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

Hu Wan[*]
City University of Hong Kong

Xuan Sun[*]
City University of Hong Kong

Yufei Cui
City University of Hong Kong

Chia-Lin Yang
National Taiwan University

Tei-Wei Kuo
City University of Hong Kong
National Taiwan University

Chun Jason Xue
City University of Hong Kong

ABSTRACT

We present FlashEmbedding, a hardware/software co-design solution for storing embedding tables on SSDs for large-scale recommendation inference under memory capacity-limited systems. FlashEmbedding leverages an embedding semantic-aware SSD, an embedding-oriented software cache, and pipeline techniques to improve the overall performance. We evaluate the performance of FlashEmbedding with our FPGA-based prototype SSD on a real-world public dataset. FlashEmbedding achieves up to 17.44× lower latency in embedding lookups and 2.89× lower end-to-end latency than baseline solution in a memory capacity–limted system.

CCS CONCEPTS

• Information systems → Recommender systems; **• Software and its engineering → Secondary storage**.

KEYWORDS

Recommender systems, Embedding, Solid-state drive (SSD)

ACM Reference Format:
Hu Wan, Xuan Sun, Yufei Cui, Chia-Lin Yang, Tei-Wei Kuo, and Chun Jason Xue. 2021. FlashEmbedding: Storing Embedding Tables in SSD for Large-Scale Recommender Systems. In *ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '21), August 24–25, 2021, Hong Kong, China.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3476886.3477511

[*]Both authors contributed equally to this research.

1 INTRODUCTION

Modern deep learning–based recommender systems are data-intensive due to the need to handle sparse, categorical input features. Recommender systems exploit the categorical information while maintaining the natural diversity within the features by using embedding tables to map each category to a unique embedding vector within a lower-dimensional embedded space. Typically, they store all embedding tables in memory to reduce latency for serving user requests. However, embedding tables can commonly contain billions of embedding vectors and would take up hundreds of gigabytes of memory [8, 20] that can exceed the system memory capacity (typically 16GB–256GB [22]). As the model size for production-scale recommender systems scales up, the embedding layer becomes so large that it cannot be entirely stored in memory. One promising solution to this problem is storing embedding tables on much larger solid-state drives (SSDs). However, naively substituting SSD for DRAM significantly reduces the performance of recommender systems.

Given this landscape, this paper focuses on addressing the challenge of recommendation inference in large-scale deep learning–based recommender systems, where the limited physical memory capacity constrains the recommendation model that can be deployed. After careful analysis, we find that the performance degradation of SSD-based recommendation inference is mainly attributed to two factors: the long I/O stack and the tremendous read amplifications of the block-interface SSD. We propose FlashEmbedding, a hardware/software co-design solution for storing embedding tables on SSDs for large-scale recommender systems, to overcome these issues. First, we design embedding vector SSD (EV-SSD) to support embedding vector lookup requests by taking advantage of two-stage fine-grained reading and MMIO interface. Second, we design embedding vector cache (EV-Cache) to reduce the access to EV-SSD by applying a bitmap filter and robin hood hashing–based LRU caches. Third, we use the pipeline to hide the I/O latency further by overlapping CPU computation with EV-SSD handling.

We implement and evaluate FlashEmbedding based on an FPGA-based prototype SSD. Experimental results show that FlashEmbedding delivers lower latencies by up to 17.44× in embedding lookups and 2.89× in end-to-end latency than a typical baseline.

2 BACKGROUND AND MOTIVATION

2.1 Embedding

A typical architecture of deep learning-based personalized recommendation model comprises bottom MLP, top MLP, embedding and feature interaction layers [6, 7, 12, 21]. Its output is the predicted click-through rate. Its inputs are a collection of dense and sparse features. Dense features describe continuous inputs that are processed with MLP layers. Sparse features are transformed to a dense representation by looking up embedding tables in the embedding layer. An embedding table is a sort of lookup table in which each row is a vector of floating-point value that corresponds to a category for a sparse feature. An embedding table lookup reads a few vectors from the table when given a set of indices. The embedding vectors returned are aggregated into a single vector through simple pooling operations such as sum.

The size of the embedding layer scales proportionally to the number of embedding tables, the number of embedding vectors inside each table, and the dimension of the embedding vector. As the model for production-scale recommender systems scales up, it consistently exceeds the memory capacity limits of typical computer systems.

2.2 SSD-based Recommendation Inference

When the entire set of embedding tables cannot fit in main memory, one promising approach is storing huge embedding tables to slower but larger SSDs [5, 13]. To better understand the use of SSDs for recommendation inference, we utilize the deep learning recommendation model (DLRM) framework and Criteo Ad dataset [4] to evaluate how well the recommender system performs.

We use the NVMe SSD attached to the AWS F1 instance to store embedding tables. Following previous work [5, 13], we manage the embedding layer by adopting an on-demand caching strategy. On-demand caching keeps the recent or often-used part of the embedding layer in memory via file system page cache while storing the rest in the slower but larger SSD. To analyze the impact of the limited capacity of working memory, we use the Linux cgroup tool to limit the amount of memory available for the recommender system processes to specified sizes. We evaluate three different configurations of memory size, i.e., large (30GB), medium (10GB), and small (1GB), denoted as SSD-L, SSD-M, and SSD-S, respectively. We compare their latency to the original DLRM version, denoted as DRAM-only, which assumes infinite memory capacity and stores entire embeddings in DRAM, indicating the upper bound performance. Each run is preceded by a warmup period of sufficient length to stress the memory usage and ensure that we measure steady-state execution only.

End-to-End Inference Latency. Figure 1 shows the average inference latency for every 1000 inferences. As the amount of memory decreases, the inference latency of baseline SSD increases significantly and shows 5× slower than the ideal performance. We further break down the end-to-end inference latency into different components, as shown in Figure 2. Inference latency is dominated by the embedding layer, while the rest of the layers are almost evenly distributed. The recommender system must improve the efficiency of memory-intensive operations such as embedding table lookups to optimize the performance.

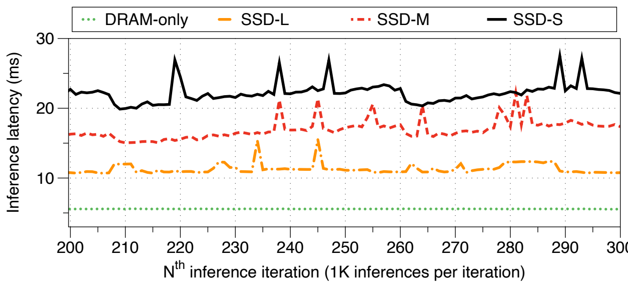

Figure 1: Inference latency of a recommender system.

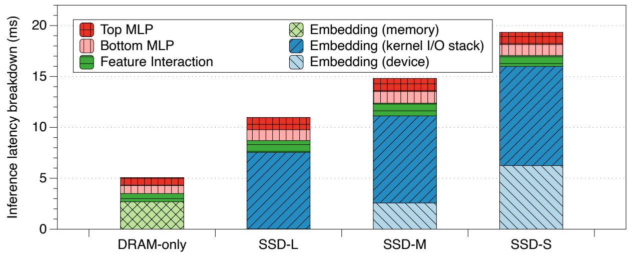

Figure 2: Inference latency breakdown.

Performance Overhead Analysis. Operations inside the kernel I/O stack account for a large fraction of total I/O latency for all SSD-based recommender systems. These overheads demonstrate that I/O stack costs are the primary limiter on SSD-based recommender system performance. Because page cache will cache frequently used embeddings and serve most embedding lookup requests if it has the request embedding. Notice that device layer latency increases rapidly as the size of available memory decreases. Under high memory contention, the recommender system suffers from thrashing, or excessive page swapping, as the memory is not enough to contain the input data. As shown in Figure 3, the system is pressured to continuously move memory pages in and out of memory and swap space. The system can discard cached pages directly, but follow-up requests for the same embedding must load from SSD again.

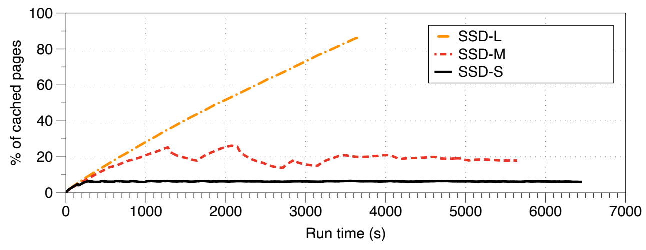

Figure 3: Pages cached in page cache during inference.

Figure 4 shows the read I/O traffic of the recommender system. As the size of memory decreases, the read traffic increases 3.7×, 41.4×, and 89.6× compared to ideally-needed traffic, which assumes underline SSD is byte-addressable. The reason is two-fold. First, because of a critical shortage of DRAM, thrashing occurs, leading to more I/O requests to SSD. Second, the I/O granularity of recommender systems and underlying SSDs mismatch, leading to significant read amplification. The typical size of an embedding vector ranges from 128B to 2KB [5, 13] while the flash page size is usually larger than 4KB. Worse, prior work [12] shows that embedding table lookup operations have little to no spatial locality. Therefore, reading more data in such a large size doesn't help increase the utilization of SSD read bandwidth or improve page cache efficiency. Instead, it would degrade the overall I/O throughput.

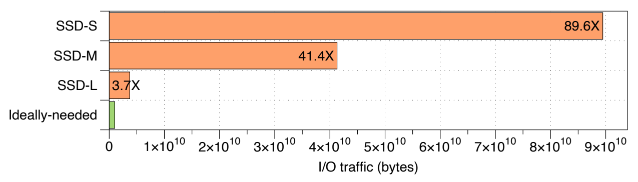

Figure 4: Read I/O traffic of a recommender system.

In summary, through the characterization study, we root-cause two factors as crucial limiters of the prior approach relying on SSD for storing embeddings: high I/O stack overhead and large SSD read amplification. Motivated by the above issues, we aim to allow recommender systems to directly access SSD and support lookup embedding tables in fine-grained granularity.

3 FLASHEMBEDDING DESIGN

3.1 System Overview

We present FlashEmbedding, a full system solution for addressing the memory capacity-limited challenge of the embedding layer in large-scale deep learning-based recommender systems. On the hardware side, we propose an embedding semantic-aware SSD, referred to as EV-SSD, to fast access the embedding vectors in SSD by reducing read amplification and I/O stacks. On the software side, we propose an embedding vector cache, referred to as EV-Cache, dedicated to efficiently cache the embedding vector using a small amount of memory to exploit the modest level of temporal locality within embedding tables and to reduce long SSD read latency. Putting them together, we reveal an efficient solution for storing embedding tables on SSDs.

3.2 Embedding Vector SSD

3.2.1 EV-SSD Architecture. Conventional NVMe SSD consists of flash memory channels and SSD controllers, including *NVMe Controller*, Flash Translation Layer (*FTL*), and Flash Controllers (*FC*s). EV-SSD extends conventional NVMe SSD with the following modules: *MMIO Manager*, Embedding Vector Translator (*EV Translator*), Embedding Vector Flash Controller (*EV-FC*) and Embedding Vector Merger (*EV Merger*).

To address the read amplification, we adopt a two-stage fine-grained reading strategy. At the 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*. At the flash channel stage, *EV-FC* only fetches the desired EV data from flash channels.

MMIO Manager is dedicated as EV lookup interface. As shown in Figure 5, EV-SSD supports both block I/O requests and EV lookups, which are handled by the *NVMe Controller* and *MMIO Manager* separately. The *MMIO Manger* supports both host-side memory interface and DMA mode data transmission. The former method can write the EV Register (*EV Reg.*) directly. It is used to exchange small control parameters with low latency, such as embedding table ID. The DMA transmission is applied to transfer data blocks in bulk, including the lookup EV indices and qualified EVs.

EV Translator is designed to parse the EV index in a table to the LBA. As shown in Figure 6, 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 according to 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)

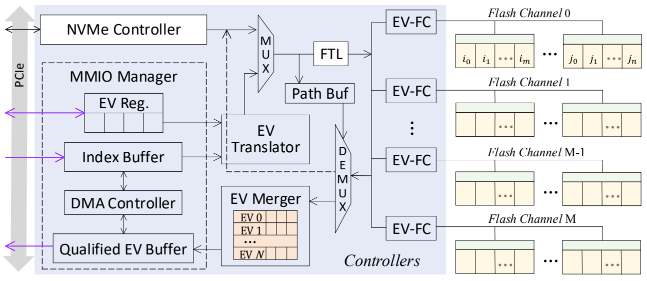

Figure 5: EV-SSD architecture.

Based on the index offset in this extent and $EV_{\text{Dimension}}$, we figure out the final LBA of the EV. To achieve the fine-grained reading, the size of read request is set to matching with the EV size, $EV_{\text{size}} = EV_{\text{Dimension}} \cdot \text{sizeof}(float)$.

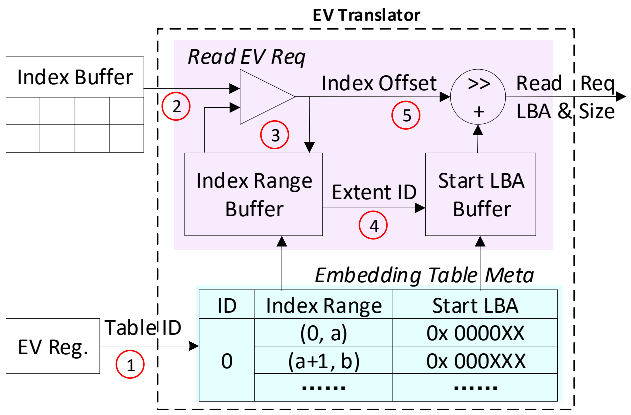

Figure 6: Working mechanism of *EV Translator*.

In order to fully utilize the parallelism of flash channels, *FTL* algorithm is designed for striping EVs on all flash channels, with the stripe unit size $EV_{\text{flash\_size}} = \lceil \frac{EV_{\text{size}}}{\#channel \cdot CH_{\text{witdh}}} \rceil \cdot CH_{\text{witdh}}$ [1]. After *FTL* translation, the physical block read requests are assigned to the related *EV-FC*.

EV-FC. The minimal storage unit of flash memory is the page. When performing a single page read, the whole page data are flushed from the flash cell to the flash buffer (marked in green in Figure 5) and then transferred to the outside controllers. Thanks to this characteristic, the data transfer can start from the offset, and size is configured to $EV_{\text{flash\_size}}$ when performing the EV reading request. 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 $EV_{\text{flash\_size}}/P_{\text{size}}$ of the original transfer time due to the flash channel stage fine-grained reading.

[1] CH_{witdh} is denoted as channel width.

EV Merger is in charge of merging EVs from different channels, which will be kept in the *Qualified EV Buffer* temporarily. 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.

3.3 Embedding Vector Cache

3.3.1 EV-Cache Workflow. We incorporate an in-memory embedding vector cache for each embedding table that resides in SSD to exploit the embedding vectors reuse. Figure 7 shows the workflow of embedding table lookup with EV-Cache. EV-Cache employs dedicated bitmap as filters to prune paths early. It takes hints of the access count from the bitmap to determine whether an embedding vector should be cached or bypassed. 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. Together with those missed in the embedding vector cache (a rare case), they are forwarded to SSD to initiate an SSD embedding lookup. Embedding vectors loaded from SSD will send back to the upper layer for further process. 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 cache reduces the quantity of SSD requests thanks to the moderate locality, reducing SSD accesses.

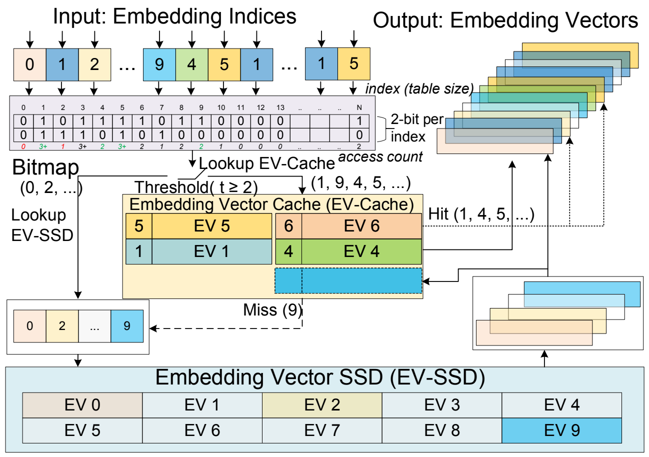

Figure 7: Workflow of embedding lookup.

3.3.2 Bitmap Filter. We devise an index filter based on the bitmap. We tightly integrate it with the embedding vector lookup to exploit the insight, i.e., during recommender system runs, a small subset of embedding entries exhibit relatively higher reuse characteristics while most vectors are rarely accessed. It provides the critical functionality to quickly identify the rarely accessed vectors in the lookup requests efficiently. As illustrated in Figure 7, 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.

3.3.3 Cache Design. Figure 8 shows the overall EV-Cache architecture. EV-Cache uses LRU (least recently used) based vector cache as its building block, which uses a hashmap to store key-value pairs, and a circular double-linked list to record the access order of each element. An efficient hashmap for EV-Cache should meet the following requirements: First, look up and insert should fast regardless of the number of items stored. Second, look up should fail fast for searching a non-existent key, too, allowing us to send the index to SSD as soon as possible. For this, we adopt a hashmap based on open addressing with robin hood, a collision resolution strategy proposed by Celis et al. [3]. Robin Hood hashing reduces the maximum probe sequence length and the long tail in probe sequence length distribution, achieving high performance. EV-Cache internally uses a set of LRU caches to serve embedding lookup requests to improve performance in a multi-threaded environment.

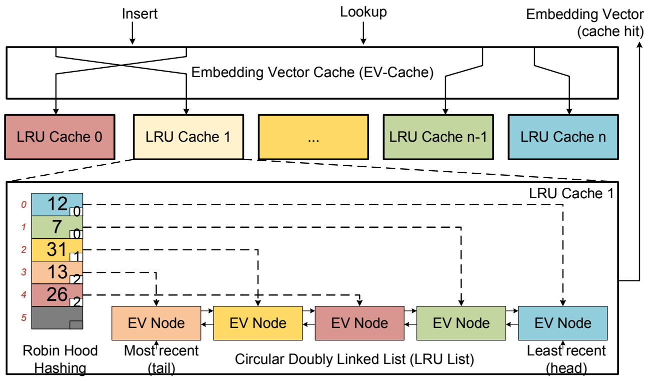

Figure 8: EV-Cache architecture.

3.4 Putting It All Together

FlashEmbedding follows three steps to process indices and lookup an embedding table: prepare indices, send indices, and lookup embedding vectors in SSD. Figure 9 (a) illustrates the timeline of embedding lookup. The host device and storage device work in a serial, synchronous fashion. We further hide I/O latency by overlapping computation of EV-Cache (first two steps) with I/O of EV-SSD (third step) in a coarse-level pipeline manner. We partition the input indices in a large batch into small parts and send the prepared sub-indices to SSD in time. Once EV-SSD receives the indices, it can then perform an embedding lookup, and the host CPU continues to prepare the next part of indices. Figure 9 (b) illustrates the optimized workflow with pipeline. The pipeline takes advantage of the parallelism between CPU and SSD and mitigates the idle time on the host side.

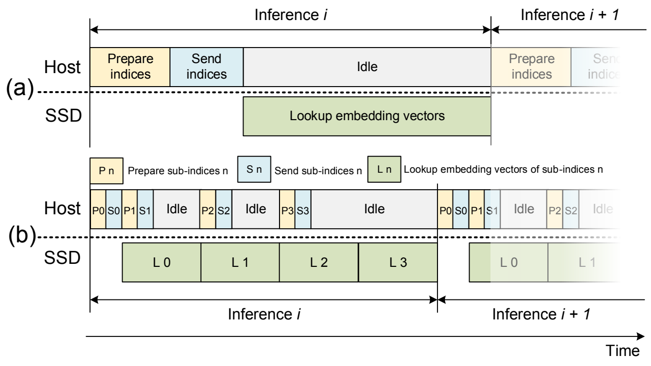

Figure 9: Data process pipeline of embedding lookup.

4 IMPLEMENTATION

We implement our prototype EV-SSD based on FPGA by enriching the FPGA chip to realize the functionality of EV-SSD controllers and employing four DDR4 banks to emulate four flash channels. The IP cores of DMA controller and DDR4 controller are provided by Xilinx. We modify the NVMe driver developed by Insider framework [19] to imitate the working mechanism of real NVMe protocol. The linear mapping function is applied in the *FTL* design, and each EV data are scattered around the four DDR4 chips. The main read latency of an SSD comes from the flash channel T_{flash}. We add a delay unit in the FPGA to emulate the read latency. FIO tool-based evaluation shows that the latency of emulated SSDs matches real SSDs accurately.

We implement EV-Cache in C++ and integrate it into the PyTorch-based DLRM framework. We intercept the function call for embedding lookups from `EmbeddingBag` operator to EV-SSD APIs provided by the EV-SSD user library.

The memory required for FlashEmbedding is mainly from EV-Cache. Our evaluations show that set EV-Cache size to 1% of its embedding table size is sufficient for most cases. Bitmap filter only takes two bits of memory space for each index; the total size is neglectable for modern memory systems.

5 EVALUATION

5.1 Experimental Setup

We evaluate FlashEmbedding on AWS EC2 F1 instance with a Xilinx Virtex 57 Plus UltraScale XCVU9P card. We use the DLRM [7] as our primary benchmark as it accurately represents production-scale models. We conduct experiments on the public dataset from Criteo [4]. We store the five largest embedding tables (around 4GB total) on SSD and the remaining 21 small tables in DRAM (around 70MB total).

Based on prior work [6, 7], we set the default batch size to 128 and sweep batch size from 64 to 512 for sensitivity studies. The default embedding dimension is 32. Embeddings are stored in fp32 format. For EV-Cache, we use a threshold of 2 for bitmap, 1% of embedding table size for cache size (total cache size is around 40MB) as our default configuration.

We evaluate three typical read latency of low- to high-end SSDs. The read latency of Micron B17A SLC flash memory for consumer SSDs is around 30 us. Optane SSD [11] from Intel reduces the latency to 10 us. The XL-FLASH [14] from Toshiba can achieve even lower 4 us.

We compare FlashEmbedding to a *Baseline SSD* implementation using SSD-S setting and an *Ideal DRAM* implementation using DRAM-only setting as in Section 2.2. We also evaluate the EV-Cache and EV-SSD techniques in detail.

5.2 Effectiveness of Proposed Techniques

Figure 10 shows the embedding lookup performance of the proposed techniques. We normalized the embedding lookup latency to that of FlashEmbedding. Compared to Baseline SSD, EV-SSD significantly reduces the normalized latency from 17.44× to 3.08×. The reason is two-fold. First, EV-SSD bypasses most parts of the I/O stack, allowing direct access to SSD. Second, EV-SSD supports fine-grained access, and read amplification is reduced remarkably (see below). EV-Cache further reduces the latency to 1.12×. This is because embedding cache can reduce the quantity of SSD requests by exploiting the temporal reuse opportunity of embeddings in the dataset. FlashEmbedding achieves the best performance by combining proposed EV-SSD and EV-Cache techniques and using pipeline to hide EV-SSD processing time partially.

Figure 11 shows read I/O traffic normalized to FlashEmbedding. FlashEmbedding and its variants incur much less read I/O traffic than Baseline SSD. FlashEmbedding incurs much less read I/O traffic than Baseline SSD. As the amount of memory is at an extremely low level, the I/O traffic incurred by Baseline SSD increases significantly. In contrast, the amount of read traffic in FlashEmbedding remains relatively low. EV-SSD's lighter I/O stack and finer access granularity efficiently reduce the read amplification. EV-Cache efficiently caches the embeddings and serves embedding lookup requests directly whenever possible, reducing traffic to SSD.

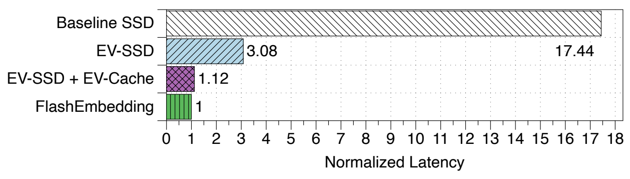

Figure 10: Embedding table lookup latency.

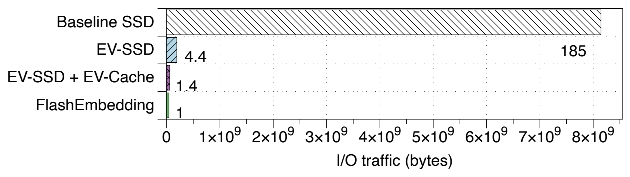

Figure 11: Comparsion of read I/O traffic.

FlashEmbedding further reduces I/O with pipeline because it can promptly update bitmap status once embedding vectors were accessed in the previous sub-indices, leading to better utilization of EV-Cache.

5.3 End-to-End Performance

We compare the performance of end-to-end recommendation inference between the Baseline SSD, Ideal DRAM, and FlashEmbedding. Figure 12 depicts the performance of end-to-end recommendation inference, including MLP, embedding lookup, and feature interaction operators.

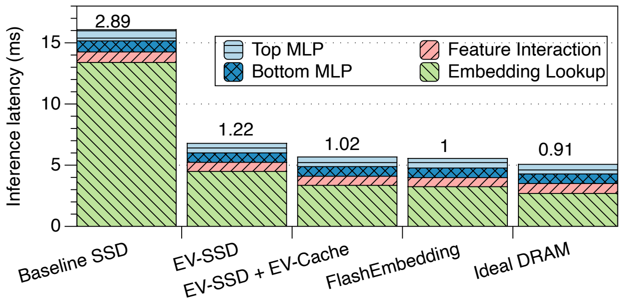

Figure 12: End-to-end performance.

FlashEmbedding shows comparable performance to Ideal DRAM (less than 10% latency overhead) but with much lower memory consumption (approximate 110MB memory versus 30GB memory), opening up new possibilities to store and lookup embedding tables on SSDs for large-scale recommendation under memory capacity limited systems. Compared to Baseline SSD, FlashEmbedding achieves up to 2.89× speedup. Moreover, the end-to-end latency to infer a single query is 6 ms, much lower than common latency requirements, tens of milliseconds. The result shows that FlashEmbedding is overall efficient thanks to our proposed techniques.

5.4 Sensitivity Study

5.4.1 Sensitivity to batch size. Figure 13 shows the normalized embedding lookup latency of Baseline SSD and EV-SSD as varying the batch size from 64 to 512. The latency is normalized to that of EV-SSD with a batch size of 512. As the batch size increases, the latency of EV-SSD decreases. Because the CPU utilization is scaled by increasing batch size, larger batch sizes lead to exploit the benefits of the wider SIMD units of CPU better. As the batch size increases, the fraction of time spent on the embedding table operations decreases with larger batch size, from 65.4% to 58.5% for batch size 64 and 512, respectively.

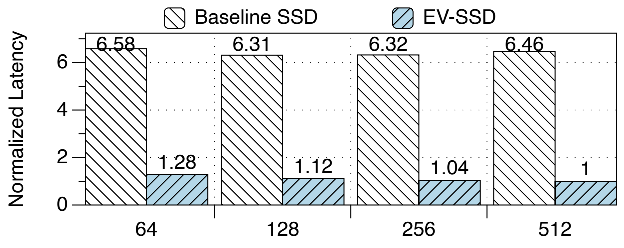

Figure 13: Sensitivity to batch size.

5.4.2 Sensitivity to cache size. Table 1 shows the impact of the size of the embedding vector caches. The performance of embedding lookup increases slightly as we increase the cache size from 0.1% to 5%. Because bitmap filters out less frequently accessed indices and only pass frequently accessed indices to the cache. The cache hit ratio is almost 100% for all cache sizes. Our workload characterization shows that a small subset of embedding entries has relatively higher reuse characteristics. 1% of embedding table cache size is sufficient to exploit the moderate locality for this dataset. Further increasing the cache size will increase the cache management overhead, leading to worse performance.

Table 1: Sensitivity to cache size.

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

.

5.4.3 Sensitivity to SSD Latency. Figure 14 shows how our proposed techniques scale with different SSD access latency. For EV-SSD, embedding lookup latency increases to 2.08× and 5.66× when SSD latency increases from 4 us to 10 us and 30 us, respectively. EV-Cache and FlashEmbedding mitigate performance degradation, indicating better scalability. This is mainly because EV-Cache reduces the number of SSD accesses by exploiting the data locality.

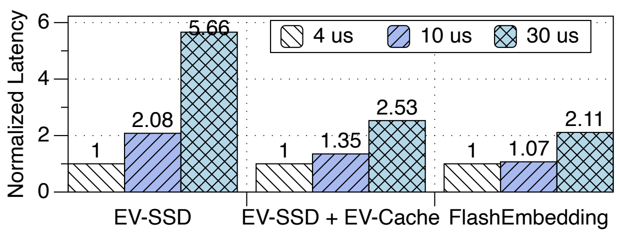

Figure 14: Sensitivity to SSD latency.

6 RELATED WORK

Recent researches [6, 7, 9, 10, 17, 18] perform detailed characterization of recommendation models in data center, demonstrating recommendation models with large embedding tables have unique storage and memory requirements. TensorDIMM [15] and RecNMP [12] utilize near-memory processing to accelerate recommendation models. MERCI [16] reduces the number of memory accesses and compute cycles at the cost of additional memory space. They all assume sufficient memory while we focus on the limited memory capacity in large-scale recommender systems.

Bandana [5] explores the use of NVMs to stored relatively less popular vectors in the NVM. The most similar related works to ours are EMB-SSD [13] and RecSSD [21], both offload the embedding lookup and sum operations into SSD for accelerating recommendation inference. However, low-level requests for reading embedding vectors from underlying SSDs are still in flash page granularity in these approaches. In contrast, FlashEmbedding explores accessing embedding tables in finer embedding vector granularity to reduce read amplification and improve performance.

General byte-addressable SSD designs [1, 2] only provide an MMIO interface to reduce read latency without flash-level finer-grained read optimization. FlashEmbedding avoids whole page data transfer overhead.

7 CONCLUSION

We propose FlashEmbedding, a novel system architecture for storing embedding tables on SSDs for large-scale recommendation inference under memory capacity-limited systems. First, we identify two performance problems of current architecture, i.e., the long I/O stack and the tremendous read amplifications of the block-interface SSD. Then, we propose three key optimizations, i.e., embedding vector SSD, embedding vector cache, and pipeline, to address both problems. The evaluation results show that FlashEmbedding outperforms the baseline.

REFERENCES

[1] Ahmed Abulila, Vikram Sharma Mailthody, Zaid Qureshi, Jian Huang, Nam Sung Kim, Jinjun Xiong, and Wen mei Hwu. 2019. FlatFlash: Exploiting the Byte-Accessibility of SSDs within a Unified Memory-Storage Hierarchy. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, Providence, RI, USA, 971–985.

[2] Duck-Ho Bae, Insoon Jo, Youra Adel Choi, Joo-Young Hwang, Sangyeun Cho, Dong-Gi Lee, and Jaeheon Jeong. 2018. 2B-SSD: the case for dual, byte-and block-addressable solid-state drives. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, Los Angeles, CA, USA, 425–438.

[3] Pedro Celis, Per-Ake Larson, and J. Ian Munro. 1985. Robin hood hashing. In *26th Annual Symposium on Foundations of Computer Science (SFCS)*. IEEE, Portland, OR, USA, 281–288.

[4] Criteo. 2014. *Kaggle Display Advertising Challenge Dataset*. Criteo. https://labs.criteo.com/2014/02/kaggle-display-advertising-challenge-dataset/

[5] Assaf Eisenman, Maxim Naumov, Darryl Gardner, Misha Smelyanskiy, Sergey Pupyrev, Kim M. Hazelwood, Asaf Cidon, and Sachin Katti. 2019. Bandana: Using Non-Volatile Memory for Storing Deep Learning Models. In *Proceedings of Machine Learning and Systems (MLSys)*, Vol. 1. Systems and Machine Learning Foundation, Indio, CA, USA, 40–52.

[6] Udit Gupta, Samuel Hsia, Vikram Saraph, Xiaodong Wang, Brandon Reagen, Gu-Yeon Wei, Hsien-Hsin S. Lee, David Brooks, and Carole-Jean Wu. 2020. DeepRecSys: A System for Optimizing End-To-End At-Scale Neural Recommendation Inference. In *ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, Valencia, Spain, 982–995.

[7] Udit Gupta, Carole-Jean Wu, Xiaodong Wang, Maxim Naumov, Brandon Reagen, David Brooks, Bradford Cottel, Kim Hazelwood, Mark Hempstead, Bill Jia, Hsien-Hsin S. Lee, Andrey Malevich, Dheevatsa Mudigere, Mikhail Smelyanskiy, Liang Xiong, and Xuan Zhang. 2020. The Architectural Implications of Facebook's DNN-Based Personalized Recommendation. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, San Diego, CA, USA, 488–501.

[8] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, James Law, Kevin Lee, Jason Lu, Pieter Noordhuis, Misha Smelyanskiy, Liang Xiong, and Xiaodong Wang. 2018. Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, Vienna, Austria, 620–629.

[9] Samuel Hsia, Udit Gupta, Mark Wilkening, Carole-Jean Wu, Gu-Yeon Wei, and David Brooks. 2020. Cross-Stack Workload Characterization of Deep Recommendation Systems. In *2020 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, Beijing, China, 157–168.

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

[11] Intel. 2018. *Breakthrough Performance for Demanding Storage Workloads*. Intel Corporation. https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/optane-ssd-905p-product-brief.pdf

[12] Liu Ke, Udit Gupta, Benjamin Youngjae Cho, 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, Xiaodong Wang, Brandon Reagen, Carole-Jean Wu, Mark Hempstead, and Xuan Zhang. 2020. RecNMP: Accelerating Personalized Recommendation with Near-Memory Processing. In *ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, Valencia, Spain, 790–803.

[13] Minsub Kim and Sungjin Lee. 2020. Reducing Tail Latency of DNN-Based Recommender Systems Using in-Storage Processing. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys)*. ACM, Tsukuba, Japan, 90–97.

[14] Toshiyuki Kouchi, Noriyasu Kumazaki, Masashi Yamaoka, Sanad Bushnaq, Takuyo Kodama, Yuki Ishizaki, Yoko Deguchi, Akio Sugahara, Akihiro Imamoto, Norichika Asaoka, et al. 2020. 13.5 A 128Gb 1b/Cell 96-Word-Line-Layer 3D Flash Memory to Improve Random Read Latency with t PROG= 75μs and t R= 4μs. In *2020 IEEE International Solid-State Circuits Conference (ISSCC)*. IEEE, IEEE, San Francisco, CA, USA, 226–228.

[15] 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 (MICRO)*. ACM, Columbus, OH, USA, 740–753.

[16] 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 (ASPLOS)*. ACM, New York, NY, USA, 302–313.

[17] Maxim Naumov, John Kim, Dheevatsa Mudigere, Srinivas Sridharan, Xiaodong Wang, Whitney Zhao, Serhat Yilmaz, Changkyu Kim, Hector Yuen, Mustafa Ozdal, Krishnakumar Nair, Isabel Gao, Bor-Yiing Su, Jiyan Yang, and Mikhail Smelyanskiy. 2020. Deep Learning Training in Facebook Data Centers: Design of Scale-up and Scale-out Systems. arXiv:1403.1349

[18] Jongsoo Park, Maxim Naumov, Protonu Basu, Summer Deng, Aravind Kalaiah, Daya Khudia, James Law, Parth Malani, Andrey Malevich, Satish Nadathur, Juan Pino, Martin Schatz, Alexander Sidorov, Viswanath Sivakumar, Andrew Tulloch, Xiaodong Wang, Yiming Wu, Hector Yuen, Utku Diril, Dmytro Dzhulgakov, Kim Hazelwood, Bill Jia, Yangqing Jia, Lin Qiao, Vijay Rao, Nadav Rotem, Sungjoo Yoo, and Mikhail Smelyanskiy. 2018. Deep Learning Inference in Facebook Data Centers: Characterization, Performance Optimizations and Hardware Implications. arXiv:1811.09886

[19] Zhenyuan Ruan, Tong He, and Jason Cong. 2019. INSIDER: Designing In-Storage Computing System for Emerging High-Performance Drive. In *USENIX Annual Technical Conference (USENIX ATC)*. USENIX Association, Renton, WA, USA, 379–394.

[20] Jizhe Wang, Pipei Huang, Huan Zhao, Zhibo Zhang, Binqiang Zhao, and Dik Lun Lee. 2018. Billion-scale Commodity Embedding for E-commerce Recommendation in Alibaba. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD)*. ACM, London, United Kingdom, 839–848.

[21] Mark Wilkening, Udit Gupta, Samuel Hsia, Caroline Trippel, Carole-Jean Wu, David Brooks, and Gu-Yeon Wei. 2021. RecSSD: near data processing for solid state drive based recommendation inference. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, Lausanne, Switzerland, 717–729.

[22] Carole-Jean Wu, Robin Burke, Ed Chi, Joseph A. Konstan, Julian J. McAuley, Yves Raimond, and Hao Zhang. 2020. Developing a Recommendation Benchmark for MLPerf Training and Inference. arXiv:2003.07336