香港城市大學
City University of Hong Kong

國立臺灣大學
National Taiwan University

FlashEmbedding: Storing Embedding Tables in SSD for Large-Scale Recommender Systems

Hu Wan[1*], Xuan Sun[1*], Yufei Cui[1], Chia-Lin Yang[2], Tei-wei Kuo[1 2], and Chun Jason Xue[1]

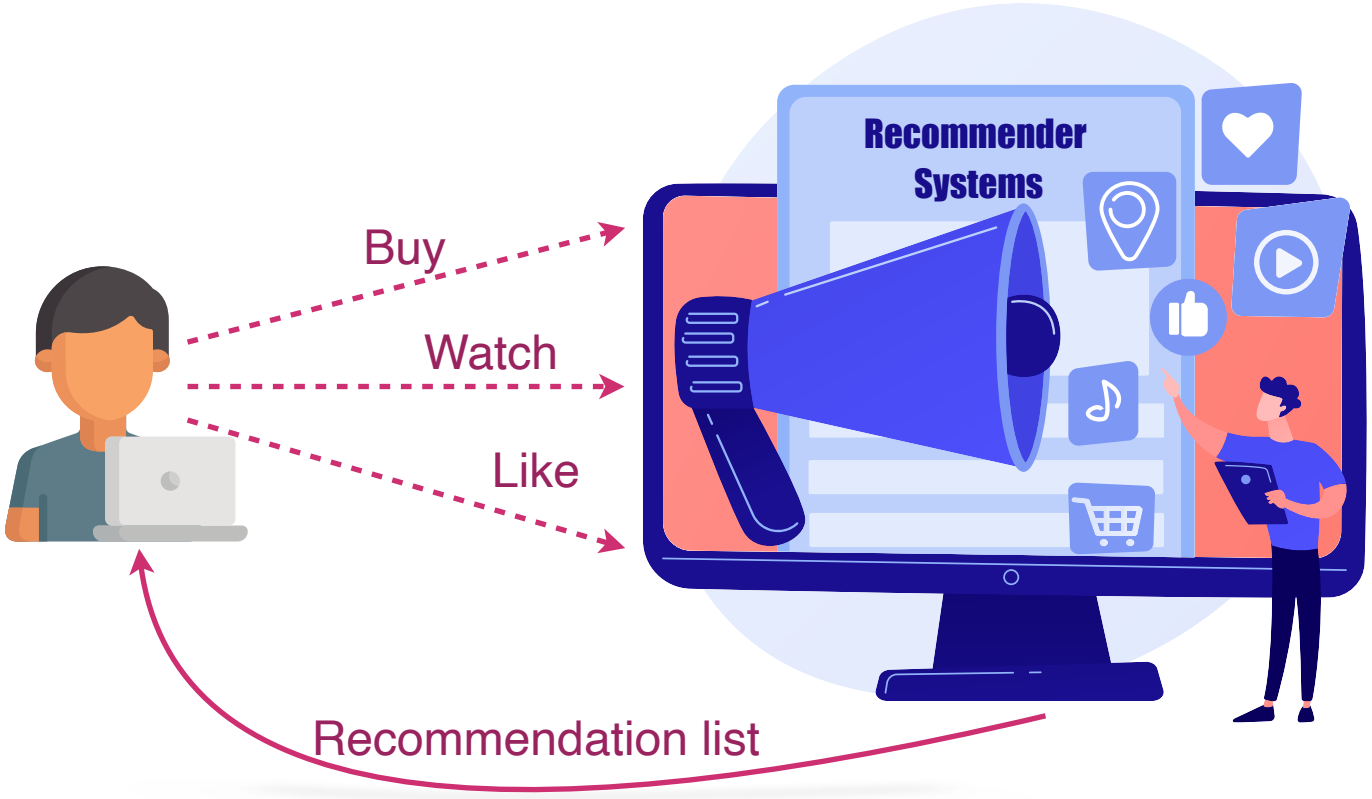
12th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys 2021)

August 24-25, 2021

[1] City University of Hong Kong
[2] National Taiwan University

* Co-first author

Personalized Recommender Systems



Personalized recommender systems are at the core of a number of online services providers such as Amazon, Netflix, Youtube, and Facebook.

Recommender systems contribute:

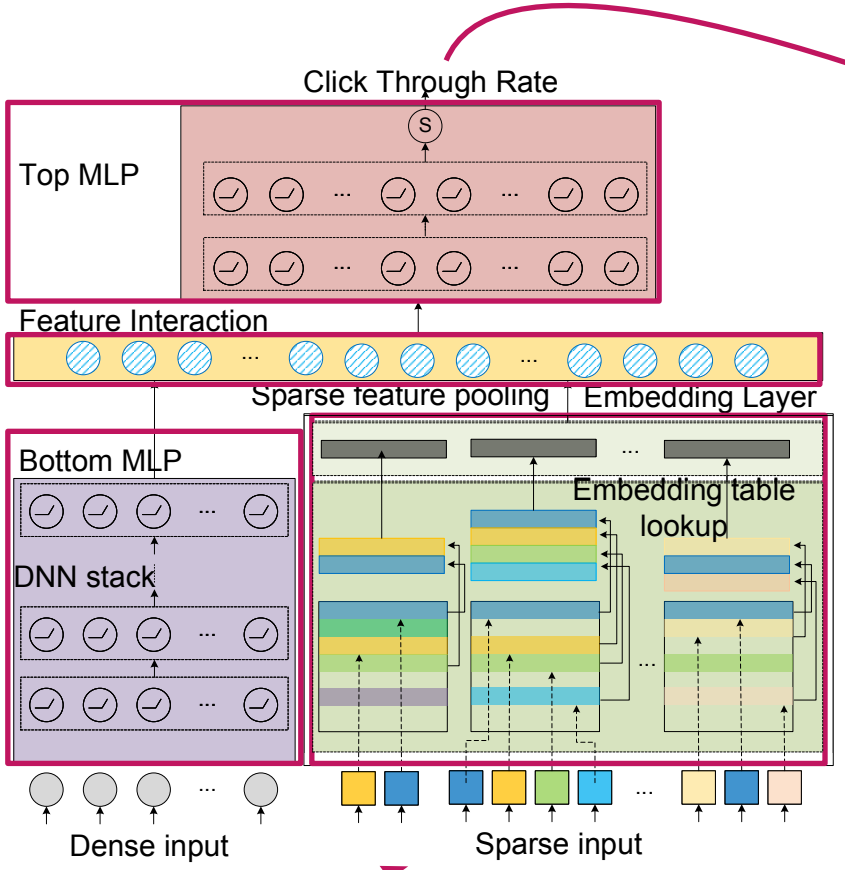
- Amazon: 35% of annual revenue[1]
- Netflix: 75% of all content consumed[1]
- Youtube: 60% of the clicks on the home screen[1]
- Facebook: 79% of datacenter inference cycles[2]

1 Dietmar Jannach and Michael Jugovac. Measuring the Business Value of Recommender Systems, Trans. Manage. Inf. Syst. 2019.
2 Gupta, et. al. The Architectural Implications of Facebook's DNN-based Personalized Recommendation Models. HPCA 2020.

Image credit: freepik.com

Modern DNN-based Recommender Systems

DLRM: Bottom MLP, top MLP, embedding layer, and feature interaction layer



▲ A typical architecture of deep learning recommendation model (DLRM).

The output is the predicted click-through rate (CTR).

Inputs to the model are a collection of dense and sparse features.

Dense (numerical) features

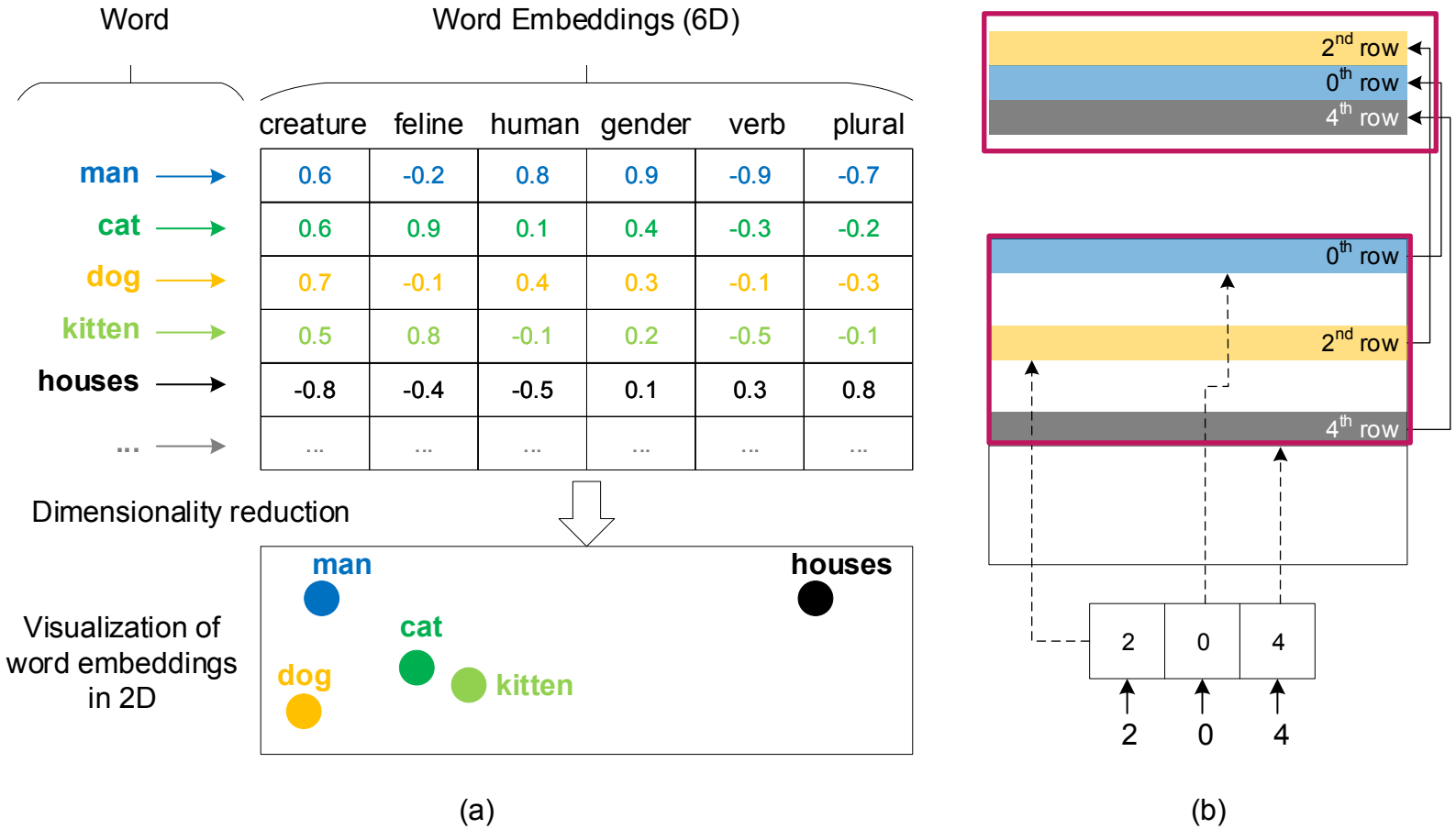
Dense features describe continuous inputs that are processed with MLP layers i.e., fully-connected layers.

Sparse (categorical) features

Sparse features are transformed to a dense representation by looking up **embedding tables** in the embedding layer.

Modern DNN-based Recommender Systems

Embeddings and embedding tables



▲ An embedding table is a sort of "lookup table"', indexed by the unique IDs of the sparse categorical features.

Memory Capacity Scaling Challenge of Embeddings

Embedding layer size

The size of an embedding layer scales proportionally to:

- the number of embedding tables
- the number of embedding vectors, or rows in the table, and
- the dimension of the embedding vector, or columns in the table.

Memory capacity scaling trends

Embedding tables can commonly contain billions of embedding vectors and would take up hundreds of GB of memory[1].

Models at the 1-10TB scale have also been deployed at Baidu and Google.

Both number of features and embeddings have grown an order of magnitude in only three years[2].

Call for new solutions

Just use more DRAM?

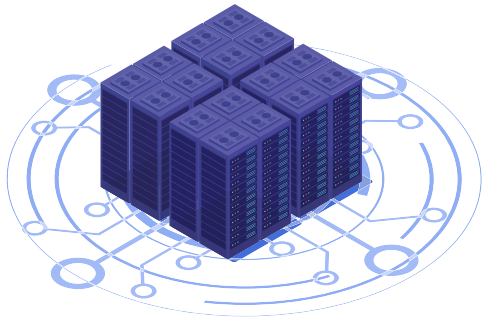
- Total cost of ownership (TCO) is high as data grows fast.
- Keeping these memory capacity hungry embeddings in memory is practically impossible for large scale recommender systems.

How about use solid-state drive (SSD)?

- Large capacity: storage capacity up to 100 TB[3]
- Good performance: low latency and high throughput
- Low cost: cost per gigabyte between $.11 and $.30

In large-scale deep learning-based recommender systems, limited physical memory capacity constrains the recommendation model that can be deployed.

One promising solution to this problem is storing embedding tables on much larger solid-state drives (SSDs).



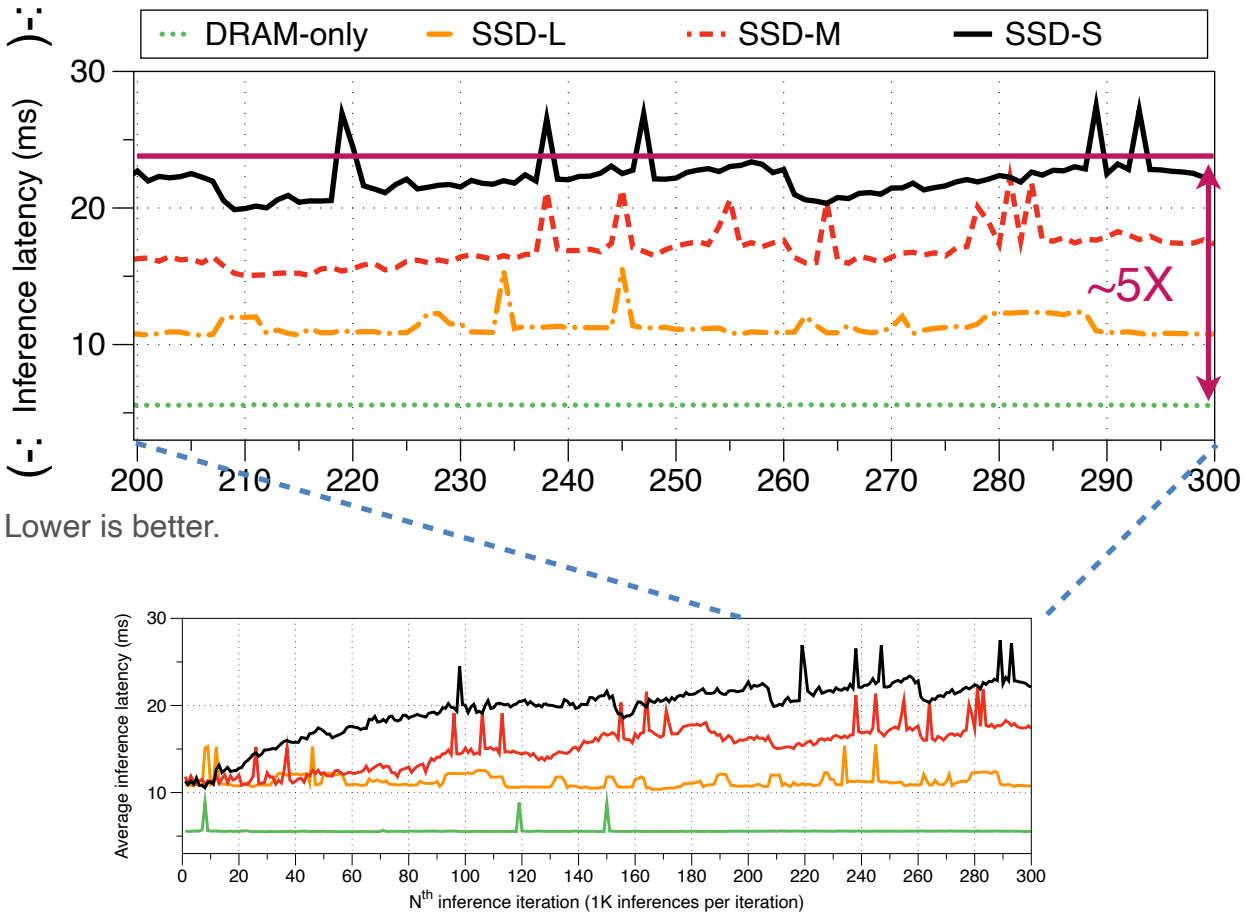
1 Gupta, et. al. DeepRecSys: A System for Optimizing End-To-End At-Scale Neural Recommendation Inference. ISCA 2020.
2 Lui, et. al. Understanding Capacity-Driven Scale-Out Neural Recommendation Inference. ISPASS 2021.
3. https://www.techradar.com/best/large-hard-drives-and-ssds

Problems of Storing Embedding Tables in SSD

Motivation Experiments

- Setup:
  - Open-source deep learning recommendation model (DLRM) framework and Criteo Ad dataset.
  - On-demand caching: Store embedding table in SSD and keep the recent and often-used part of the embedding layer in memory via file system page cache.
  - Evaluate three configurations of memory size, i.e., large (30GB), medium (10GB), and small (1GB).
  - Compare their latency to the original DLRM, which assumes infinite memory capacity and stores entire embeddings in DRAM, indicating the upper bound performance.
- Result:

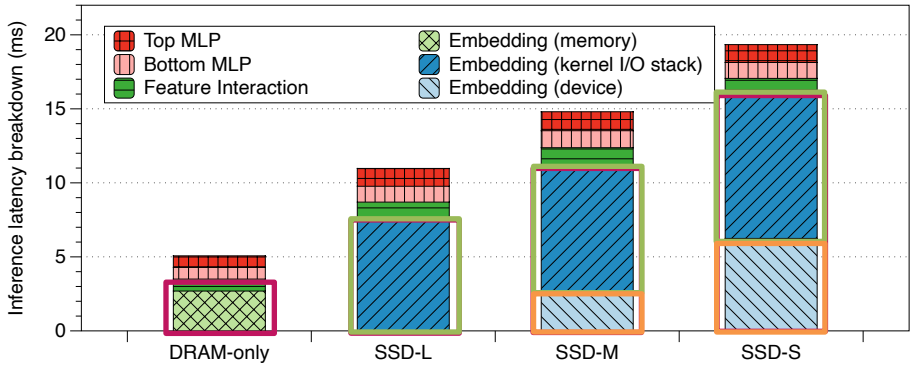
  As the amount of memory decreases, the inference latency of baseline SSD increases significantly and shows 5x slower than the ideal performance.



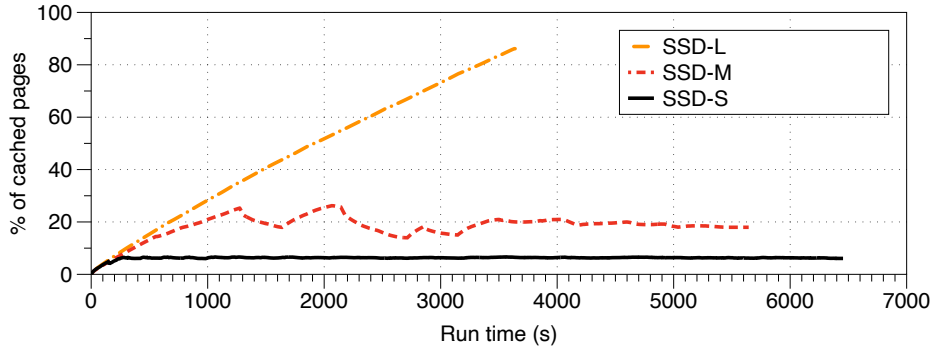
▲ Inference latency of a recommender system.

Problems of Storing Embedding Tables in SSD

Performance Overhead Analysis



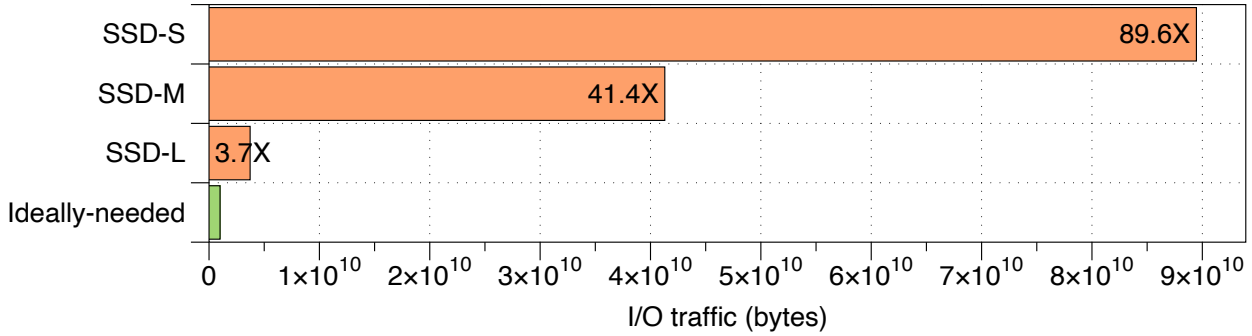
▲ Inference latency breakdown.



▲ % of pages cached in page cache during inference.

- Under high memory contention, recommender system suffers from thrashing, or excessive page swapping.

- Inference latency is dominated by the embedding layer, while the rest of the layers are almost evenly distributed.
- Operations inside the kernel I/O stack account for a large fraction of total I/O latency for all SSD-based recommender systems.
- Device layer latency increases rapidly as the size of available memory decreases.

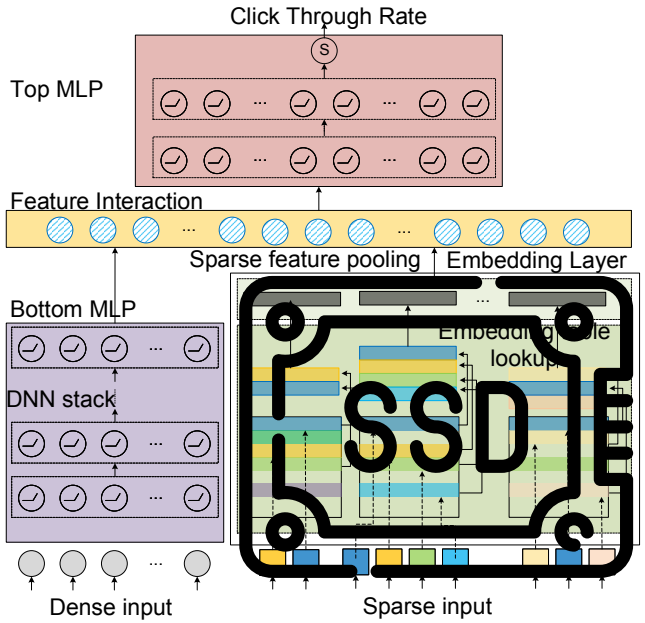


▲ Read I/O traffic of a recommender system.

- Because of a critical shortage of DRAM, thrashing occurs, leading to more I/O requests to SSD.
- The I/O granularity of recommender systems and underlying SSDs mismatch

Problems of Storing Embedding Tables in SSD

- Key Challenges
  - Challenge 1: High I/O stack overhead
  - Challenge 2: Large SSD read amplification
- Our solution: FlashEmbedding
  - Key idea 1: Allow recommender systems to directly access SSD
  - Key idea 2: Support lookup embedding tables in fine-grained granularity

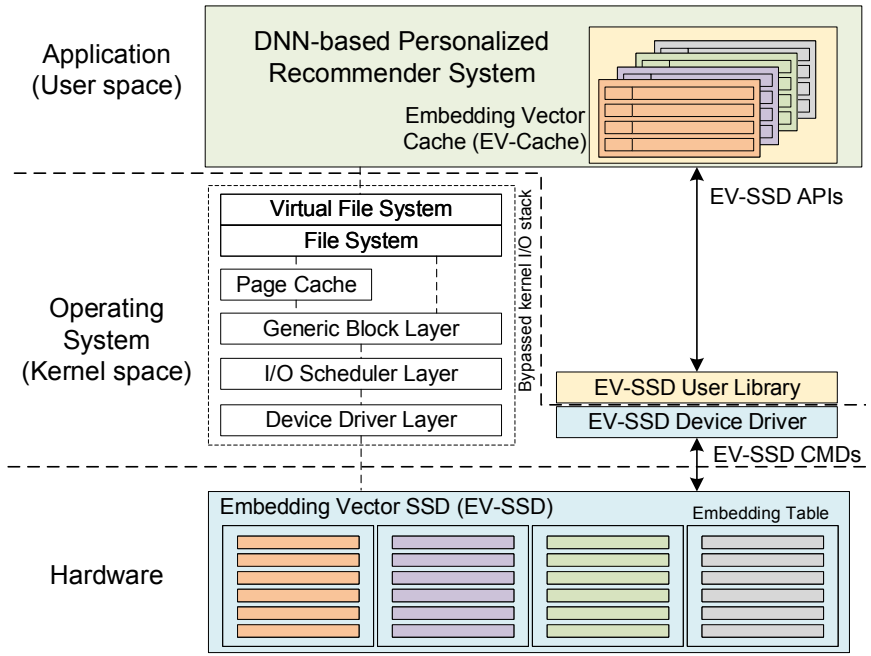


▲ Storing embedding tables in SSD

FlashEmbedding Design Overview

Storing Embedding Tables in SSD for Large-Scale Recommender Systems

- Hardware: EV-SSD, an embedding semantic-aware SSD
  - To support embedding vector lookup requests by taking advantage of two-stage fine-grained reading and MMIO interface
- Software: EV-Cache: an embedding vector cache
  - To reduce the access to EV-SSD by applying a bitmap filter and robin hood hashing-based LRU caches
- Putting it all together: Pipeline
  - To hide the I/O latency further by overlapping CPU computation with EV-SSD handling



▲ FlashEmbedding design overview

Embedding Vector SSD (EV-SSD)

Overview

- EV-SSD extends conventional SSD with the following modules:
  - MMIO Manager
  - Embedding Vector Translator (EV Translator)
  - Embedding Vector Flash Controller (EV-FC)
  - Embedding Vector Merger (EV Merger)

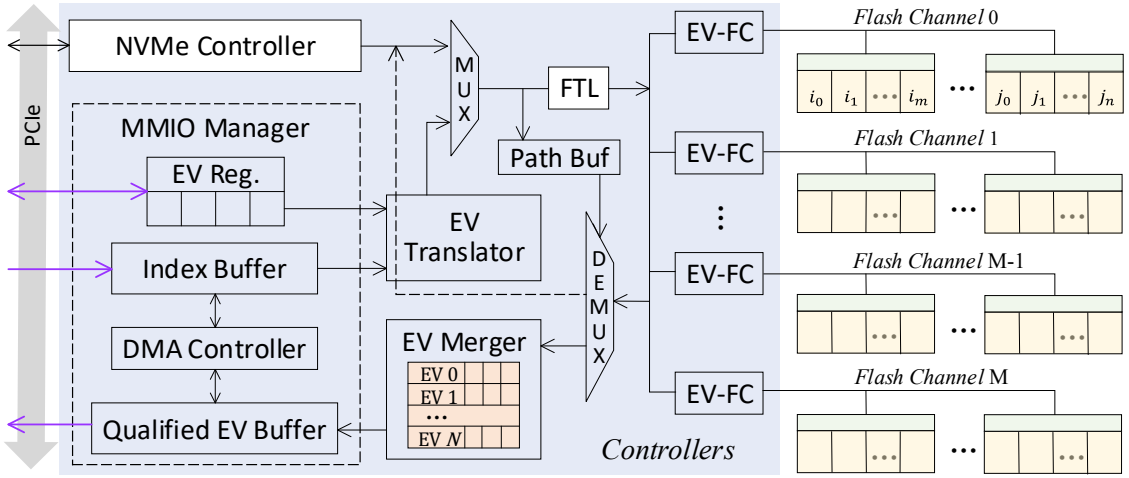
Two-stage fine-grained reading strategy

Device stage

EV Translator is used to translate EV lookup indices, and EV Merger takes charge of merging the scattered qualified EV lookup results returning by EV-FC.

Flash channel stage

EV-FC only fetches the desired EV data from flash channels.



▲ EV-SSD architecture

Embedding Vector SSD (EV-SSD)

MMIO Manager

- MMIO Manager is dedicated as EV lookup interface:
  - EV-SSD supports both block I/O requests and EV lookups, which are handled by the NVMe Controller and MMIO Manager separately.

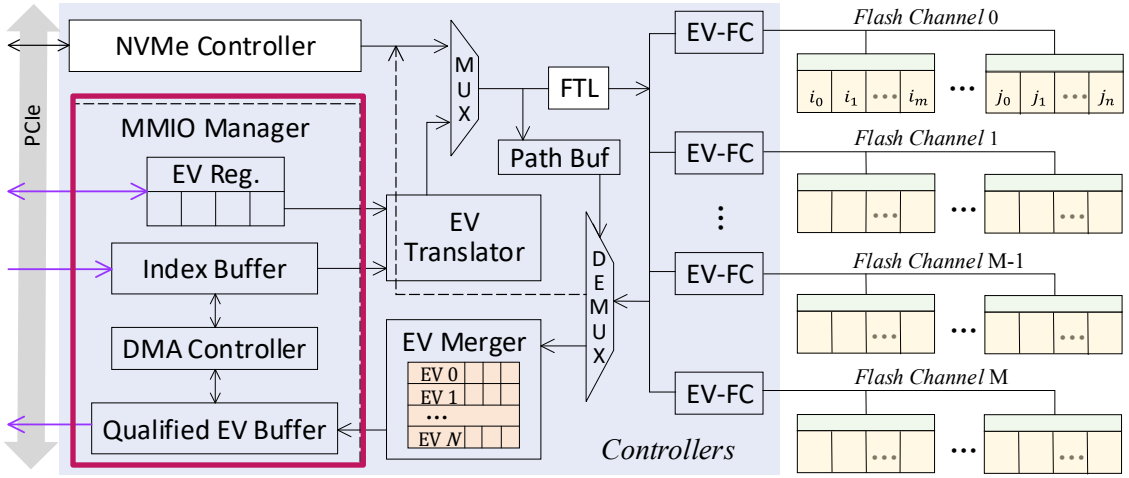
MMIO Manager support two modes:

Host-side memory interface

Exchange small control parameters with low latency, such as embedding table ID.

DMA data transmission

Transfer data blocks in bulk, including the lookup EV indices and qualified EVs.



▲ EV-SSD architecture

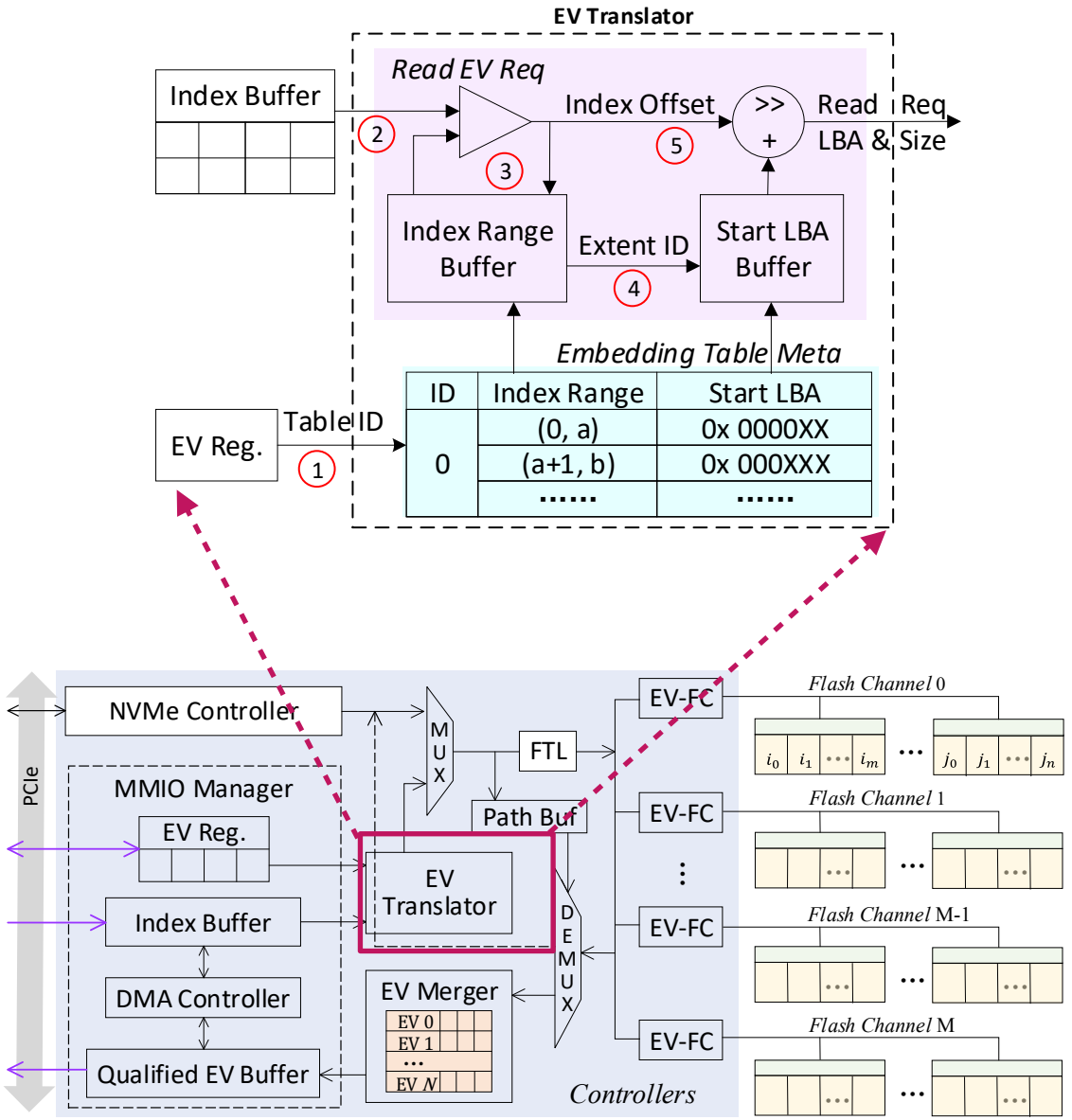
Embedding Vector SSD (EV-SSD)

EV Translator

- EV Translator is designed to parse the EV index in a table to the LBA (logical block addressing):
  - When a batch of EV lookups comes, the host side informs the EV translator with table ID and batch size and sends indices group to the Index Buffer in DMA mode

EV-SSD side translates the EV request in the following steps:

1. EV Translator fetches the metadata of the requested table to the SRAM for faster access.
2. Read EV Req fetches one index from the Index Buffer each time.
3. The extent ID of the current index is calculated by repeatedly comparing with the index ranges.
4. According to the extent ID, the start LBA of the EV is determined.
5. Figure out the final LBA of the EV and size of read request (EV size).

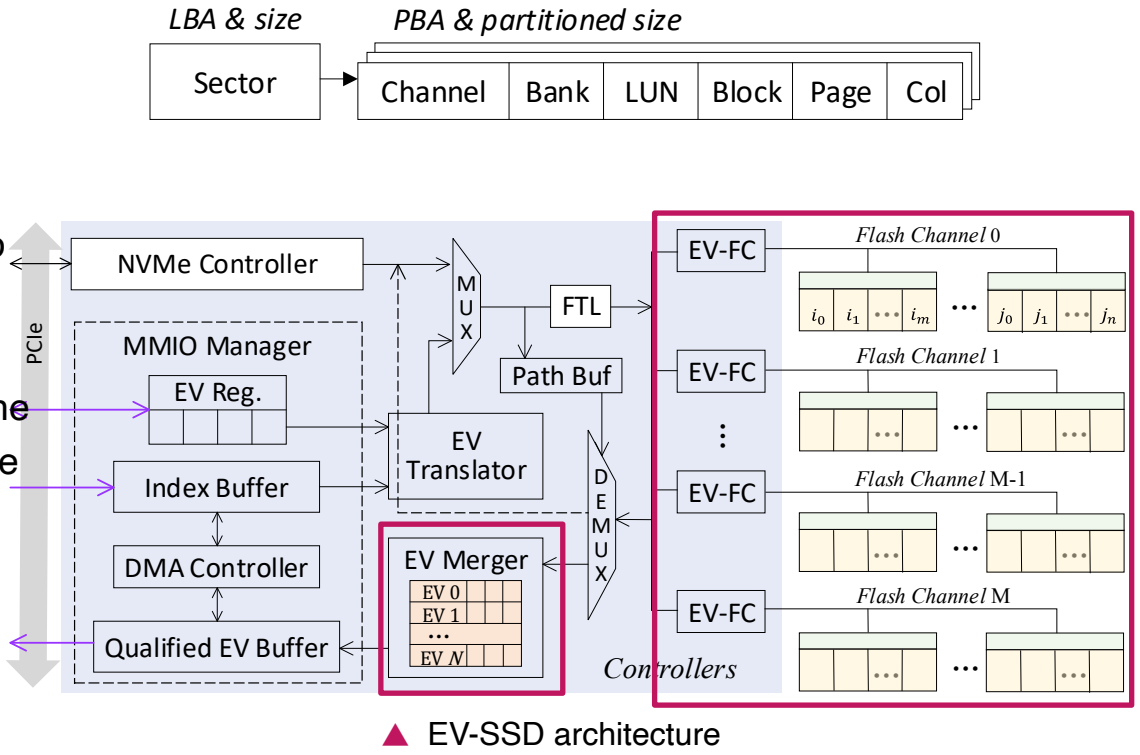


▲ EV-SSD architecture

Embedding Vector SSD (EV-SSD)

EV-FC

- Fine-grained data transfer:
  - LBA are transformed to PBA (physical block addressing), read request size is partitioned.
  - The data transfer starts from the offset, and size is configured to EV_{flash_size} when performing the EV reading request.
  - The whole page data are flushed from the flash cell to the flash buffer and then transferred to the outside controllers. We drop the remaining data on this page due to the overall poor locality of the embedding workload.
  - The read latency of data transfer is reduced to EV_{flash_size}/Page size of the original transfer time.



▲ EV-SSD architecture

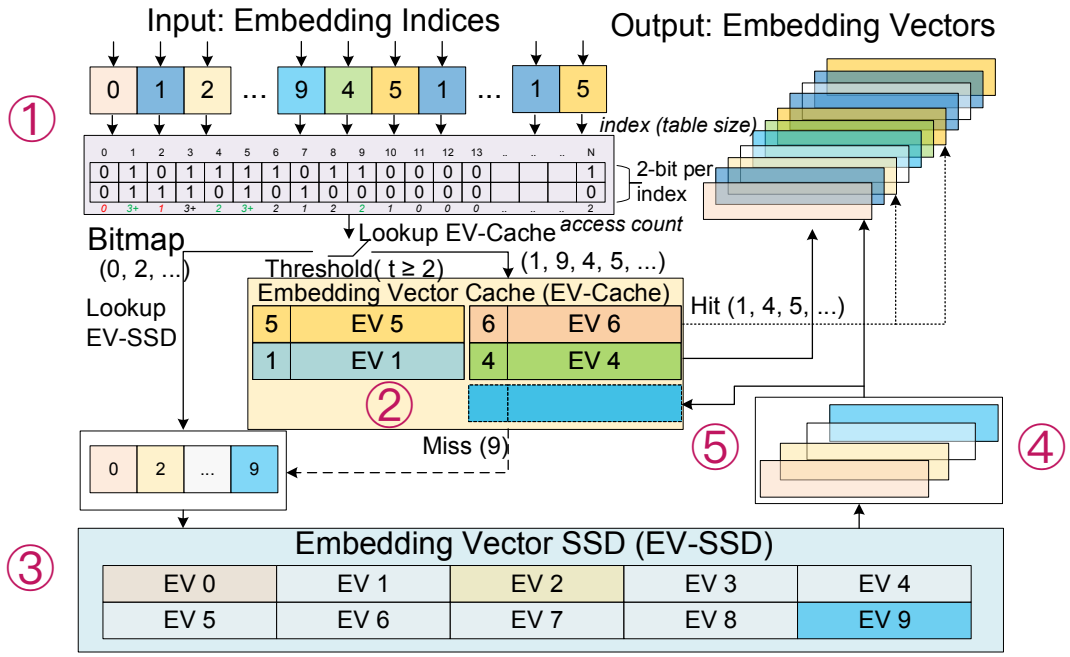
EV-Merger

- EV Merger is in charge of merging EVs from different channels:
  - When the batch of EV lockups are all finished, the status register in EV Reg. is set to ready.
  - Before reading SSD results, the host will check the result status register in SSD first using MMIO. Once it is ready, data in Qualified EV Buffer will be transmitted to the host in DMA mode.

Embedding Vector Cache (EV-Cache)

Overview

- Bitmap-based index filter
- Robin hood hashing-based LRU caches



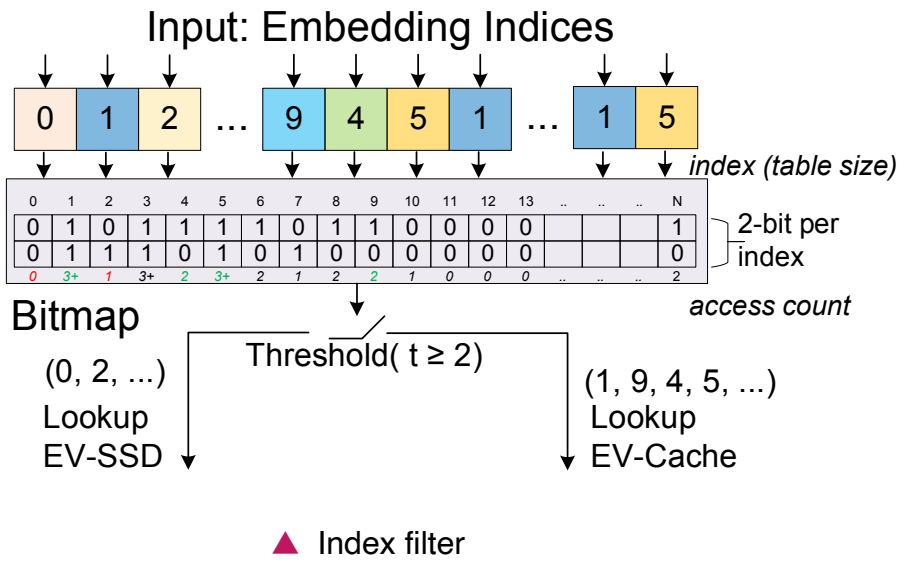
▲ Workflow of embedding lookup with EV-Cache

1. Input indices are filtered by index filter. Index filter use access count from the bitmap to determine whether an embedding vector should be cached or bypassed.
2. If the access history bits indicate high locality than a user-specified threshold *t*, it will look up and load corresponding internal vectors from the embedding vector cache. Otherwise, the requests bypass the embedding table cache.
3. Together with those missed in the embedding vector cache (a rare case), they are forwarded to SSD to initiate an SSD embedding lookup.
4. Embedding vectors loaded from SSD will send back to the upper layer for further process.
5. If the access count is greater than or equal to the threshold after this access, we will add newly fetched embedding vectors to the embedding table cache.

Embedding Vector Cache (EV-Cache)

Bitmap-based index filter

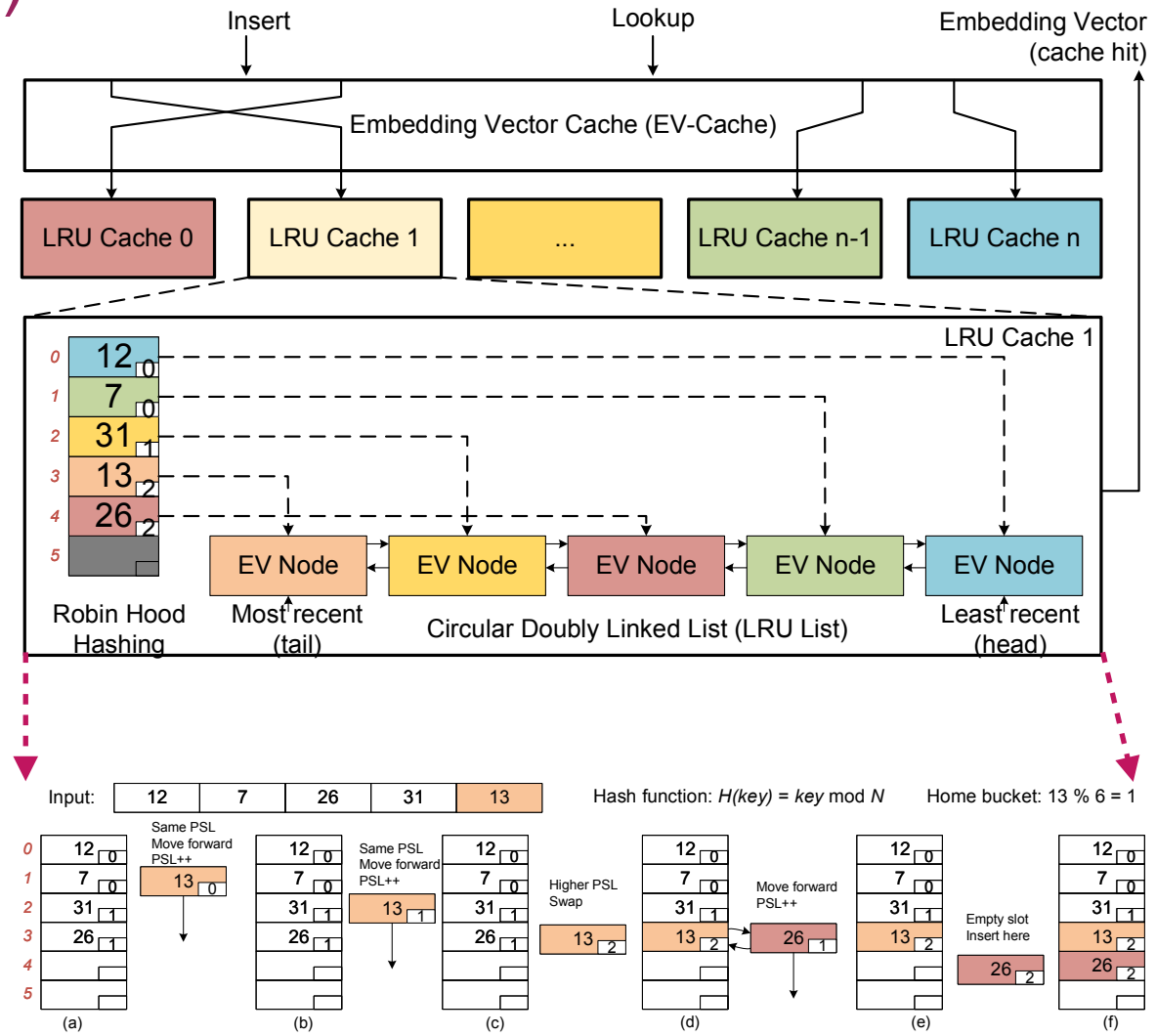
- Insight:
  - A small subset of embedding entries exhibit relatively higher reuse characteristics while most vectors are rarely accessed.
- An index filter based on the bitmap.
  - Two bits are given for each embedding vector in an embedding table to represent the access frequency: never accessed, once, twice, and three or more times.
- A given threshold decides whether a vector is to be added to the cache or not.
  - If the access frequency is smaller than the threshold, send indices to SSD.
  - Otherwise, send indices to SSD and add the loaded vectors to the embedding vector cache.



Embedding Vector Cache (EV-Cache)

Robin hood hashing-based LRU caches

- LRU (least recently used) based vector cache:
  - Use a hashmap to store key-value pairs, and a circular double-linked list to record the access order of each element.
  - Use a set of LRU caches to serve embedding lookup requests in a multi-threaded environment.
- Requirements for an efficient hashmap for EV-Cache:
  - Look up and insert should fast.
  - Look up should fail fast for searching a non-existent key.
- Robin hood hashing[1]:
  - Open addressing with robin hood collision resolution strategy.
  - Robin Hood hashing reduces the maximum probe sequence length and the long tail in probe sequence length distribution.



▲ EV-Cache architecture

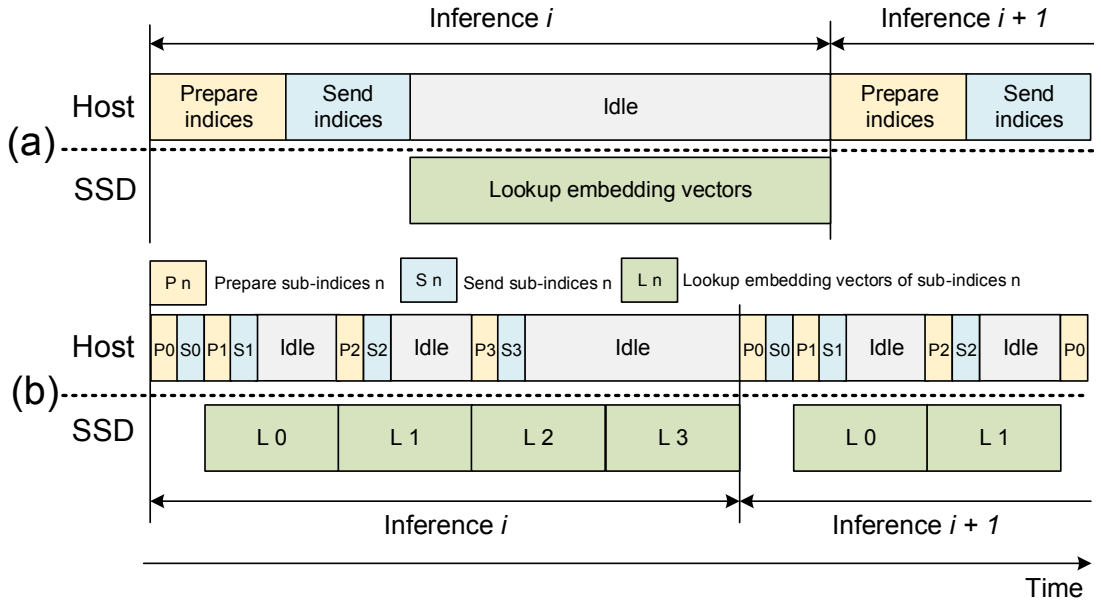
1 Pedro Celis, et. al. Robin hood hashing.SFCS 1985.

Putting It All Together

Pipeline

- FlashEmbedding follows three steps to process indices and lookup an embedding table
  - prepare indices,
  - send indices, and
  - lookup embedding vectors in SSD.

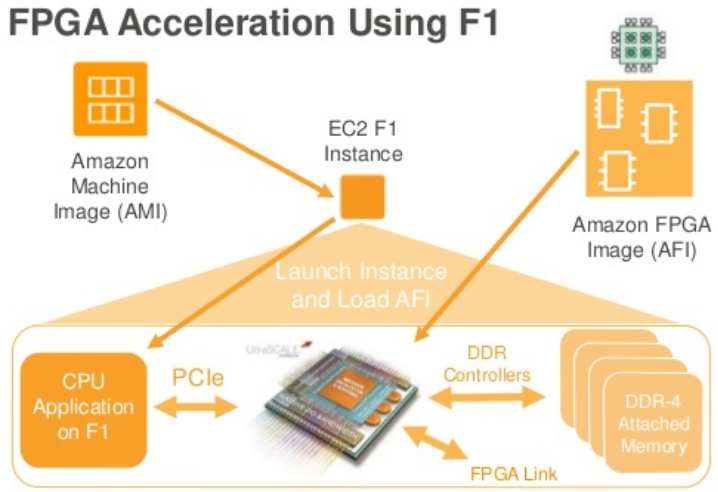
- Pipeline
  - partition the input indices in a large batch into several small parts, and
  - send the prepared sub-indices to SSD in time,
  - once the EV-SSD receives the indices, it can then perform an embedding look up, and
  - the host CPU continues to prepare the next part of indices.



Implementation

EV-SSD

- We implement our prototype EV-SSD based on FPGA on AWS F1:
  - Enrich the FPGA chip to realize the functionality of EV-SSD controllers, and
  - Employ four DDR4 banks to emulate four flash channels,
  - Modify the NVMe driver developed by Insider framework[1] to imitate the working mechanism of real NVMe protocol,
  - Add a delay unit in the FPGA to emulate the read latency.



EV-Cache

- We implement EV-Cache in C++ and integrate it into the PyTorch-based DLRM framework[2]:
  - Intercept the function call for embedding lookups from EmbeddingBag operator to EV-SSD
  - Robin hood hashing[3]

PyTorch

1 https://github.com/zainryan/INSIDER-System
2 https://github.com/facebookresearch/dlrm
3 https://github.com/martinus/robin-hood-hashing

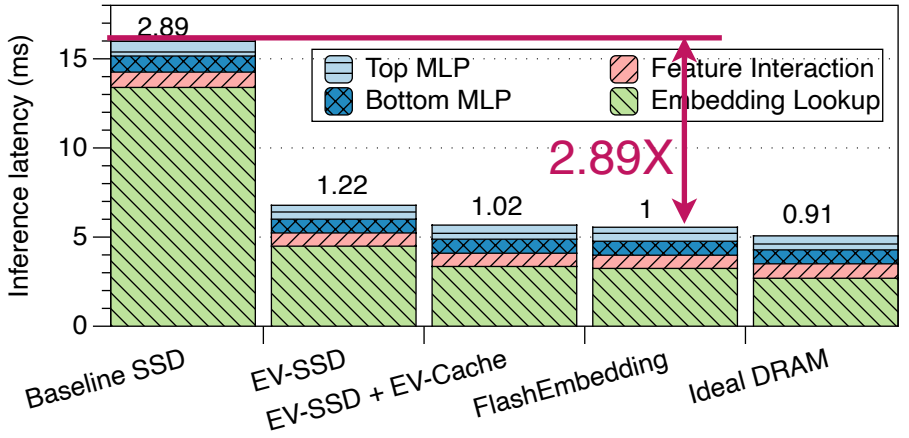
Image credit: Amazon AWS, Pytorch Project

Evaluation

Experimental setup:

- Evaluate on AWS EC2 F1 instance with a Xilinx Virtex 57 Plus UltraScale XCVU9P card.
- Use DLRM framework and conduct experiments on public real-world dataset from Criteo AI Labs.
- We store the five largest embedding tables (around 4GB total) on SSD and the remaining 21 small tables in DRAM (around 70MB total).
- Default batch size to 128. Default embedding dimension is 32. Embeddings are stored in fp32 format.
- EV-Cache: use a threshold of 2 for bitmap, 1% of embedding table size for cache size (total cache size is around 40MB).
- EV-SSD: evaluate three typical read latency of low- to high-end SSDs, i.e., 4us (default), 10us, and 30us.

End-to-end Performance

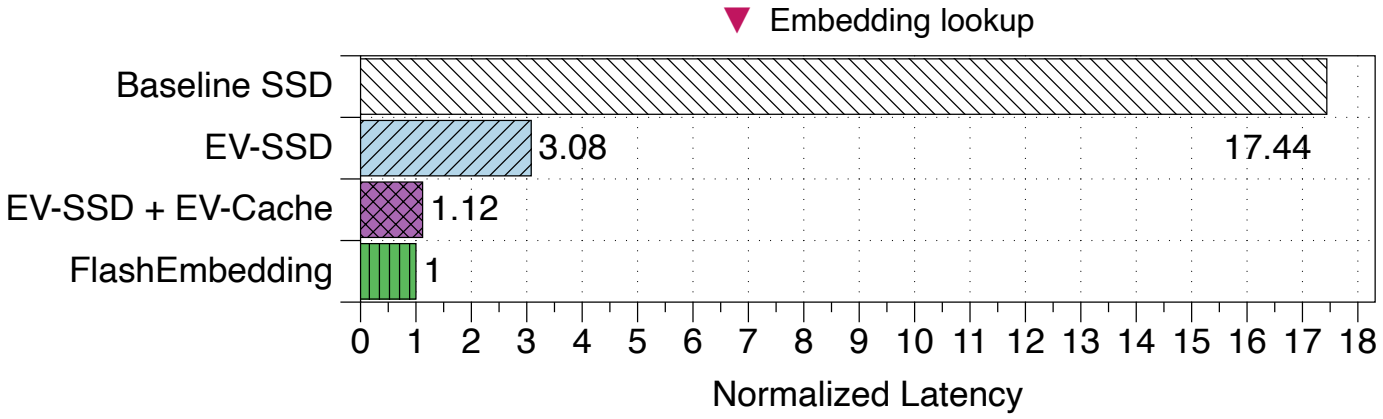


Up to 2.89x speedup compared to Baseline SSD

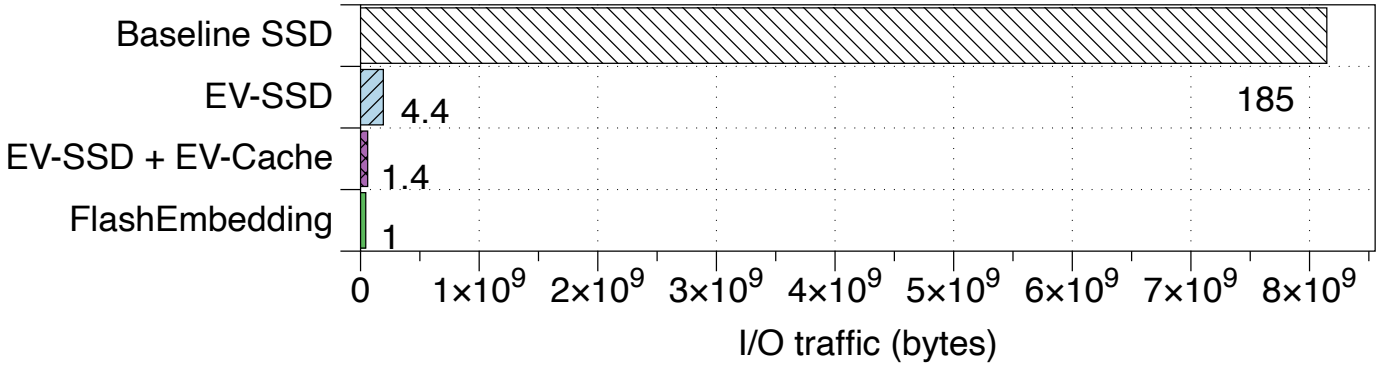
Comparable performance to Ideal DRAM. This opens up new possibilities to store and lookup embedding tables on SSDs for large-scale recommendation with limited memory capacity.

Evaluation

Effectiveness of Proposed Techniques

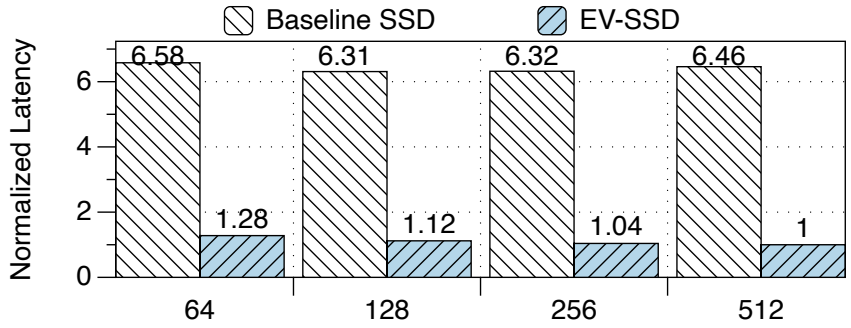


FlashEmbedding achieves up to 17.44× lower latency in embedding lookups

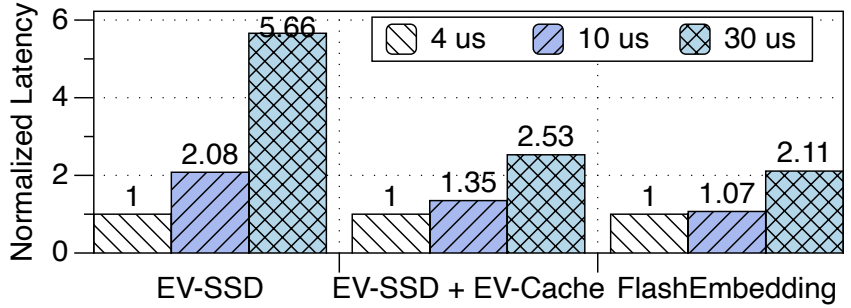


FlashEmbedding and its variants incur much less read I/O traffic than Baseline SSD.

Sensitivity Study



Sensitivity to batch size: EV-SSD scale well.



Sensitivity to SSD latency: FlashEmbedding shows better scalability.

Cache size	01.%	1%	5%	10%
Normalized Latency	1.19	1	0.99	1.09

Sensitivity to EV-Cache size

Conclusion

- We propose FlashEmbedding, a novel system architecture for storing embedding tables on SSDs for large-scale recommendation inference under memory capacity-limited systems.

- We identify two performance problems of current architecture:
  - the long I/O stack and
  - the tremendous read amplifications of the block-interface SSD.

- We propose three key optimizations to address both problems:
  - embedding vector SSD,
  - embedding vector cache, and
  - pipeline.

- The evaluation results show that FlashEmbedding outperforms the baseline.

香港城市大學
City University of Hong Kong

國立臺灣大學
National Taiwan University

Q & A

Session 1: Storage

August 24, 10:15 - 11:35