2025 IEEE International Symposium on High-Performance Computer Architecture (HPCA)

Machine Learning-Guided Memory Optimization for DLRM Inference on Tiered Memory

Jie Ren[†,*], Bin Ma[‡,*], Shuangyan Yang[‡], Benjamin Francis[§],
Ehsan K. Ardestani[§], Min Si[§], and Dong Li[‡]
William & Mary[†], Meta[§], University of California, Merced[‡]
{jren03}@wm.edu, {bf0428, ehsanardestani, msi}@meta.com, {bma100, syang127, dli35}@ucmerced.edu

***Abstract*—Deep learning recommendation models (DLRMs) are widely used in industry, and their memory capacity requirements reach the terabyte scale. Tiered memory architectures provide a cost-effective solution but introduce challenges in embedding-vector placement due to complex embedding-access patterns. We propose RecMG, a machine learning (ML)-guided system for vector caching and prefetching on tiered memory. RecMG accurately predicts accesses to embedding vectors with long reuse distances or few reuses. The design of RecMG focuses on making ML feasible in the context of DLRM inference by addressing unique challenges in data labeling and navigating the search space for embedding-vector placement. By employing separate ML models for caching and prefetching, plus a novel differentiable loss function, RecMG narrows the prefetching search space and minimizes on-demand fetches. Compared to state-of-the-art temporal, spatial, and ML-based prefetchers, RecMG reduces on-demand fetches by 2.2×, 2.8×, and 1.5×, respectively. In industrial-scale DLRM inference scenarios, RecMG effectively reduces end-to-end DLRM inference time by up to 43%.**

I. Introduction

Deep learning recommendation models (DLRM) are widely used in industry [18], [54]. The DLRM inference takes 80% of total AI inference cycles at some data centers [41]. The DLRM processes continuous features with compute-intensive deep neural networks and categorical features with data-intensive embedding operators [24], [94] to generate recommendation results. GPUs have been widely used to deploy DLRM inferences. Massive categorical features stored as embedding tables (EMBs) creates deployment challenges - due to the sheer size of EMBs, which is proportional to the cardinality of categorical features and dimensionality of the latent space, the DLRM is often too large to fit onto a single device memory.

The existing work deploys EMBs on tiered memory [4]: a small portion of frequently accessed embedding vectors in EMBs are cached in GPU memory, while the rest embedding vectors are placed in host CPU memory or SSD which is much slower to access than GPU's local memory. On-demand fetching of embedding vectors from the host CPU memory/SSD to GPU memory causes up to $O(10\mu s)$ latency. However, large-scale DLRM inferences often have to achieve latency-related Service-Level Agreements (SLA). In the real-world, the SLA target is in an order of up to a few hundreds of ms [4]. Hence, the efficient management of limited GPU memory for EMBs to minimize the violation of the SLA target is the key to the success of DLRM inferences.

Existing work on performance optimization of recommendation models [3], [68], [81] and our observations with real production inference traces show that embedding vector accesses in DLRM inferences follow a power law distribution. With such a distribution, a portion of embedding vectors (about 20%) takes about 80% of accesses to EMBs. This memory access pattern provides opportunities for traditional LRU-like caching mechanisms to manage a buffer on GPU memory. However, this method faces two challenges. First, a significant portion of embedding vector accesses still occur on external memory, causing on-demand data fetching. Although these vectors are not frequently reused individually, their accumulated accesses constitute a large portion of total memory accesses. These vectors cannot be held in a traditional LRU-cache due to the lack of temporal locality.

Second, some embedding vectors have long reuse distances. Based on datasets from production environments, we observe that 20% of embedding vector accesses have a reuse distance larger than 2^{20}. Such long reuse distances can exceed the size of most software-managed GPU buffers in production, reducing the effectiveness of traditional fully-associative caching.

In this paper, we explore the feasibility of using machine learning (ML) to address these data locality issues difficult to be solved by the existing method to manage GPU buffer for DLRM (i.e., the LRU-like caching). ML has been successful in enhancing cache-line prefetchers [32], [71], [74] and page prefetchers in heterogeneous memory tiers [22], [23]. ML can learn implicit memory access patterns from the sequence of memory accesses, enabling the prediction of irregular or streaming memory access patterns for prefetching, which are difficult to handle with traditional approaches.

The feasibility of using ML to cache and prefetch embedding vectors in the GPU buffer stems from the strong correlation in user access behaviors, both across users and for individual users [48], [80]. This correlation leads to implicit relationships between consecutive vector accesses, making the access patterns learnable and predictable. Importantly, the ability of ML to learn and exploit these correlations is not constrained by the presence of long reuse distances or irregular memory access patterns [19], [71], which are common in DLRM inference workloads. This property enables the development of effective ML-based caching and prefetching

*Equal contribution

strategies tailored to embedding vector access patterns.

However, using ML for GPU buffer management presents unique challenges not encountered in existing ML-based prefetching problems. First, the prefetch model must make predictions in a large search space consisting of billions of vectors from embedding tables. Vector accesses within such a large search space have different characteristics than memory accesses in a large address space. Traditional memory prefetchers [32], [71], [74], [95], [96] target memory accesses that are sparsely distributed in the address space [32], allowing for a *delta*-based approach that predicts the address difference between two contiguous memory accesses. This approach can narrow down the search space because the sparse distribution leads to a limited number of deltas. In contrast, the dense distribution of vectors in the search space results in a large number of deltas, invalidating the effectiveness of the delta-based approach. Moreover, two deltas with the same value can be calculated from different EMBs, representing different semantics and memory access patterns. Hence, simply using deltas loses feature distinctiveness, an ML property required for accurate prediction.

Second, using ML for prefetching faces a data-labeling problem. To train the ML model, one must label training dataset to establish ground truth: within the search space of model prediction, the model learns which memory address (or embedding vector in the context of DLRM) will be accessed. Given the large search space, the number of labels is huge, causing high complexity in the ML model to enable high prediction accuracy and coverage. Existing solutions, such as Voyager [71], address this problem by decomposing the memory address into page address and offset, and predicting them separately, significantly reducing the number of labels (especially for offsets). However, this decomposition method cannot work in DLRM because there are still a large number of offsets (or vectors) within an embedding table when mapping the idea of offset to the context of DLRM. Furthermore, existing work [6] classifying memory access patterns into a handful of categories to reduce the number of labels cannot work either, because of random nature in memory accesses.

To address the above challenges, we introduce RecMG, a GPU buffer management system customized for DLRM inferences. RecMG introduces a novel approach that employs two separate neural models, unlike the unified neural modeling used in existing prefetching techniques [32], [71], [74]. The first model focuses on prefetching embedding vectors with few reuses or long reuse distances, while the second model targets caching, emphasizing temporal locality and effective eviction of embedding vectors. The caching model (the second model), once trained with endless user data, provides the flexibility to outperform LRU-like models in terms of caching effectiveness, thereby more effectively reducing the search space for the prefetch model. By using two separate models, RecMG effectively improves prediction accuracy.

To address the data labeling problem, RecMG transforms the embedding-vector prediction problem into a binary classification problem for the caching model. Given a sequence of prior accesses as input, the caching model predicts which vectors should be kept in the cache, requiring only two labels. This significantly reduces the complexity of the data labeling process. Furthermore, RecMG's prefetcher outputs a sequence of embedding-vector indices for prefetching, differing from traditional ML-based memory prefetchers that predict only the next address. By prefetching a sequence of vectors [32], [74], RecMG aims to improve the chance of prefetch hits in the GPU buffer. The prefetched embedding vectors remain valid in the buffer for a time duration, expecting hits in the near future.

Using the prefetch model introduces a challenge in designing the loss function for optimization during prefetch model training. To improve prefetch effectiveness, we extend the length of the evaluation window used for deciding prefetch hits. However, this extension introduces a mismatch between the evaluation window length and the model-output sequence length, making it difficult to create a differentiable loss function that quantifies prediction error without introducing bias into the model output. To address this challenge, RecMG introduces a loss function based on the Chamfer Measure [13].

This paper makes the following contributions:

- We study the patterns of embedding vector accesses in DLRM inferences and reveal the inability of using traditional LRU-like caching to buffer embedding vectors effectively;
- We propose an ML approach to learn vector access patterns for caching and prefetching, and introduce techniques centering around reducing the search space of ML and improving accuracy;
- Compared with state-of-the-art prefetchers, including a temporal prefetcher (Domino [8]), a spatial prefetcher (Bingo [10]), and an ML-based prefetcher (TransFetch [96]), RecMG outperforms by 190×, 400×, and 27× respectively in terms of prefetch sequence prediction correctness, and reduces the on-demand fetches by 2.2×, 2.8×, and 1.5×, respectively. We also demonstrate that existing ML-based prefetchers (Voyager and TransFetch) cannot work effectively for DLRM inferences due to high model training and inference overhead.
- Evaluating RecMG on a production-like platform with five datasets, we show that our approach reduces end-to-end DLRM inference time by 31% on average (up to 43%), outperforming LRU caching in production.

II. Background

DLRM architecture. Figure 1 presents major components of an industry-scale DLRM [54]. A DLRM input (an inference query) is composed of categorical and continuous features. The categorical features are sparse, representing categorical data (such as what subject a user is interested in). The continuous features are dense, representing user information (such as a user's age). A categorical feature is represented as a one-hot or multi-hot binary vector where one or multiple positions in the vector corresponding to one or multiple categories are 1 and others are 0. Hence, the representation of the categorical

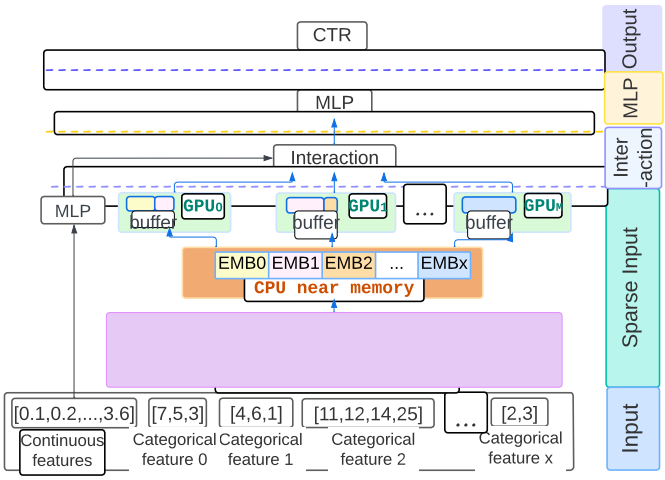

Fig. 1. DLRM architecture on tiered memory. Embeddings map the categorical features into dense representations.

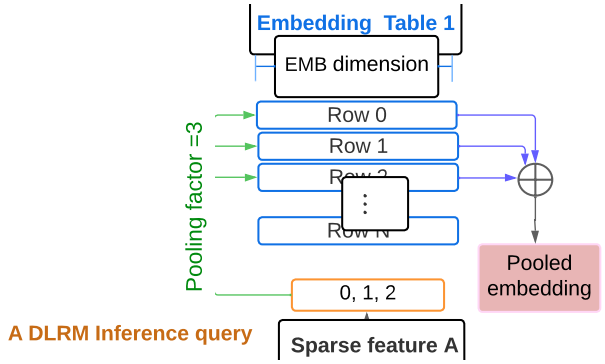

Fig. 2. Embedding tables and pooling factor.

features is *sparse*. Different categorical features have varying cardinality. The categorical features can have big cardinalities in the scale of billions [40], [97]. Given the input features, DLRM predicts the probability that a user will interact with a particular piece of content. Such a probability is referred to as the click-through-rate (CTR).

DLRM often has two major components: embeddings and interaction, shown in Figure 1. Embeddings map the categorical features into dense representations. Interaction aggregates the continuous features and dense representation of categorical features by dot-product summation concatenation, and then captures their interaction by top multi-layer perceptrons (MLP). There is also a bottom MLP which processes the dense inputs to reproject the continuous features to dense ones.

Embeddings are based on embedding tables (EMBs) that map categorical features from a high-dimensional, sparse space to a low-dimensional, dense space. In essence, EMBs act as large lookup tables, where each row functions as a latent vector encoding of a category (i.e., a sparse-feature value). The input categorical features activate specific categories, which are used as indices to gather one or more embedding vectors from the EMBs, as depicted in Figure 2. These embedding vectors are gathered (or pooled) on a per-EMB basis. The

TABLE I
EXTRA OVERHEAD OF EMBEDDING-VECTOR ACCESSES IN THE DLRM INFERENCE. "CACHING RATIO" DENOTES THE PROPORTION OF EMBEDDING VECTORS STORED IN THE GPU BUFFER.

	# emb tables	# accesses	# unique indices	batch size	caching ratio	emb access overhead
DS1	24	20.1M	2.2M	6K	100%	0%
DS2	24	20.1M	2.2M	6K	20%	52.7%
DS3	192	191.4M	6.4M	6K	7%	30.1%
DS4	192	191.4M	6.4M	18K	7%	58.7%

gather operation, often using summation or concatenation, is called *feature pooling*.

EMBs are large: a single EMB in a production-scale DLRM can be at the scale of 100s of GB; the total memory capacity of EMBs can be at the scale of multi-TB. Because of large memory consumption of EMBs, GPU memory is often a bottleneck. Using tiered memory (i.e., GPU memory tier plus CPU memory tier) is a solution.

The DLRM inference is different from the DLRM training in terms of EMBs accesses. In DLRM inference, a batch of items (i.e., categories) is accessed simultaneously, resulting in a more diverse pooling factor. In contrast, DLRM training focuses on learning from individual items and their specific interactions, accessing one item at a time. Furthermore, in training, the dataset and memory accesses are predetermined, allowing for well-planned prefetching [12], [68], while the inference does not have such luxury. Therefore, embedding vector prefetching is more challenging in DLRM inference compared to training.

Limitations of existing data caching and prefetching techniques for DLRM inferences. To reduce EMB accesses outside of GPU memory, the PyTorch Library [58] maintains a software-managed buffer inside GPU memory for caching frequently accessed embedding vectors using an LRU policy (detailed in Section VII-E). However, even with this optimization, embedding-vector accesses remains expensive. We study execution time of DLRM inference [27] with various datasets using NVIDIA A100 GPU. Table I shows the results. When all embedding vectors fit into the GPU buffer (DS1), there is no extra overhead for embedding vector accesses. As the number of EMB tables/indices/batch size increases and the caching ratio decreases, the embedding vector accesses takes larger overhead (up to 58.7% of the total execution time). Such overhead comes from the access misses in GPU buffer (leading to CPU memory accesses).

Prefetch has been employed to mitigate the performance impact of accessing embedding vectors in memory with the PyTorch library [58], [59] using Unified Virtual Memory (UVM). However, it lacks guidance on *which* embedding vectors to prefetch and *when* to prefetch them. Furthermore, our evaluation in Section VII-B demonstrates that many existing rule-based prefetchers [2], [9], [11], [51], [70], [75] struggle to capture the irregular nature of embedding accesses, which exhibit extremely low spatial locality and limited temporal

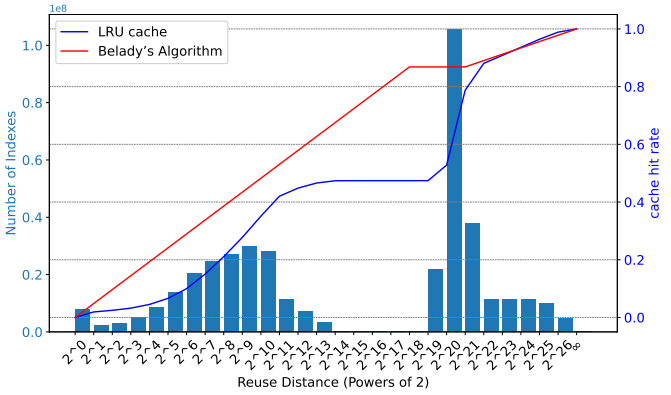

Fig. 3. Reuse distance of embedding-vector accesses in 856 sparse features.

locality. Also, ML-based prefetchers [32], [44], [45], [66], [69], [73], [74], [95] fail to handle the extremely large search space (i.e., tens of billions of unique indices) and hence incur large decision-making overheads, making them impractical in production environment. As shown in Section VII-B, existing rule-based and ML-based prefetchers only achieve prefetch accuracy of less than 1%.

III. Study of Sparse Features Accesses

Sparse features represent categorical data. The categorical feature space can be arbitrarily large. During a DLRM inference, the values of the input categorical feature indicate or activate some categories in the feature space as indices, and indices are used to access sparse features (i.e., EMBs and their embedding vectors). We characterize the accesses to sparse features using sixteen datasets from Meta [26] and each includes over 400 million accesses to 856 sparse features. There are 62-million unique embedding vectors in each dataset. These datasets represent memory access patterns in production recommendation workloads, and studying such datasets is unprecedented.

Reuse distance analysis. We measure the reuse distance of embedding vectors in EMBs. The reuse distance is a metric for locality analysis [21], [67]. In the context of DLRM, the reuse distance defines the number of distinct embedding-vectors accessed between two consecutive references to the same vector. Reuse distance quantifies the likelihood of a cache hit for an embedding vector access in a fully associative LRU cache. If the reuse distance of a vector access is larger than the cache size, then the latter access (reuse) is likely to cause a cache miss. Figure 3 shows the reuse distance for 410 million embedding vector accesses. We have two observations.

(1) The reuse distance of 20% accesses is larger than 2^{20}, which is larger than the capacity of a typical GPU buffer (hosting embedding-vectors at the scale of $O(100,000)$), and hence invalidates the effectiveness of the traditional LRU cache. Given the large number of accesses with long reuse-distance, there is a strong need to reduce on-demand fetches.

(2) There is room to improve the efficiency of the LRU cache. We compare the hit rate of a fully associative LRU cache with the optimal hit rate collected from the Belady algorithm [14]. The results show that to achieve an 80% hit rate, the optimal cache requires only 1/16 of the capacity of the LRU cache. This motivates us to introduce new caching and prefetching algorithms to reduce cache misses during DLRM inference. This approach is expected to outperform the traditional LRU cache with the same capacity as ours.

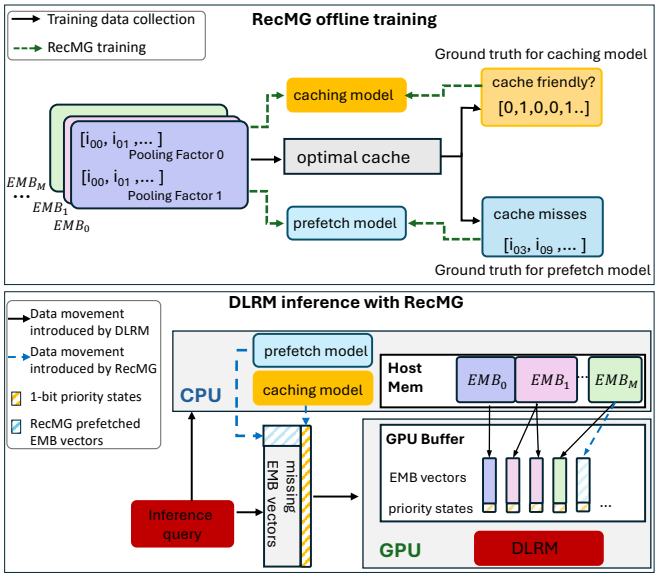

Fig. 4. Design overview of RecMG.

Pooling factor. The previous work [4] reveals that the average pooling factor during DLRM inferences in a production environment varies a lot across DLRM inference queries (in the range of 1 to hundreds). Such a wide distribution of pool factor calls for effective handling of input sequences.

IV. Overview

RecMG aims to learn correlations between consecutive embedding-vectors accessed by prior DLRM inferences to predict the embedding-vector accesses by future DLRM inferences. Similar to the traditional temporal prefetching [8], [34], [72], [82], [83], this learning task can be formulated as a probabilistic prediction problem: Given a sequence of historical accesses (vec_1, vec_2, ..., vec_k, called *input features*) up to k embedding-vectors, predicts the probability that an embedding vector will be accessed.

$$P(vec|vec_1, vec_2, .., vec_k) \tag{1}$$

RecMG aims to achieve both high prediction accuracy and high coverage. The coverage is defined as follows. Given a sequence of embedding-vector accesses as ground truth (denoted as vec_seq_{gt}), a prefetcher outputs a sequence of embedding-vector accesses, denoted as vec_seq_{out}, where both vec_seq_{gt} and vec_seq_{out} can have repeated vector accesses. The coverage is about unique vectors in vec_seq_{gt} and vec_seq_{out}.

$$coverage = \frac{|vec\_seq_{out} \cap vec\_seq_{gt}|}{|vec\_seq_{gt}|} \tag{2}$$

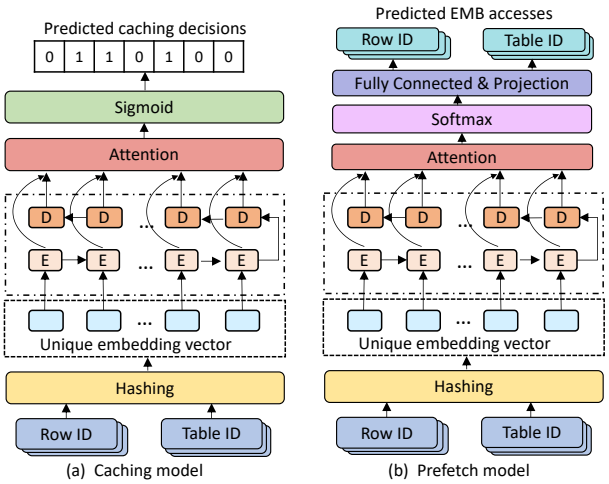

Fig. 5. The architecture of (a) caching and (b) prefetch models. The dashed rectangle represents one LSTM stack. "E" and "D" stand for encoder and decoder in LSTM respectively.

Having high coverage is important in the context of DLRM inferences, because some embedding vectors may not be accessed often, but should be covered in order to minimize the violation of SLA.

RecMG uses a combination of a caching model and a prefetch model (Section V). The caching model makes prediction based on temporal locality and reduces the search space of the prefetch model; The prefetch model makes prediction for irregular accesses. Both models are sequential models with an attention mechanism to concentrating on key correlations between embedding-vector accesses (even if they are far away from each other).

Figure 4 illustrates the workflow of RecMG. Offline training of the two models in RecMG uses DLRM inference traces and their theoretically optimal caching analysis as training data. During online DLRM inference, the caching model generates a 1-bit priority for each embedding, guiding the eviction of embedding vectors (Section V-A), while the prefetching model predicts which embeddings will be accessed, enabling the insertion of embedding vectors (Section V-B). Section VI discusses the deployment of RecMG in production systems.

V. Learned Placement of Embedding Vectors

Figure 5 shows the caching and prefetch model architecture in RecMG. The backbone of both models is sequence-to-sequence (seq2seq) LSTM model with attention mechanism. Each model takes a sequence of prior accesses as input. The order of the embedding-vector accesses within the sequence matters to the prediction accuracy, and the sequence reveals implicit correlations between the accesses. The LSTM has been utilized for handling this type of sequence [5], [7], [17], making it a natural choice in our scenario.

Each LSTM stack includes a pair of an encoder and a decoder. The pair naturally generates a dense representation of embedding vectors in a continuous space and captures the relationship between different embedding vectors within the same input sequence. The caching model in RecMG consists of a single LSTM stack with 37K learnable parameters, while the prefetch model comprises two LSTM stacks with a total of 74K learnable parameters (details in Table III).

Moreover, we introduce an attention mechanism into both models, which is able to learn the relationship between input sequences, even when the accesses to the same embedding vectors across DLRM inference requests are far apart. The attention mechanism assigns higher weights to the relevant embedding vectors from distant input sequences, enabling the model to effectively capture long-range dependencies.

Recent works [15], [20], [39], [77] show that transformers are powerful for handling long sequences because of their high computational parallelism. However, when deploying transformer-based RecMG for memory management in production, we find a lack of extra computation cycles on the GPU, making it challenging to leverage transformers' parallelism effectively. Therefore, we use CPU for RecMG deployment. We choose LSTM (instead of transformers) as the bone structure because LSTMs are more CPU-friendly and still provide good model accuracy. This choice allows RecMG to be deployed in resource-constrained environments while maintaining effective memory management, making it a practical solution.

A. Caching Model

We introduce a caching model to capture temporal locality and enable effective eviction of embedding vectors.

Model input. The input to the caching model is a sequence of embedding vector accesses, consisting of a list of row IDs and their corresponding table IDs.

RecMG truncates the sequence of prior vector accesses into a set of fix-sized shorter sequences to handle DLRM inferences with wide distribution of pooling factor. Each shorter sequence is a *chunk* and serves as the basic unit of input for the caching model. This indicates that an input sequence may come from the same or multiple inference queries. We do not impose an requirement that the embedding-vector accesses must come from the same DLRM inference query, such that the input sequence can include correlation information between queries, which is useful for improving model accuracy.

Model output. The output of the caching model is a binary sequence with the same length as the input sequence of the caching model. Each element of the output sequence indicates whether the corresponding element in the input sequence should have higher priority to stay in the GPU buffer than other elements ("1" means higher priority and "0" indicates otherwise). The priority of an embedding vector is used to decide whether a vector should be eliminated out of the buffer when the buffer size is not enough for fetching new vectors (see Section VI).

Model training. We use the *cross entropy loss* as the loss function to train the caching model, since the caching model works on a classification task (deciding high or low priority). The cross entropy loss uses a sigmoid function to estimate the binary probability distribution of candidate labels, predicting whether or not an event is likely to happen.

B. Prefetch Model

The prefetch model aims to prefetch embedding vectors with few reuse or long reuse distance into the GPU buffer. It uses the same input as the caching model.

Model output. The output of the prefetch model is a sequence of embedding vectors to be accessed. The sequence length of the output is smaller than that of the input, in order to improve prefetch accuracy.

Model structure. The prefetch model also uses a seq2seq LSTM model. Different from the caching model that in essence works on a classification problem with the output of a binary sequence, the prefetch model predicts embedding vectors to be accessed in a large search space. As a result, the prefetch model has an output embedding layer (i.e., fully connected and projection layer) after the attention layer to convert the attention vectors into the indices of embedding vectors as the output. See Figure 5 for model structure.

Model training. To train the prefetch model, we must calculate the difference between the model output and ground-truth in order to calculate the gradient. We decide the ground-truth based on how the output of the prefetch model is used for managing the GPU buffer. Specifically, the ground-truth represents the optimal prefetch decisions that would minimize cache misses in the GPU buffer for the next batch of DLRM inference requests.

In particular, the prefetch-model output (named PO, which is a sequence of embedding vectors to be accessed with a length of $|PO|$) will be fetched into the GPU buffer, but can stay longer than the next $|PO|$ embedding vectors referenced by DLRM inference queries. There are two reasons why the prefetched vectors stay longer. First, we aim to maximize the benefits of prefetching. If there is any mismatch between the $|PO|$ prefetched vectors and the next $|PO|$ vectors referenced by DLRM inference queries, we do not immediately replace the prefetched vectors. Instead, we expect them to be accessed in the near future. Second, prefetching the vectors from CPU memory takes time. During the prefetch process, the DLRM inference queries can access embedding vectors in parallel. This indicates that the prefetch may not be useful for the immediately next N vector accesses from DLRM inference queries, but rather for accesses in the near future, where N is the number of vector accesses that take the same time as fetching PO embedding vectors.

Based on the above observation, when deciding the ground-truth for an output of the prefetch model, we look at a sequence of embedding vector accesses from DLRM inference queries as ground truth, and the sequence (named W) has a length of $|W|$ (where $|W| > |PO|$). We compare the difference between vectors in PO and vectors in W. The training objective is to minimize the difference between PO and W. We formulate the objective function of the prefetch model in Equation 3.

$$f : min|dist(PO, W)| \tag{3}$$

where $dist()$ quantifies the distance. $dist()$ can be constructed as a function that counts the number of non-overlapped vectors between PO and W. However, this function is not differentiable needed by optimization of the objective function (i.e., AI models using such an objective function cannot be trained.

Hence, we introduce the Chamfer Measure [13] (CM) to build $dist()$. Given two sets of data points S_1 and S_2, CM is defined in Equation 4 and differentiable.

$$d_{CM}(S_1, S_2) = \sum_{x \in S_1} \min_{y \in S_2} |x - y| \tag{4}$$

With CM, each point in $x \in S_1$ finds its closet point in S_2. In the context of the prefetch model, S_1 and S_2 are PO and W respectively. However, using CM introduces shortcuts in the prefetch training, and the prediction result (the output of the prefetch model) tends to have the same value in all elements in PO. For example, during the training process, assume that PO=$\{1, 2, 3\}$ and W=$\{2, 6, 7, 8\}$. "2" in W tends to be chosen to minimize the distance from each point in $x \in PO$, which leads to a uniform value in all elements of PO during DLRM inference. This problem comes from the fact that CM loses locality information.

To address the above problem, we introduce two CMs instead of one, defined in Equation 5. Compared with Equation 4 using $d_{CM}(PO, W)$, Equation 5 adds a term $d_{CM}(W, PO)$. Using Equation 5, minimizing the object function requires that not only each point in PO finds the closet point in W, but also each point in W finds the closet point in PO, which prevents the locality problem in Equation 4. Also, Equation 5 normalizes the first and second terms by $\frac{1}{|PO|}$ and $\frac{1}{|W|}$ respectively to enable meaningful comparison across training iterations for optimization.

$$\begin{aligned} dist(PO, W) = \alpha \times \frac{1}{|PO|} \times d_{CM}(PO, W) \quad + \\ (1 - \alpha) \times \frac{1}{|W|} \times d_{CM}(W, PO) \end{aligned} \tag{5}$$

Equation 5 introduces a hyperparameter, α ($\alpha \in (0, 1)$). α balances the contributions of the two terms to $dist(PO, W)$. Since the first term directly minimizes the distance from PO to W, we put more weight to it ($\alpha = 0.7$). Given the length of the prefetch model output $|PO|$, the ratio $|W|/|PO|$ is a user-specified hyperparameter. With this ratio, W is determined. We evaluate the sensitivity of model accuracy to this ratio.

VI. RecMG in Practice

A. Model Offline Training

The caching and prefetch models use the same training data but different ground-truth during offline training due to their distinct goals for memory optimization.To generate the ground-truth labels, we first collect traces of embedding-vector accesses from DLRM inferences. Each trace is then fed into `optgen` [35], which determines what would have been cached if Belady's algorithm [14] were used for caching, providing the minimum miss ratio when temporal locality is optimally exploited. `optgen` generates a new trace (called a caching trace) based on a user-specified buffer size, where each element is either "1" or "0", indicating if the corresponding vector should

stay in the buffer. We set the buffer size in `optgen` to 80% of the GPU buffer capacity to ensure sufficient space for placing prefetched embedding vectors. The caching trace serves as the ground-truth for training the caching model. The prefetch trace, derived from the caching trace, consists of embedding vectors leading to cache misses, which serves as the ground-truth for prefetch model training.

In practice, we periodically retrain the caching and prefetch models when the DLRM requires retraining, which typically happens when embedding tables are updated in production because of the changes in content popularity across different domains or time periods.

B. Model Deployment

We implement RecMG based on TorchRec [58], leveraging its software-managed cache buffer to bring high thread-level parallelism for processing of embedding vectors. We change TorchRec's buffer management policy from LRU/LFU to the policies generated by our caching and prefetch models. Specifically, the GPU buffer is co-managed by the two ML models. Each buffer entry is an embedding vector accompanied by metadata indicating the element's status, which enables efficient buffer management and elimination. The caching model handles embedding-vector eviction, while the prefetch model guides the insertion of embedding vectors.

At the end of each batch of DLRM inference, the `load_embeddings()` function (Algorithm 1) is executed. When a chunk of embedding vectors is accessed, their indices form a sequence that serves as input to the caching and prefetch models. The caching model assigns either high or low priority to each vector based on its output (Lines 4-7 in Algorithm 1). Simultaneously, the prefetching model prefetches each vector in its output into the GPU buffer (Lines 9-13) and sets its priority to high (Line 14) to prevent premature eviction. If necessary, `gpu_buffer_populate()` is called to ensure sufficient space for prefetching.

The metadata `priority` (Line 5 in Algorithm 1) reuses the metadata space in TorchRec without incurring extra space overhead. The priority is determined by `eviction_speed`, which balances the impact of the caching and prefetching models on eviction. In our evaluation, we set the `eviction_speed` to 4, inspired by the RRIP hardware prefetcher algorithm [38]. A larger `eviction_speed` value allows the prefetched embeddings to stay longer in the GPU buffer, while those placed by the caching model are evicted sooner. Although the `eviction_speed` does not affect the accuracy of the caching and prefetching models, it influences the overall system hit rate.

Algorithm 2 shows the embedding vector eviction algorithm based on the priorities generated in the caching model.

C. System Implementation

We integrate the caching and prefetch models to a DLRM implementation from Meta [49], [50], [58]. To save GPU cycles for DLRM inferences and avoid throughput loss, the two models are executed on CPU. When a batch of DLRM inference requests arrives, the caching and prefetch models take them as input, outputting (1) the caching priority for the embeddings fetched in the current batch and (2) embeddings to prefetch for the subsequent inference batches, respectively. The output information is then appended to the fetched embeddings and sent to the GPU, where it will be used in the GPU buffer for loading and evicting embedding vectors, as detailed in Section VI-B. The additional communication from CPU to GPU is minimal because the caching state is represented by only one bit per embedding vector. Since DLRM processes millions of embeddings, the communication volume is just a few hundred KB. Figure 6 shows the workflow.

The major challenge of our system implementation lies in reducing the performance impact of the inferences of the two models on DLRM inference. We use: (1) pipeline execution and less synchronization between CPU and GPU, (2) maximizing thread-level parallelism on CPU, and (3) quantization.

Algorithm 1 : function `load_embeddings`

1: **Input:** Caching model output C, prefetch model output P, and the most recently accessed trunk T.
2:
3: //set the priority for the rows in T
4: **for** i = 0 to sizeof(T) **do**
5: priority[T[i]] = C[i] + $eviction\_speed$
6: **end for**
7:
8: //prefetching
9: **for** i = 0 to sizeof(P) **do**
10: **if** the buffer is full **then**
11: gpu_buffer_populate()
12: **end if**
13: fetch(P[i])
14: priority[P[i]] = $eviction\_speed$
15: **end for**

Algorithm 2 : function `gpu_buffer_populate`

1: **Input:** Cached embedding trunk T, and its priority array $priority$.
2:
3: $evict\_id = 0$
4: **for** i = 0 to sizeof(T) **do**
5: **if** priority[T[i]] $<$ priority[$T[evict\_id]$] **then**
6: $evict\_id = i$
7: **end if**
8: $priority[T[i]] = max(0, priority[T[i]] - 1)$
9: **end for**
10: $T[evict\_id]$ is evicted

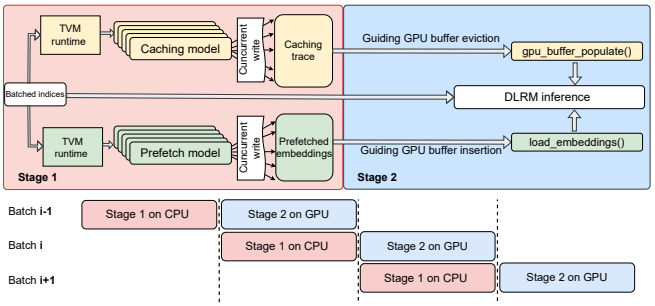

Fig. 6. Applying the caching and prefetch models to the DLRM inference.

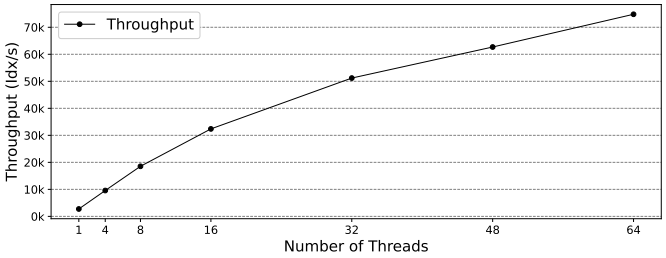

Fig. 7. Caching/prefetching model throughput with different thread numbers

With the **pipeline execution**, the DLRM serves the DLRM inference batch i on GPU, while the caching and prefetch models infer the results for the future DLRM inference batch $i+1$ on CPU. The executions of the two models and DLRM are overlapped. In the case that the inference time of the two models on CPU is longer than the DLRM inference time on GPU for a batch i, the DLRM inference does not wait for the CPU completion. Instead, GPU moves on to the next DLRM inference batch $i+1$, and CPU moves on to infer for the future DRLM inference batch $i+2$. This design results in less synchronization between GPU and CPU. As a result, the states of some cached items cannot be updated by the two models. But this does not impact the effectiveness of the original caching policy for the GPU buffer.

We **maximize thread-level parallelism** of the two models on CPU. This is implemented by wrapping up a batch of DLRM inference requests into n inference requests, and sending them to CPU (where n is the number of idle CPU cores). Each request is served by one thread and the n requests are served in parallel. Alternatively, we can use multiple threads to serve each inference request in parallel and use less inference requests on CPU. But we do not find performance benefits when doing so, comparing with using one thread per inference request. This may be because of the lack of instruction-level parallelism and frequent thread launch overhead.

Based on the throughput results in the Figure 7, it becomes evident why using one thread per inference request is more beneficial than using multiple threads to serve each request. The near-linear increase in throughput as the number of threads increases indicates that the system effectively handles thread-level parallelism without significant overhead. Using idle CPU cores and reducing unnecessary overhead, this approach maximizes throughput for inference tasks.

Besides the above techniques, we aggressively employ vectorization based on AVX512 instructions and use C++ for implementation. Overall, we get more than 10× performance improvement, compared with no optimization.

VII. Evaluation

A. Methodology

Datasets. We evaluate RecMG with five datasets [26]. Each includes 856 embedding tables and records more than 500 million accesses to the embedding tables, which represent memory access patterns in Meta production. The datasets differ in terms of embedding table IDs and row IDs which are most frequently accessed. Such differences in datasets reflect variations in user behavior and content popularity across different domains or time periods.

Baseline Strategies. We compare RecMG against six caching policies including rule-based caching LRU, LFU, SRRIP, adaptive policy DRRIP [38], and reuse distance prediction based caching policies including Mockingjay [69] and Hawkeye [36]. We further compare RecMG against seven prefetchers, including spatial prefetcher Bingo [1], [10], temporal prefetcher Domino [8], ML-based prefetchers Voyager [71], which is built on LSTM layers, and TransFetch [96], which is built on transformer layers, and state-of-the-art delta-based prefetcher Berti [55], offset-based prefetcher Best Offset Prefetcher (BOP) [52], and reinforcement learning based prefetcher coordinator Micro-Armed Bandit (MAB) [30] which orchestrates multiple simple prefetchers. To evaluate these policies, we treat each embedding-vector index as a memory address and use the access sequence of embedding-vector indices as memory access traces. Some strategies leverage Program Counter (PC) or Instruction Pointer (IP) to distinguish memory accesses and capture implicit semantic locality for improving prediction accuracy. Since DLRM inference operations lack PC information, we use embedding table IDs as proxies for PC/IP when evaluating these localization-based prediction policies.

GPU buffer. The GPU buffer size significantly impacts the evaluation results of caching and prefetching. For a fair comparison, unless otherwise specified, we consistently set the GPU buffer size to 20% of the unique embedding vectors in each dataset. This setting aligns with the power law distribution of embedding vector accesses [4], ensuring that most frequently accessed embedding vectors have opportunities to be cached in GPU buffer.

Default configurations of RecMG. Unless indicated otherwise, the lengths of the input and output sequences are 15 and 5, respectively, and the evaluation window size is 15; there is one LSTM stack in the caching model, and two LSTM stacks in the prefetch model.

Evaluation platform. We use a server equipped with a 48-core Intel Xeon Gold 5318Y CPU@2.10GHz with 1.2TB host memory and an NVIDIA A100 GPU with 40GB memory. Given the CPU memory constraint, we evaluate 256 EMBs consuming 900GB memory.

B. Model Evaluation

Caching model. We first study the effectiveness of the caching model. We compare the number of cache hits using five datasets with different caching policies, including fully-associative LRU, 32-way LRU, 32-way LFU, and an approximation of the optimal caching policy `optgen` [14]. Both 32-way LRU and LFU are commonly used in production DLRM embedding vector caching policies. Figure 8 shows the results.

Figure 8 indicates that the optimal caching policy provides, on average, 67% more cache hits than LRU or LFU caching policies. However, the optimal caching policy cannot be used

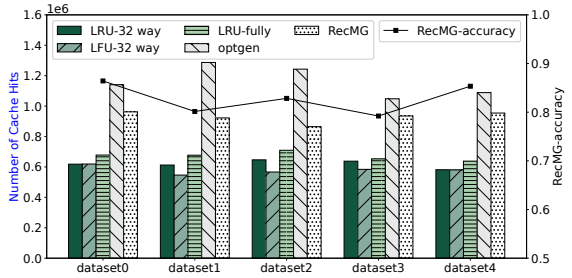

Fig. 8. Comparison in terms of cache hits between LRU, RecMG, and optgen.

online since it requires information about future accesses. Our caching model achieves an accuracy of 83%. As a result, the caching model increases the number of cache hits by at least 38% on all five datasets compared to using LRU or LFU with various associativities. The outstanding performance of the caching model comes from its training process, which uses the optimal caching decision (`optgen`) as ground truth, allowing the model to approximate the optimal policy.

Prefetch model. We study the quality of the prefetch model with 25-million records of embedding vector accesses in the five datasets. We first compare prefetch sequence prediction correctness, which measures the percentage of prefetched embeddings (output sequences) that will be needed within the evaluation window size of future accesses. Voyager uses one-hot vector to label the prefetch address where the vector length is the total number of unique embedding vectors in the datasets. Using one-hot vector works well in the context of labeling address offsets within a page, because the number of offsets is small, but cannot work in the context of labeling embedding vectors, because the length of one-hot vector is at the scale of millions and training Voyager using this vector as output leads to out-of-memory (even on CPU with 512GB DDR). So we compare the prefetch model with Bingo, Domino and TransFetch.

Figure 9 shows the prediction correctness results. The correctness of the spatial prefetcher (Bingo) is less than 0.1%, which is aligned with our observation that there is few spatial locality in the embedding-table accesses during DLRM inferences. The temporal prefetcher (Domino) records cache miss history with multiple streams to capture multiple prefetch targets. Those multiple streams are called metadata. We set the metadata memory overhead as 10% of the unique indices accessed, which is large enough for Domino to record history information for prefetching. Nevertheless, the prefetch accuracy of Domino is only 0.3%. The ML-based prefetcher (TransFetch) achieves 10% on average accuracy because it cannot handle a large amount of embedding vectors within one embedding table. In contrast, the accuracy of our prefetch model is 37% on average for the five datasets, significantly larger than that of all baseline solutions. Our prefetch accuracy metric focuses on prediction quality and does not directly reflect runtime performance benefits as it does not account for cache behavior. For a more comprehensive evaluation, we analyze the actual cache and prefetch hits compared to baseline approaches, with results shown in Figure 14.

We further compare Bingo, Domino, TransFetch and our prefetch model in terms of coverage, which is defined in Equation 2. See Figure 10. Our prefetch model largely outperforms Bingo and Domino by 400x and 190x on average, respectively, because of the ability of the attention mechanism in the prefetch model to capture implicit correlations between vectors (even though they are not accessed often). RecMG outperforms TransFetch by 10% on average for model coverage, although both RecMG and TransFetch are good at predicting embedding vectors to be accessed in the near future.

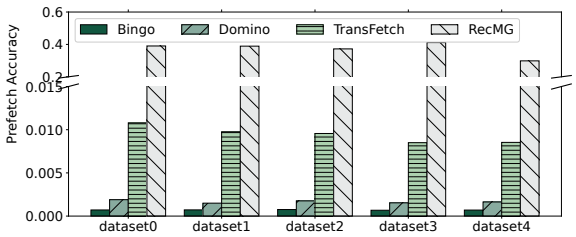

Fig. 9. Comparison in terms of prefetch accuracy between Bingo, Domino, TransFetch, and RecMG.

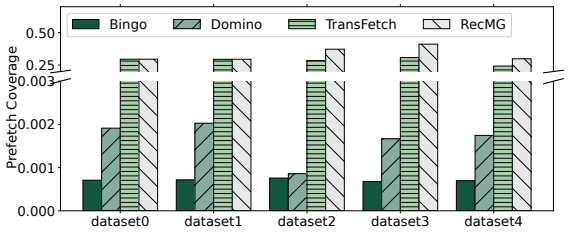

Fig. 10. Comparison in terms of prefetch coverage between Bingo, Domino, TransFetch, and RecMG.

TABLE II
AVERAGE COST OF PREDICTING NEXT WILL BE ACCESS EMBEDDING VECTOR

	Bingo	**Domino**	**Voyager**	**Tranfetch**	**RecMG**
Cost	32μs	100μs	1521μs	1052μs	92μs

We further evaluate the cost of predicting the next embedding vector for all the baselines. Table II shows the results. The rule-based prefetcher Bingo can generate a prediction in just 32 μs, while Domino, which needs to scan historical access information, takes 100 μs. To enable a fair comparison, we use CPU to generate all predictions (i.e., using CPU for ML-based prefetch inference). ML-based prefetchers are more expensive to use, with Voyager and TransFetch requiring 16$\times$ and 10.6$\times$ longer than the prefetch model in RecMG to generate one prediction. This is because Voyager has a large search space, and TransFetch is based on a transformer, which requires much more computation compared to RecMG. The evaluation results show the prefetch model in RecMG strikes a good balance between cost and accuracy in predicting the next accessed embedding vector.

Fig. 11. Evaluating the effectiveness of using the decoupling design and Chamfer measure.

TABLE III
TRAINING TIME, MODEL SIZE, AND ACCURACY OF USING VARIOUS NUMBER OF LSTM STACKS. RECMG USES ONE AND TWO LSTM STACKS FOR CACHING PREFETCH MODEL RESPECTIVELY.

# LSTM Stack	Caching Model: Training Time (mins)	Caching Model: Model Size (# of params)	Caching Model: Acc	Prefetch Model: Training Time (mins)	Prefetch Model: Model Size (# of params)	Prefetch Model: Acc
1	**429**	**37,055**	**80%**	765	38,290	39%
2	603	45,055	82%	**962**	**74,290**	**50%**
3	745	63,055	83%	1,059	110,290	50%

C. Ablation Study

In the prefetch model, RecMG decouples the evaluation window from the output of the prefetch model to improve the prefetch hit rate, and uses customized loss function. We evaluate the effectiveness of this design. We compare the prefetch model with a baseline using the L2 loss and the evaluation window length equal to the output length. Figure 11 shows the results using the dataset0 which has 100 million records of embedding table accesses. With the baseline, the training loss does not decrease after 10 training steps. With RecMG, the training loss continuously decreases with more training samples, which shows the effectiveness of our design.

D. Sensitivity Study

Number of LSTM stacks. The backbone of the caching model and prefetch model are seq2seq LSTM stacks with attention. We evaluate how the model accuracy is sensitive to the number of LSTM stacks with dataset-0. Table III shows the results. As the number of LSTM stacks increases, the accuracy of the cache model slightly increases (less than 5%). In contrast, the prefetch model is sensitive to the number of LSTM stacks. By increasing the number of LSTM stacks, the accuracy of the prefetch model increases by 11%.

Although adding more LSTM stacks is helpful to improve model accuracy, they are not free - adding them leads to a larger model and longer training time. We report the training time and model sizes for caching and prefetch models with various number of LSTM stacks in Table III. As the caching model adds one LSTM stack, the number of parameters in the caching model increases by at least 21%, and the training time increases by 24%. As the prefetch model adds one LSTM stack, the number of parameters in the prefetch model increases by at least 48%, and the training time increase by 21%. Considering both model training/deploy overhead and model accuracy, we use one LSTM stack for the caching model, and two LSTM stacks for the prefetch model.

Evaluation of window size. We evaluate how the accuracy of the prefetch model is sensitive to the evaluation window size with dataset-0. We change the evaluation window size. Figure 12 shows the results. Compared with the case of evaluation window size equal to the output sequence length, increasing the evaluation window size increases the model accuracy by at least 39%. In contrast, as the evaluation window size is larger than 3 times of the output sequence length, the prediction coverage does not increase. Hence, RecMG sets the evaluation window size equal to 3 times of the output sequence length to achieve both high accuracy and coverage.

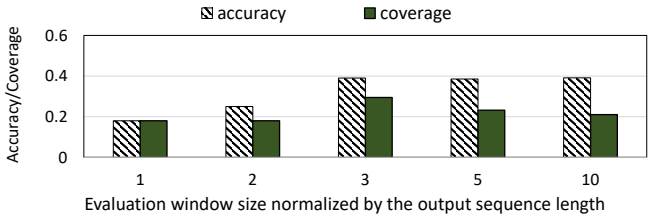

Fig. 12. Evaluating the sensitivity of prefetch model accuracy to the evaluation window size.

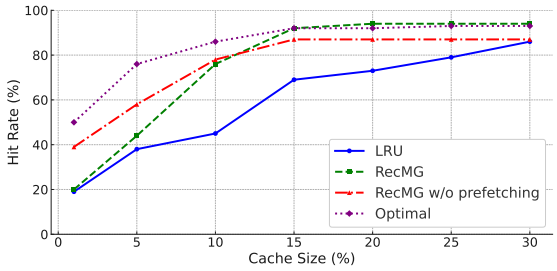

Fig. 13. The impact of buffer size on access hit rate.

Evaluation of GPU buffer size. We evaluate the access hit rate with RecMG, RecMG without prefetch model, fully associative LRU, and `optgen` (the Belady's algorithm) with various GPU buffer sizes. We set the GPU buffer size to 1% to 30% of unique embedding-vectors in the dataset-0. We use a GPU buffer emulator (a tool to play various caching algorithms to evaluate the functionality of buffer management). Figure 13 shows the result. We observe that RecMG outperforms LRU and RecMG without prefetch when the cache size is above 10%, and is close to the optimal when the cache size is above 15%. However, the prefetch model is not very helpful when the cache size is too small (e.g., less than 10%), because the caching model (guiding the frequently accessed embedding-vectors) largely dominates the performance.

E. Using Caching Model and Prefetch Model Together

We evaluate how embedding vectors are accessed, and compare with Domino, Bingo, TransFetch and LRU+PF (i.e., the fully associative LRU plus our prefetch model). LRU+PF is

a case of using a single ML model (instead of two) for memory optimization. We perform the evaluation on the GPU buffer emulator. The buffer size is set as 20% of unique embedding-vectors referenced by DLRM. We break down the accesses to the GPU buffer(s) into three components: (1) buffer hit because of the caching policy (i.e., LRU or the caching model); (2) buffer hit because of the prefetch model (not the caching policy); and (3) on-demand fetches from CPU memory.

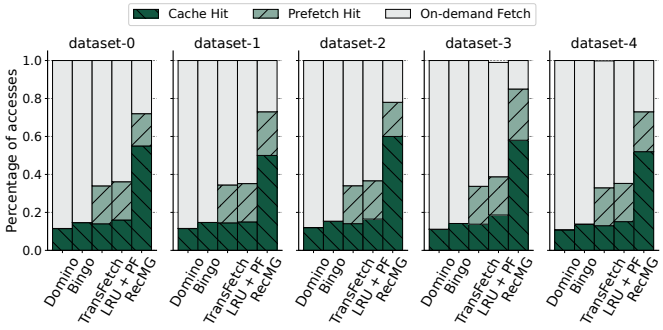

Fig. 14. Embedding vector accesses breakdown for Domino, Bingo, TransFetch, LRU+PF, and RecMG.

Figure 14 shows the results. RecMG reduces the number of access misses on the critical path by $4.5\times$, $4.8\times$, $2.8\times$ and $2.7\times$ on average, compared to Domino, Bingo, TransFetch and LRU+PF, respectively. We observe that the caching model in RecMG significantly improves the access hit rate by $2.2\times$, $2.8\times$, $1.45\times$ and $1.45\times$ on average, compared with the caching mechanisms in Domino, Bingo, TransFetch and LRU, respectively. Moreover, the prefetch model in RecMG brings additional improvements. It boasts a $1.9\times$ increase in the access hit rate, compared to LRU+PF.

RecMG's prefetch model is particularly effective at the use of resource. Unlike Domino, which consumes excessive GPU buffer capacity for metadata recording, RecMG utilizes all available buffer space for embedding vectors, improving the number of access hit by $283\times$. Also, the prefetch model in RecMG significantly outperforms the spatial prefetcher Bingo, increasing the access hit by $16,000\times$ on average, due to lacking of spatial locality in embedding vector accesses.

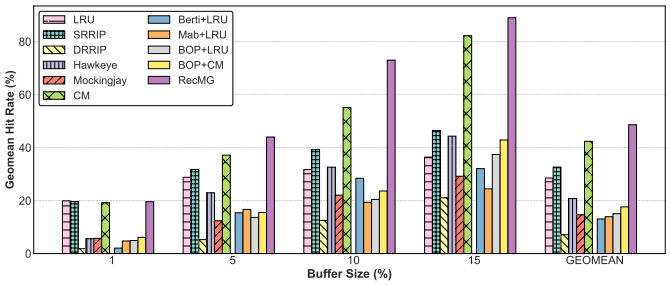

Fig. 15. Geometric mean of GPU buffer hit rates across three datasets. Buffer size is defined as percentage of unique embedding vectors. The rightmost GEOMEAN group shows the geometric mean across all buffer sizes. For each buffer size, patterned bars represent caching strategies and solid bars represent prefetchers. LRU refers to ChampSim with a 32-way LRU cache, and CM stands for caching model only.

Comparison with advanced caching and prefetching strategies. Leveraging Champsim [31], we further compare RecMG with advanced cache replacement strategies (SRRIP, DRRIP [38], and Mockingjay [69], Hawkeye [36]) and prefetchers (Berti [55], Best offset prefetcher (BOP) [52], and Micro-Armed-Bandit (MAB) [30]). To apply the above existing works to DLRM inferences traces, we treat each access as a read operation. We map the embedding table ID to Program Counter (PC) or Instruction Pointer (IP), and map embedding vector to the address targeted by the load instruction. The embedding vectors, typically larger than traditional cache lines, are treated as atomic units for replacement decisions. Using ChampSim configured with a 32-way set-associative cache, we compare hit rates across strategies for various GPU buffer sizes. Baseline prefetchers (Berti, BOP, MAB) are evaluated with 32-way LRU or caching model as the underlying cache replacement policy. Figure 15 shows the geometric mean performance across dataset-0, dataset-1, and dataset-2.

At 1% buffer size, LRU, SRRIP and caching model achieve similar performance. More complex caching strategies like Hawkeye and Mockingjay show significantly lower performance. This aligns with the short-term skewed nature of embedding table accesses during DLRM inferences, where access patterns are primarily determined by ad-hoc user behaviors rather than code structure. PC/IP-independent strategies like LRU, SRRIP, and caching model prove more effective in this scenario. Furthermore, all prefetchers (Berti+LRU, BOP+LRU, and MAB+LRU) achieve lower hit rates than basic LRU at this small buffer size. This suggests that prefetching approaches, regardless of their prediction mechanisms, are less effective at small buffer sizes due to short-term skewed nature of embedding table accesses.

As buffer size increases from 5% to 15%, caching strategies can better leverage long-term structured patterns in the embedding table accesses. Such patterns emerge when multiple users sharing interests access related features across different embedding tables. Therefore, all caching strategies show improved hit rates. Specifically, SRRIP outperforms LRU by 14% on geomean across different buffer sizes, demonstrating the effectiveness of PC/IP-independent replacement policies. At larger buffer sizes (10% and 15%), Hawkeye also shows improved performance. Hawkeye achieves 10% higher hit rates than LRU at 15% buffer size. Caching model demonstrates superior performance, achieving 29% higher hit rate than LRU and 30% higher than SRRIP on geomean.

Prefetching strategies show varied effectiveness when compared to their underlying cache strategies when buffer sizes increase. BOP+LRU shows the best scaling among traditional prefetchers, achieving 3% higher hit rate than LRU at 15% buffer size. However, Berti+LRU and Mab+LRU demonstrate limited benefits over LRU. This is because Berti's delta-based prefetching and MAB's reinforcement learning coordination of traditional prefetchers are both designed for regular program patterns, making them less effective for the dynamic, user-driven patterns of embedding accesses. Meanwhile, a simpler single global offset design in BOP capture the coarse-grained

spatial locality better when given sufficient buffer space. In general, RecMG consistently outperforms all baselines across different GPU buffer sizes. As the buffer size is 15%, RecMG outperforms other baselines by 20% - 425%, across three datasets, which is attributed to its two ML models that efficiently capture the needs of both immediate and far-future accesses to embedding tables.

We further quantify prefetcher effectiveness with three metrics: prefetch utilization rate, total prefetches, and prefetch hits. Prefetch utilization rate measures useful embedding prefetches to total prefetches issued. Prefetch hits measures number of successful cache accesses to prefetched embeddings. For evaluation, Berti and Mab only use 32-way LRU as their underlying cache replacement policy, and BOP are evaluated with both 32-way LRU and caching model. Table IV shows prefetcher statistics across three datasets. RecMG achieves the highest buffer hit rate by combining efficient prefetching (35% utilization rate) with selective prefetch issuance (2 million prefetches), resulting in 1 million prefetch hits. While BOP+LRU shows higher buffer hit rate than LRU, BOP+CM underperforms compared to caching model alone, which is reflected in the prefetch statistics. Although BOP maintains similar total prefetches (3M) with both policies, its prefetch hits drop substantially from 0.7M with LRU to 0.3M with caching model, and its utilization rate decreases from 12% to 9%, indicating BOP's prefetching strategy conflicts with CM's replacement decisions. Berti and Mab perform worse than baseline LRU because their aggressive and inefficient prefetching issues 10M to 12M prefetches with very low utilization rates between 5% and 6%, causing significant cache pollution.

TABLE IV
PREFETCHER STATISTICS. "CM" AND "PM" ARE SHORT FOR CACHING MODEL AND PREFETCHING MODEL, RESPECTIVELY.

Strategy	Utilization Rate (geomean)	Total Prefetches (arith mean)	Prefetch Hits (arith mean)
Berti + LRU	6%	12M	0.8M
Mab + LRU	5%	10M	0.6M
BOP + LRU	12%	3M	0.7M
BOP + CM	9%	3M	0.3M
RecMG	35%	2M	1M

F. Speedup of End-to-End DLRM Inference

Real-world DLRM performance evaluation. We evaluate the inference time of DLRM using different buffer management solutions. We set the DLRM inference batch size to 512, involving over 600K embedding vectors in one batch. The buffer holds 2.3 million embedding vectors, which are approximately 18% of the unique embedding vectors involved in the inferences. "LRU" is a 32-way set-associative LRU cache. "CM" is RecMG using the caching model alone. We break down the execution time, including (1) the time to send embedding vectors and caching priority according to the caching and prefetch models (labeled as "embedding copy to GPU"), (2) GPU computation, (3) GPU buffer management overhead (including on-demand fetches of embedding vectors), and (4) others (e.g., synchronization within FBGEMM). Figure 16 shows the result.

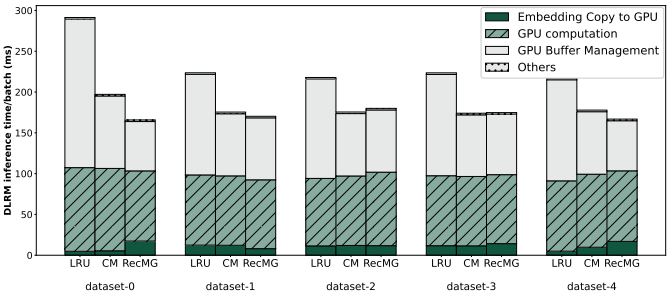

Fig. 16. Performance breakdown for DLRM inference for one inference batch. "CM" is short for caching model.

In general, RecMG effectively reduces the inference time. Compared with the LRU, RecMG reduces the inference time by 31% on average (up to 43%). The major performance benefit comes from the reduction of on-demand fetches during the buffer management: RecMG reduces it by 29.8% on average, compared to the LRU. The prefetch model, targeting accesses to embedding-vectors difficult to predict, reduces the inference time by up to 16%, compared to using the caching model alone. RecMG effectively increases the number of access hit on the GPU buffer (not shown in Figure 16). Across all datasets (geometric mean), RecMG outperforms the LRU by 49.9%. Without the prefetch model, RecMG shows a geometric mean improvement of 41.6% over LRU across datasets.

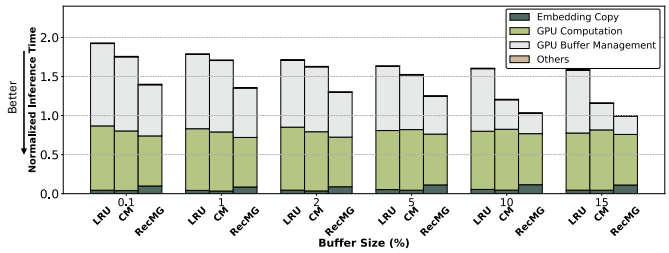

Fig. 17. Normalized DLRM inference time when using RecMG with various GPU buffer sizes. The buffer size is defined in terms of percentage of unique embedding-vectors in the dataset. "CM" is short for caching model.

Figure 17 shows the normalized DLRM inference time on dataset-0 across different buffer sizes. All results are normalized to the execution time when using RecMG with 15% buffer size. The performance benefit of RecMG includes two components. The benefit from the caching model is shown as the difference between LRU and using caching model only (CM), while the benefit from the prefetch model is represented by the difference between CM and RecMG. The buffer size is small (e.g., 0.1%), the prefetching model contributes 67.5% of the performance benefit in RecMG, while the caching model accounts for only 32.5%. This is because with limited buffer space, prefetching is more crucial for reducing memory access latency. As the GPU buffer size increases from 0.1% to 15%, the caching model's contribution to RecMG's benefit grows

to 72.3%. This is because the larger buffer size allows the caching model to retain more frequently accessed embeddings, reducing the need of costly CPU memory accesses. The impact of improved caching is evident in the significant reduction of GPU buffer management time. Specifically, when the GPU buffer size increases from 0.1% to 15%, the GPU buffer management time reduces by 2.79 × and 2.91 × in using caching model only and RecMG, respectively.

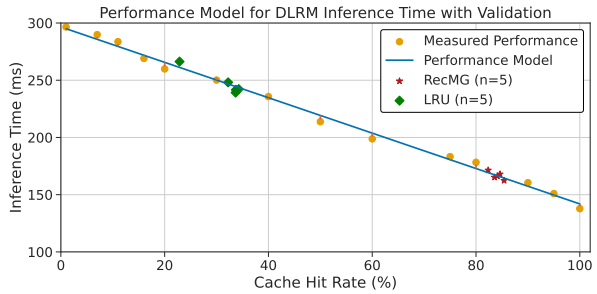

Fig. 18. A performance model captures the linear relationship between DLRM inference time (ms) and cache hit rate. The model is validated using LRU and RecMG cache policies across five different datasets (n=5).

Model based performance analysis. To quantitatively understand the impact of caching on DLRM inference latency, we develop a performance model. Specifically, we construct multiple synthetic traces from dataset-0, each containing 10M access records derived from reordering 10,000 unique embedding vectors, with each trace designed to achieve a cache hit rate between 0-100%. The orange dots in Figure 18 show the measured DLRM inference time under different cache hit rates, and the blue line represents our linear performance model. The averaged root mean square error (RMSE) between the performance model and the measured performance is less than 3.75 ms, which is only 1.7% difference. To further validate the correctness of the performance model, we evaluate it using DLRM's default 32-way LRU cache policy and RecMG, testing each policy with 5 different datasets (shown as green and red marks in Figure 18). Validation results from LRU and RecMG show less than 3.6% deviation from model predictions, demonstrating the robustness and generality of the performance model.

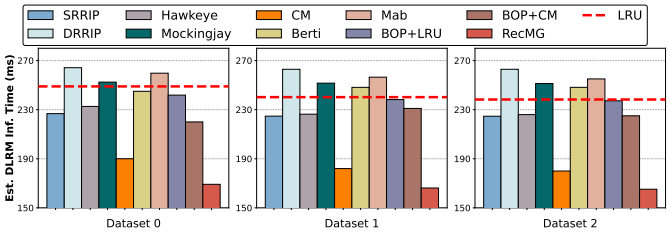

Fig. 19. Estimated DLRM inference latency across ten caching/prefetch strategies on three datasets. The dashed line shows inference time with 32-way LRU caching, the default solution for DLRM embeddings. "CM" denotes caching model only.

We apply our performance model to estimate DLRM inference time for various cache replacement and prefetching strategies, using their measured cache hit rates under the same buffer size (15% of unique embedding vectors). As shown in Figure 19, compared to the default GPU buffer management solution of 32-way LRU (red dashed line), approaches including SRRIP, Hawkeye, caching model only, BOP+LRU, and RecMG improve performance by 7%, 5.8%, 24%, 1.4% and 31%, on average across three datasets with geometric mean. In contrast, DRRIP, Mockingjay, Berti (with LRU) and Mab (with LRU) show comparable or slightly worse performance than LRU, with performance degradation of 8.7%, 3.9%, 2% and 6% respectively. Among these strategies, BOP is promising to complement caching mechanisms with prefetching. We further study the impact of BOP by combining it with different caching strategies (BOP+LRU and BOP+CM). Compared to baseline LRU, BOP+LRU and BOP+CM achieve 1.4% and 7% performance improvements respectively, averaged across three datasets using geometric mean.

Note that our performance model reflects the best-case performance for most baseline strategies, as it does not account for their additional decision-making overheads. For example, Mockingjay, Hawkeye, Berti and BOP require both extra metadata storage space and lookup time. Given the limited tiered memory capacity in DLRM inference scenario, these metadata storage and lookup overheads can significantly impact performance.

VIII. Related Work

Memory optimization for DLRM. Existing works propose various techniques to mitigate the memory capacity challenges posed by EMBs in DLRM, including hierarchical caching [4], [12], [25], [37], [53], [68], [81], [90], tensor train compression [93], mixed precision embedding [90], compiler-based acceleration [57], and domain-specific approximation [42]. Specifically, RecShard [68] and cDLRM [12]) target DLRM training. They profile the embedding-table access traces and collect statistical features from the traces to systematically plan embedding table partitioning and caching in memory tiers. However, DLRM inferences do not have information on the accesses of the embedding table in advance.

RecMG is different from the above efforts from multiple perspectives. First, RecMG is for DLRM inferences, unlike some efforts that focus on DLRM training [12], [25], [28], [43], [53], [68], [79], [90], [93], which makes offline memory profiling infeasible. Second, most of existing efforts [25], [37], [53], [81] focus on hot embedding vectors with high reuse, and cannot effectively handle sparse accesses or accesses with long reuse distance to other embedding vectors. Third, RecMG does not change DLRM model, and hence does not impact model accuracy. Finally, in contrast to DLRM inference optimizations on CPU [37], which focus on small caches (tens of MBs) and rely on the power-law distribution of embedding accesses for easier prefetching, RecMG targets DLRM inference on GPU, dealing with higher computation speeds and the challenge of timely data prefetching for long reuse distances.

AutoScratch [29] uses a cache replacement strategy for GPU L2 cache for ML inference, leveraging predictable and

TABLE V
COMPARISON OF VARIOUS CACHING/PREFETCHING TECHNIQUES

	Usage scenario	Scalability concerns	Algorithmic prefetcher	PC/IP indication
RRIP [38], BOP [52]	General	N	N	N
Bingo [11], Prodigy [76]	General	N	N	Y
HawkEye [36], Mockingjay [69]	General	Y	N	Y
Berti [55], Helper Thread [16], Domino [9]	General	Y	N	Y
AutoScratch [29]	DL inference	Y	Y	N
PrefEdge [56]	Graph	Y	Y	N
RecMG	DLRM Inference	N	Y	N

invariant access patterns interleaved between weights and activations. AutoScratch cannot be used for prefetching in DLRM, because of variant access patterns in EMBs.

Data prefetchers. Rule-based data prefetchers [9], [11], [51], [70], [75] predict memory accesses with temporal or spatial relations. However, rule-based data prefetchers are not efficient in predicting irregular data accesses.

To address this limitation, ML-based prefetchers [32], [44], [45], [66], [69], [73], [74], [95] leverage machine learning techniques to learn complex relations between memory accesses. Existing works have explored various problem formulations, such as treating cache-line prefetch as a classification problem [32], [71] or a regression problem [44], [95]. While these approaches have shown promising results in improving prefetch accuracy, most of them do not fully consider the practical challenges of deploying the ML models in real-world systems with limited resources.

Table V compares RecMG with a set of representative efforts. We make the comparison from multiple perspectives. RecMG leverages limited "algorithm" knowledge that there are implicit correlations between embedding vector accesses between users. This "algorithm" knowledge is different from the algorithm knowledge in the existing work, such as node connectivity in a graph as in PrefEdge [56] or program semantics as in Prodigy [76]. DLRM does not offer structural data or program semantics that can be leveraged by prefetch.

Many existing efforts [9], [11], [36], [55], [69], [76] rely on PC as an indicator of future memory accesses. This makes sense in the context of regular programs, but does not make sense in the context of DLRM, because the embedding vector accesses in DLRM are related to the user behavior of DLRM (e.g., following the trending news), not the DLRM program itself. Using table ID as PC, we can apply those existing efforts, but such a "PC" does not show the tendency to access the same addresses or have a predictable address delta.

Existing efforts [11], [36] store historical memory access traces in a lookup table to guide prefetching. Such a table-based approach may not be scalable when applied to embedding vector accesses, because of the tadeoff between memory consumption and prefetch effectiveness.

Tiered memory. Tiered memory systems [33], [46], [47], [60]–[65], [78], [84]–[89], [91], [92] manage multiple memory components with different properties (e.g., latency, cost, and capacity). In essence, RecMG is a memory tiering solution. RecMG is the first ML-guided memory tiering solution for DLRM.

IX. CONCLUSIONS

In this paper, we use ML for prefetching and caching of embedding vectors based on modeling of implicit correlations between consecutive accesses to embedding vectors. RecMG largely reduces the on-demand fetches.

ACKNOWLEDGMENT

This work was partially supported by U.S. National Science Foundation (2104116, 2316202 and 2348350) and the Chameleon Cloud. This project was partially supported by Meta. We would like to thank the anonymous reviewers, as well as our shepherd, for their feedback on the paper.

REFERENCES

[1] "Bingo," https://github.com/bakhshalipour/Bingo.
[2] S. Ainsworth and L. Mukhanov, "Triangel: A high-performance, accurate, timely on-chip temporal prefetcher," 2024. [Online]. Available: https://arxiv.org/abs/2406.10627
[3] N. Ardalani, C.-J. Wu, Z. Chen, B. Bhushanam, and A. Aziz, "Understanding Scaling Laws for Recommendation Models," 2022. [Online]. Available: https://arxiv.org/abs/2208.08489
[4] E. K. Ardestani, C. Kim, S. J. Lee, L. Pan, V. Rampersad, J. Axboe, B. Agrawal, F. Yu, A. Yu, T. Le, H. Yuen, S. Juluri, A. Nanda, M. Wodekar, D. Mudigere, K. Nair, M. Naumov, C. Peterson, M. Smelyanskiy, and V. Rao, "Supporting Massive DLRM Inference Through Software Defined Memory," *CoRR*, vol. abs/2110.11489, 2021.
[5] B. Athiwaratkun and J. W. Stokes, "Malware classification with lstm and gru language models and a character-level cnn," in *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2017, pp. 2482–2486.
[6] G. Ayers, H. Litz, C. Kozyrakis, and P. Ranganathan, "Classifying Memory Access Patterns for Prefetching," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.
[7] P. Badjatiya, S. Gupta, M. Gupta, and V. Varma, "Deep learning for hate speech detection in tweets," in *Proceedings of the 26th International Conference on World Wide Web Companion*, ser. WWW '17 Companion. Republic and Canton of Geneva, CHE: International World Wide Web Conferences Steering Committee, 2017, p. 759–760. [Online]. Available: https://doi.org/10.1145/3041021.3054223
[8] M. Bakhshalipour, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Domino Temporal Data Prefetcher," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018.
[9] M. Bakhshalipour, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Domino Temporal Data Prefetcher," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Vienna: IEEE, Feb. 2018, pp. 131–142. [Online]. Available: http://ieeexplore.ieee.org/document/8327004/
[10] M. Bakhshalipour, M. Shakerinava, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Bingo Spatial Data Prefetcher," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019.

[11] M. Bakhshalipour, M. Shakerinava, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Bingo Spatial Data Prefetcher," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Washington, DC, USA: IEEE, Feb. 2019, pp. 399–411. [Online]. Available: https://ieeexplore.ieee.org/document/8675188/

[12] K. Balasubramanian, A. Alshabanah, J. D. Choe, and M. Annavaram, "CDLRM: Look Ahead Caching for Scalable Training of Recommendation Models," in *ACM Conference on Recommender Systems*, 2021.

[13] H. Barrow, J. Tenenbaum, R. Boles, and H. Wolf, "Parametric Correspondence and Chamfer Matching: Two New Techniques for Image Matching," in *International Joint Conference on Artificial Intelligence*, 1977.

[14] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Systems Journal*, vol. 5, no. 2, pp. 78–101, 1966.

[15] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., vol. 33. Curran Associates, Inc., 2020, pp. 1877–1901. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf

[16] L. Ceze, K. Strauss, J. Tuck, and J. Torrellas, "Cava: Using checkpoint-assisted value prediction to hide l2 misses," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006. [Online]. Available: https://iacoma.cs.uiuc.edu/iacoma-papers/mteac01.pdf

[17] Q. Chen, X. Zhu, Z.-H. Ling, S. Wei, H. Jiang, and D. Inkpen, "Enhanced LSTM for natural language inference," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, R. Barzilay and M.-Y. Kan, Eds. Vancouver, Canada: Association for Computational Linguistics, Jul. 2017, pp. 1657–1668. [Online]. Available: https://aclanthology.org/P17-1152

[18] P. Covington, J. Adams, and E. Sargin, "Deep neural networks for youtube recommendations," in *Proceedings of the 10th ACM Conference on Recommender Systems*, ser. RecSys '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 191–198. [Online]. Available: https://doi.org/10.1145/2959100.2959190

[19] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," *CoRR*, vol. abs/1810.04805, 2018. [Online]. Available: http://arxiv.org/abs/1810.04805

[20] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "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)*, J. Burstein, C. Doran, and T. Solorio, Eds. Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 4171–4186. [Online]. Available: https://aclanthology.org/N19-1423

[21] C. Ding and Y. Zhong, "Predicting whole-program locality with reuse distance analysis," in *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Diego, CA, June 2003, pp. 245–257.

[22] T. D. Doudali, S. Blagodurov, A. Vishnu, S. Gurumurthi, and A. Gavrilovska, "Kleio: A Hybrid Memory Page Scheduler with Machine Intelligence," in *International Symposium on High-Performance Parallel and Distributed Computing*, 2019.

[23] T. D. Doudali and A. Gavrilovska, "Coeus: Clustering (A)like Patterns for Practical Machine Intelligent Hybrid Memory Management," in *IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 2022.

[24] A. Eisenman, K. K. Matam, S. Ingram, D. Mudigere, R. Krishnamoorthi, K. Nair, M. Smelyanskiy, and M. Annavaram, "Check-N-Run: a checkpointing system for training deep learning recommendation models," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. Renton, WA: USENIX Association, Apr. 2022, pp. 929–943. [Online]. Available: https://www.usenix.org/conference/nsdi22/presentation/eisenman

[25] A. Eisenman, M. Naumov, D. Gardner, M. Smelyanskiy, S. Pupyrev, K. M. Hazelwood, A. Cidon, and S. Katti, "Bandana: Using Non-volatile Memory for Storing Deep Learning Models," in *Conference on Machine Learning and Systems*, 2018.

[26] Facebook Research, "Embedding Lookup Synthetic Dataset," https://github.com/facebookresearch/dlrm_datasets.

[27] facebook research, "Deep learning recommendation model for personalization and recommendation systems," 2022. [Online]. Available: https://github.com/facebookresearch/dlrm

[28] J. Fang, G. Zhang, J. Han, S. Li, Z. Bian, Y. Li, J. Liu, and Y. You, "A frequency-aware software cache for large recommendation system embeddings," *arXiv preprint arXiv:2208.05321*, 2022.

[29] Y. Fu, E. Bolotin, A. Jaleel, G. Dalal, S. Mannor, J. Subag, N. Korem, M. Behar, and D. Nellans, "AutoScratch: ML-Optimized Cache Management for Inference-Oriented GPUs," *Proceedings of Machine Learning and Systems*, vol. 5, pp. 495–512, 2023.

[30] G. Gerogiannis and J. Torrellas, "Micro-Armed Bandit: Lightweight & Reusable Reinforcement Learning for Microarchitecture Decision-Making," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023.

[31] N. Gober, G. Chacon, L. Wang, P. V. Gratz, D. A. Jimenez, E. Teran, S. Pugsley, and J. Kim, "The Championship Simulator: Architectural Simulation for Education and Competition," 2022. [Online]. Available: https://arxiv.org/abs/2210.14324

[32] M. Hashemi, K. Swersky, J. A. Smith, G. Ayers, H. Litz, J. Chang, C. Kozyrakis, and P. Ranganathan, "Learning Memory Access Patterns," in *International Conference on Machine Learning*, 2018.

[33] Y. Huang and D. Li, "Performance Modeling for Optimal Data Placement on GPU with Heterogeneous Memory Systems," in *IEEE International Conference on Cluster Computing*, 2017.

[34] A. Jain and C. Lin, "Linearizing irregular memory accesses for improved correlated prefetching," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013.

[35] A. Jain and C. Lin, "Back to the Future: Leveraging Belady's Algorithm for Improved Cache Replacement," in *International Symposium on Computer Architecture (ISCA)*, 2016.

[36] A. Jain and C. Lin, "Back to the future: Leveraging belady's algorithm for improved cache replacement," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016, pp. 78–89.

[37] R. Jain, S. Cheng, V. Kalagi, V. Sanghavi, S. Kaul, M. Arunachalam, K. Maeng, A. Jog, A. Sivasubramaniam, M. T. Kandemir, and C. R. Das, "Optimizing cpu performance for recommendation systems at-scale," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ser. ISCA '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3579371.3589112

[38] A. Jaleel, K. B. Theobald, S. C. Steely, and J. Emer, "High performance cache replacement using re-reference interval prediction (rrip)," *SIGARCH Comput. Archit. News*, vol. 38, no. 3, p. 60–71, jun 2010. [Online]. Available: https://doi.org/10.1145/1816038.1815971

[39] J. Jumper, R. Evans, A. Pritzel, T. Green, M. Figurnov, O. Ronneberger, K. Tunyasuvunakool, R. Bates, A. Ž'ıdek, A. Potapenko *et al.*, "Highly accurate protein structure prediction with alphafold," *Nature*, vol. 596, no. 7873, pp. 583–589, 2021.

[40] W.-C. Kang, D. Z. Cheng, T. Yao, X. Yi, T. Chen, L. Hong, and E. H. Chi, "Learning to Embed Categorical Features without Embedding Tables for Recommendation," in *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, 2021.

[41] L. Ke, U. Gupta, B. Y. Cho, D. Brooks, V. Chandra, U. Diril, A. Firoozshahian, K. Hazelwood, B. Jia, H.-H. S. Lee, M. Li, B. Maher, D. Mudigere, M. Naumov, M. Schatz, M. Smelyanskiy, X. Wang, B. Reagen, C.-J. Wu, M. Hempstead, and X. Zhang, "RecNMP: Accelerating Personalized Recommendation with Near-Memory Processing," in *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2020.

[42] D. H. Kurniawan, R. Wang, K. S. Zulkifli, F. A. Wiranata, J. Bent, Y. Vigfusson, and H. S. Gunawi, "EVStore: Storage and Caching Capabilities for Scaling Embedding Tables in Deep Recommendation Systems," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023.

[43] Y. Lee, H. Kim, and M. Rhu, "PreSto: An In-Storage Data Preprocessing System for Training Recommendation Models," 2024. [Online]. Available: https://arxiv.org/abs/2406.14571

[44] P. Li, Y. Guo, and Y. Gu, "Predicting reuse interval for optimized web caching: an lstm-based machine learning approach," in *Proceedings of*

the International Conference on High Performance Computing, Networking, Storage and Analysis, ser. SC '22. IEEE Press, 2022.

[45] E. Z. Liu, M. Hashemi, K. Swersky, P. Ranganathan, and J. Ahn, "An imitation learning approach for cache replacement," 2020.

[46] J. Liu, D. Li, and J. Li, "Athena: High-Performance Sparse Tensor Contraction Sequence on Heterogeneous Memory," in *International Conference on Supercomputing (ICS)*, 2021.

[47] J. Liu, J. Ren, R. Gioiosa, D. Li, and J. Li, "Sparta: High-Performance, Element-Wise Sparse Tensor Contraction on Heterogeneous Memory," in *Principles and Practice of Parallel Programming*, 2021.

[48] Q. Lu and F. Guo, "Personalized information recommendation model based on context contribution and item correlation," *Measurement*, vol. 142, pp. 30–39, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0263224118311497

[49] Meta, "Deep Learning Recommendation Model for Personalization and Recommendation Systems," https://github.com/facebookresearch/dlrm.

[50] Meta, "Facebook GEneral Matrix Multiplication," https://github.com/pytorch/FBGEMM.

[51] P. Michaud, "Best-offset hardware prefetching," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 469–480.

[52] P. Michaud, "Best-offset hardware prefetching," in *Proceedings of the 2016 International Symposium on High-Performance Computer Architecture (HPCA)*, Barcelona, Spain, 2016. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01254863

[53] D. Mudigere, Y. Hao, J. Huang, A. Tulloch, S. Sridharan, X. Liu, M. Ozdal, J. Nie, J. Park, L. Luo, J. A. Yang, L. Gao, D. Ivchenko, A. Basant, Y. Hu, J. Yang, E. K. Ardestani, X. Wang, R. Komuravelli, C. Chu, S. Yilmaz, H. Li, J. Qian, Z. Feng, Y. Ma, J. Yang, E. Wen, H. Li, L. Yang, C. Sun, W. Zhao, D. Melts, K. Dhulipala, K. R. Kishore, T. Graf, A. Eisenman, K. K. Matam, A. Gangidi, G. J. Chen, M. Krishnan, A. Nayak, K. Nair, B. Muthiah, M. khorashadi, P. Bhattacharya, P. Lapukhov, M. Naumov, L. Qiao, M. Smelyanskiy, B. Jia, and V. Rao, "High-performance, Distributed Training of Large-scale DLRM," *arXiv*, 2021.

[54] M. Naumov, D. Mudigere, H. M. Shi, J. Huang, N. Sundaraman, J. Park, X. Wang, U. Gupta, C. Wu, A. G. Azzolini, D. Dzhulgakov, A. Mallevich, I. Cherniavskii, Y. Lu, R. Krishnamoorthi, A. Yu, V. Kondratenko, S. Pereira, X. Chen, W. Chen, V. Rao, B. Jia, L. Xiong, and M. Smelyanskiy, "Deep Learning Recommendation Model for Personalization and Recommendation Systems," *CoRR*, vol. abs/1906.00091, 2019. [Online]. Available: http://arxiv.org/abs/1906.00091

[55] A. Navarro-Torres, B. Panda, J. Alastruey-Benedé, P. Ibáñez, V. Viñals Yúfera, and A. Ros, "Berti: An accurate local-delta data prefetcher," in *Proceedings of the 55th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '22. IEEE Press, 2023, p. 975–991. [Online]. Available: https://doi.org/10.1109/MICRO56248.2022.00072

[56] K. Nilakant, V. Dalibard, A. Roy, and E. Yoneki, "Prefedge: Ssd prefetcher for large-scale graph traversal," in *Proceedings of International Conference on Systems and Storage*, ser. SYSTOR 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 1–12. [Online]. Available: https://doi.org/10.1145/2611354.2611365

[57] Z. Pan, Z. Zheng, F. Zhang, R. Wu, H. Liang, D. Wang, X. Qiu, J. Bai, W. Lin, and X. Du, "RECom: A Compiler Approach to Accelerating Recommendation Model Inference with Massive Embedding Columns," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2024.

[58] PyTorch, "TorchRec," https://github.com/pytorch/torchrec.

[59] pytorch, "Support uvm for embedding," 2022. [Online]. Available: https://github.com/pytorch/torchrec/blob/main/examples/sharding/uvm.ipynb

[60] J. Ren, J. Luo, I. Peng, K. Wu, and D. Li, "Optimizing Large-Scale Plasma Simulations on Persistent Memory-based Heterogeneous Memory with Effective Data Placement Across Memory Hierarchy," in *International Conference on Supercomputing (ICS)*, 2021.

[61] J. Ren, J. Luo, K. Wu, M. Zhang, H. Jeon, and D. Li, "Sentinel: Efficient Tensor Migration and Allocation on Heterogeneous Memory Systems for Deep Learning," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2020.

[62] J. Ren, S. Rajbhandari, R. Y. Aminabadi, O. Ruwase, S. Yang, M. Zhang, D. Li, and Y. He, "ZeRO-Offload: Democratizing Billion-Scale Model Training," in *USENIX Annual Technical Conference*, 2021.

[63] J. Ren, D. Xu, J. Ryu, K. Shin, D. Kim, and D. Li, "MTM: Rethinking Memory Profiling and Migration for Multi-Tiered Large Memory Systems," in *European Conference on Computer Systems*, 2024.

[64] J. Ren, S. Yang, D. Xu, J. Li, Z. Zhang, C. Navasca, C. Wang, G. H. Xu, and D. Li, "DyNN-Offload: Enabling Large Dynamic Neural Network Training with Learning-based Memory Management," in *International Symposium on High-Performance Computer Architecture (HPCA)*, 2024.

[65] J. Ren, M. Zhang, and D. Li, "HM-ANN: Efficient Billion-Point Nearest Neighbor Search on Heterogeneous Memory," in *Conference on Neural Information Processing Systems (NeurIPS)*, 2020.

[66] L. V. Rodriguez, F. Yusuf, S. Lyons, E. Paz, R. Rangaswami, J. Liu, M. Zhao, and G. Narasimhan, "Learning cache replacement with CACHEUS," in *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, Feb. 2021, pp. 341–354. [Online]. Available: https://www.usenix.org/conference/fast21/presentation/rodriguez

[67] D. Schuff, M. Kulkarni, and V. Pai, "Accelerating multicore reuse distance analysis with sampling and parallelization," in *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, 2010, pp. 53–64.

[68] G. Sethi, B. Acun, N. Agarwal, C. Kozyrakis, C. Trippel, and C.-J. Wu, "RecShard: Statistical Feature-Based Memory Optimization for Industry-Scale Neural Recommendation," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022.

[69] I. Shah, A. Jain, and C. Lin, "Effective mimicry of belady's min policy," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022, pp. 558–572.

[70] Z. Shi, X. Huang, A. Jain, and C. Lin, "Applying deep learning to the cache replacement problem," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 413–425. [Online]. Available: https://doi.org/10.1145/3352460.3358319

[71] Z. Shi, A. Jain, K. Swersky, M. Hashemi, P. Ranganathan, and C. Lin, "A Hierarchical Neural Model of Data Prefetching," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[72] S. Somogyi, T. F. Wenisch, A. Ailamaki, and B. Falsafi, "Spatio-Temporal Memory Streaming," in *International Symposium on Computer Architecture*, 2009.

[73] Z. Song, D. S. Berger, K. Li, and W. Lloyd, "Learning relaxed belady for content distribution network caching," in *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation*, ser. NSDI'20. USA: USENIX Association, 2020, p. 529–544.

[74] A. Srivastava, A. Lazaris, B. Brooks, R. Kannan, and V. K. Prasanna, "Predicting Memory Accesses: The Road to Compact ML-Driven Prefetcher," in *Proceedings of the International Symposium on Memory Systems*, 2019.

[75] N. Talati, K. May, A. Behroozi, Y. Yang, K. Kaszyk, C. Vasiladiotis, T. Verma, L. Li, B. Nguyen, J. Sun, J. M. Morton, A. Ahmadi, T. Austin, M. O'Boyle, S. Mahlke, T. Mudge, and R. Dreslinski, "Prodigy: Improving the memory latency of data-indirect irregular workloads using hardware-software co-design," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 654–667.

[76] N. Talati, K. May, A. Behroozi, Y. Yang, K. Kaszyk, C. Vasiladiotis, T. Verma, L. Li, B. Nguyen, J. Sun, J. M. Morton, A. Ahmadi, T. Austin, M. O'Boyle, S. Mahlke, T. Mudge, and R. Dreslinski, "Prodigy: Improving the memory latency of data-indirect irregular workloads using hardware-software co-design," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 654–667.

[77] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30. Curran Associates, Inc., 2017. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[78] X. S. Wang, J. Liu, J. Wu, S. Yang, J. Ren, B. Shankar, and D. Li, "Performance Characterization of CXL Memory and Its Use Cases," in *International Parallel and Distributed Processing Symposium*, 2025.

[79] Z. Wang, Y. Wang, J. Deng, D. Zheng, A. Li, and Y. Ding, "RAP: Resource-aware Automated GPU Sharing for Multi-GPU Recommendation Model Training and Input Preprocessing," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2024.

[80] T. Widiyaningtyas, I. Hidayah, and T. B. Adji, "User Profile Correlation-based Similarity (UPCSim) Algorithm in Movie Recommendation System," *Journal of Big Data*, vol. 52, 2021.

[81] M. Wilkening, U. Gupta, S. Hsia, C. Trippel, C.-J. Wu, D. Brooks, and G.-Y. Wei, "RecSSD: Near Data Processing for SSD Based Recommendation Inference," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[82] H. Wu, K. Nathella, J. Pusdesris, D. Sunwoo, A. Jain, and C. Lin, "Temporal Prefetching Without the Off-Chip Metadata," in *IEEE/ACM International Symposium on Microarchitecture*, 2019.

[83] H. Wu, K. Nathella, D. Sunwoo, A. Jain, and C. Lin, "Efficient Metadata Management for Irregular Data Prefetching," in *International Symposium on Computer Architecture (ISCA)*, 2019.

[84] K. Wu, Y. Huang, and D. Li, "Unimem: Runtime Data Management on Non-Volatile Memory-based Heterogeneous Main Memory," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017.

[85] K. Wu, J. Ren, and D. Li, "Runtime Data Management on Non-volatile Memory-based Heterogeneous Memory for Task-parallel Programs," in *International Conference for High Performance Computing, Networking, Storage, and Analysis*, 2018.

[86] Z. Xie, W. Dong, J. Liu, I. Peng, Y. Ma, and D. Li, "MD-HM: Memoization-based Molecular Dynamics Simulations on Big Memory System," in *International Conference on Supercomputing (ICS)*, 2021.

[87] Z. Xie, J. Liu, J. Li, and D. Li, "Merchandiser: Data Placement on Heterogeneous Memory for Task-Parallel HPC Applications with Load-Balance Awareness," in *Proceedings of the Symposium on Principles and Practices of Parallel Programming (PPoPP)*, 2023.

[88] D. Xu, Y. Feng, K. Shin, D. Kim, H. Jeon, and D. Li, "Efficient Tensor Offloading for Large Deep-Learning Model Training based on Compute Express Link," in *36th ACM/IEEE International Conference for High Performance Computing, Performance Measurement, Modeling and Tools*, 2024.

[89] D. Xu, J. Ryu, K. Shin, P. Su, and D. Li, "FlexMem: Adaptive page profiling and migration for tiered memory," in *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. Santa Clara, CA: USENIX Association, Jul. 2024, pp. 817–833. [Online]. Available: https://www.usenix.org/conference/atc24/presentation/xu-dong

[90] J. A. Yang, J. Huang, J. Park, P. T. P. Tang, and A. Tulloch, "Mixed-Precision Embedding Using a Cache," in *Conference on Machine Learning and Systems*, 2020.

[91] S. Yang, K. Wu, Y. Qiao, D. Li, and J. Zhai, "Algorithm-directed crash consistence in non-volatile memory for hpc," in *IEEE Cluster Computing*, 2017.

[92] S. Yang, M. Zhang, and D. Li, "Buffalo: Enabling large-scale gnn training via memory-efficient bucketization," in *Proceedings of the 2025 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2025.

[93] C. Yin, B. Acun, X. Liu, and C. Wu, "TT-Rec: Tensor Train Compression for Deep Learning Recommendation Models," in *Conference on Machine Learning and Systems*, 2021.

[94] C. Zeng, L. Luo, Q. Ning, Y. Han, Y. Jiang, D. Tang, Z. Wang, K. Chen, and C. Guo, "FAERY: An FPGA-accelerated embedding-based retrieval system," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA: USENIX Association, Jul. 2022, pp. 841–856. [Online]. Available: https://www.usenix.org/conference/osdi22/presentation/zeng

[95] P. Zhang, R. Kannan, and V. K. Prasanna, "Phases, modalities, spatial and temporal locality: Domain specific ml prefetcher for accelerating graph analytics," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3581784.3607043

[96] P. Zhang, A. Srivastava, A. V. Nori, R. Kannan, and V. K. Prasanna, "Fine-grained address segmentation for attention-based variable-degree prefetching," in *Proceedings of the 19th ACM International Conference on Computing Frontiers*, ser. CF '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 103–112. [Online]. Available: https://doi.org/10.1145/3528416.3530236

[97] W. Zhao, D. Xie, R. Jia, Y. Qian, R. Ding, M. Sun, and P. Li, "Distributed Hierarchical GPU Parameter Server for Massive Scale Deep Learning Ads Systems," in *Proceedings of Machine Learning and Systems*, 2020.