Fleche: An Efficient GPU Embedding Cache for Personalized Recommendations

Minhui Xie[†] Youyou Lu[†*] Jiazhen Lin[†]
Qing Wang[†] Jian Gao[†] Kai Ren[‡] Jiwu Shu[†]

xmh19@mails.tsinghua.edu.cn,luyouyou@tsinghua.edu.cn

[†]*Department of Computer Science and Technology, BNRist, Tsinghua University* [‡]*Kuaishou*

Abstract

Deep learning based models have dominated current production recommendation systems. However, the gap between CPU-side DRAM data accessing and GPU processing still impedes their inference performance. GPU-resident cache can bridge this gap, but we find that existing systems leave the benefits to cache the embedding table, a huge sparse structure, on GPU unexploited. In this paper, we present Fleche, a holistic cache scheme with detailed designs for efficient GPU-resident embedding caching. Fleche (1) uses one cache backend for all embedding tables to improve the total cache utilization, and (2) merges small kernel calls into one unitary call to reduce the overhead of kernel maintenance (e.g., kernel launching and synchronizing). Furthermore, we carefully design the cache query workflow for finer-grain parallelism. Evaluations with real-world datasets show that compared with the prior art, Fleche significantly improves the throughput of embedding layer by 2.0 – 5.4×, and gets up to 2.4× speedup of end-to-end inference throughput.

CCS Concepts: **• Software and its engineering → Memory management**; **• Information systems → Recommender systems**.

Keywords: Memory management, GPU cache, Deep learning recommendation models, Embedding lookup

ACM Reference Format:
Minhui Xie, Youyou Lu, Jiazhen Lin, Qing Wang, Jian Gao, Kai Ren, and Jiwu Shu. 2022. Fleche: An Efficient GPU Embedding Cache for Personalized Recommendations. In *Seventeenth European Conference on Computer Systems (EuroSys '22), April 5–8, 2022, RENNES, France.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3492321.3519554

[*]Youyou Lu is the corresponding author.

1 Introduction

Exploding information production calls for the recommendation system to extract the essentials from massive information sources and generate customized streams for users. The key determinants of recommendation quality include not only the prediction accuracy that has always been the algorithmic optimization target, but also the system performance (e.g., latency and throughput). Given the same requirement of service-level agreement (SLA), a higher-performance recommendation system can examine more candidate items, and thus is more likely to follow users' interests.

Emerging Deep Learning Recommendation Model [21, 35] (DLRM) comprises a huge portion in today's recommendation systems. DLRM usually has a two-part structure (Figure 1), an ultra-huge embedding layer (sparse part) with hundreds of gigabytes of memory footprint, and several fully-connected layers (dense part), a.k.a., multilayer perceptron (MLP), running on GPU with only hundreds of megabytes of memory footprint. The embedding layer contains dozens of *embedding tables*, mapping high-dimensional categorical inputs (i.e., IDs) to latent low-dimensional dense representations (called *embedding vectors*, or *embeddings* for short). These embedding tables are typically stored as hash tables on CPU-side DRAM due to the limited GPU memory capacity. Previous studies from both academia [31, 32] and industry [17, 26, 27, 29, 45] show that the CPU-side DRAM bandwidth scarcity caused by irregular and sparse accesses of embedding tables has become the major performance impediment of recommendation models. For example, Alibaba reports that over 60% of prediction latency in their production models comes from the embedding layers [26, 27]. Furthermore, model developers are driving the growth of model capacity and complexity for better accuracy. Facing with the ever-increasing embedding size, the bandwidth scarcity problem will be more acute in future.

To tackle the bandwidth scarcity, caching on GPUs' high bandwidth memory (HBM) can be efficient owing to the high access locality of real-world workloads. However, we find that the existing cache scheme (called the *static per-table cache structure*) leaves several GPU-resident cache's benefits unexploited. This cache scheme maintains a fixed-size cache table for each embedding table to prevent massive data movement during tables' repartition. Our evaluation

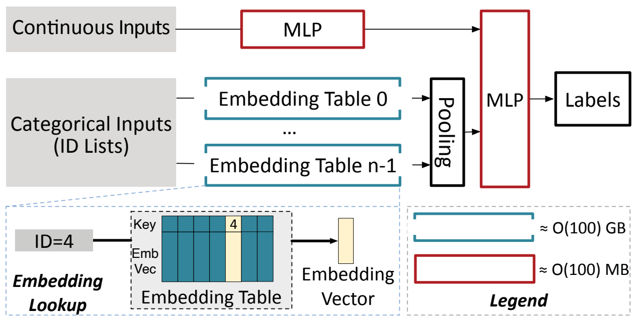


Figure 1. The structure of Deep Learning Recommendation Model (DLRM).

of a highly-optimized industry-level recommendation inference system, NVIDIA HugeCTR-Inference [7] (HugeCTR for short, see §2.2), shows that this design suffers from two critical deficiencies: cache under-utilization and overhead of kernel maintenance.

- **Issue 1: cache under-utilization.** Compared with the optimal case, HugeCTR suffers from 11%-42% degradation in hit rate across different datasets. This huge gap arises from the structural defect of *static* per-table cache.
- **Issue 2: overhead of kernel maintenance.** Up to 70% of cache query time is spent on tasks other than kernel execution (called *kernel maintainence* in this paper, including CPU launching, context initialization, and CPU synchronization), due to excessive small cache-query kernels.

We propose Fleche [1], a novel cache scheme with detailed designs for efficient GPU-resident embedding caching. The key idea of Fleche is *co-designing the cache structure and workflow to improve cache utilization and reduce query time.* Fleche follows the existing two-layer architecture, a GPU-HBM layer caching hot embeddings and a CPU-DRAM layer storing all embeddings [2].

To address Issue 1, we propose *flat cache* (FC). Different from the previous scheme, FC does not partition cache tables for each embedding table. Instead, it lets all embedding tables share one global cache backend by re-encoding input IDs from all embedding tables to *flat keys* with a unified format. Thus, cache tables of different embedding tables can elastically expand or contract, further exploiting the cache memory for a higher hit rate. Moreover, with *size-aware coding*, Fleche tailors the flat key format for FC to reduce key conflicts and accuracy losses brought by re-encoding.

To address Issue 2, based on the observation that most kernel launches are not calls to different kernel functions but to the same function with different parameters, we propose an efficient fusing method for these kernel calls, namely *self-identified kernel fusion*. It merges small kernel calls into a unitary call, and lets each thread in the kernel identify which kernel call it should serve originally. With this technique, Fleche reduces the number of query kernels to one, alleviating kernel maintenance overhead, while also keeping the semantics of multi-cache-table query.

Moreover, we adopt two techniques to optimize the cache query workflow. First, Fleche decouples the copy of hit embeddings from indexing FC. With this optimization, Fleche enjoys the parallelism between the GPU-HBM and CPU-DRAM layers. Specifically, Fleche can query the CPU-DRAM layer for missing embeddings ahead without waiting for the completion of copying hit embeddings. Second, Fleche adopts a *unified index* technique to offload partial index of CPU-DRAM layer to GPU. Thus, we can further sidestep indexing overhead of missing keys and enjoy the turbo-boost lookup performance of GPU.

We evaluate Fleche with three public real-world datasets. Compared with HugeCTR, Fleche significantly improves the throughput of embedding layer by 2.0 – 5.4×, and gets up to 2.4× speedup of end-to-end inference throughput with the same cache size.

In summary, our work makes the following contributions:

- We profile the existing GPU-resident cache scheme with real-world recommendation workloads, and discover two main culprits which hinder the embedding performance.
- We propose Fleche, a holistic GPU-resident cache scheme for efficient embedding lookup based on our profiling results.
- We implement Fleche on HugeCTR, evaluate it with real-world workloads and show its effectiveness.

2 Background & Motivation

We first describe the structure of Deep Learning Recommendation Models (DLRMs) in §2.1, and then analyze the existing cache scheme with experiments in §2.2.

2.1 Deep Learning Recommendation Models

Overcoming the limited expression ability of conventional models [15, 30, 34, 39], DLRM has become prevalent both in academia and industry. It leverages neural networks to predict the probability of certain interactions between users and items (e.g., videos), such as the like action in YouTube [14] and the click action in Facebook [36], and delivers those with high probabilities to users.

Model structure. Figure 1 sketches a representative DLRM structure. The model requires two kinds of inputs: continuous and categorical inputs. Continuous inputs (e.g., user age, video click count, recent product sales) represent features with continuous values, while categorical inputs (e.g., user

[1]Short for FLat Embedding caCHE.
[2]We only consider the case where models can be fit into the CPU-DRAM but not into the GPU-HBM. Larger models will be discussed in §5.

ID, video ID and list of favorite videos) are one-hot or multi-hot high-dimensional features and are usually encoded as a list of IDs. The neural network does not usually take in these IDs with huge value domains directly, but utilizes the embedding technique to project feature IDs to dense *embedding vectors* (or *embeddings* for short). Specifically, for each kind of categorical feature, it maintains a hash table with a capacity of c as an *embedding table*, where keys are IDs, and values are d-dimensional embedding vectors. Note that the embedding table count typically ranges from dozens to hundreds in production models [9, 26]. The *embedding lookup* process is similar to the lookup of a hash table with the input ID as the key. After embedding lookup, these embedding vectors of each category are further compressed into one dense vector through *pooling* operation. Finally, all pooled vectors are concatenated and served as the input of MLP together with original dense features. Despite the differences in various concrete designs [13, 20, 33, 37, 38, 40, 47], all DLRMs follow this Embedding-MLP paradigm [23], with the embedding layer capturing low-dimensional representations of IDs and MLP layers learning the latent relationships between input features and output labels.

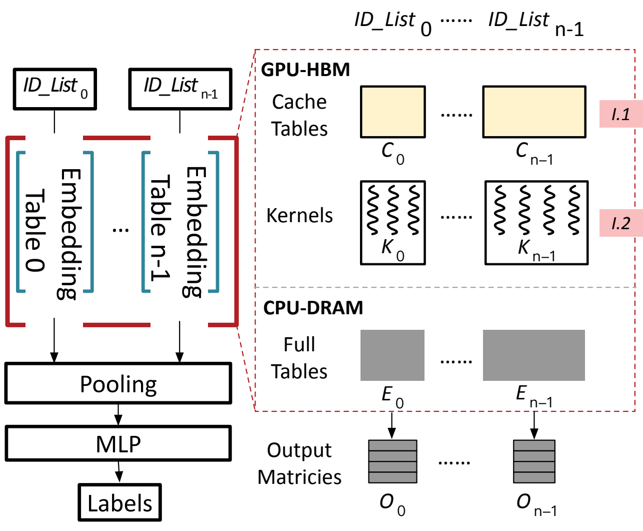
Memory-bounded embedding layers. Many companies (e.g., Alibaba [26, 27] and Facebook [29, 45]) reported that the embedding layers of their production DLRM models account for over 60% of the prediction latency. The underlying reason for this high latency is DRAM bandwidth scarcity [21, 32]. Random lookups of the embedding layer result in a large number of CPU cache misses, and simultaneous accesses to multiple embedding tables by different threads exhaust DRAM's bandwidth [32].

To solve this problem, one promising method [7] is to cache hot embeddings on GPU. It lifts the burden of DRAM with GPU's surplus memory bandwidth and thus performs better than no-caching ones (more than 5× improvement of the embedding layer by our measurement). However, our analysis indicates that there are still two issues which hinder caching performance in existing cache schemes.

2.2 Analysis of Existing Cache Schemes

We begin with describing how existing embedding cache schemes deploy the aforementioned models in the scenario of inference (Figure 2). Suppose there is a DLRM with a set of embedding tables $\{E_0, \ldots, E_{n-1}\}$, where E_i is implemented as a hash table with capacity c_i and value dimension d_i. Here c_i and d_i respectively denote the corpus size and embedding dimension of i-th embedding table E_i. Existing schemes maintain a separate *cache table* for each embedding table, i.e., $\{C_0, \ldots, C_{n-1}\}$, where each C_i is a smaller hash table with capacity s_i and value dimension d_i. E_i resides in the CPU-DRAM layer, while C_i resides in the GPU-HBM layer.

When querying the cache, existing schemes launch a kernel for each cache table C_i with parameters including the input IDs ID_List_i, and the address of the output matrix



Issue	Performance Implication
I.1 Cache under-utilization.	Low cache hit rate.
I.2 Overhead of kernel maintenance.	High GPU query latency.

Figure 2. The sparse part of DLRM and the existing GPU-resident cache scheme.

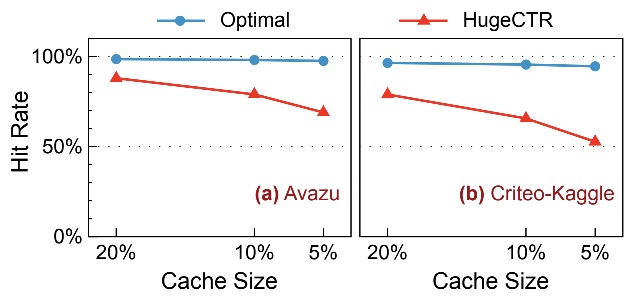


Figure 3. Cache hit rate of HugeCTR with different cache sizes in two real-world datasets, Avazu and Criteo-Kaggle. *Optimal denotes the ideal case where the cache knows all accesses of datasets. 5% means that the cache size is 5% of the size of all embedding tables.*

O_i with shape $len(ID\_List_i) \times d_i$. This kernel searches the embedding cache table C_i and copies hit embeddings to O_i. Once the kernel finishes, CPU gets the missing ID list from GPU, queries E_i, and copies their embeddings to O_i. For kernel concurrency, n kernels are placed into different streams (e.g., cudaStream in CUDA).

To quantitatively analyze existing cache schemes in depth, we conduct a series of experiments on HugeCTR, discovering two main deficiencies.

Issue 1: cache under-utilization. Figure 3 depicts the cache hit rates of HugeCTR with different cache sizes in two real-world datasets, Avazu [3] and Criteo-Kaggle [5]. It also

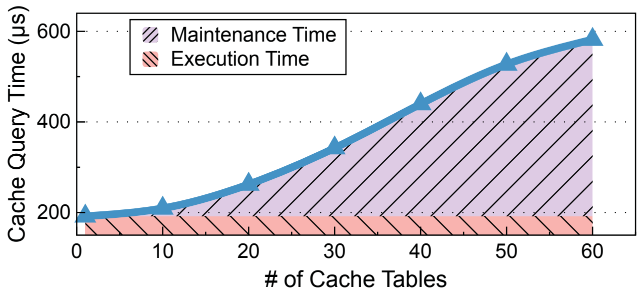


Figure 4. Cache query performance degradation of HugeCTR with the increasing cache table count. *The number of aggregate query IDs of all embedding caches is set to 10K. We use the cache query latency with single embedding table as an approximation to the actual execution time, since all cases query the same total number of IDs.*

plots the theoretical upper limit ("Optimal") of hit rate with the same cache size, where the cache knows all accesses of datasets. We observe that there is quite a huge gap of hit rate between HugeCTR and Optimal. Also, the under-utilization is more severe with smaller cache sizes. Specifically, when the cache size is 5% of the size of all embedding tables, this gap reaches 29% for Avazu and is almost 42% for Criteo-Kaggle.

The root cause of this gap is the structural defect of *static* per-table cache. HugeCTR statically sets the same proportion of cache size for all embedding tables. Since the hotspot size of each embedding table is usually different and constantly changes with time, the static structure tends to cache the local hotspots of each embedding table instead of the global hotspots of all tables, leading to a low hit rate.

Issue 2: overhead of kernel maintenance. We measure kernel execution time and maintenance time of HugeCTR's cache query with a synthetic workload[3]. Here, we regard the time not spent doing actual execution in GPU as the kernel maintenance time, which includes CPU launching, context initialization, CPU synchronization, communication between CPU and GPU, etc. We fix the aggregate number of query IDs to 10K (the results of 1K and 100K are similar), and vary the number of cache tables, n. These query IDs are equally spread to n cache tables. From Figure 4 it is clear that as n increases, the kernel maintenance takes longer time. Specifically, when n comes to 60, maintenance time is over 2× than execution time. Note that the number of embedding tables is commonly greater than 60 in real-world production models. For example, Facebook and Alibaba state that the embedding table counts of their two production models are 61 and 98 respectively [9, 26]. We also optimize HugeCTR using cudaGraph [4] to alleviate the kernel launch overhead and repeat the experiment. The findings are similar.

[3]The synthetic workload follows a power law distribution ($\alpha = -1.2$). The number of embedding tables in real datasets is fixed, which cannot be used for this experiment.

The underlying reason is the excessive small kernels. First, with scattering all IDs to each cache table, the number of IDs carried in each kernel is very small. As a result, maintenance time cannot be hidden by the execution time, and the overall maintenance time is proportional to the number of query kernels. Second, HugeCTR needs to launch excessive kernels whose number is proportional to the number of cache tables, because these kernels have dependencies on the cache table parameters (e.g., table address, size, embedding dimension).

To summarize our discussion so far, existing cache schemes suffer from the *static per-table cache structure*. Undesirable low cache utilization increases the time in the CPU-DRAM layer; excessive small kernels' maintenance overhead increases the time spent in the GPU-HBM layer.

3 Design

Motivated by our analysis, we propose Fleche. Fleche introduces *flat cache* (§3.1) and *self-identified kernel fusion* (§3.2) to improve cache utilization and alleviate kernel maintenance overhead, respectively. Moreover, we adopt two techniques to optimizes the cache query workflow (§3.3).

3.1 Flat Cache

Figure 5a sketches the query process of flat cache (FC). When querying the GPU-resident cache, Fleche re-encodes all feature IDs from different embedding tables to *flat keys* in a uniform format, and then launches one unitary kernel on GPU to query the cache backend.

In this section, we first describe the detailed structure of our FC and the customized design of mapping the original multi-table caches to FC with re-encoding, and then present how FC elastically rescales different embedding cache tables. Finally we analyze the advantages of FC.

Cache table structure. FC is organized as a fashion of key value separation, with a *memory pool* storing all embeddings (i.e., values) from different cache tables, and a GPU-resident index maintaining the mapping from *flat keys* to locations in the memory pool; see Figure 5c. The memory pool inherits the existing slab memory allocator. Fleche avoids memory fragmentation by pre-defining the size of each slab according to the embedding dimension, since all embeddings in an embedding table shares the same size known in advance. Fleche pre-allocates a bulk for the memory pool during booting up and makes finer management inside at runtime. This sidesteps the latency of cudaMalloc API, up to a dozen microseconds. The GPU-resident index can be an arbitrary existing GPU hash index (e.g., MegaKV [42], SlabHash [11]). To implement an approximate LRU algorithm, FC embraces a timestamp into each slot of the index.

FC minimizes the size of metadata to save HBM capacity with domain knowledge. First, FC does not bookkeep any information of embedding sizes unlike traditional key-value stores, because the input IDs are bound to specific embedding

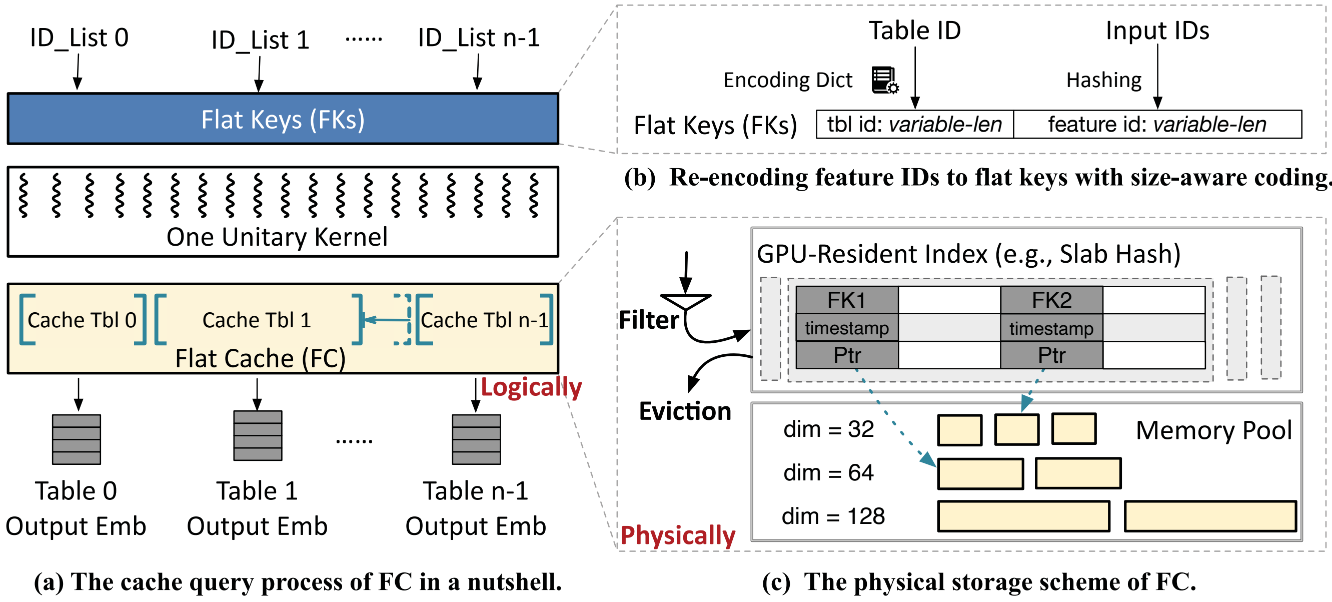


Figure 5. The flat cache (FC) structure of Fleche. *FK: flat key. Emb: embedding.*

tables which imply the size. Second, the per-slot timestamp also acts as a version number to detect concurrent read-write conflicts. Note that the write-write conflicts are resolved by our deduplicating & restoring mechanism (see §4). Thus, we need no extra metadata for concurrency control.

Re-encoding IDs to flat keys with size-aware coding. We unify all IDs from different embedding tables to flat keys. Thus the underlying index of FC is unaware of multi-table structure. With the abstraction of flat keys, all cache tables can share one global cache, to achieve high utilization.

To keep the semantic differences of the embeddings with the same feature ID in different tables (e.g., both the user table and city table may share the same feature ID 212, but the one in the user table represents a user while the other represents New York), we need to re-encode all feature IDs with a suitable key format for FC. One plain method called *fixed-length coding* [41] is to reserve several high bits (e.g., 8 bits) of keys (e.g., int32) for table IDs to identify different embedding tables, and encapsulate the original feature IDs into remaining bits with hashing. In the example above, the key structure is shown as following:

tbl id: 8-bit	feature id (hashed): 24-bit

However, this method simply reserves the same space for all embedding tables despite their different corpus sizes, which suffers from imbalanced utilization of key space. E.g., there are only several cities while the number of users can be billions, leading to severe hash collisions for the user table and under-utilization of the key space for the city table.

To solve this problem, we propose *size-aware encoding* method (see Figure 5b). The key idea is to assign shorter table IDs to larger tables in order to squeeze more space for features. Specifically, similar to the fixed-length coding, the flat key structure is also separated to table ID bits and feature ID (hashed) bits. For each embedding table (sorted ascending by the corpus size), we assign it the longest table ID bits whose remaining feature ID bits are enough for accommodating the key space of this embedding table. Once a table ID is assigned, the future use of all bits prefixed by it should be prohibited, in order to avoid inter-table collision. If there are no available table ID bits for some embedding tables, we reserve several bits and allocate them in proportional to the corpus sizes, which may introduce intra-table collision.

Note that the process of encoding is ultra-fast and at almost no cost, because the mapping metadata can be stored as a hash table with only dozens of entries, and for all feature IDs of one embedding table, it requires only one transformation since they share the same table ID.

Cache replacement & eviction. Fleche performs cache replacement for those missing embeddings. To reduce FC's swap-in-swap-out overhead of excessive less-frequent IDs, we apply a probability-based filter policy [34]. With this policy, each embedding gets swapped into the cache with a certain probability p, so that features that occur less than $1/p$ times would bypass cache on the mathematical expectation.

When the utilization of the memory pool exceeds a certain threshold, FC performs cache eviction with a full table scan and keeps evicting cold embeddings until memory utilization falls below another threshold. Note that there may be a case of *read-after-delete* during eviction, which means the deallocated embeddings are then read by other threads. FC embraces the epoch-based space reclamation [18] to ensure the consistency, which first marks the evicted embedding

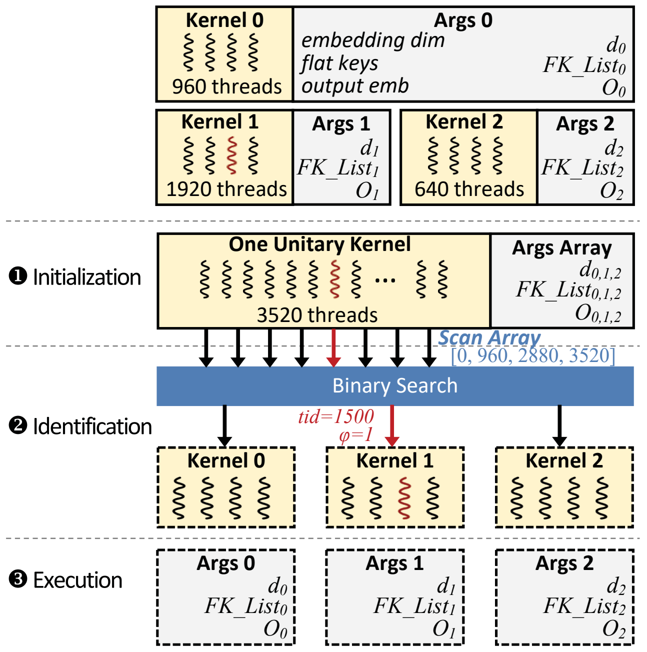


Figure 6. Self-identified kernel fusion method. *In this example, we fuse three tables' cache query kernels, which originally have 960, 1920, 640 threads respectively.*

as deleted logically and delays the real reclamation until a grace period where all readers no longer have access to it.

Advantages. With the flat key abstraction as a shim layer, FC abstracts a multi-table external cache interface, and internally enjoys the benefits of high cache utilization as a single physical cache. All embedding table caches can elastically rescale to capture the hotspots of the whole picture.

3.2 Self-identified Kernel Fusion

Upon the single cache structure of flat cache, we propose an efficient kernel-fusion method to squeeze the number of cache query kernels to only one.

We follow the previous notations (see §2.2) for convenience of expression. To perform cache queries of multiple tables, existing method launches a separate kernel for each set of parameters (including the embedding dimension d_i, input flat key list FK_List_i and output matrix address O_i), which suffers severe kernel maintenance overhead. Since these kernel launches are calls to the same kernel function with various arguments instead of different functions, it provides us the opportunity of kernel fusion.

Here we propose self-identified kernel fusion to squeeze the number of embedding cache query kernels to one, which still supports the original interface of multi-cache queries but alleviates kernel maintenance overhead. Specifically, this method consists of three phases. Suppose the original ith kernel contains m_i threads.

1) Initialization phase. CPU initializes an array (called *Args Array*) to store the original n kernels' arguments, and calculates a prefix-sum array *scan*, with each element containing the sum of the first i kernels' thread count, formally

$$scan = [0, m_0, m_0 + m_1, \cdots, \sum_{j=0}^{n-1} m_j]$$

Then we issue a kernel with $\sum_{j=0}^{n-1} m_j$ threads.

2) Identification phase. In this phase, every GPU thread needs to identify its position in the original multi-kernel scenario. Specifically, for the thread with ID *tid*, it starts by performing a binary search on the *scan* to find the largest element which is smaller than *tid*, whose index is denoted as φ. This means the thread *tid* corresponds to the $(tid - scan[\varphi])$-th thread of φ-th kernel. Since all threads share the same *scan* and *Args Array* arrays, FLECHE can cache them in low-latency shared memory of GPU's streaming multiprocessors (SMs) to speed up repeated accesses. Though binary search appears to introduce branch divergence (leading to poor performance), it does not actually exist in our case. An important property to understand it is that, since the *tids* are consecutive, the branch conditions of every 32 threads (a warp) walking through are exactly the same, if the thread count in each kernel rounds up to multiples of the warp size.

3) Execution phase. Now each thread can take its kernel arguments from the *Args Array* and execute the corresponding job just as the multi-kernel fashion.

Figure 6 shows a running example of fusing three cache query kernels. The original three kernels consist of 960, 1920, 640 threads respectively. With the aforementioned fusion method, we only need to launch a kernel of 3520 threads once, reduing maintenance overhead.

The fusion process is executed during the building of computational graph. Since DLRMs first transform feature IDs with embedding, the queries for all embedding tables are launched simultaneously. It helps us find all cache query operators and fuse them easily.

3.3 Optimizing the Workflow of Cache Query

Leveraging the key-value separation structure of FC, we adopt two techniques to further optimize the workflow of cache query: 1) decoupling copying from indexing to enable parallelism between GPU-HBM's copying and CPU-DRAM layer's indexing, and 2) using unified index to bypass indexing of CPU-DRAM layer.

Decoupling copying from indexing. To reduce the number of kernels (as more kernels come with more maintenance overhead), existing per-table-cache scheme fuses both indexing and copying in a tightly-coupled kernel, but suffers from in-critical-path copy. Figure 7a shows a specific example. Suppose there are two warps trying to access a certain key-value pair at the same time. If warp 1 locks it first, warp 2 will keep retrying until warp 1 finishes the copy process

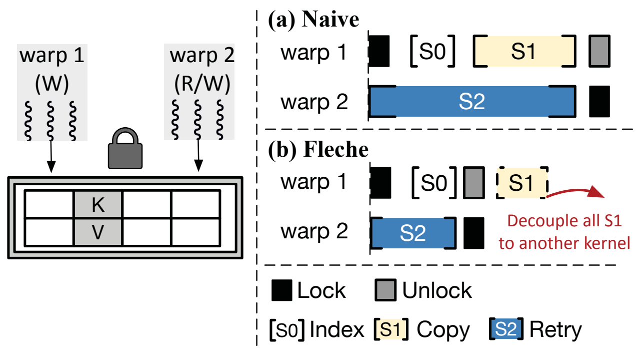


Figure 7. Decoupling the copying operation out of critical path. *The right part compares the timing diagrams of the naive approach and Fleche. Warp 1 is a writer, and warp 2 can be a reader or writer.*

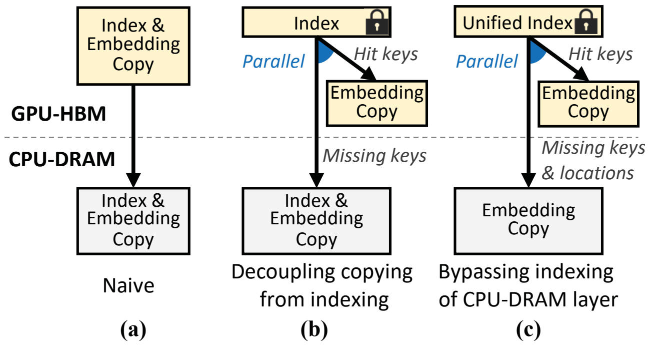


Figure 8. Optimizing the workflow of cache query. *In subfigure (c), indexing of these missing keys is offloaded to unified index.*

and releases the lock. This copy time is not short on GPU due to the high latency of global memory access. Since the cache usually only uses one warp (32 threads) to query one key and copy the desired embedding, if the embedding dimension is larger than 32, this warp needs to perform more rounds of global memory access, further aggravating the lock time.

Self-identified kernel fusion gives us the opportunity to achieve the best of both worlds. It features Fleche's decoupling of indexing and copying (Figure 7b) at the cost of just incrementing the number of kernels from 1 to 2. In contrast, that of per-table cache schemes increments from n to $2n$.

Fleche's decoupling design adjusts the workflows of cache operations. For querying, the indexing kernel locates all hit embeddings' addresses and the copying kernel copies from these addresses to output matrices. There is no need for the copying kernel to consider the thread safety thanks to our epoch-based space reclamation mechanism. For replacement, we first launch a copying kernel to copy the embeddings to the addresses allocated from the memory pool, and then launch an indexing kernel to modify ⟨key, address⟩ mappings of FC. This is achievable since the process of copying embedding is invisible to indexing.

This decoupling design enjoys three benefits. First, copy operations are moved out of the critical path. Second, we shorten the copying time. Instead of being limited to a single warp copying the entire embedding, our copying kernel can launch more threads according to embedding dimensions, improving the SM utilization of GPU. Third, Fleche can query the CPU-DRAM layer ahead without waiting for the completion of copying kernel; see Figure 8b. This is because once the indexing kernel finishes its execution, we already know which key is missing.

Bypassing indexing of CPU-DRAM layer. Fleche proposes a *unified index* technique (Figure 8c) to further shorten the cache query workflow opportunistically. This technique offloads lookups of partial embeddings stored in the CPU-DRAM layer to GPU. Specifically, we record these embeddings' locations in FC and set the least significant bit of pointers to indicate a CPU-DRAM pointer. Therefore, slow indexing for some missing keys in DRAM gets bypassed, and is replaced by extremely fast parallel GPU queries.

Although maintaining an index in GPU for the CPU-DRAM layer improves performance, it requires additional memory space which could have been used to cache embedding. Thus, we need carefully consider the tradeoffs.

Here we empirically give a simple method to tune the memory capacity for unified index. We start with an empty unified index, and keep gradually increasing its capacity, by replacing the cache of cold embeddings with CPU-DRAM pointers. We pause increasing until the performance peak is reached. If a significant performance decline is detected (which means that workload changes), we clear the unified index and repeat the above process.

4 Implementation

We implement Fleche on HugeCTR by replacing the original cache module. Fleche leverages Slab-Hash [11] as the underlying index of flat cache, same as HugeCTR.

Deduplicating and restoring. Since there are usually many duplicate IDs among different samples in a batch, in order to eliminate redundant cache queries, Fleche first deduplicates all IDs, and restores the full output matrix according to the deduplication results after querying embedding. The deduplicating mechanism also ensures no concurrent readers exist for the same key on GPU-resident index, which allows us to implement concurrency control with existing timestamps.

Fast memory copy between host and device with GPU-Direct RDMA. There are fragmented copies between CPU and GPU for various metadata in Fleche, e.g., the *Args Array*, *scan* array in the self-identified kernel fusion method. The vanilla cudaMemcpy API incurs in a 6–7 µs overhead, not suitable for excessive small copies. Fleche exploits the

GDRCopy library [6], which exposes HBM to CPU and uses CPU-driven copies based on NVIDIA GPUDirect RDMA technology, to speed up the small copy to about 0.1 µs latency.

5 Discussion

Dealing with giant models. We describe Fleche in the context of a single machine. However, the size of industrial recommendation models can exceed a single machine's DRAM capability. In this case, the local CPU-DRAM layer is no longer an immutable layer with all parameters, but becomes another cache layer, and the full amount of parameters are stored in a new additional layer (e.g., remote parameter servers). All our designs still work in this scenario, but we should carefully deal with a corner case: unified index's pointers to DRAM may be invalidated due to the cache eviction of the CPU-DRAM layer.

Dealing with multi-GPU. Fleche focuses on single-GPU caching only. The main reason is that according to various datasets, the size of hotspots is commonly small enough for a single GPU. Multi-GPU caching expands the size of cache system and removes the redundancy between GPUs with model parallelism. We leave it for future research.

Generality of self-identified kernel fusion method. Our self-identified kernel fusion method has its generality, not only for multi-table cache queries, but also for a specific type of maintenance overhead issue where we need to wait the completion of too many small kernels issued simultaneously.

These kernels need to meet two assumptions. 1) They should share the same thread block size, otherwise the block size of fused kernel may fail to satisfy the original block synchronization semantics and produce unintended results. 2) They should not introduce greater-than-block-granularity synchronization (e.g. grid_group sync), as the problem of hanging on will occur after fusion. Fortunately, multi-table cache query kernels satisfy the above assumptions.

Alternative designs. We discuss alternative designs, and why we do not adopt them.

1) Reduction cache. Fleche relies on the point-cache scheme similar to traditional hash-based key-value stores. As another way of caching, the reduction cache [28, 32] brings the memoization technique to cache the reduced results (after the pooling layer) of co-appearing embeddings.

Fleche does not adopt the reduction-cache scheme, because it is only applicable to simple pooling layers (e.g., sum, avg, max) instead of more complicated layers such as attention layers [47], which damages model generality. Still, if model generality is not considered, implementing a reduction cache on GPU would be considered.

2) Persistent kernels. Existing GPU-resident KV store uses *persistent kernel* [1] to alleviate the maintenance overhead, i.e., it specially issues a dead-loop kernel to indefinitely poll requests and generate results, which does not work in our scenario either. This is because recommendation models contain not only the sparse embedding part but also the following dense MLP part, and with persistent kernel occupying SM resources, the MLP part would suffer a slower computation speed, which is unacceptable.

	CPU	**NVIDIA T4 GPU**
Cores	64	2560
Cache Sizes	1-16-22 MB	96-512 KB
Memory Capability	512 GB	15 GB
Memory Bandwidth	60 GB/s	300 GB/s
TDP	300 W	70 W

Table 1. Hardware platform.

Datasets	**# Emb Tbls**	**# Samples**	**# Sparse IDs**	**Param Size**
Avazu	22	40M	49M	5.8GB
Criteo-Kaggle	26	45M	34M	4.1GB
Criteo-TB	26	4.4B	0.9B	461GB

Table 2. Datasets for evaluation.

Applicability to other embedding models. We discuss the applicability of Fleche to two other types of models leveraging embedding: natural language processing (NLP) which embeds words, and graph neural network (GNN) which embeds graph nodes, edges, and their attributes. 1) Fleche does not apply to NLP models because the word embedding table is small (e.g., ~100 MB in Google BERT [16]) due to limited words, and can be fully cached in GPU. 2) In terms of GNN, the categorical features of graph nodes and edges also introduce many large embedding tables, which exhibit similar behaviors as recommendation models, so we believe it is practical for GNN to gain benefits from Fleche and it will be considered as our future work.

6 Evaluation

6.1 Evaluation Setup

Testbed. We run experiments on a machine equipped with an Intel Xeon Gold 6252 CPU and an NVIDIA T4 GPU (15 GB HBM available), which shares the same configuration as our inference cluster in production; the bandwidth of HBM on GPU and CPU-side DRAM is 300 GB/s and 60 GB/s respectively; see Table 1 for detailed specifications. All codes are compiled by GCC 9.3 and nvcc 11.3 with -O3. We use CUDA Toolkit 11.3 and cuDNN 8.2 for GPU.

Datasets. We use real-world datasets to evaluate Fleche's performance and synthetic datasets to evaluate its sensitivity. We use the preprocessing scripts in HugeCTR to remove low frequent features of all datasets.

For real-world datasets, we use Avazu [3], Criteo-Kaggle [5], and Criteo-TB [8]; see Table 2 for their detailed characteristics. For Avazu and Criteo-Kaggle, the embedding dimension

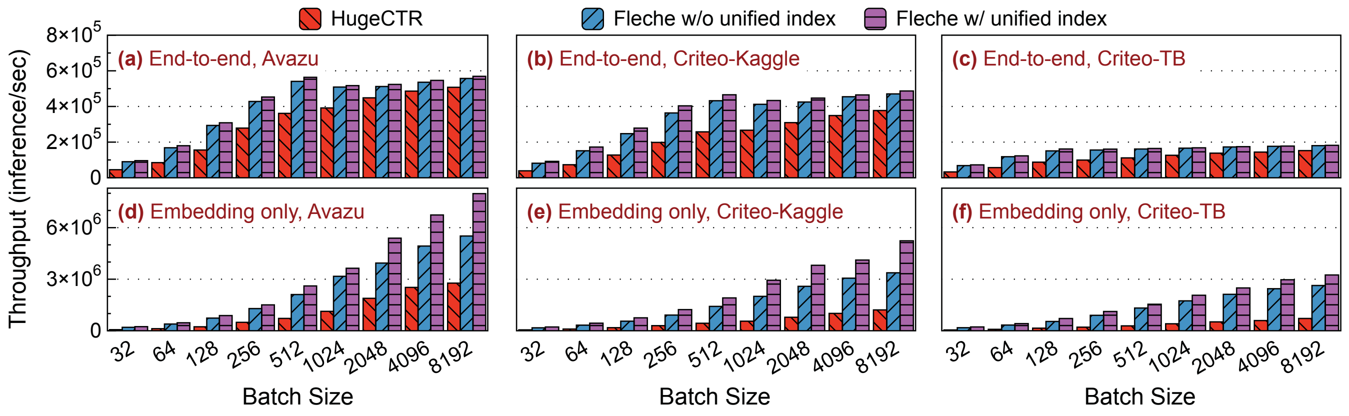


Figure 9. (Exp #1) Overall throughput improvement.

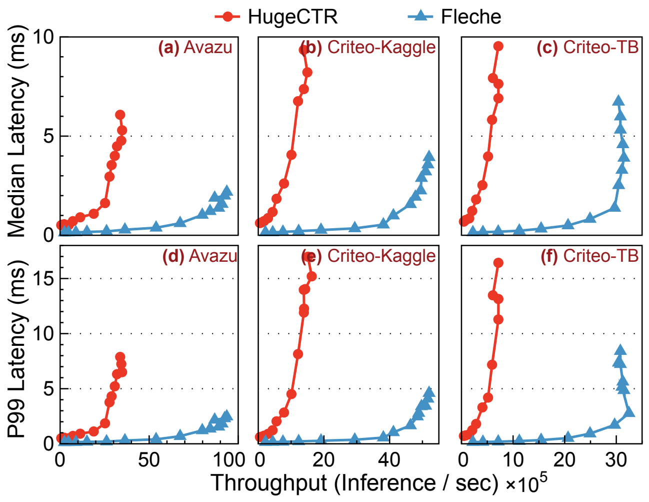


Figure 10. (Exp #2) Throughput vs. median/P99 latency of the embedding layer.

is configured as 32, while for Criteo-TB, the dimension is configured as 128. Note that all these datasets are open source, and thus our experiments can be easily reproduced.

For synthetic datasets, we generate feature IDs subjecting to a power law distribution. Unless otherwise stated, the α of power law distribution is -1.2, the embedding dimension is configured as 32, and the embedding table count is 40 with each table containing $0.25M$ features.

Model. We evaluate Fleche on a Deep Cross Network [40] with 6 multi-cross layers and a MLP layer with (1024, 1024) hidden units. We mainly focus on a single model instead of several ones for evaluation, because all our techniques target at the embedding part, and the most important difference between different kinds of recommendation models lies in their MLP parts, e.g., the extra FM layer [20], Cross layer [40], and transformer layer [46], while their embedding parts remain similar. For comprehensiveness of evaluation, we also benchmark Fleche with several models for evaluating the sensitivity of MLP layers in §6.4, since different models may exhibit different ratios of embedding/MLP costs, which affects the gain of Fleche for end-to-end performance.

Systems in comparison. We present comparisons with HugeCTR. It is the only recommendation inference system supporting GPU-resident cache. For fair comparison, both Fleche and HugeCTR are configured with the same HBM cache size. We also apply the GDR-Copy library to optimize small copies for HugeCTR. If not specified, the cache size of Avazu and Criteo-Kaggle is set to 5% of the size of all embedding tables, and that of Criteo-TB is set to 0.5%. The batch size is set to 4096. We omit the results of a no-caching system, because caching brings a more than five-fold performance improvement according to our evaluation; we focus on how to use caching well.

For all experiments, we first issue requests to warm up the cache, and then collect metrics. We run each experiment 3 times and show the average.

6.2 Overall Performance

Exp #1: Throughput. In Figure 9 (a-c), we compare the end-to-end throughput of HugeCTR and Fleche (both with and without unified index) in three datasets with batch sizes ranging from 32 to 8192. Fleche with unified index achieves 1.1–2.1×, 1.3–2.4×, and 1.2–2.2× speedup over HugeCTR in Avazu, Criteo-Kaggle, and Criteo-TB respectively, which justifies the designs of Fleche. Meanwhile, the variant without unified index outperforms HugeCTR by 1.1 – 2.0×, 1.2 – 2.1×, and 1.2 – 2.1×. We also find that the larger the batch size is, the less performance Fleche improves, because all the techniques of Fleche optimize the embedding part only, which takes up a smaller proportion of the end-to-end latency in the larger batch size case.

We repeat our experiments on a model with only embedding layers, i.e., sidestep the computation of MLP layers, to evaluate Fleche's performance improvement on embedding layers. As shown in Figure 9 (d-f), the standalone improvement of embedding layer ranges from 2.7 – 3.9×,

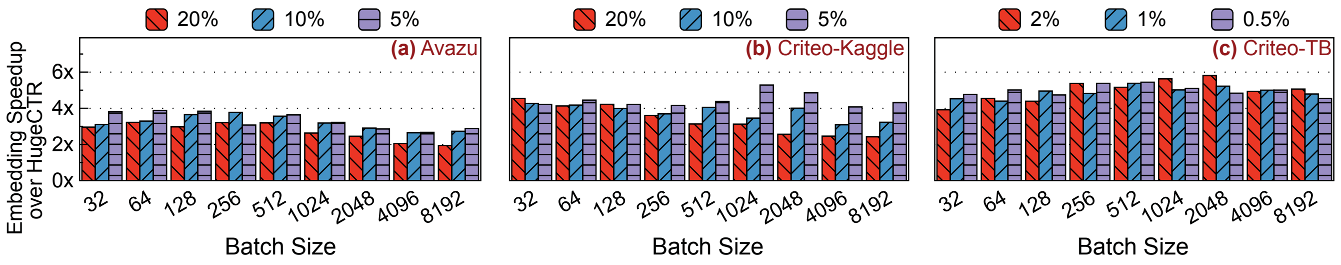


Figure 11. (Exp #3) Throughput improvement of the embedding part in FLECHE compared with HugeCTR under different cache sizes. *20% means that the cache size is 20% of the size of all embedding tables.*

4.1 – 5.3×, 4.5 – 5.4×(with unified index), and 2.0 – 3.3×, 2.8 – 3.6×, 3.6 – 4.7×(without unified index) respectively in Avazu, Criteo-Kaggle, and Criteo-TB. We observe that: 1) the improvement in Criteo is higher compared with that in Avazu, since Criteo has more embedding tables and a more skewed hotness distribution among tables. 2) Compared with Criteo-Kaggle, Criteo-TB suffers from low throughput due to the large corpus size.

Exp #2: Throughput vs. Latency. Figure 10 shows the median / 99 percent (P99) latencies and the corresponding throughput of HugeCTR and FLECHE for the embedding part. We find that FLECHE exhibits much lower latency (both median and P99) and significantly higher throughput. For example, given the median latency of 1 ms in Avazu, the throughput of FLECHE achieves 80*M* inference per second, around 4.2× to that of HugeCTR. Put differently, given the throughput of 40*M* in Avazu, the reduction of latency brought by FLECHE over HugeCTR reaches one order of magnitude. Similar with Exp #1, FLECHE improves more in Criteo-Kaggle and Criteo-TB than in Avazu.

Exp #3: Performance with different cache sizes. Since the performance of the MLP part remains the same as cache size changes, we only focus on the embedding part. Specifically, as shown in Figure 11, we evaluate the performance of embedding layer under cache sizes accounting for 5%, 10%, 20% (for Avazu and Criteo-Kaggle) and 0.5%, 1%, 2% (for Criteo-TB) of the size of all embedding tables. FLECHE outperforms HugeCTR by 1.9 – 3.8×, 2.4 – 5.3× and 3.9 – 5.8× in Avazu, Criteo-Kaggle and Criteo-TB, respectively.

We observe that: 1) in Avazu and Criteo-Kaggle, the smaller the cache size, the more performance FLECHE boosts, while in Criteo-TB, there is no such phenomenon. This is because in Criteo-TB, the cache hit rates improved by FLECHE's flat cache are similar in different cache sizes. However, in Avazu and Criteo-Kaggle, the cache hit rates improved are higher with smaller cache sizes, which means that HugeCTR's per-table cache structure suffers from severer under-utilization with scarce capacity in these two datasets. The detailed hit rates in different datasets will be reported in Exp #4.

2) The increasing in the batch size brings down the improvement of FLECHE. The reason is that with a larger batch, unoptimized operations such as deduplicating and restoring take more time. See Exp #8 for details.

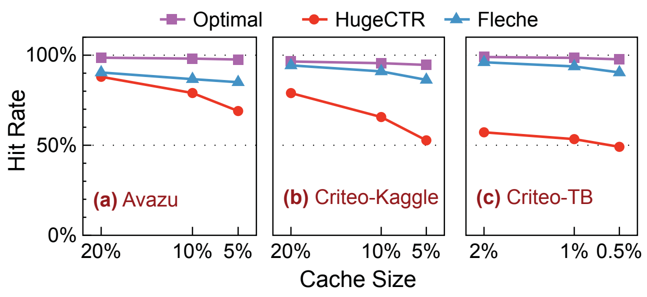


Figure 12. (Exp #4) The cache hit rate improvement brought by flat cache. *Optimal denotes the ideal case where the cache knows all accesses of datasets.*

6.3 Techniques

We evaluate the effectiveness of our proposed techniques and show how much they contribute to the final performance.

Exp #4: Flat cache. Figure 12 shows the cache hit rate improvement of flat cache in three datasets. Specifically, FLECHE achieves hit rates of 85% – 90%, 85% – 94% and 90% – 96% with different cache sizes in Avazu, Criteo-Kaggle, and Criteo-TB respectively, improving by 2% – 15%, 11% – 27% and 39% – 41% compared with HugeCTR. FLECHE is quite close to the optimal case and we attribute these results to the efficiency of flat cache's sharing structure.

Exp #5: Re-encoding with size-aware coding. Figure 13 shows the model accuracy after re-encoding with different methods by varying the number of bits of flat keys. The common metric for recommendation models, Area Under Curve [22] (AUC, the higher the better), is applied here to evaluate the model performance. We evaluate a fixed-length encoding method [41] ("Kraken" in Figure 13), and our size-aware encoding method (denoted as "FLECHE"). We also evaluate the AUC upper bound with an ideal case where no key conflicts occurs. As we can see, our encoding approach always exhibits 1) much higher AUC with the same number of bits, or 2) significantly less bit number with the same AUC than Kraken. This is mainly because our variable-length

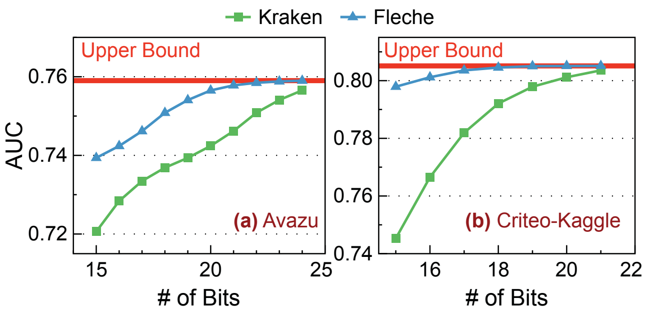


Figure 13. (Exp #5) Model performance (AUC) of different flat-key encoding methods. *The red line denotes the AUC of ideal case without conflicts, which is the upper bound.*

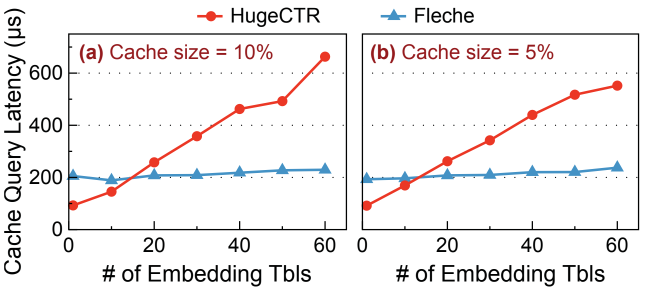


Figure 14. (Exp #6) Latency of cache query under different embedding table counts. *The total number of queried keys spreading all embedding caches is 10K. Other queried keys share a similar result.*

encoding method makes full use of the limited bit representation space, while Kraken's fixed-length encoding causes violent key conflicts.

Exp #6: Self-identified kernel fusion. Similar to §2.2, we fix the total number of query keys and evaluate the embedding cache query latency by varying the number of apportioned cache tables. As shown in Figure 14, we make the following two observations.

1) When the embedding table count is smaller than 15, the maintenance overhead is hidden by the long execution time. The decoupling design of Fleche introduces one more kernel and therefore gives higher latency. Fortunately, the number of embedding tables in the real-world inference scenario is usually much larger than 15.

2) As the number of tables increases, HugeCTR suffers from a rising latency, while Fleche keeps an almost stable latency (the slightly increasing latency of Fleche is due to more host-to-device metadata copies).

Our fusion method cuts out the kernel maintenance overhead almost completely. By our measurement, the overhead of initialization phase and identification phase is only several microseconds, which is negligible.

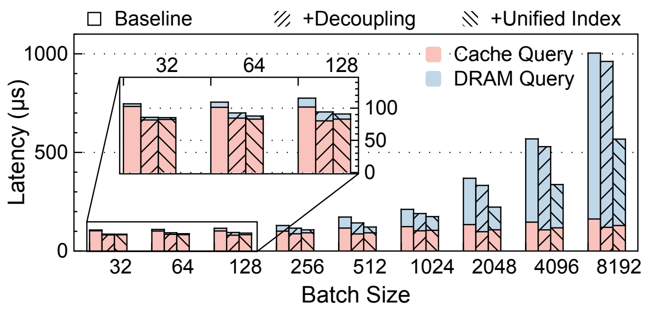


Figure 15. (Exp #7) Benefits of unified index and decoupling copying from indexing. *Design techniques are cumulative. Baseline: HugeCTR with flat cache and self-identified kernel fusion. Decoupling: decoupling copying from indexing. Cache Query = Cache Index + Cache Copy, so is DRAM.*

Exp #7: Optimizing the cache query workflow. Figure 15 illustrates the improvement brought by unified index and decoupling copying from indexing. We only show the results of Avazu and 5% cache size, and omit other cases that have similar results due to space limitations. The benefits of two techniques differ as batch size changes. With small batch sizes (32–128), decoupling copying from indexing can reduce the latency by 15.3% to 19.7% since the overhead of GPU query is the bottleneck. As batch size increases (e.g., larger than 4096), the proportion of cache query gets smaller, so the latency of decoupling copying from indexing only decreases by 4.2% to 9.8%. Conversely, unified index performs better with larger batch sizes, where the reduction reaches 33.0% to 40.9%. This is because in this case, DRAM query accounts for a larger proportion of the overhead.

Exp #8: Contributions of techniques to performance. We analyze Fleche's performance by breaking down the gap between HugeCTR and Fleche. Figure 16 illustrates that using flat cache respectively reduces the latency by 3.6% – 19.7% in Avazu, 3.5% – 13.7% in Criteo-Kaggle, and 8.0%–32.4% in Criteo-TB, mainly steming from the reduction in CPU-DRAM layer time due to the improved hit rate. By introducing self-identified kernel fusion, the cache query latency is further reduced by 64.3% – 81.9% in Avazu, 68.9% – 89.7% in Criteo-Kaggle, and 65.7%–91.9% in Criteo-TB, which results in an end-to-end reduction of 38.8% – 61.2%, 55.4% – 66.9% and 61.7% – 73.0% respectively. Finally, we optimize the workflow of cache query, which reduces the latency by 60.2% – 68.6% in Avazu, 71.0% – 79.9% in Criteo-Kaggle, and 74.5% – 79.6% in Criteo-TB cumulatively.

6.4 Sensitivity

In this subsection, we mainly use the synthetic dataset to evaluate the sensitivity of Fleche (except Exp #12). Note that there is almost no difference between HugeCTR and Fleche in the hit rate for the synthetic dataset. This is because every embedding table has the same size and hotness distribution,

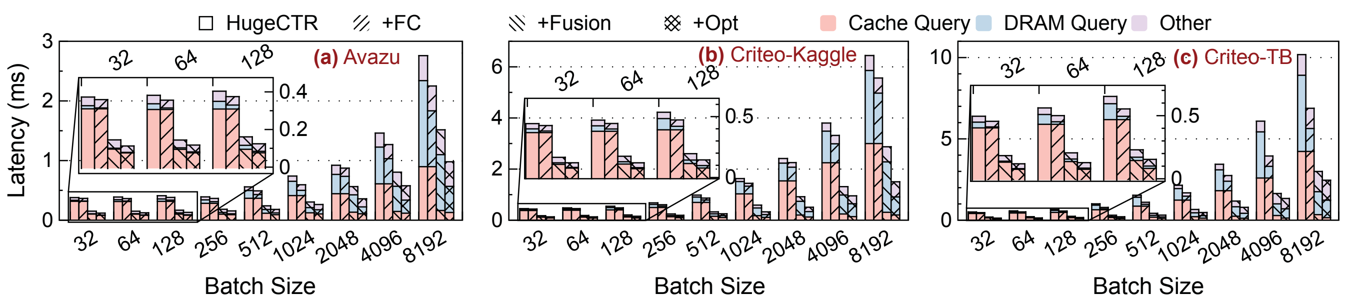


Figure 16. (Exp #8) Contributions of techniques to performance. *Design techniques are cumulative. Fusion: self-identified kernel fusion. Opt: unified index and decoupling copying from indexing. Other: other operations not related to querying, including deduplicating, restoring, etc. Cache Query = Cache Index + Cache Copy, so is DRAM.*

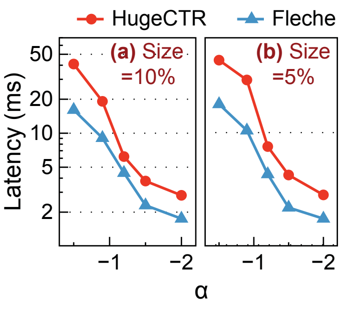


Figure 17. (Exp #9) Impact of embedding skewness.

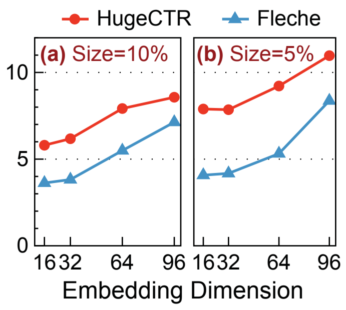


Figure 18. (Exp #10) Impact of embedding dimension.

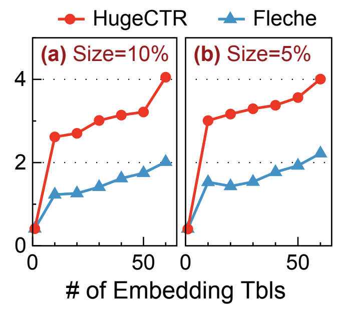


Figure 19. (Exp #11) Impact of embedding table number.

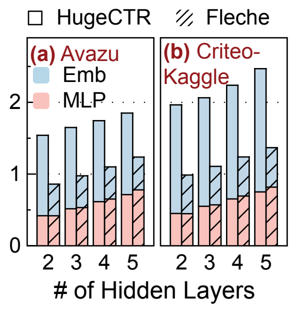


Figure 20. (Exp #12) Impact of MLP layers.

for ease of sensitivity tests (eliminating all benefits of FC). Yet, even with this ideal distribution, Fleche still improves the embedding performance significantly in different cases (see below). To exclude the effects of cache sizes on sensitivity, we evaluate synthetic datasets with two cache sizes, 5% and 10% (achieving hit rates of 69% and 78% respectively).

Exp #9: Impact of embedding skewness. Figure 17 shows how skewness of workloads affects Fleche's performance. We vary the α parameter of power law distribution from -0.5 to -2.0 (a smaller α implies larger skewness of the data distribution). First, Fleche constantly benefits the embedding lookup by 1.4 – 2.8× under different distributions. Second, for the low-skewness case (e.g., $\alpha = -0.5$), the latency of both Fleche and HugeCTR increases due to low hit rate. However, Fleche gains more improvement than the high-skewness case (e.g., $\alpha = -2.0$). This is due to the fact that more indexing operations of CPU-DRAM layer are offloaded to Fleche's unified index at low hit rates.

Exp #10: Impact of embedding dimension. Figure 18 depicts the latency of embedding layer in HugeCTR and Fleche with varying embedding dimensions. We make the following two observations.

1) Generally, the larger the embedding dimension is, the slower the embedding part is. This is due to larger copy size for both GPU and CPU. Still, Fleche outperforms HugeCTR by 1.2 – 1.9× under different dimensions.

2) Interestingly, we can see that the performance of each system with 16 embedding dimensions is similar to that with 32, because the memory coalescing characteristic of GPU results in no difference between copying 16 dimensions and 32 dimensions. Thus, their difference only exists in the DRAM part, yet it is quite small, less than 100 µs.

Exp #11: Impact of embedding table number. We analyze how two systems behave with varying embedding table numbers. We fix the number of querying IDs to 100K and change the number of embedding tables (Figure 19). Overall, Fleche achieves 1.8 – 2.2× and 1.8 – 2.1× improvement over HugeCTR with 5% and 10% cache size respectively (except the case where the table count is 1). When the table count is 1, Fleche shares a similar performance because of low kernel maintenance overhead at this point. As the embedding table count increases, Fleche's increased latency is mainly attributed to sorting out fragmented memory copies from query requests for each embedding table.

Exp #12: Impact of MLP layers. Figure 20 shows the end-to-end prediction latency of HugeCTR and Fleche with different numbers of hidden layers on Avazu and Criteo-Kaggle. Each of the layer has 1024 hidden units. The batch size is configured as 256. Similar findings are found for other batch sizes and the Criteo-TB dataset. First, Fleche's MLP time is similar with that of HugeCTR, since our techniques involve only the embedding part, and they do not interfere with the computation of the MLP part. Second, as the MLP gets deeper, the MLP time increases accordingly, which results in less performance gain from Fleche. Yet, Fleche constantly benefits the end-to-end performance compared with HugeCTR with all different models.

7 Related Work

The most related work is HugeCTR [7], an optimized inference system for recommendation models proposed by NVIDIA, which employs GPU-resident cache to resolve the DRAM's bandwidth bottleneck. We have detailedly described it in §2.2. Next, we organize the related work into two types: 1) with *different goal and similar mechanism*, which uses GPU-resident cache but for other goals, and 2) with *similar goal and different mechanism*, which uses other techniques to solve the DRAM bandwidth problem of embeddings.

7.1 Different Goal, Similar Mechanism

GPU-accelerated in-memory key-value stores (KVSs) [11, 24, 42] enjoy the benefit of massive threads' lookup in parallel to boost throughput. However, GPU-KVS is a bespoke system designed for high peak throughput of general KV requests. Simply using a traditional GPU-KVS as embedding cache suffers from unsatisfied performance under the recommendation scenario. First, a separate GPU-KVS for each embedding cache table only captures local hot spots, causing under-utilization, while a global GPU-KVS encounters the problem of key conflicts. Second, unlike GPU-KVS, which usually carries up to hundreds of thousands of keys in one kernel to amortize maintainence overhead, the batch size of IDs in a recommendation inference request is small, which exposes the kernel maintenance latency due to the relatively shorter kernel execution time. These two arising problems are the main focuses of Fleche.

Industrial large-scale recommendation model training systems (AIBOX [44], HierPS [43], ScaleFreeCTR [19]) mostly maintain the embedding parameters to be used soon on GPU-resident cache tables and the overall parameters on DRAMs or SSDs. The key intuition is that, each training batch only requires accesses to a small portion of substantial model parameters. With training datasets known in advance, these systems can prefetch embeddings of next several batches and write them into GPUs, successfully hiding the overhead of embedding layer accesses. However, unlike training, the inference process faces more challenges due to the unpredictable IDs carried by the incoming requests.

Kraken [41] shares the idea of sharing the space of different embedding tables, and uses a fixed-length encoding method for flat keys. However, Kraken is designed for continuous training in CPU while Fleche targets specifically at GPU caching. Moreover, the accuracy degradation caused by conflicts with fixed-length coding is unacceptable in inference. We propose variable-length size-aware coding in Fleche to effectively solve this problem.

7.2 Similar Goal, Different Mechanism

MERCI [32] applies memoization for recording the reduced results of embeddings to alleviate bandwidth pressure. It is only applicable to static datasets due to its static renumbering technique, but not operative in the real online query scenario where IDs keep pouring in constantly.

Some studies seek near memory processing (NMP) architecture to reduce unnecessary data movements [10, 29, 31], while others offload embedding operations to FPGAs with high-bandwidth memory (HBM) [25–27]. Nonetheless, these studies either use gem5 [12] simulator and cannot get validated on real hardware, or require additional expensive hardware, e.g., HBM-equipped FPGA ($6,495), which costs about 3× the price of NVIDIA Tesla T4 datacenter GPU ($2,268) [2]. These studies only aim at exploring research interests and can not practically deploy on a large scale in a real industrial production setting in foreseeable several years. Unlike the above two types, Fleche leverages the off-the-shelf GPU in datacenters without the need of additional hardware.

8 Conclusion

In this paper, we study the problem of accelerating inference of deep learning based recommendation models (DLRMs). This problem comes from the bandwidth gap of CPU-side DRAM data accessing and GPU processing. We evaluate the existing systems and find their common pitfalls, 1) cache under-utilization, 2) overhead of kernel maintenance. We then apply the instructional insights on how to cache the embedding table on GPU, including sharing one global cache backend among all embedding tables, merging small kernel calls into a single large one, decoupling copying from indexing. And we present Fleche, a holistic cache scheme with detailed designs for efficient GPU-resident embedding caching. Experiments with real-world datasets show that compared with the prior art, Fleche significantly improves the throughput of embedding layer, and gets speedup of end-to-end inference throughput.

Acknowledgements

We sincerely thank our shepherd Y. Charlie Hu and anonymous reviewers for their valuable feedback, which greatly improved this paper. We also thank Xiao Liang, Hongbo Ao for their insightful suggestions. This work is funded by the National Natural Science Foundation of China (Grant No. 62022051, 61832011), and Kuaishou.

References

[1] 2018. Design high-performance in-memory key-value operations with persistent GPU kernels and openshem. https://www.csm.ornl.gov/workshops/openshmem2018/presentations/openshmem2018-NVIDIA-ORNL.pdf.

[2] 2021. Amazon.com: HP R0W29A Tesla T4 Graphic Card - 1 Gpus - 16 GB: Computers & Accessories. https://www.amazon.com/HP-R0W29A-Tesla-Graphic-Card/dp/B07PGY6QPT.

[3] 2021. Click-Through Rate Prediction | Kaggle. https://www.kaggle.com/c/avazu-ctr-prediction.

[4] 2021. CUDA Runtime API::CUDA Toolkit Documentation. https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__GRAPH.html.

[5] 2021. Display Advertising Challenge | Kaggle. https://www.kaggle.com/c/criteo-display-ad-challenge.

[6] 2021. NVIDIA/gdrcopy: A fast GPU memory copy library based on NVIDIA GPUDirect RDMA technology. https://github.com/NVIDIA/gdrcopy.

[7] 2021. NVIDIA/HugeCTR: HugeCTR is a high efficiency GPU framework designed for Click-Through-Rate (CTR) estimating training. https://github.com/NVIDIA/HugeCTR.

[8] 2022. Download Criteo 1TB Click Logs dataset - Criteo AI Lab. https://ailab.criteo.com/download-criteo-1tb-click-logs-dataset/.

[9] Ehsan K Ardestani, Changkyu Kim, Seung Jae Lee, Luoshang Pan, Valmiki Rampersad, Jens Axboe, Banit Agrawal, Fuxun Yu, Ansha Yu, Trung Le, et al. 2021. Supporting Massive DLRM Inference Through Software Defined Memory. *arXiv preprint arXiv:2110.11489* (2021).

[10] Bahar Asgari, Ramyad Hadidi, Jiashen Cao, Da Eun Shim, Sung-Kyu Lim, and Hyesoon Kim. 2021. FAFNIR: Accelerating Sparse Gathering by Using Efficient near-Memory Intelligent Reduction. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, Seoul, Korea (South), 908–920. https://doi.org/10/gkg3vr

[11] Saman Ashkiani, Martin Farach-Colton, and John D Owens. 2018. A dynamic hash table for the GPU. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 419–429.

[12] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH computer architecture news* 39, 2 (2011), 1–7.

[13] Heng-Tze Cheng, Mustafa Ispir, Rohan Anil, Zakaria Haque, Lichan Hong, Vihan Jain, Xiaobing Liu, Hemal Shah, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, and Wei Chai. 2016. Wide & Deep Learning for Recommender Systems. In *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems - DLRS 2016*. ACM Press, Boston, MA, USA, 7–10. https://doi.org/10.1145/2988450.2988454

[14] Paul Covington, Jay Adams, and Emre Sargin. [n.d.]. Deep Neural Networks for YouTube Recommendations. ([n. d.]), 8. https://doi.org/10/gfvb44

[15] Christian Desrosiers and George Karypis. 2011. A comprehensive survey of neighborhood-based recommendation methods. *Recommender systems handbook* (2011), 107–144.

[16] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[17] Assaf Eisenman, Maxim Naumov, Darryl Gardner, Misha Smelyanskiy, Sergey Pupyrev, Kim Hazelwood, Asaf Cidon, and Sachin Katti. 2018. Bandana: Using non-volatile memory for storing deep learning models. *arXiv preprint arXiv:1811.05922* (2018).

[18] Keir Fraser. 2004. *Practical lock-freedom*. Technical Report. University of Cambridge, Computer Laboratory.

[19] Huifeng Guo, Wei Guo, Yong Gao, Ruiming Tang, Xiuqiang He, and Wenzhi Liu. 2021. ScaleFreeCTR: MixCache-Based Distributed Training System for CTR Models with Huge Embedding Table. *arXiv:2104.08542 [cs]* (May 2021). arXiv:2104.08542 [cs]

[20] Huifeng Guo, Ruiming Tang, Yunming Ye, Zhenguo Li, and Xiuqiang He. 2017. DeepFM: a factorization-machine based neural network for CTR prediction. *arXiv preprint arXiv:1703.04247* (2017).

[21] Udit Gupta, Carole-Jean Wu, Xiaodong Wang, Maxim Naumov, Brandon Reagen, David Brooks, Bradford Cottel, Kim Hazelwood, Bill Jia, Hsien-Hsin S. Lee, Andrey Malevich, Dheevatsa Mudigere, Mikhail Smelyanskiy, Liang Xiong, and Xuan Zhang. 2020. The Architectural Implications of Facebook's DNN-Based Personalized Recommendation. *arXiv:1906.03109 [cs]* (Feb. 2020). arXiv:1906.03109 [cs]

[22] James A Hanley and Barbara J McNeil. 1982. The meaning and use of the area under a receiver operating characteristic (ROC) curve. *Radiology* 143, 1 (1982), 29–36.

[23] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. 2017. Neural Collaborative Filtering. In *Proceedings of the 26th International Conference on World Wide Web* (Perth, Australia) *(WWW '17)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 173–182. https://doi.org/10.1145/3038912.3052569

[24] Tayler H Hetherington, Mike O'Connor, and Tor M Aamodt. 2015. Memcachedgpu: Scaling-up scale-out key-value stores. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*. 43–57.

[25] Ranggi Hwang, Taehun Kim, Youngeun Kwon, and Minsoo Rhu. 2020. Centaur: A Chiplet-Based, Hybrid Sparse-Dense Accelerator for Personalized Recommendations. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, Valencia, Spain, 968–981. https://doi.org/10/gh7zp6

[26] Wenqi Jiang, Zhenhao He, Shuai Zhang, Thomas B Preußer, Kai Zeng, Liang Feng, Jiansong Zhang, Tongxuan Liu, Yong Li, Jingren Zhou, et al. 2021. MicroRec: efficient recommendation inference by hardware and data structure solutions. *Proceedings of Machine Learning and Systems* 3 (2021).

[27] Wenqi Jiang, Zhenhao He, Shuai Zhang, Kai Zeng, Liang Feng, Jiansong Zhang, Tongxuan Liu, Yong Li, and Jingren Zhou. 2021. FleetRec: Large-Scale Recommendation Inference on Hybrid GPU-FPGA Clusters. (2021), 10.

[28] Hongju Kal, Yonsei University, Seokmin Lee, Yonsei University, Gun Ko, Yonsei University, Won Woo Ro, and Yonsei University. [n.d.]. SPACE: Locality-Aware Processing in Heterogeneous Memory for Personalized Recommendations. ([n. d.]), 13.

[29] Liu Ke, Udit Gupta, Carole-Jean Wu, Benjamin Youngjae Cho, Mark Hempstead, Brandon Reagen, Xuan Zhang, David Brooks, Vikas Chandra, Utku Diril, Amin Firoozshahian, Kim Hazelwood, Bill Jia, Hsien-Hsin S. Lee, Meng Li, Bert Maher, Dheevatsa Mudigere, Maxim Naumov, Martin Schatz, Mikhail Smelyanskiy, and Xiaodong Wang. 2019. RecNMP: Accelerating Personalized Recommendation with near-Memory Processing. *arXiv:1912.12953 [cs]* (Dec. 2019). arXiv:1912.12953 [cs]

[30] Yehuda Koren, Robert Bell, and Chris Volinsky. 2009. Matrix factorization techniques for recommender systems. *Computer* 42, 8 (2009), 30–37.

[31] Youngeun Kwon, Yunjae Lee, and Minsoo Rhu. 2019. TensorDIMM: A Practical near-Memory Processing Architecture for Embeddings and Tensor Operations in Deep Learning. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, Columbus OH USA, 740–753. https://doi.org/10/gkfjhs

[32] Yejin Lee, Seong Hoon Seo, Hyunji Choi, Hyoung Uk Sul, Soosung Kim, Jae W Lee, and Tae Jun Ham. 2021. MERCI: efficient embedding reduction on commodity hardware via sub-query memoization. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 302–313.

[33] Jianxun Lian, Xiaohuan Zhou, Fuzheng Zhang, Zhongxia Chen, Xing Xie, and Guangzhong Sun. 2018. xdeepfm: Combining explicit and implicit feature interactions for recommender systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge*

Discovery & Data Mining. 1754–1763.

[34] H. Brendan McMahan, Daniel Golovin, Sharat Chikkerur, Dan Liu, Martin Wattenberg, Arnar Mar Hrafnkelsson, Tom Boulos, Jeremy Kubica, Gary Holt, D. Sculley, Michael Young, Dietmar Ebner, Julian Grady, Lan Nie, Todd Phillips, and Eugene Davydov. 2013. Ad Click Prediction: A View from the Trenches. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '13*. ACM Press, Chicago, Illinois, USA, 1222. https://doi.org/10.1145/2487575.2488200

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

[36] Jongsoo Park, Maxim Naumov, Protonu Basu, Summer Deng, Aravind Kalaiah, Daya Khudia, James Law, Parth Malani, Andrey Malevich, Satish Nadathur, et al. 2018. Deep learning inference in facebook data centers: Characterization, performance optimizations and hardware implications. *arXiv preprint arXiv:1811.09886* (2018).

[37] Yanru Qu, Han Cai, Kan Ren, Weinan Zhang, Yong Yu, Ying Wen, and Jun Wang. 2016. Product-based neural networks for user response prediction. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*. IEEE, 1149–1154.

[38] Weiping Song, Chence Shi, Zhiping Xiao, Zhijian Duan, Yewen Xu, Ming Zhang, and Jian Tang. 2019. Autoint: Automatic feature interaction learning via self-attentive neural networks. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*. 1161–1170.

[39] Strother H Walker and David B Duncan. 1967. Estimation of the probability of an event as a function of several independent variables. *Biometrika* 54, 1-2 (1967), 167–179.

[40] Ruoxi Wang, Bin Fu, Gang Fu, and Mingliang Wang. 2017. Deep & cross network for ad click predictions. In *Proceedings of the ADKDD'17*. 1–7.

[41] Minhui Xie, Kai Ren, Youyou Lu, Guangxu Yang, Qingxing Xu, Bihai Wu, Jiazhen Lin, Hongbo Ao, Wanhong Xu, and Jiwu Shu. 2020. Kraken: Memory-Efficient Continual Learning for Large-Scale Real-Time Recommendations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Atlanta, Georgia) *(SC '20)*. IEEE Press, Article 21, 17 pages.

[42] Kai Zhang, Kaibo Wang, Yuan Yuan, Lei Guo, Rubao Lee, and Xiaodong Zhang. 2015. Mega-kv: A case for gpus to maximize the throughput of in-memory key-value stores. *Proceedings of the VLDB Endowment* 8, 11 (2015), 1226–1237.

[43] Weijie Zhao, Deping Xie, Ronglai Jia, Yulei Qian, Ruiquan Ding, Mingming Sun, and Ping Li. 2020. Distributed hierarchical gpu parameter server for massive scale deep learning ads systems. *arXiv preprint arXiv:2003.05622* (2020).

[44] Weijie Zhao, Jingyuan Zhang, Deping Xie, Yulei Qian, Ronglai Jia, and Ping Li. 2019. AIBox: CTR Prediction Model Training on a Single Node. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management - CIKM '19*. ACM Press, Beijing, China, 319–328. https://doi.org/10.1145/3357384.3358045

[45] Guorui Zhou, Na Mou, Ying Fan, Qi Pi, Weijie Bian, Chang Zhou, Xiaoqiang Zhu, and Kun Gai. 2019. Deep interest evolution network for click-through rate prediction. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 33. 5941–5948.

[46] Guorui Zhou, Chengru Song, Xiaoqiang Zhu, Ying Fan, Han Zhu, Xiao Ma, Yanghui Yan, Junqi Jin, Han Li, and Kun Gai. 2018. Deep Interest Network for Click-through Rate Prediction. *arXiv:1706.06978 [cs, stat]* (Sept. 2018). arXiv:1706.06978 [cs, stat]

[47] Guorui Zhou, Xiaoqiang Zhu, Chenru Song, Ying Fan, Han Zhu, Xiao Ma, Yanghui Yan, Junqi Jin, Han Li, and Kun Gai. 2018. Deep interest network for click-through rate prediction. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 1059–1068.