NextBatch(D_{train}, b)$ **do**
    if k % n == 0 **then**
      $unique\_embeds \leftarrow Q_{lookahead}.pull()$
      $CacheEmbeds(C, unique\_embeds, Q_{writeback})$
    end if
    $Forward(M^{bot}, M^{top}, C, V, b_k)$
    $Backward(M^{bot}, M^{top}, C, V)$
    $Update(M^{bot}, M^{top}, C, V)$
    $FlushVictimTables(V, Q_{writeback})$
  end while
end for

any *unpinned* entries in that set. The reason for choosing ways at random is performance. Our implementation is heavily parallelized. If we were to choose an alternate policy such as the next available way, the caching procedure would have to be serial, making the entire process slower. Thus, random way selection allows multiple indices to randomly and concurrently pick an available way in a set. Multiple original indices that map to the same set could end up choosing the same way to cache their embedding in. By using different random seeds one can minimize such overlap. But in the worst case, to resolve a conflict when two indices map to the same index within the set, we once again choose one of the colliding indices at random to occupy the way. The embeddings corresponding to all the other colliding indices are considered *conflict victims* of the caching procedure. These conflict victims are unable to stay within a given set and are handle differently.

In practice, such a collision is extremely rare. This is because two unique indices from J^i_{miss} can only have a potential collision if they map to the same set. We evaluated different policies on handling conflict victims, but given the rarity of this scenario we simply do not place a conflict victim into the GPU cache. Instead, during training we re-fetch the missing embedding vector (conflict victim) from the CPU directly into a victim buffer as we describe in section 5.2.2.

Evicted Embeddings. Any way in a set that is evicted to make room for embeddings of indices in J^i_{miss} needs to be written back to the CPU to update e_i. The caching+training process works with the eviction process (section 5.3) to orchestrate this writeback. The caching+training process simply pushes the evicted embeddings into an eviction queue, $Q_{writeback}$, from which the eviction process will pull the evicted embeddings and write them to e_i.

5.2.2 Forward Propagation. The phase that follows the caching phase is the forward propagation phase. In this phase, the caching+training process performs a forward propagation on a single batch of training examples. The continuous features in the batch are transformed through the MLPs, while the categorical feature vector consists of indices for which embeddings need to be obtained.

Indices that hit in $\mathbf{c_i}$. After the caching phase, the embeddings for most indices needed over the current lookahead window of n batches will be resident in c_i. The reason we cannot guarantee that the embedding for every index in every batch in the lookahead window will be in c_i is because some of them might be conflict victims as explained in section 5.2.1. The hitting indices can simply lookup their embeddings in c_i by computing the set and the way.
Indices that miss in $\mathbf{c_i}$. While indices that miss in a cache are rare, we still require the conflict victim embeddings on the GPU to enable GPU-contained training. To address this issue, we maintain a *victim table* $v_i \in \mathcal{R}^{b \times d}$, similar to a victim cache [7], in GPU DRAM, into which the conflict victim embeddings will be cached on demand during forward propagation. This involves a CPU to GPU data copy step that adds overhead. An important note about the size of the victim table is that the number of rows need only be as large as b, the size of the batch. This is because in a batch of size b, there can only be at most b indices that miss in c_i. In practice we find that hit rates in the primary cache are very high and that we rarely use more than a few rows in the victim table to handle conflicts. Once the conflict victims have been placed into the victim table, all of the required embeddings for the current batch are in GPU memory, and the GPU-contained forward propagation can proceed.

5.2.3 Backpropagation and Parameter Update. Backpropagation, gradient computation, and parameter updates happen entirely in GPU memory. Embeddings of hitting indices are updated in c_i, and embeddings of conflict victims are updated in v_i. All MLPs are updated in GPU memory as well. After updating all the parameters, the conflict victims that are buffered in v_i are explicitly flushed into the eviction queue while the cache resident embeddings are left as is where they may be evicted later. Algorithm 2 outlines the entire training procedure.

5.3 Eviction Process

The last of the three processes is the eviction process. Its role is simply to pull embeddings that have been pushed into the eviction queue by the training process and write them back to the original embedding tables in CPU DRAM, namely, **E**. It is computationally the least intensive process of the three.

Several of the cDLRM design choices were made in favor of high performance rather than strictly enforcing an identical behavior of a baseline uncached system, that performs forward passes on embeddings that have been updated *only after* the previous backward pass. For example, consider the following illustrative scenario: let j be a unique index that is needed during lookahead window w and $w + 2$, but not in $w + 1$. If the prefetching process is prefetching data for window $w + 2$ it may fetch embedding vector for j from the CPU and place it in $Q_{lookahead}$. But during window $w + 1$, j may be evicted and pushed back to CPU. Finally during window $w + 2$ the caching+training process may simply read the data from $Q_{lookahead}$, which may be a stale embedding. While such cases occur rarely, it is important to note them to understand their impact on accuracy. The only way to avoid this race condition is to use locks, which would make the whole system slower. Hence, for performance reasons, we opt to allow rare stale embeddings. We

Algorithm 3 Multi GPU Training (process perspective)

Input: training set: $\mathbf{D_{train}}$, lookahead window size: **n**, training batch size: **b**, embedding table caches: $\mathbf{C^r} = \{c_1^r, ..., c_m^r\}$, victim tables $\mathbf{V^r} = \{v_1^r, ..., v_m^r\}$, DLRM MLPs: $\mathbf{M_r^{bot}}, \mathbf{M_r^{top}}$, lookahead window queue: $\mathbf{Q_{lookahead}}$, eviction queue: $\mathbf{Q_{writeback}}$, number of epochs: **num_epochs**, process rank: **r**, communication world: **W**, cache aggregate granularity: λ

for *epochs* ← 1 **to** *num_epochs* **do**
  while $b_k^r \leftarrow NextBatch(D_{train}, b)$ **do**
    if k % n == 0 **and** r == 0 **then**
      $unique\_embeds \leftarrow Q_{lookahead}.pull()$
      $CacheEmbeds(C^0, unique\_embeds, Q_{writeback})$
      $BroadcastCacheState(C^0, W)$
    end if
    $Forward(M_r^{bot}, M_r^{top}, C^r, V^r, b_k^r)$
    $Backward(M_r^{bot}, M_r^{top}, C^r, V^r)$
    $AggregateMLPs(M_r^{bot}, M_r^{top}, W)$
    $Update(M_r^{bot}, M_r^{top}, C^r, V^r)$
    if k % λ == 0 **then**
      $AggregateCaches(C^r, W)$
    end if
    $FlushVictimTables(V^r, Q_{writeback})$
  end while
end for

show empirically that the allowance of this staleness does not hurt model accuracy.

6 DATA PARALLEL TRAINING WITH CACHING

While cDLRM enables the training of DLRM on a single GPU regardless of embedding table sizes, when multiple GPUs are available it is possible to improve the training speed by using data parallelism in MLPs to speed up the overall computation. In such cases, cDLRM can easily scale out to multiple GPUs in a purely data parallel fashion by replicating the embedding table caches as well as MLPs on all participating GPUs. While prior approaches used model replication for data parallel training, our approach is unique in that we cache the necessary model parameters across multiple GPUs thereby emulating data parallelism on top of model caching.

6.1 Maintaining Cache Coherency in Data Parallel Training

When training in a data parallel fashion with replicated caches on all GPUs, cache coherency needs to maintained. Such a situation occurs when two GPUs may need to access the same embedding table index in different training samples. cDLRM enforces coherency at two places in the training pipeline: at the beginning of a lookahead window when caches are loaded with the contents of the upcoming batches; and after each GPU individually updates its replicas of the embedding tables and MLPs.

Coherency at the beginning of a lookahead window: Recall from section 5.2.1 that caches are preloaded/warmed up at the beginning of every lookahead window. In the case of single GPU training, the cached table entries reside on the singular GPU used to train and hence there are no consistency issues across different caches. However, when training in a data parallel fashion on multiple GPUs, the caches on all GPUs need to have the same state after caching. We facilitate this through a broadcast of cache state. More specifically, the process with rank 0 is the only process that interfaces with the prefetching process. It caches the embeddings for the upcoming lookahead window on GPU0 as described in section 5.2.1 and broadcasts the cache state to all other GPUs over a high bandwidth NVLink interconnect. For aggregation simplicity the broadcast process replicates the cache across all GPUs, even though some of the embedding table entries may only be needed on a single GPU. Through this broadcast we guarantee that all GPUs see a consistent copy of any shared embedding entry before they start the next training batch.

Coherency after individual parameter updates: Since each process involved in data parallelism is training on their own batches, the individual updates to the parameters (both MLPs and embedding tables) in each process will once again lead to a lack of consistency between the parameters on each GPU. Note that this inconsistency is similar to traditional data parallel training where each GPU may have its own model updates locally first. These updates must be aggregated for a global model. cDLRM employs the following practices to synchronize parameters:

MLP aggregation. MLPs are synchronized by the standard practice of gradient averaging. At the end of every minibatch, the gradients for the MLPs across all GPUs are averaged via an all-reduce. The resulting gradient is used to update parameters on all GPUs.

Embedding Table Cache aggregation. Synchronizing embedding table caches in a similar manner is infeasible. Despite being much smaller than whole embedding tables, accesses to the caches in a per batch granularity is still sparse, thereby making the gradients sparse. Performing an all-reduce on sparse gradients can only be accomplished using at least two communication calls: one to aggregate non-zero elements first and another to aggregate values. This is an artifact of the current PyTorch implementation. To avoid this overhead, we choose to average the caches post parameter update via a single all-reduce. Recall that GPU0 broadcasts the entire cache to all GPUs, even though only some of these embedding indices are actually shared. But by sharing the entire cache the aggregation process is simplified. Gradients of shared embedding table entries will be produced by multiple GPUs and these are aggregated to create a single weight update for all the shared entries. On the other hand, if an embedding table entry is only used on a single GPU only that GPU produces the update for that model parameter and all other GPUs produce a zero gradient update. Hence, our approach lets each GPU first update its own cache with the gradients it computed. Once the local caches are updated these caches are then averaged to create a global model update.

Interestingly, since only a few cached entries are truly shared across GPUs the sparse model update of each GPU is sufficient to update its local cache. Hence, averaging the caches across multiple GPUs can be done much less frequently than the MLPs. cDLRM thus tracks the fraction of sparse updates across caches and waits until the fraction of cache updates across all GPUs exceeds a threshold. The granularity at which we average caches is a hyperparameter of the optimization process. Algorithm 3 illustrates how Algorithm 2 changes in the multi-gpu setting from the perspective of each

process. *AggregateMLPs* and *AggregateCaches* are collective communication calls in which all ranks participate.

7 EXPERIMENTAL EVALUATION

We implemented cDLRM on top of Facebook's open source DLRM [12] using PyTorch for cache management.

Datasets: We use two publicly available datasets to evaluate cDLRM relative to the baseline open source DLRM: the Criteo AI Labs Ad Kaggle and Terabyte datasets. Both datasets consist of click logs for ad click-through-rate prediction. To our knowledge, these are the only publicly available datasets in this domain, a sentiment reiterated by [12]. The dataset details are outlined in Table 1(a). Each datapoint in both datasets contains 13 dense features (C) and 26 sparse features (M). The model architecture used to train on each dataset is also listed in the table. All performance results reported are on the Terabyte dataset.

Experimental Setup: All our experiments are run on a server machine with 4 Intel(R) Xeon(R) Gold 5220R CPUs, each with 24 cores, and 8 Nvidia Quadro RTX 5000 GPUs, each equipped with 16GB of GDDR6 high bandwidth memory.

7.1 Experimental Results

We evaluate cDLRM on the following criteria: (1) model accuracy on a single GPU; (2) sensitivity of computational performance on cache parameters when training on a single GPU; and (3) accuracy and scalability when using multiple GPUs. In all cases, we use the original DLRM [12] as the baseline for comparison, using the same hyperparameters and no hyperparameter tuning or optimization. We use a bottom MLP arch of 13-512-256-64, a top MLP arch of 512-512-256-1, an embedding table dimension of 64 and a 16-way associative cache to evaluate (1) and (2). For (3) we use the model architecture used in the MLPerf [10] benchmark in which the only difference is a bottom MLP arch of 13-512-256-128 and an embedding table dimension of 128.

Model Accuracy on a single GPU. As mentioned earlier, there are rare cases when the embedding vectors may be stale. The comparison of test accuracy between the final DLRM model and the cDLRM model obtained at convergence shows that accurracy obtained by cDRLM is within 0.02%. These results are shown in Table 1(b). Both cDLRM and DLRM were trained with the same batch size of 2048 for 2 epochs.

Caching overheads and training speed on a single GPU. To analyze the impact of caching on the training speed of cDLRM, we split the training time into two components: (1) the time it takes to execute lines 7, 8, 9 and 10 in Algorithm 2 - *batch computation time*. (2) The time it takes to execute lines 4 and 5 - *caching overhead*. The *amortized caching overhead*, is the average per batch overhead due to caching incurred by every batch in the lookahead window. It is computed by dividing the caching overhead by the lookahead window size. The *total per batch execution time* is the sum of the batch computation time and the amortized caching overhead.

cDLRM is heavily parallelized with vector operations. Hence, caching overhead costs can be amortized when using large lookahead windows. By using a larger lookahead window there are more training examples to pre-process concurrently which benefit from vectorization. On the other hand, using a larger lookahead window requires more memory for $Q_{lookahead}$, and larger embedding table cache sizes in GPU DRAM to hold more embeddings. Hence, there is a tradeoff in amortizing caching overhead vs. reducing cache size. As shown in Figure 3a, with a large cache size of up to 50,000 sets per table, the total per batch computation time decreases with increase in the lookahead window size. In general, the larger the lookahead window, the smaller the per-batch amortized caching overhead. This rule holds up to the maximum available hardware parallelism to execute vectorized code. Figure 3b shows the amortized caching overhead gradually decreasing with increasing lookahead window size. In fact, the amortized caching overhead decreases independent of the cache size on GPU DRAM.

However, as mentioned above there is a tradeoff between lookahead window size and cache size, even when sufficient hardware parallelism is available. Figure 3a also demonstrates the detrimental effects of smaller cache sizes when using larger lookahead windows. As explained before, when using a larger lookahead window more embedding vectors need to be brought into the GPU cache. This results in a higher probability of there being more unique indices in the lookahead window than there are empty cache slots available. This leads to more conflict victims. Thus, the forward propagation becomes the bottleneck as a result of having to fetch these conflict victims into the victim tables. This fact is corroborated by Figure 3c which shows the cache hit rate for different cache sizes and lookahead windows. The cache hit rate measures the number of embedding vectors that are already cached in GPU DRAM and those that miss in the cache are conflict victims. When the cache size is small increasing the lookahead window causes more conflict victims. Hence, the total per batch execution time increases with lookahead window size when the cache size is not properly configured to account for the lookahead window size. This result shows that the cache size and lookahead window size selection must be coordinated for optimal performance.

Accuracy and Scalability of cDLRM on multiple GPUs. As explained in section 6 cDLRM can leverage multiple GPUs when they are available by training in a purely data parallel fashion. As a result, cDLRM shows strong scaling when using multiple GPUs. Figure 4(a) illustrates the benefit of using multiple GPUs when they are available. The model architecture used is the same model architecture used in the MLPerf training benchmark. We use a cache size of 150,000 sets with 16-way set associativity. We use a lookahead window size of 3000 for batch sizes upto 8192. For a batch size of 16384 we use a lookahead window size of 1500 and for a batch size of 32768 we use 500. These values give us the best performance for the respective batch sizes. As we can see, larger batch sizes warrant the use of additional GPUs much more so than smaller batch sizes. Unlike the DLRM baseline, cDLRM can select the number of GPUs based on optimal batch sizes as opposed to being constrained by embedding table memory requirements. Another effect of more data parallelism is that caching overhead starts to become a larger percentage of the total computation time. Figure 4(b) illustrates this phenomenon. Caching can add over 25% overhead to the total computation time. We believe that this overhead can be reduced or even eliminated with some additional software engineering. But we leave this to future work. Recall that for performance reasons that cDLRM aggregates caches at a granularity specified by the hyperparameter λ due their sparse

Dataset	C	M	Training/Test Examples
Kaggle	13	26	39,291,958 / 3,274,330
Terabyte	13	26	645,753,353 / 13,760,492

(a) Dataset dimensions and split

Dataset	DLRM	cDLRM
Kaggle	78.967±0.01	78.971±0.01
Terabyte	81.07±0.01	81.06±0.01

(b) Accuracies at convergence

Table 1: Dataset Details

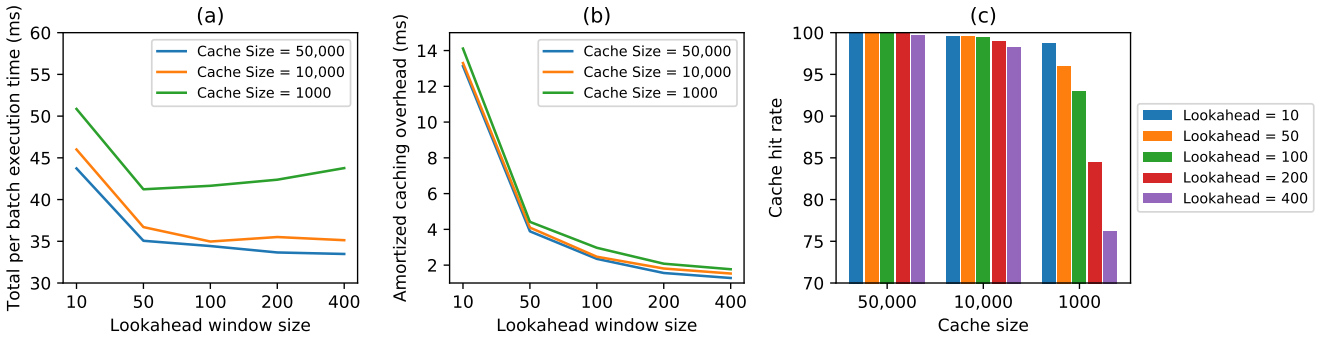


Figure 3: (a) Single GPU cDLRM Performance (b) Caching cost (c) Cache Performance (All data from Terabyte dataset, Bot MLP: 13-512-256-64, Top MLP: 512-512-256-1, Embedding Dim: 64)

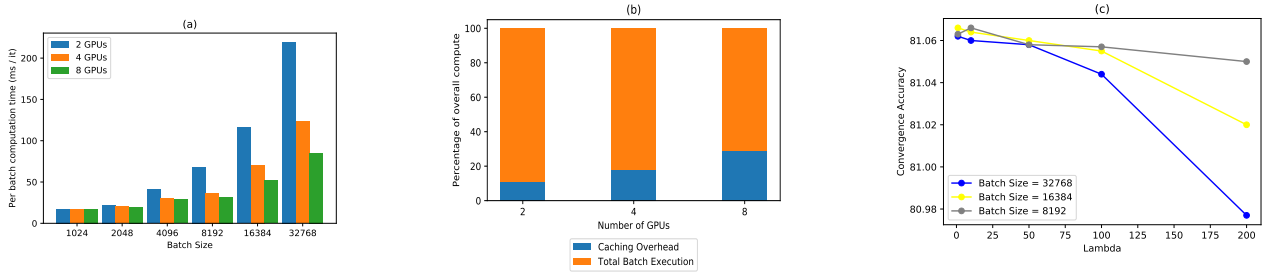


Figure 4: (a) cDLRM multigpu scaling; (b) Caching overhead with 32K batch size; (c) Sensitivity of convergence accuracy on cache aggregation frequency λ. Cache Size=150,00 ways / 16 sets. Lookahead size=500

nature. Figure 4c illustrates the effect of lambda on convergence accuracy for different batch sizes. The takeaway is that for a fixed cache and lookahead size, λ can be larger for smaller batch sizes without incurring significant penalty in convergence accuracy.

8 RELATED WORK

The problem of large recommendation systems have been recognized in previous work. [18] proposes a hierarchical parameter server distributed between NVMe, CPU and GPU memory to alleviate the demands on GPU memory for storing model parameters. Their approach is centered around a distributed hash table, split between multiple GPUs, as well as an execution pipeline involving the SSD, CPU and GPU. Our approach takes an orthogonal approach of reducing the need for many GPUs to start with using caching. However, when a cached system still needs to be distributed across many GPUs we believe [18] can be applied on top of cDLRM for cached parameter aggregations. AIBox [19] takes a similar approach to [18]. They seek to keep model parameters in NVMe and train on GPUs while using CPU memory as a cache for frequently accessed parameters. The key difference between cDLRM and AIBox is that AIBox doesn't proactively identify the necessary model parameters for a given window of batches and instead caches parameters in CPU memory. They use various storage access optimizations to reduce I/O overhead. As we described earlier the locality of embedding table accesses is very poor in DLRM like models. Hence, pre-processing batches and prefetching the unique indices is an efficient way to get near 100% cache hit rate. We also address how we deal with various consistency issues with caching embeddings. Other approaches to reducing GPU memory usage such as [14] are tailored for models whose intermediate activations require large

amounts of GPU memory. DLRM intermediate activations are fairly small compared to embedding table size. Hence such approaches are ill-suited to DLRM like models. Other pipelined approaches such as [6] are designed for models in which sparse parameters and dense parameters are never part of the same layer of computation. These are models that lend themselves to efficient pipelining. DLRM on the other hand contains layers with both sparse and dense parameters, hence designing pipelined systems that do not degrade model performance can be very challenging. We would like to stress that the objective of cDLRM is not to be the fastest recommendation model training system, but to democratize recommendation model training to the point where it does not require hundreds of thousands of dollars of hardware and GPUs to be able to just fit the model.

9 CONCLUSION

cDLRM is a recommendation model training system that enables cost efficient training by storing the large embedding tables entirely on a CPU. It is based on the key insight that only a small subset of embedding table entries are updated by each training batch. Furthermore, these small subset of entries can be identified by looking ahead in the training batches. As such, cDLRM uses a CPU based lookahead thread that pre-processes many training batches ahead of their training and prefetches the set of unique embedding vectors needed for training these batches into GPU DRAM. cDLRM decouples the memory demands of the recommendation model from its computational demands and thus enables the training of large models on a single GPU. When multiple GPUs are available, cDLRM shows strong scaling across GPUs and enables pure data parallelism, all while preserving model accuracy. More importantly, by eliminating the need for hundreds of thousands of dollars of hardware and GPUs just to store recommendation models, cDLRM takes an important step in the direction towards democratizing recommendation model training.

REFERENCES

[1] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 265–283. https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf

[2] M. T. Ahamed and S. Afroge. 2019. A Recommender System Based on Deep Neural Network and Matrix Factorization for Collaborative Filtering. In *2019 International Conference on Electrical, Computer and Communication Engineering (ECCE)*. 1–5.

[3] Paul Covington, Jay Adams, and Emre Sargin. 2016. Deep Neural Networks for YouTube Recommendations. In *Proceedings of the 10th ACM Conference on Recommender Systems*. New York, NY, USA.

[4] Carsten Felden and Peter Chamoni. 2007. Recommender Systems Based on an Active Data Warehouse with Text Documents. In *Proceedings of the 40th Annual Hawaii International Conference on System Sciences (HICSS '07)*. IEEE Computer Society, USA, 168a. https://doi.org/10.1109/HICSS.2007.460

[5] Sahin Cem Geyik, Qi Guo, Bo Hu, Cagri Ozcaglar, Ketan Thakkar, Xianren Wu, and Krishnaram Kenthapadi. 2018. Talent Search and Recommendation Systems at LinkedIn: Practical Challenges and Lessons Learned. In *The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval* (Ann Arbor, MI, USA) *(SIGIR '18)*. Association for Computing Machinery, New York, NY, USA, 1353–1354. https://doi.org/10.1145/3209978.3210205

[6] Biye Jiang, Chao Deng, H. Yi, Zelin Hu, Guorui Zhou, Y. Zheng, Sui Huang, X. Guo, D. Wang, Y. Song, Liqin Zhao, Z. Wang, P. Sun, Y. Zhang, Di Zhang, Jin hui Li, Jian Xu, Xiaoqiang Zhu, and Kun Gai. 2019. XDL: an industrial deep learning framework for high-dimensional sparse data. *Proceedings of the 1st International Workshop on Deep Learning Practice for High-Dimensional Sparse Data* (2019).

[7] Norman P. Jouppi. 1990. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture* (Seattle, Washington, USA) *(ISCA '90)*. Association for Computing Machinery, New York, NY, USA, 364–373. https://doi.org/10.1145/325164.325162

[8] Jiahui Liu, Peter Dolan, and Elin Rønby Pedersen. 2010. Personalized News Recommendation Based on Click Behavior. In *Proceedings of the 15th International Conference on Intelligent User Interfaces* (Hong Kong, China) *(IUI '10)*. Association for Computing Machinery, New York, NY, USA, 31–40. https://doi.org/10.1145/1719970.1719976

[9] M. Marović, M. Mihoković, M. Mikša, S. Pribil, and A. Tus. 2011. Automatic movie ratings prediction using machine learning. In *2011 Proceedings of the 34th International Convention MIPRO*. 1640–1645.

[10] Peter Mattson, Christine Cheng, Cody Coleman, Greg Diamos, Paulius Micikevicius, David A. Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, David Brooks, Dehao Chen, Debojyoti Dutta, Udit Gupta, Kim M. Hazelwood, Andrew Hock, Xinyuan Huang, Bill Jia, Daniel Kang, David Kanter, Naveen Kumar, Jeffery Liao, Guokai Ma, 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. 2019. MLPerf Training Benchmark. *CoRR* abs/1910.01500 (2019). arXiv:1910.01500 http://arxiv.org/abs/1910.01500

[11] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, Liang Luo, et al. 2021. High-performance, Distributed Training of Large-scale Deep Learning Recommendation Models. *arXiv preprint arXiv:2104.05158* (2021).

[12] 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. *CoRR* abs/1906.00091 (2019). arXiv:1906.00091 http://arxiv.org/abs/1906.00091

[13] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. arXiv:1912.01703 [cs.LG]

[14] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. 2016. Virtualizing Deep Neural Networks for Memory-Efficient Neural Network Design. *CoRR* abs/1602.08124 (2016). arXiv:1602.08124 http://arxiv.org/abs/1602.08124

[15] Sebastian Schelter, Christoph Boden, Martin Schenck, Alexander Alexandrov, and Volker Markl. 2013. Distributed Matrix Factorization with Mapreduce Using a Series of Broadcast-Joins. In *Proceedings of the 7th ACM Conference on Recommender Systems* (Hong Kong, China) *(RecSys '13)*. Association for Computing Machinery, New York, NY, USA, 281–284. https://doi.org/10.1145/2507157.2507195

[16] Y. Wang, S. C. Chan, and G. Ngai. 2012. Applicability of Demographic Recommender System to Tourist Attractions: A Case Study on Trip Advisor. In *2012 IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology*, Vol. 3. 97–101.

[17] 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*, I. Dhillon, D. Papailiopoulos, and V. Sze (Eds.). Vol. 2. 412–428. https://proceedings.mlsys.org/paper/2020/file/f7e6c85504ce6e82442c770f7c8606f0-Paper.pdf

[18] 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*, I. Dhillon, D. Papailiopoulos, and V. Sze (Eds.). Vol. 2. 412–428. https://proceedings.mlsys.org/paper/2020/file/f7e6c85504ce6e82442c770f7c8606f0-Paper.pdf

[19] Weijie Zhao, Jingyuan Zhang, Deping Xie, Yulei Qian, Ronglai Jia, and Ping Li. 2019. AIBox: CTR prediction model training on a single node. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*. 319–328.