Reducing Tail Latency of DNN-based Recommender Systems using In-storage Processing

Minsub Kim
DGIST
appleeji@dgist.ac.kr

Sungjin Lee
DGIST
sungjin.lee@dgist.ac.kr

ABSTRACT

Most recommender systems are designed to comply with service level agreement (SLA) because prompt response to users' requests is the most important factor that decides the quality of service. Existing recommender systems, however, seriously suffer from long tail latency when the embedding tables cannot be entirely loaded in the main memory. In this paper, we propose a new SSD architecture called EMB-SSD, which mitigates the tail latency problem of recommender systems by leveraging in-storage processing. By offloading the data-intensive parts of the recommendation algorithm into an SSD, EMB-SSD not only reduces the data traffic between the host and the SSD, but also lowers software overheads caused by deep I/O stacks. Results show that EMB-SSD exhibits 47% and 25% shorter 99th percentile latency and average latency, respectively, over existing systems.

CCS CONCEPTS

• **Computer systems organization** → **Architectures**; • **Computing methodologies** → *Machine learning*.

KEYWORDS

Recommender system, Machine learning, In-storage processing

ACM Reference Format:
Minsub Kim and Sungjin Lee. 2020. Reducing Tail Latency of DNN-based Recommender Systems using In-storage Processing. In *ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '20), August 24–25, 2020, Tsukuba, Japan.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3409963.3410501

APSys '20, August 24–25, 2020, Tsukuba, Japan

ACM ISBN 978-1-4503-8069-0/20/08…$15.00
https://doi.org/10.1145/3409963.3410501

1 INTRODUCTION

Deep-learning-based algorithms are becoming popular because they demonstrate high accuracy and efficiency in many problems which cannot be effectively solved by traditional algorithms. The recommender system (RS), which recommends contents that users may be interested in, is one of the popular applications based on deep-learning algorithms.

Unlike typical DNN models such as CNN and RNN which are compute-intensive [6] [13], recommender systems are data-intensive [2]. Particularly, an embedding layer, which is a core part of the recommender system, maintains a large number of huge embedding tables and requires irregular accesses to them. To comply with service level agreement (SLA) requirements and to provide prompt response to users' requests, existing recommender systems usually keep the entire embedding layer in the fast DRAM. In proportion to the rapid increase of web contents, however, the size of the embedding layer becomes so huge that it cannot be entirely loaded in the main memory. Some recommender systems manage the huge embedding layer by adopting on-demand caching strategies which keep only the popular part of the embedding layer in the main memory, while storing the rest in the slower but larger disks [1]. While generally effective, when the desired data is not found in the DRAM, it has to undergo costly disk accesses, which result in long tails of responses and degrade the quality of service.

In this paper, we propose a new SSD architecture, called *EMB-SSD*, which mitigates the long tail problem of recommender systems when the embedding layer cannot fit in the main memory. This study is motivated by the observations that rather than storage hardware, software stacks and network actually act as the main bottleneck in providing quick response to users. For example, modern SSDs offer short I/O latency close to 50 μs [12]. On the other hand, existing I/O software stacks (e.g., a file system and a block I/O layer) where the recommender systems runs are not fast enough and cause moderate overheads. Additionally, in data centers, recommender systems are often connected to network-attached storage (e.g., NAS and SAS) where a number of large embedding layers are stored in. When many service instances of recommender systems are simultaneously accessing the embedding layers, the network bandwidth gets

saturated quickly, which causes a serious delay in transferring the embedding layers to the remote host [4].

The proposed EMB-SSD is designed to address the aforementioned problems by leveraging the concept of in-storage computing [5] [8]. Instead of executing the entire recommender algorithm in an x86 host, EMB-SSD offloads data-intensive parts to an SSD controller. By running them inside an SSD, EMB-SSD eliminates almost all of the data transfers between the host and the storage, and thus, the network does not become a bottleneck anymore. EMB-SSD is also designed to expose *EMBLib* APIs so that the host can directly retrieve the results from EMB-SSD without going through the deep software stacks. This enables us to remove the overheads caused by frequent I/O syscall invocations. Moreover, by redesigning the flash translation layer (FTL), firmware in an SSD controller, we accelerate the embedding operations while efficiently utilizing hardware resources of an SSD.

Considering the limited computing resources, running parts of the recommender algorithm inside an SSD controller may seem infeasible. However, this is not the case because most of the data-intensive parts of the recommender algorithm require simple arithmetic operations (*e.g.,* addition) which do not require lots of CPU cycles and can be accelerated by using SIMD instructions offered by ARM CPUs.

We designed and implemented a proof-of-concept prototype of EMB-SSD in our custom SSD board. Our experiments using Criteo benchmark [11] showed that EMB-SSD exhibited 25% shorter average response time than the existing on-demand recommender systems. EMB-SSD also reduced the 99th percentile latency by 47%. Moreover, we confirmed that by removing almost all of the network traffic, EMB-SSD provided much better scalability than others.

The rest of the paper is organized as follows. In Section 2, we explain the basics of the recommender systems. Section 3 analyzes the access patterns of the embedding vectors and explains why tail latency problems occur. In Section 4, we explain EMB-SSD in detail. After showing our experimental results in Section 5, we review the related works in Section 6. Section 7 concludes with future work.

2 BACKGROUND

2.1 Recommender System

Figure 1 shows the overall structure of DNN-based recommender models. The recommender system receives the input data associated with the users and the contents and return the predicted click-through-rate (CTR). The features of the input data are either dense or sparse. Continuous inputs (*e.g.,* users' age) have dense features, whereas categorical inputs (*e.g.,* users' content preferences) have sparse features. Sparse features are encoded as multi-hot vectors. For example, if a user likes the contents whose unique IDs are 1 and 4, the multi-hot vector would be (0, 1, 0, 0, 1). Dense features are small and can easily be processed by FC layers. However, owing to the large number of categorical items, sparse features are usually too huge to be handled in FC layers. Moreover, the naive representation with binary numbers (*i.e.,* '0' or '1') makes it difficult to capture any of the contextual information between items or contents.

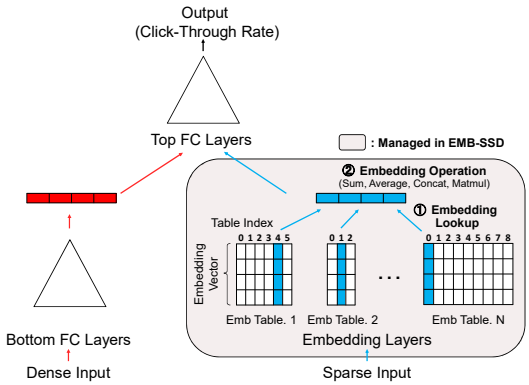


Figure 1: Simplified recommender models

To overcome such limitations, the recommender system employs an *embedding layer* which transforms sparse features into dense ones. The embedding layer consists of several embedding tables, each of which is also composed of multiple embedding vectors. As illustrated in Figure 1, the embedding tables are indexed by the unique IDs of the sparse features. Given the unique IDs, the embedding layer looks up the embedding tables, and the corresponding embedding vectors are returned. This process, which converts a sparse feature into dense vectors using embedding tables, is called an *embedding lookup*. The embedding vectors returned from embedding tables are aggregated into a single vector through simple arithmetic operations such as sum, average, concatenation, and matrix multiplication, before moving to the top FC layers. These operations required to generate a single vector are called an *embedding operations*.

Embedding lookup and embedding operations require large memory space to keep the embedding tables and involve irregular memory accesses to the tables. However, those operations only involve simple arithmetic, and thus their computation costs are relatively low.

2.2 Embedding Table Size

The size of the embedding table is determined by the dimension of the embedding vector and the number of columns. The dimension of the embedding vector affects the accuracy of the recommender system. It is thus preferred to have a higher dimension to improve accuracy. The number of columns in the table represents the total number of items or

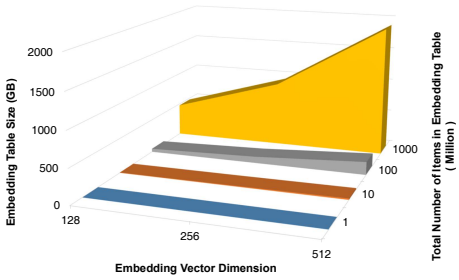


Figure 2: The size of the embedding table depending on the vector dimension and the number of items

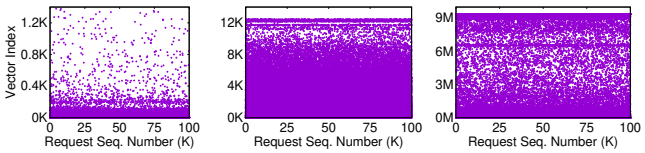
contents. Therefore, the more items are included in the recommender system, the larger the size of the embedding table is. As a result, considering the increasing number of contents in the web services and the high demand for more accurate content recommendations, both the row and column of the embedding table would keep growing.

Figure 2 shows the size of the embedding table depending on the dimension of the embedding vector and the number of items. We increased the vector dimension from 128 to 512 and the number of items from 1 million to 1 billion, respectively. The embedding table size is 490MB when the vector dimension and the number of items are 128 and 1 million, respectively. However, the table size increases up to 1.8TB when the numbers are increased to 512 and 1 billion, respectively. The embedding table of size 1.8TB is so huge to be loaded in the main memory. Even worse, the table size tends to increase as pointed out before. The existing recommender systems usually attempt to address the issue associated with the huge embedding tables by adopting the idea of on-demand caching which keeps the popular vectors in fast DRAM, while storing the unpopular ones in large disks.

3 ANALYSIS OF RECOMMENDER SYSTEM

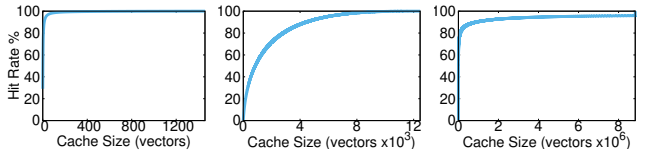
The on-demand caching approach would work pretty well if the locality of the embedding vectors is sufficiently high since most of the requests can be directly served by the DRAM. If not, however, frequent accesses to disks inevitably occur, which may cause long tail latency.

In this section, we analyze the tail latency of a recommender system when the entire embedding tables cannot be loaded in the DRAM. We use Criteo [11], which is a real-world dataset for predicting click-through rates of online advertising services. Criteo consists of 13 dense features and 26 sparse features. The sparse features are encoded using a 32-bit hash function to ensure the anonymity and privacy of the participants. On the Criteo dataset, we ran the Deep Learning Recommendation Model (DLRM) [10], which is a recommender system open-sourced by Facebook. DLRM



(a) Emb table 1 **(b) Emb table 7** **(c) Emb table 3**

Figure 3: I/O access patterns of embedding tables



(a) Emb table 1 **(b) Emb table 7** **(c) Emb table 3**

Figure 4: Hit rate curves of embedding tables

supports both the training and inference of content recommendation. During the training, DLRM creates 26 embedding tables by analyzing the 26 sparse features. The resulting embedding tables are stored in disks.

The DLRM inference predicts click-through rates by referring to the embedding tables. The inference step directly affects the user-perceived latency. Therefore, we analyzed the I/O access patterns during the inference of the DLRM and measured the cache hit ratios for different cache sizes (*i.e.,* the size of the main memory available for caching popular embedding vectors). The cache hit ratio is estimated by computing the stack distance of I/O references [9]. To serve a single inference request of the click-through rate, all the embedding tables (*i.e.,* 26 tables) need to be searched.

Workload Analysis: We analyzed the patterns of I/O accesses to the embedding tables and categorized the tables into three types based on the degree of locality and the working-set size: (1) high locality with a small working-set size (*Type-A*), (2) low locality with a small working-set size (*Type-B*), and (3) moderate locality with a large working-set size (*Type-C*). Figures 3 and 4 show our analyses of the three embedding tables, embedding tables 1, 7, and 3, each of which represents the unique I/O patterns of their respective type. Figure 3 plots the patterns of I/O accesses to the embedding tables, where the X-axis represents the sequence number of requests and the Y-axis represents the index of the accessed vectors in the table. Figure 4 illustrates the cache hit ratio curves of the tables for different cache sizes.

The embedding table 1 is categorized as Type-A. The dimension of the table 1 is 1,400, but as illustrated in Figure 3, almost all of the embedding vectors referenced range from 0 to 200. Thanks to such a small working-set size as well as high locality, the cache hit ratio reaches almost 100% when 14% of the total embedding vectors (about 200 vectors) are

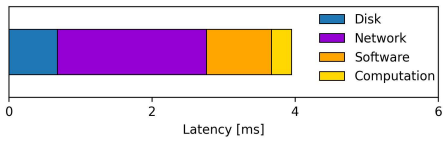


Figure 5: Latency breakdown

cached in the main memory. In our analysis, the size of each embedding vector is assumed to be 512B, so about 100KB of DRAM is enough to cache the popular vectors.

The embedding table 7 belongs to Type-B. I/O requests to the table 7 have low locality – the referenced vectors are uniformly distributed across a wide range of indices in the table, and this results in low cache hit ratio when the cache size is small. However, since table 7 only has 14,000 vectors, its working-set size is relatively small. Without employing a large amount of DRAM, we are able to achieve high cache hit ratio. As depicted in Figure 4, caching 10,000 vectors (= 4.8MB) in DRAM is sufficient to achieve a hit ratio of 100%.

Finally, Type-C is represented by embedding table 3, which exhibits moderate locality for a large number of vectors. As shown in Figure 3, the dots are highly populated around the top and bottom of the plot and the corresponding vectors are frequently referenced during the inference step. This enables us to achieve a relatively high hit ratio – 80% – even with a very small cache. However, the rest of the vectors are referenced randomly, exhibiting low locality. Even if the size of the cache increases to hold 8M vectors (= 3.8GB DRAM), the overall cache hit ratio does not improve.

According to our analysis, five out of 26 embedding tables are categorized as Type-C. However, owing to their huge size, they occupy almost all of the disk space – 98.3%. Moreover, these Type-C tables actually decide the tail latency of the recommender systems. As mentioned before, to predict the click-through rate, we have to access all the embedding tables. If one of the Type-C tables involve disk accesses due to cache misses, it would delay the entire recommendation process, causing long tail latency. Even worse, because of their huge size and relatively low locality, no matter how much DRAM the recommender system employs to cache the embedding vectors, the long tail latency caused by disk accesses cannot be eliminated.

Latency Analysis: We analyzed the latency of the recommender system when a cache miss occurs. Figure 5 shows our experimental result. The latency of the recommender system is decided by four main components, (1) disk access time, (2) data transfer time over the network, (3) software time, and (4) computation time for embedding operations. The computation time accounts for a trivial proportion of the total latency. This is an expected result because embedding operations require simple arithmetic operations. The disk I/O time is also relatively short – reading data from an SSD

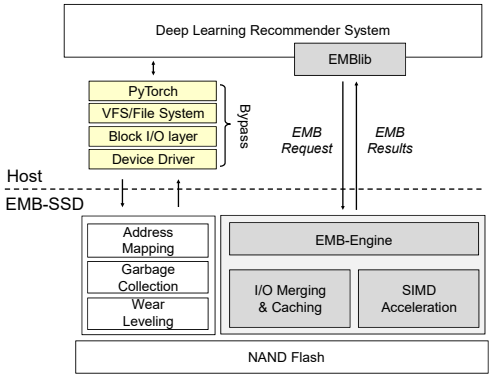


Figure 6: Overall architecture of the recommender system with EMB-SSD

takes about 0.67 *m*s, which is 17% of the total latency. This is owing to high performance of modern SSDs. Recent NAND chips provide 50 μs ~ 100 μs latency, along with the high throughput achieved by utilizing the parallelism of multiple channels and ways.

On the other hand, the software and network overheads are much higher than our expectation, accounting for 76% of the total latency. The existing recommender systems rely on several software layers, including deep-learning frameworks (e.g., PyTorch), a local file system, and a block I/O layer. As reported by [14], such a deep software stack causes non-trivial overheads especially when the underlying storage media is fast. Frequent cache misses also involve many I/O requests to disks, which increase the volume of data traffic between the host and the SSD.

The above observations lead us to consider running the embedding operations inside an SSD. Because embedding operations are relatively cheap, executing them in an SSD controller would not be a serious burden. By offloading part of the recommender system into the SSD controller, data-intensive operations that require the majority of the data accesses can be directly performed inside the SSD, and only small intermediate results need to be delivered to the x86 host. This not only eliminates the software stack overheads caused by the frequent I/O syscalls, but reduces a large amount of data transfer between the host and the SSD.

4 DESIGN OF EMB-SSD

4.1 Overall Architecture

Figure 6 shows the overall architecture of the recommender system with EMB-SSD. The x86 host and EMB-SSD are connected to each other through high-speed interconnects like PCIe (when EMB-SSD is used as a local storage) and Ethernet (when EMB-SSD is enclosed in a storage box like JBOF [3]). In our system, the embedding layer in the recommender system (see Figure 1) is offloaded into an SSD controller. This

is a reasonable design choice because the embedding layer retrieves lots of data from embedding tables, but performs simple arithmetic operations. The size of the data that are transferred to the host is just a few kilobytes (*i.e.,* 2KB).

EMB-SSD supports typical block I/O commands (*e.g.,* `READ`, `WRITE`, and `TRIM`) and in addition to them, it exposes a set of new commands (*e.g.,* `EMB_READ` and `EMB_WRITE`) so that the host can use embedding functions (see Section 4.2). These new commands are encapsulated by a user-space library, *EMBLib*, which can be directly invoked by existing recommender frameworks like DLRM. Except for the device drivers, no kernel I/O stacks are involved in accessing EMB-SSD. Please be advised that a new NVMe specification provides flexibility that such custom interfaces can be easily added.

The internal architecture of EMB-SSD is not much different from conventional SSDs, except that it should maintain data structures to manage embedding tables in NAND flash and perform embedding operations. This is the responsibility of *EMB-Engine*. To manage NAND flash better, EMB-SSD performs *I/O merging* and *caching* (see Section 4.4). EMB-SSD also accelerates embedding operations by making use of SIMD instructions provided by ARM CPUs (see Section 4.5).

4.2 User-space Library for EMB-SSD

EMBLib, which is a user-space library for EMB-SSD, exposes three new commands, `EMB_READ`, `EMB_WRITE`, and `EMB_REMOVE` to recommender frameworks. `EMB_WRITE` accepts two parameters, (1) an embedding table number and (2) a list of embedding vectors, which are required to create a new embedding table in NAND flash. `EMB_READ` is invoked when recommender frameworks want to perform embedding operations in storage. To reduce the number of command exchanges between the host and EMB-SSD, `EMB_READ` is designed to support vectored I/Os by default, which enable us to deliver a group of commands to the EMB-SSD at once. `EMB_READ` gets three parameters: (1) a list of table numbers, (2) a list of embedding vector indices per table, and (3) the type of operations (*e.g.,* addition) to perform on the tables. `EMB_READ` returns the computed embedding vectors for the requested tables. `EMB_REMOVE` is called to remove the embedding tables from EMB-SSD.

4.3 In-storage Embedding Engine

EMB-Engine is an in-storage embedding engine which is responsible for managing embedding tables and performing embedding operations. Thanks to the simplicity of the embedding layer, the overall architecture of EMB-Engine is so simple that it runs efficiently on the SSD controller.

When an `EMB_WRITE` arrives, it gets a list of free logical block addresses (LBAs) which are not used to store user data from the FTL. Then, it writes the embedding vectors to the LBAs sequentially. All the vectors are written in a sorted manner by their index numbers. Note that the mapping between the LBAs and the NAND pages are done by the FTL. Finally, it updates a redirection table that keeps track of the embedding tables and their vectors in flash. The redirection table is small enough; it is indexed by an embedding table number, and each entry contains a start LBA offset of the embedding table as well as its length.

Performing embedding operations on tables is straightforward. When EMB-SSD receives `EMB_READ` from the host, it figures out which LBAs are associated with the requested tables. A list of LBAs can be retrieved by looking up the redirection table. Locating the positions of the vectors in a table is easy as well, since all the vectors are stored sorted by their indices in flash. Finally, EMB-SSD performs the requested arithmetic operations on the vectors and returns the results to the host.

4.4 I/O Merging and Caching

The size of an embedding vector ranges from 256B to 2KB, and it is much smaller than that of a flash page which is usually 16KB. This size difference causes read amplification, which degrades the overall I/O throughput. EMB-SSD uses an I/O merging technique to solve this problem. In EMB-SSD, thousands of requests for embedding vectors come at once. For example, if there are 26 embedding tables and 64 user requests are processed at once, a total of 1,664 requests are delivered to EMB-SSD. Instead of handling vector requests one by one according to their arrival times, EMB-SSD merges the vector requests destined to the same NAND pages so that they are served by a single NAND operation.

In addition to I/O merging, EMB-SSD uses a controller DRAM to cache popular embedding vectors. As illustrated in Section 3, some embedding tables like Type-A embedding tables have high temporal locality but do not require a large amount of DRAM for caching. By selectively caching embedding vectors belonging to such tables, EMB-SSD is able to reduce the number of NAND page reads greatly.

4.5 SIMD Acceleration

The embedding layer requires relatively simple operations such as sum and average, but they could still be a burden on the ARM CPU, causing throughput drops, particularly when a large number of embedding vectors arrive simultaneously. EMB-SSD mitigates this problem by leveraging the parallelism of SIMD instructions. The embedding layer repeatedly performs homogeneous arithmetic operations over a group of associated vectors, and this makes it easier for EMB-engine to support SIMD instructions.

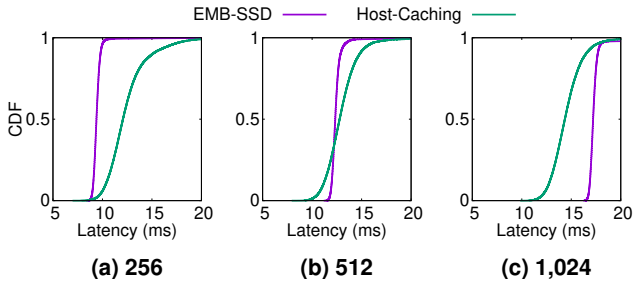


Figure 7: CDFs of I/O latency according to the dimension of the embedding vector (256, 512, and 1,024)

5 EXPERIMENTS

5.1 Experimental Setup

In order to evaluate the effectiveness of EMB-SSD, we implemented EMB-SSD in a Xilinx ZCU102 board with a custom flash card attached. Our flash card provides 1.2 GB/s read throughput and 500 MB/s write throughput, respectively. The ZCU102 board has a quad-core ARM Cortex-A53 CPU running at 1.2 GHz and 4GB of DRAM. The 10 GbE network is used to exchange data with the x86 host. Note that the network interface provides enough bandwidth to saturate all the NAND chips in our custom flash card.

We used Facebook's open source Deep Learning Recommendation Model (DLRM) to evaluate the performance of EMB-SSD. We used PyTorch as the deep learning library. The real-world dataset, Criteo, and the challenge dataset were used in our experiment. The default dimension of the embedding vector and batch size was 512 and 64, respectively.

We compared EMB-SSD with a recommender system employing on-demand caching, denoted as Host-Caching. We assumed that 25% of the total embedding tables are cached in the main memory. On the other hand, except for the memory required to manage the existing FTLs, EMB-SSD uses only 52MB of memory to cache the embedding vectors in the SSD controller side.

5.2 Experimental Results

Figure 7 shows the end-to-end latency of EMB-SSD and Host-Caching according to the dimension of the embedding vectors. Host-Caching showed irregular latency with long tails. Conversely, EMB-SSD exhibited consistent latency with shorter tails. In the case of Host-Caching, the latency was low when all the embedding vectors were cached in the memory. However, when cache misses occurr, it suffered from long tail latency. In the case of EMB-SSD, all the requests are sent directly to the SSD together, so network access exists only once. As a result, EMB-SSD always exhibit uniform latency for all requests.

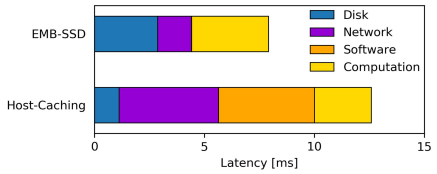


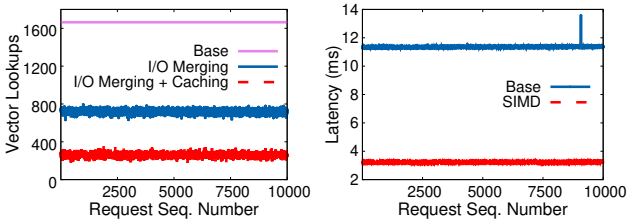
Figure 8: Latency comparison of each technique

When the dimension of the embedding vector was 256, EMB-SSD reduced the 99th percentile latency by 47% and the average latency by 25%, respectively, in comparison to Host-Caching. However, unlike Host-Caching which utilized high-speed x86 CPUs, EMB-SSD showed a significantly increased average latency when the dimension of the embedding vector was increased from 256 to 1,024. EMB-SSD used SIMD instructions to accelerate the embedding operations. However, in the case where a large number of vectors were issued simultaneously, the limited speed of ARM CPUs became a main bottleneck. As pointed out in Section 4.5, we expect this overhead to be removed by implementing embedding operations in FPGA accelerators.

Detailed Analysis of System Latency: We analyze the latency of Host-Caching and EMB-SSD in detail. Figure 8 shows the latency breakdown graph of the two systems. In Host-Caching, 70% of the total time was spent in the network and software layers. Conversely, the disk and computation time accounted for 30% of the total latency. As expected, by caching popular vectors in the host memory, Host-Caching was able to reduce a large number of I/Os sent to the SSD. Moreover, by leveraging the high-speed x86 CPU, embedding operations were processed quickly.

By offloading the embedding layer into the SSD, EMB-SSD eliminated almost all of the network traffic between the host and the SSD. Therefore, the time taken to transfer data over the network was reduced greatly. This also resulted in the reduction of software overheads associated with data access. Owing to the low speed of the ARM CPU, however, EMB-SSD suffered from higher computation overheads compared to Host-Caching. Moreover, EMB-SSD required longer disk access time than Host-Caching. Unlike Host-Caching which keeps many vectors in the large host DRAM, EMB-SSD used a small controller DRAM to cache highly localized vectors. Therefore, EMB-SSD had to read data more frequently from NAND flash. However, the increase in the computation and disk costs were fully offset by the reduced costs in the network and software layers. As a result, EMB-SSD achieved 38% shorter latency than Host-Caching.

Impact of I/O Merging and Caching: We analyzed how much I/O merging and caching techniques improved the performance. Figure 9(a) shows the number of NAND flash accesses with and without the optimization techniques applied. Here, Base employed no optimization techniques, I/O



(a) I/O merging and caching **(b) SIMD acceleration**

Figure 9: Impact of optimization techniques

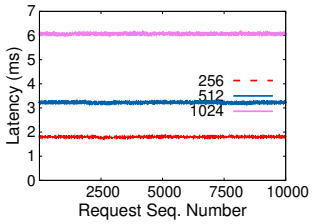


Figure 10: Embedding operation overhead

Merging used the I/O merging technique, and I/O Merging + Caching employed both the I/O merging technique and the caching techniques. In Base, 1,664 NAND flash accesses were required. On the other hand, I/O Merging reduced the number of disk accesses by 58% over Base by coalescing I/Os which were destined for the same NAND pages. I/O Merging + Caching reduced the number of NAND accesses by 85% compared to Base. As explained in Section 3, some embedding tables (*e.g.,* Type-A) have high temporal locality but require a small amount of DRAM owing to their small working-set sizes. By selectively caching their vectors in DRAM, EMB-SSD was able to get rid of many NAND accesses.

Impact of SIMD Acceleration: We analyzed the impact of accelerating embedding operations with SIMD instructions. Figure 9(b) shows the latency of EMB-SSD when SIMD instructions were used or not. We observed that the performance of EMB-SSD improved by 3.4 times with SIMD optimization. Considering that ARM Cortex-A53 supported the parallel computation of four vectorized float variables, the results in Figure 9(b) showed that EMB-SSD parallelized the embedding operations efficiently and took full advantage of available hardware support. The small performance drop was caused by copying the results to another vector.

Embedding Operation Overhead: Finally, we analyzed the latency of EMB-SSD depending on the dimension of the embedding vectors. We plot the latency of EMB-SSD in Figure 10 while varying the dimension from 256 to 1,024. The latency of EMB-SSD increased linearly as the vector dimension increased. This is an expected result – the more the vectors in the table are, the more the arithmetic operations to compute the outputs are involved in. As a result, the time spent on embedding operations increased from approximately 1.7 *ms* to 6 *ms* when the dimension of the embedding vector was increased from 256 to 1,024.

6 RELATED WORK

There have been several attempts to address the throughput and latency degradation problem of recommender systems. Bandana presented an idea of storing some of the embedding tables in the NVM devices (*e.g.,* Intel's Optane) which are larger but slower than DRAM [1]. In Bandana, NVM devices were used as an extension of small DRAM to keep the embedding tables. Bandana stored relatively less popular vectors in the NVM. In addition, by grouping spatially localized ones in the same I/O unit, it was able to fully utilize the available throughput of the NVM devices. EMB-SSD provides much higher scalability than Bandana, considering that NAND flash offers much larger capacity than NVM. Moreover, by performing embedding operations on the storage side, EMB-SSD not only improves I/O latency but utilizes a system bus shared by other system components better.

TensorDIMM [7] aimed at increasing DRAM capacity for recommendation systems by combining disaggregated DIMM devices attached to GPU-to-GPU interconnection fabrics (*e.g.,* NVIDIA's NVLINK). By leveraging high scalability of the GPU interconnection, TensorDIMM made it possible to connect up to 128 DIMMs to the GPUs. To avoid GPU interconnection bottleneck, TensorDIMM also put a near-memory processing unit onto a DIMM controller so that it performed embedding operations near memory and only returned the results to the GPUs. EMB-SSD is similar to TensorDIMM but extends its idea by offloading the embedding operations to the storage side. TensorDIMM might provide better responsiveness to users' requests thanks to the high speed of DRAM, but EMB-SSD would be more scalable in terms of capacity and more energy efficient owing to the non-volatile nature of NAND flash.

7 CONCLUSION

In this paper, we proposed a new SSD architecture, called EMB-SSD, which mitigated the tail latency of the recommender systems by using in-storage processing. By offloading the data-intensive algorithm into the SSD, EMB-SSD not only reduced the data movements between the host and the SSD, but also lowered the software overheads caused by deep I/O stacks. Our results showed that EMB-SSD exhibit 47% and 25% shorter 99th percentile latency and average latency, respectively, than the existing systems. As future work, we plan to use specialized hardware units (*e.g.,* FPGA) to accelerate the embedding operations in storage. This would resolve the long latency problem which occurs when the embedding dimension is too large.

ACKNOWLEDGMENTS

We would like to thank anonymous reviewers for all their helpful comments. This work was supported by Samsung Research Funding Incubation Center of Samsung Electronics under Project Number SRFC-IT1902-03. Sungjin Lee is a corresponding author.

REFERENCES

[1] A. Eisenman, M. Naumov, D. Gardner, M. Smelyanskiy, S. Pupyrev, K. Hazelwood, A. Cidon, and S. Katti. 2019. Bandana: Using Non-Volatile Memory for Storing Deep Learning Models. In *Proceedings of the Machine Learning and Systems*.

[2] U. Gupta, C.-J. Wu, X. Wang, M. Naumov, B. Reage, D. Brooks, B. Cottel, K. Hazelwood, M. Hempstead, B. Jia, H.-H. S. Lee, A. Malevich, D. Mudigere, M. Smelyanskiy, L. Xiong, and X. Zhang. 2020. The Architectural Implications of Facebook's DNN-Based Personalized Recommendation. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*.

[3] Y. Jin, S. Ahn, and S. Lee. 2018. Performance Analysis of NVMe SSD-Based All-flash Array Systems. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*.

[4] B. Kim, J. Kim, and S.H. Noh. 2017. Managing Array of SSDs When the Storage Device Is No Longer the Performance Bottleneck. In *Proceedings of the USENIX Workshop on Hot Topics in Storage and File Systems*.

[5] M. Kim, J. Kung, and S. Lee. 2020. Towards Scalable Analytics with Inference-Enabled Solid-State Drives. *IEEE Computer Architecture Letters*.

[6] A. Krizhevsky, I. Sutskever, and G. E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Proceedings of the International Conference on Neural Information Processing Systems*.

[7] Y. Kwon, Y. Lee, and M. Rhu. 2019. TensorDIMM: A Practical Near-Memory Processing Architecture for Embeddings and Tensor Operations in Deep Learning. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*.

[8] S. Liang, Y. Wang, Y. Lu, Z. Yang, H. Li, and Xiaowei Li. 2019. Cognitive SSD: A Deep Learning Engine for In-Storage Data Retrieval. In *Proceedings of the USENIX Annual Technical Conference*.

[9] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. 1970. Evaluation techniques for storage hierarchies. *IBM Systems Journal*.

[10] M. Naumov, D. Mudigere, H.-J. M. Shi, J. Huang, N. Sundaraman, J. Park, X. Wang, U. Gupta, C.-J. Wu, A. G. Azzolini, D. Dzhulgakov, A. Mallevich, I. Cherniavskii, Y. Lu, R. Krishnamoorthi, A. Yu, V. Kondratenko, S. Pereira, X. Chen, W. Chen, V. Rao, B. Jia, L. Xiong, and M. Smelyanskiy. 2019. Deep Learning Recommendation Model for Personalization and Recommendation Systems. *Computing Research Repository*.

[11] "Kaggle." [Online]. 2014. *Criteo Display Advertising Challenge Dataset*. https://www.kaggle.com/c/criteo-display-ad-challenge

[12] "Micron." [Online]. 2017. *Micron Nand Flash*. https://www.micron.com/products/nand-flash

[13] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G. Wei, and D. Brooks. 2016. Minerva: Enabling Low-Power, Highly-Accurate Deep Neural Network Accelerators. In *Proceedings of the ACM/IEEE International Symposium on Computer Architecture*.

[14] Y. Son, N. Y. Song, H. Han, H. Eom, and H. Y. Yeom. 2014. A User-Level File System for Fast Storage Devices. In *Proceedings of the International Conference on Cloud and Autonomic Computing*.