FAERY: An FPGA-accelerated Embedding-based Retrieval System

Chaoliang Zeng, *Hong Kong University of Science and Technology;* Layong Luo, Qingsong Ning, Yaodong Han, and Yuhang Jiang, *ByteDance;* Ding Tang, Zilong Wang, and Kai Chen, *Hong Kong University of Science and Technology;* Chuanxiong Guo, *ByteDance*

https://www.usenix.org/conference/osdi22/presentation/zeng

This paper is included in the Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation.

July 11–13, 2022 • Carlsbad, CA, USA

978-1-939133-28-1

Open access to the Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation is sponsored by

ARTIFACT EVALUATED
usenix ASSOCIATION
FUNCTIONAL

FAERY: An FPGA-accelerated Embedding-based Retrieval System

Chaoliang Zeng[1*] Layong Luo[2] Qingsong Ning[2] Yaodong Han[2] Yuhang Jiang[2] Ding Tang[1*]
Zilong Wang[1*] Kai Chen[1] Chuanxiong Guo[2]
[1]*Hong Kong University of Science and Technology* [2]*ByteDance*

Abstract

Embedding-based retrieval (EBR) is widely used in recommendation systems to retrieve thousands of relevant candidates from a large corpus with millions or more items. A good EBR system needs to achieve both high throughput and low latency, as high throughput usually means cost saving and low latency improves user experience. Unfortunately, the performance of existing CPU- and GPU-based EBR are far from optimal due to their inherent architectural limitations.

In this paper, we first study how an ideal yet practical EBR system works, and then design FAERY, an FPGA-accelerated EBR, which achieves the optimal performance of the practically ideal EBR system. FAERY is composed of three key components: It uses a high bandwidth HBM for memory bandwidth-intensive corpus scanning, a data parallelism approach for similarity calculation, and a pipeline-based approach for K-selection. To further reduce hardware resources, FAERY introduces a filter to early drop the non-Top-K items. Experiments show that the degraded FAERY with the same memory bandwidth of GPU still achieves 1.21×-12.27× lower latency and up to 4.29× higher throughput under a latency target of 10 ms than GPU-based EBR.

1 Introduction

Recommendation systems have gained significant adoption in many online services [11, 12, 18, 38]. To make a recommendation from a large corpus containing millions of candidate items, industrial large-scale recommendation systems are usually divided into two layers, namely retrieval and ranking, as shown in Figure 1. Retrieval quickly selects thousands of relevant items from the large corpus with simple algorithms, while ranking utilizes sophisticated algorithms to sort the retrieval results more precisely, and then chooses dozens out of the sorted items.

Real-world retrieval systems conduct multi-channel retrieval [26, 39, 43]: It leverages different strategies in separate channels to retrieve different candidates, which are then merged and filtered to generate the final retrieval result. Among the multi-channel retrieval strategies, embedding-based retrieval (EBR) gains increasing popularity [12, 18, 20, 25, 38, 42]. EBR represents user queries and candidate items with semantic embedding vectors (embedding for short) using representation learning [9], and converts the retrieval problem into a similarity search problem in the embedding space. In particular, an EBR algorithm, as shown in Listing 1, involves *scoring*, which scans the corpus to get all items and calculates a similarity score (e.g., via inner product) between every item embedding and the given query embedding, and *K-selection*, which returns the Top-K items based on their similarity scores. The returned Top-K items of EBR are usually sorted [25, 42], to simplify merging and filtering retrieval candidates from multiple channels.

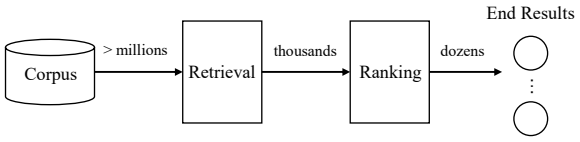


Figure 1: A typical recommendation system. Retrieval selects thousands of candidate items from a large corpus, and ranking further chooses dozens from the retrieval results.

The performance of such EBR systems is important. On the one hand, increasing the throughput of every EBR server reduces the overall server cost, as fewer servers are required to serve a target number of queries per second (QPS). On the other hand, decreasing the latency of each EBR server reduces the retrieval time, which can either shorten user's overall waiting time or leave more time for ranking computation to get better recommendation results [11]. Therefore, *latency-bounded throughput* becomes a critical metric for EBR systems.

To achieve high latency-bounded throughput, we characterize the EBR algorithm shown in Listing 1 and derive a practically ideal EBR hardware architecture (§2.2). Specifically, corpus scanning (line 3) is a memory-intensive operator which requires both large external memory capacity and high memory bandwidth. Similarity calculation (line 4) and K-selection (line 6) are both compute-intensive. They should match the memory bandwidth with a data-parallel architecture across multiple operator instances. Moreover, to overlap communications with computations among steps or operators, both inside K-selection and the entire EBR data flow require pipeline parallelism. Then, we extend the ideal architecture to support batch queries, by sharing corpus scanning among queries in a batch and providing separate compute pipelines

[*] This work is done while Chaoliang Zeng, Ding Tang, and Zilong Wang are interns in ByteDance.

```
# Scoring
for i in corpus_size:
    item_emb = corpus[i] # corpus scanning
    scores[i] = sim_calc(user_emb, item_emb) # similarity calc
# K-selection
ret_items = topk(scores) # returns the sorted top_k items
```

Listing 1: Simplified EBR algorithm for a single user query.

to serve different queries in the batch in parallel. As a result, the ideal architecture achieves the optimal query latency, and scales the latency-bounded throughput linearly with the batch size.

By comparing existing CPU- and GPU-based EBR with the ideal architecture, we realize that, unfortunately, none of the existing approaches achieve the optimal performance due to their inherent architectural limitations (§2.3). First, despite large memory capacity, CPU does not perform well in corpus scanning due to low memory bandwidth, and fails to well support the desired parallelism paradigms simultaneously due to the limited number of cores. Second, although GPU provides higher memory bandwidth and massive compute cores for data parallelism, GPU is not optimized for pipeline parallelism required by K-selection and the entire EBR data flow due to explicit resource boundaries.

We observe that FPGA, a programmable hardware device readily available in some hyper-scale cloud providers [10, 14, 41, 45], has all the desired properties of the practically ideal EBR architecture. Some modern FPGAs are equipped with large high bandwidth memory (HBM), ideal for corpus scanning. Moreover, FPGAs provide sufficient on-chip memories and fully programmable compute elements to enable appropriate parallelism paradigms for various operators (§2.4).

We exploit the above observations to design FAERY (§3), an **F**PGA-**A**ccelerated **E**mbedding-based **R**etrieval s**Y**stem, which is an embodiment of the ideal EBR architecture and achieves high performance. Specifically, FAERY stores the corpus in FPGA's HBM, which provides high bandwidth for the memory bandwidth-intensive corpus scanning. FAERY leverages a corpus manager to maximize the HBM bandwidth utilization in runtime while preserving memory-efficient storage and enabling online corpus update. FAERY follows the ideal architecture to design similarity calculation with data parallelism and K-selection with pipeline parallelism. Different from the ideal architecture, FAERY needs only a single K-selection pipeline, and adds a filter in front of it to significantly lower its throughput requirement, based on a unique property observed in the K-selection pipeline. The filter optimization lowers the resource requirements of FAERY compared with the ideal architecture by eliminating multiple K-selection pipelines.

The above ideas make a single FPGA-based EBR accelerator perform well. To further enhance its capabilities, multiple such accelerator cards can be inserted into a FAERY server (§4) and work together. When a corpus can fit into a single card, we can scale the aggregate query throughput by replicating the corpus among multiple cards. When the corpus is too large to fit into a single card, we can shard it evenly among multiple cards. FAERY supports both the *replication* and *sharding* modes and leverages a software front-end to dispatch queries and to merge retrieval results for multiple accelerator cards.

We have implemented a fully functional FAERY prototype with Xilinx FPGA cards (§5). Experiments (§6) show that the degraded FAERY with the same memory bandwidth of GPU achieves 1.21×-12.27× lower latency and up to 4.29× higher throughput under a latency target of 10 ms than an EBR system accelerated by Nvidia T4 GPU.

This paper makes the following contributions:

- We study the EBR algorithm from the first principles and derive a practically ideal EBR architecture to achieve the optimal query latency and to scale the latency-bounded throughput linearly with the batch size, constrained by hardware resources. We further identify the performance bottlenecks of CPU- and GPU-based EBR using the ideal EBR architecture as a reference (§2).
- We design FAERY, a domain specific accelerator (DSA) for EBR. FAERY arranges its key components: corpus scanning, similarity calculation, and K-selection in a perfect pipeline, and accelerates these components using appropriate data and/or pipeline parallelisms. FAERY is an embodiment of the ideal EBR architecture, with balanced filtering and buffering which matches the capability of parallel similarity score calculations with a single K-selection pipeline, based on a thorough analysis (§3 and §4).
- We implement FAERY using FPGA, evaluate its performance, and quantify its advantages over CPU- and GPU-based EBR systems, respectively (§5 and §6).

2 Background & Motivation

2.1 EBR Algorithms: KNN vs. ANN

EBR represents user queries and candidate items with embeddings, and converts the retrieval problem into a K-Nearest Neighbor (KNN) or an Approximate Nearest Neighbor (ANN) search problem in the vector space [20]. KNN-based EBR searches the **accurate** k-nearest item embeddings from the corpus, while ANN-based EBR retrieves the **approximate** k-nearest item embeddings, by sacrificing accuracy for efficiency using techniques such as indexing (e.g., IVF [34] and HNSW [28]) and quantization (e.g., PQ [21]). The tradeoff between accuracy and efficiency in various ANN algorithms is well studied in [8].

CPU provides limited memory bandwidth and computing power, so that it is challenging for CPU to perform KNN search on a large corpus due to the tremendous costs of memory accesses and computations. As a result, ANN search is

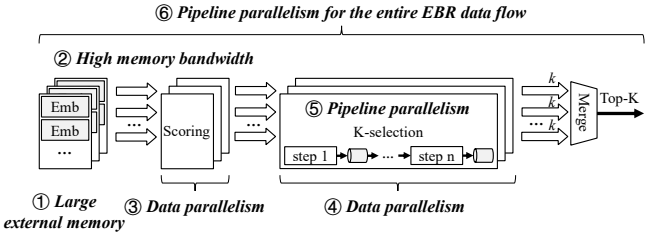


Figure 2: A practically ideal EBR architecture with the batch size of 1. It has the following properties: ① large external memory for corpus store, ② high memory bandwidth for corpus scanning, ③ data parallelism for similarity calculation, ④ data parallelism among multiple K-selection instances, ⑤ pipeline parallelism within a K-selection instance, and ⑥ pipeline parallelism for the entire EBR data flow.

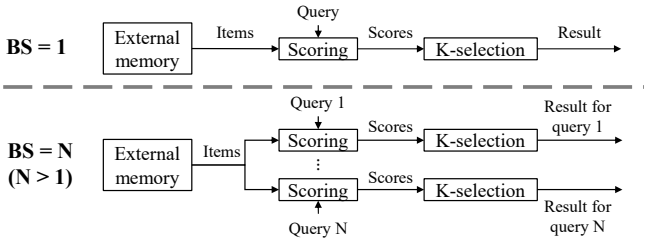


Figure 3: A practically ideal EBR architecture with the batch size of N, where the throughput scales linearly with the batch size N, while the latency remains the same as shown in Equation 1.

widely applied in CPU-based EBR in the industry. In contrast, accelerators, e.g., GPU and FPGA, provide much higher memory bandwidth and computing power, so that KNN search is usually adopted by these accelerators to trade memory bandwidth and computing power for higher accuracy and thus better recommendation quality.

To simplify discussion and comparison, we use the same KNN search (shown in Listing 1) for EBR on all platforms (CPU, GPU, and FPGA) in this paper, but our analysis results and acceleration ideas apply to ANN as well, as ANN shares similar characteristics and bottlenecks with KNN, just to different extents.

2.2 Practically Ideal EBR Architecture

To maximize latency-bounded throughput, an ideal architecture should first achieve minimal latency for each individual query (equivalent to maximal throughput with the batch size of 1), and then scale the throughput linearly with increasing batch sizes while preserving the consistent minimal latency.

In a theoretically ideal architecture, for each query, we do similarity calculation with ALL item embeddings in parallel and finish this operator in $O(1)$ time, followed by a perfect K-selection to match the parallelism. This is obviously impractical, as it requires millions of item accesses and millions of similarity calculation (e.g., inner product) operators in parallel, not to mention the design challenge of K-selection to match that extreme parallelism. A practically ideal EBR should take into account both realistic hardware constraints and the EBR characteristics which we discuss below.

Corpus scanning (line 3) is a memory-intensive operator. The size of an industrial corpus is up to several GBs [19], and scanning such a large corpus incurs millions of memory accesses for a single query. Thus, corpus store and scanning require large external memory and high memory bandwidth.

Similarity calculation (line 4) is a compute-intensive operator, which calculates similarity scores between the user query and all item embeddings. As the calculations for different item embeddings are independent, an ideal architecture should perform similarity calculation with data parallelism to match the throughput of corpus scanning.

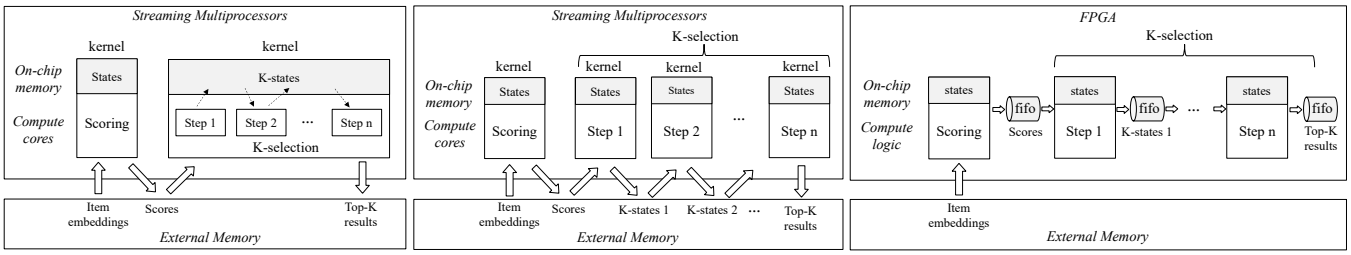
K-selection (line 6) is another compute-intensive operator. To match the throughput of multiple similarity calculation instances, K-selection requires data parallelism with multiple instances as well. Inside a single instance, K-selection can be realized by various algorithms [23, 33], among which a common practice is to partition this complex task into multiple steps, and organizes them in a pipelined manner.

Based on these characteristics, a practically ideal EBR architecture for optimal latency should have a large and high-bandwidth memory for corpus store and scanning, appropriate parallelisms for EBR operators to match their throughput to the memory bandwidth, and a perfect overlap among communications and computations of operators in the entire pipeline to minimize latency. Figure 2 describes a practically ideal EBR architecture with the batch size of 1 and its desired properties. The minimal query latency of this architecture is:

$$latency = \frac{S}{B} + C, \tag{1}$$

where S is the corpus size, B is the external memory bandwidth, and C is a constant delay, i.e., the pipeline latency, which is the time it takes for the last embedding going throughout the pipeline. Thus, the maximal throughput is $1/latency$ queries per second (QPS) with the batch size of 1.

The ideal architecture can be extended to support batch queries to increase latency-bounded throughput linearly, as shown in Figure 3. The key is to share corpus scanning among multiple queries in a batch (i.e., scan the corpus only once in each batch), and process multiple queries with separate compute pipelines in a data-parallel manner. In this way, the latency remains constant as shown in Equation 1, and the latency-bounded throughput scales linearly with the number of batched queries. In practice, the batch size cannot be increased unlimitedly due to resource constraints, and hence the maximum latency-bounded throughput will be bounded by the available hardware resource of the chosen platform.



(a) GPU-based EBR (GPU-o, K-selection with on-chip memory)

(b) GPU-based EBR (GPU-e, K-selection with external memory)

(c) FPGA-baesd EBR

Figure 4: Comparison between GPU- and FPGA-based EBR architectures. (a) GPU-o is a GPU-based EBR that stores intermediate states of K-selection in on-chip memory [23] and maintains corpus and scores in external memory. It suffers from heavy state maintenance cost and small k values; (b) GPU-e is a GPU-based EBR that moves all intermediate states of K-selection to external memory [33, 36], requiring multiple passes over the external memory; (c) is an FPGA-based EBR which stores only corpus in external memory, traverses external memory only once, and keeps the computation and communication of other operators fully on chip and in a streaming pipeline.

2.3 Existing EBR Architectures

Using the ideal EBR architecture as a reference, we analyze existing CPU- and GPU-based EBR architectures, and show that their performance are both sub-optimal due to inherent architectural limitations.

2.3.1 CPU-based EBR

Datacenter CPUs are equipped with large DDR memory (hundreds of GBs), able to store a very large corpus with millions or more item embeddings. However, CPU-based EBR does not perform well due to the following reasons.

Low memory bandwidth violates ② in Figure 2. The theoretical DDR memory bandwidth of a CPU is proportional to the limited number (typically 2∼8) of DDR channels [3], and the memory bandwidth utilization driven by a CPU is not high. Taking the server used in our evaluations (§6) as an example, a CPU with six DDR channels provides a theoretical maximum bandwidth of 140.8 GB/s, and an empirical upper bound of only 78 GB/s measured with Intel MLC [1]. The low memory bandwidth (B) significantly increases the first part (S/B) of Equation 1.

Limited number of CPU cores cannot support ③-⑥ and batch queries, simultaneously. A CPU contains dozens of processor cores that can be flexibly used for data parallelism, pipeline parallelism, and/or batch processing. However, due to the limited number of cores, CPU-based EBR fails to support all the above features well simultaneously, where the number of cores desired is the product of the number of memory channels, the number of pipeline stages, and the batch size as shown in Figure 2 and Figure 3. The poor support of data parallelism and pipeline parallelism results in throughput mismatch and imperfect overlapping among operators, leading to an increase on the second part (C) of Equation 1 as well as a sub-linear throughput increase with batch queries.

2.3.2 GPU-accelerated EBR

Compared with CPU, GPU provides high external memory bandwidth (e.g., Nvidia T4 [2] provides 300 GB/s bandwidth with GDDR6), and massive lightweight SIMT (Single Instruction Multiple Threads) cores optimized for data parallelism. Although GPU provides a smaller memory capacity (e.g., $16-80$ GB in a typical GPU and $128-640$ GB in a holistic server with 8 GPU cards), the size is still large enough to store the corpora in most recommendation services. For example, given a typical embedding size of 256 bytes, 128 GB memory can store more than 500M items that can meet the requirements of most recommendation systems [11,12,16,40]. These strengths inspire the design of GPU-accelerated EBR [23, 46] to achieve higher performance.

However, the performance of these GPU-based EBR systems are still sub-optimal, as GPU is not optimized for pipeline parallelism. GPU consists of a large number of *streaming multiprocessors* (SM), each of which contains exclusive on-chip memory and compute cores. Communication between SMs or kernels[1] is only possible via external memory, and the available on-chip memories for a single SM are very limited (e.g, 304 KB in Nvidia T4). These restrictions make GPU-based EBR not perfectly pipelined, leading to an increase on the second part (C) of Equation 1.

Inter-operator communication via external memory violates ⑥. Different EBR operators are organized as separate kernels. The similarity scores generated by scoring kernels are transmitted to the K-selection kernels via the external memory, as shown in Figure 4a and Figure 4b. The explicit kernel boundaries make it difficult to exploit pipeline parallelism

[1] A kernel is a function executed on GPU, which realizes a data-parallel portion of an application. An operator may consist of one or multiple kernels.

across EBR operators to overlap perfectly communication and computation [24, 47].

Existing K-selection pipelines violate ⑤. Existing GPU-based K-selection algorithms can be classified into the following two categories.

K-selection with on-chip memory (e.g., `WarpSelect` [23]), denoted as GPU-o, as shown in Figure 4a, fuses all K-selection sub-steps into a single kernel to avoid cross-kernel overhead, and keeps all K-states in on-chip memory. However, maintaining all K-states on chip and executing these steps in the SIMT cores introduce non-trivial computation overhead (e.g., per-thread queue sorting, sorted queues merging, and thread synchronization [23]), resulting in poor latency. To show this overhead, we measure Faiss [23], which adopts `WarpSelect`, with a 4M corpus and the same setting as §6. The result shows that the K-selection operator consumes up to 80.4% of the total time. Moreover, given the limited on-chip memory size of each SM, it fails to support a large k value, i.e., at most 2048 in this setting.

K-selection with external memory (e.g., `RadixSelect` [33, 36]), denoted as GPU-e, as shown in Figure 4b, implements different K-selection sub-steps as separate kernels, and transmits intermediate data among kernels via the external memory. As a result, the K-selection operator has to access the external memory multiple passes (well studied in [33]), leading to sub-optimal K-selection performance, which will become worse with a larger batch size due to heavy bandwidth contention on external memory. With the same setting mentioned above, the query latency of `RadixSelect` is increased by 12.36× when the batch size is increased from 1 to 16.

2.4 FPGA Opportunities

We observe that FPGA has the following properties that meet the requirements of the ideal EBR architecture.

- Similar to GPU, high-end FPGAs are equipped with HBM of large capacity (typically 8 to 32 GB). A typical HBM is a stack of 32 parallel DRAM channels (versus up to 8 DDR channels in a CPU), providing parallel memory accesses and thus high bandwidth (460 GB/s), which fundamentally eliminates the biggest memory bandwidth bottleneck in CPU-based EBR.
- Unlike GPU with exclusive and small on-chip memories for each SM, FPGA provides sufficient on-chip memories (dozens of MB in total), which are accessible to all compute elements. This could be leveraged to overcome the problems of GPU-based EBR as discussed in §2.3.2. Unlike GPU with SIMT cores optimized only for data parallelism, the massive compute elements and interconnects among them in FPGA are fully programmable, so that they can be orchestrated in any parallelism strategy (data parallelism or pipeline parallelism).

Desired features in ideal arch.	CPU	GPU	FPGA
large memory capacity	✔	✔	✔
high memory bandwidth		✔	✔
data parallelism	△	✔	✔
pipeline parallelism	△		✔
batch queries with low latency	△	△	✔

Table 1: EBR architecture comparison among CPU, GPU, and FPGA. ✔ means perfect support, while △ means limited support.

Table 1 summarizes the architecture comparison of CPU, GPU, and FPGA for EBR. Based on FPGA's advantages, we can design an FPGA-based EBR pipeline similar to that in Figure 4c: It traverses the HBM only once, passes intermediate data between operators via on-chip memory, and overlaps communications with computations of operators via careful pipeline designs. In this way, FPGA-based EBR has the potential to approach the optimal performance (Equation 1). The design details of such a system, named FAERY, are presented in the following sections.

3 FAERY Accelerator

We design the FAERY accelerator by following the most desired properties of the ideal EBR architecture, with some additional optimizations. Figure 5 presents the architecture of the FAERY accelerator, with a few major components including HBM, corpus manager, similarity calculation, filter, and K-selection. FAERY stores the corpus in HBM and uses the corpus manager (§3.1) for corpus scanning and update. FAERY applies data parallelism across multiple similarity calculation units (§3.2), and pipeline parallelism within K-selection (§3.3). Different from the ideal EBR architecture, FAERY does not need multiple K-selection pipelines with data parallelism, thanks to a new filter operator (§3.4) inserted before the K-selection pipeline to lower its throughput requirement. This optimization lowers the resource overhead compared with the ideal architecture. The above operators are perfectly pipelined and overlapped, and the resulting data streams are shown in §3.5.

3.1 Corpus Manager

FAERY stores the corpus in HBM and uses the corpus manager to perform corpus scanning and update. The corpus manager is designed to meet two objectives toward high bandwidth utilization of HBM: maximizing *single-channel performance* and maximizing *multi-channel parallelism*.

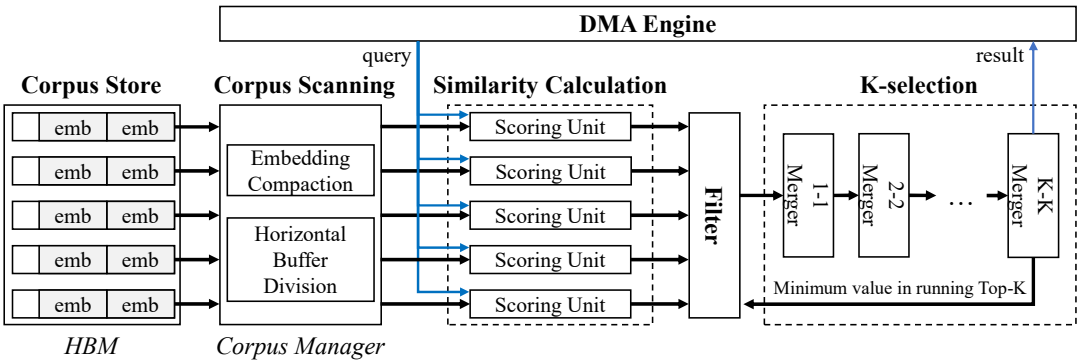


Figure 5: FAERY accelerator architecture with the batch size of 1. It stores item embeddings in high bandwidth memory (HBM), uses a corpus manager for corpus scanning and update, and applies appropriate parallelism paradigms for different key operators: data parallelism for similarity calculation and pipeline parallelism for K-selection. A filter is added between the above two operators to bridge their throughput mismatch and lower the resource requirement. The overall architecture is fully pipelined, with computations and communications perfectly overlapped to minimize latency and maximize throughput.

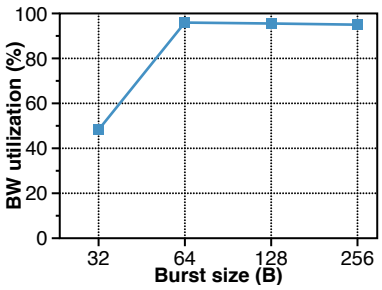


Figure 6: Bandwidth utilization of a single HBM channel with different burst sizes. The utilization is over 90% when the burst size is not smaller than 64 bytes.

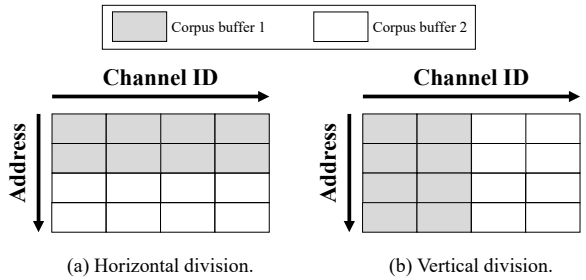


Figure 7: HBM can be divided into two buffers in two ways. (a) Horizontal division: divide buffers based on address, which preserves all memory channels and maximum memory bandwidth for each buffer. (b) Vertical division: divide buffers based on channel ID, which halves the number of available channels and the memory bandwidth for each buffer.

3.1.1 Embedding Compaction to Maximize Single-channel Performance

The bandwidth utilization of a single HBM channel is affected by two factors: *access pattern* (sequential or random access) and *burst size* (the number of bytes in a memory transaction). Given the nature of brute-force KNN search, both corpus scanning and corpus update perform sequential access, which is more efficient than random access. We show in Figure 6 the bandwidth utilization of a single HBM channel in sequential access over various burst sizes. The result reveals that, to achieve bandwidth utilization of over 90%, an ideal burst size should be not smaller than 64 bytes and be a multiple of the channel width of 32 bytes.

However, the size of embeddings could be smaller than 64 bytes, especially for those generated by quantization-aware training [30, 31, 37]. It could also be not a multiple of the channel width. To bridge the mismatch between the ideal burst size requirement and the realistic embedding size, we compact one or multiple embeddings into a burst, whose size might not be exactly a multiple of the embedding size, leaving some unused bytes in a burst. To minimize the waste, we choose an ideal burst size with minimal unused bytes.

3.1.2 Horizontal HBM Division to Maximize Multi-channel Parallelism

To support the online corpus update, the corpus manager partitions HBM into two corpus buffers: a runtime buffer to store the latest corpus and serve queries, and an update buffer reserved for update. Upon receiving a new corpus from the host, the corpus manager stores it into the update buffer, and then switches the EBR pipeline to scan corpus from that buffer for new queries. In this way, the runtime buffer and update buffer switch roles after each update.

HBM can be partitioned into two corpus buffers in two ways, horizontally or vertically, as shown in Figure 7. The horizontal division is chosen, as it keeps all the available HBM channels and thus the maximum memory bandwidth

for each buffer, while the vertical division loses half channels and thus half memory bandwidth for each buffer.

During corpus update, the HBM write caused by update and HBM read caused by query, may contend for HBM memory bandwidth with horizontal division. Such contention is negligible. Considering that the realistic HBM bandwidth is 414 GB/s (90% utilization of a typical 460 GB/s HBM), and corpus update is bounded by the PCIe Gen3 x16 bandwidth (16 GB/s), the update (HBM write) throughput over the total HBM throughput is less than 4%. Moreover, given that corpus update happens much less frequently than query, update can be further throttled to minimize its impact to query. Other update methods will be discussed in §7.

3.2 Similarity Calculation

Similarity calculation receives multiple item embeddings from multiple HBM channels simultaneously. In order to match the bandwidth of HBM, we apply data parallelism in similarity calculation, where multiple scoring units (SU) are instantiated to work in parallel, and each performs similarity calculation, e.g., inner product, between a separate item embedding and the given query embedding. The number of parallel SUs required is the product of the total number of HBM channels and the maximum number of item embeddings inside a channel width, which may contain more than one item embedding due to the embedding compaction (§3.1.1).

3.3 K-selection

There exist multiple different K-selection architectures [27,29, 44], suitable for different scenarios. In the context of recommendation systems, the value of k is from a few thousand to dozens of thousands in realistic EBR [16,25], so K-selection in FAERY aims to achieve both high performance and high scalability in supporting a large value of k. To this end, we choose an existing K-selection pipeline [29] based on bottom-up merge sort for the following two reasons.

First, the bottom-up merge sort allows processing input scores in a streaming manner to avoid storing the entire scores before computing. In contrast, some algorithms incapable of streaming processing, e.g., `RadixSelect` [36], inevitably need external memory to store the entire scores of a large size. Leveraging external memory to cache the scores should be avoided, as it will not only reduce the available storage space for the corpus, but also interfere with the performance of corpus scanning due to bandwidth contention.

Second, pipeline parallelism within K-selection is compute-efficient and scalable, e.g., the chosen K-selection pipeline [29] requires only $O(logk)$ comparators. In contrast, some data-parallel K-selection architectures [27, 44] use a large number of parallel comparators to process a batch of input scores at a time. The number of parallel comparators required by this method is $O(p*k)$, where p is the batch size

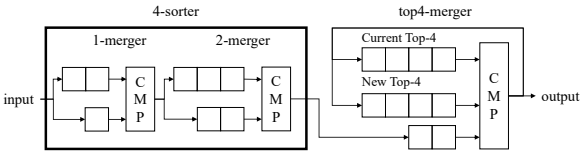


Figure 8: An example of the 4-selection pipeline in [29].

of input scores. Such design is not scalable, especially for k in the order of thousands.

Pipeline parallelism within K-selection. Figure 8 illustrates a K-selection pipeline where $k = 4$. The K-selection pipeline in [29] contains a series of *i-mergers* and a final *topk-merger*. An *i-merger* merges two sorted lists with length i into a sorted list with length $2i$, followed by a *2i-merger* in the pipeline. The pipeline starts at a *1-merger*, and log_2k sequential *i-mergers* form a *k-sorter*. At the end of the pipeline, a *topk-merger* merges the output of the *k-sorter* with the current sorted Top-K to generate a new running Top-K. All modules process data in a streaming manner, and the latency of such a pipeline is $k + log_2k$ clock cycles [29].

The above K-selection pipeline processes one score every clock cycle, which is slower than the throughput of scores generated by similarity calculation with data parallelism. According to the ideal architecture shown in Figure 2, K-selection can simply match the throughput with multiple K-selection pipelines, i.e., instantiating multiple K-selection pipelines in parallel, each processing different scores, followed by a merger at the end to get the final Top-K from multi-channel sorted Top-K. However, a single K-selection pipeline is much more resource-hungry than a single scoring unit. Instantiating multiple K-selection pipelines to match the throughput of the multi-channel similarity calculation is not resource-efficient, especially when supporting a large k and a large batch size. Based on an important observation on the K-selection pipeline, we address the throughput mismatch problem in a resource-efficient way by introducing a new operator: filter (§3.4).

3.4 Filter

The K-selection pipeline maintains inside a **running Top-K** (e.g., the *current Top-4* in Figure 8), which continuously updates the Top-K for all the past scores until the current point. We observe that, if the input score to K-selection is not greater than the minimum score of the running Top-K, the input won't change the internal running Top-K and thus can be dropped. Based on this observation, we design a filter to early drop non-Top-K scores, which significantly reduces the number of scores sent to K-selection. Figure 9 shows the throughput model of FAERY, where corpus scanning and similarity calculation are designed with data parallelism to match the HBM throughput, K-selection only provides a single pipeline to save resources, and the filter bridges the throughput mis-

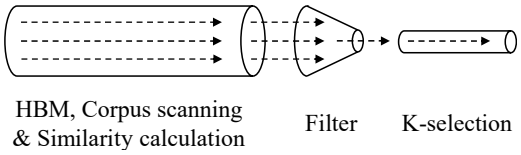


Figure 9: Throughput model of FAERY. Corpus scanning and similarity calculation are designed to fully match the HBM bandwidth, followed by a filter to early drop most of scores generated by similarity calculation, thus significantly lowering the throughput requirement of K-selection.

match between the multi-channel similarity calculation and the single-channel K-selection.

Let x ($x > 1$) denote the number of scores generated by similarity calculation per clock cycle. The throughput of similarity calculation is x scores per clock cycle, and the throughput of K-selection is one score per clock cycle, so that the throughput mismatch is $(x-1)/x$. We define filtering efficiency as the number of scores (m) dropped by the filter over the total number of scores (n), i.e., m/n. As long as $m/n \geq (x-1)/x$, the design will work well without performance degradation.

In practice, the recall ratio of EBR (the ratio of the retrieved items to the total items, i.e., $k:n$) is usually very low, e.g., $1:1000$. The majority of scores will be early dropped by the filter, and the filtering efficiency will be high enough to bridge the throughput gap. We analyze the average filtering efficiency as follows.

Filtering efficiency. Given that $n >> k$ in practice, we can derive the filtering efficiency using a simplified model. Assuming the input scores follow a random distribution, and the running Top-K values are already generated from all the previous scores when the dropping decision for a score is made, the probability of the i^{th} ($i > k$) score dropped by the filter follows

$$p(i) = \frac{i-k}{i}. \tag{2}$$

The expected number of scores dropped by the filter follows

$$m = \sum_{i=k+1}^{n} p(i) = \sum_{i=k+1}^{n} \frac{i-k}{i}. \tag{3}$$

As a result, the average filtering efficiency is

$$\begin{aligned} e &= \frac{m}{n} = \frac{\sum_{i=k+1}^{n} \frac{i-k}{i}}{n} \\ &= 1 - \frac{k}{n} - \frac{k}{n} \sum_{i=k+1}^{n} \frac{1}{i} \\ &> 1 - \frac{k}{n} - \frac{k}{n} ln(n). \end{aligned} \tag{4}$$

Given a typical setting in practice where $k = 1024$, $n = 10^6$, the filtering efficiency is larger than 98%. In our implementation (§5), x is 4, and the throughput mismatch is $3/4 = 75\%$. This shows that the filtering efficiency is much higher than the throughput mismatch in practice, and thus the filter enables K-selection to match the throughput of similarity calculation with just one pipeline, reducing resource consumption.

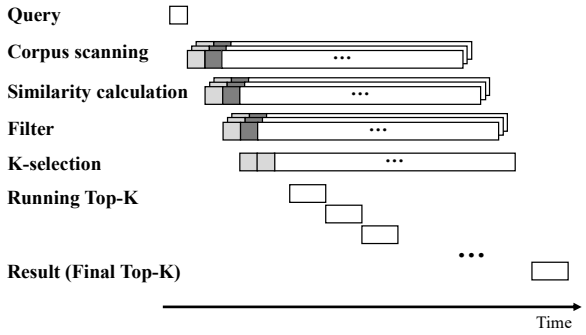


Figure 10: The perfect overlap of data streams in FAERY.

Buffer to absorb bursts. Although the filter balances good performance and low resource cost in practice, it can fail to drop any scores in the worst case when all the input scores are sorted ascendingly. Since the item embeddings are stored randomly in HBM, the probability that such worst case happens is very low. However, we do have a buffer in the filter to absorb two types of temporal bursts. The initial burst is built up while the filter is processing the first y scores of every new query, when the drop probability ($p(i)$, $i < y$) of score i is lower than the throughput mismatch $(x-1)/x$ between similarity calculation and K-selection. Based on Equation 2, y is $(k*x)$. The other type of burst is occasional score sequences in which all scores are larger than the minimum of the running Top-K. The size of this burst is variable but should be small given the increasing drop probability shown in Equation 2.

3.5 Perfect Overlap of FAERY Data Streams

As described in the above sections, all operators work in a streaming manner, i.e., all operators start processing as soon as the data begin to stream in, and the communications between operators are perfectly overlapped with computations. As a result, the data streams in this architecture exhibit a perfect overlap, as shown in Figure 10. Upon receiving a query, the corpus manager starts corpus scanning and gets a multi-stream of embeddings from 32 HBM channels, followed by similarity calculation and filter streams in the subsequent cycles. The filter operator early drops most of scores, so that a single stream of scores is sent to K-selection. As scores begin to stream into K-selection, the running Top-K is updated, and it is output as the final Top-K result soon after the last score is injected into the K-selection pipeline.

Batch is supported in FAERY in the same way as the ideal architecture (Figure 3). The data streams of multiple queries start at the same point. Therefore, the latency remains the same, and the throughput scales linearly with the batch size.

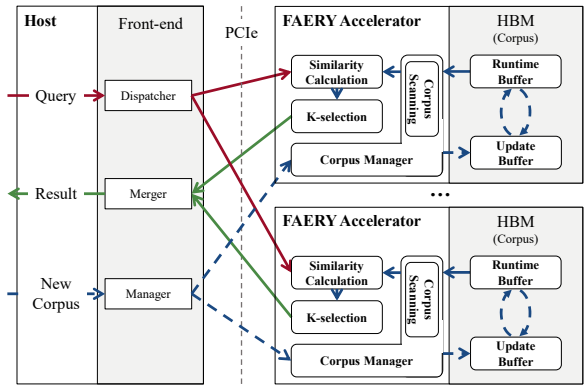


Figure 11: FAERY server architecture. A FAERY server hosts multiple FAERY accelerators to increase either the query throughput or the supported corpus size. A software front-end on the host CPU dispatches user queries to multiple accelerators, merges retrieval results from them, and updates the corpus on the fly.

4 FAERY Server

Figure 11 presents the FAERY server architecture, which includes a software front-end on the host CPU and multiple FAERY accelerators inserted in server PCIe slots.

Multiple FAERY accelerators can work together to enhance the capabilities of a single accelerator in two modes: *replication* and *sharding*. In the *replication* mode, these accelerators store separate replicas of the same corpus and serve different queries simultaneously to increase the query throughput. In the *sharding* mode, multiple accelerators store different shards of the same corpus and serve the same query simultaneously to increase the supported corpus size. The front-end running in the host CPU is responsible for query dispatching via a dispatcher module and result merging via a merger module in the above two modes.

When there is a new corpus received by the server, a manager module in the software front-end handles this update request. It determines whether corpus replicating or sharding is needed based on the working mode listed above, and then sends the corpus replicas (or shards) to the corresponding accelerators via PCIe. The corpus manager in each FAERY accelerator stores the update corpus in the update buffer and switch buffer roles as described in §3.1.2.

5 Implementation

We build a fully functional prototype of FAERY using FPGAs. The FPGA accelerator is built with Xilinx VU35P FPGA [4], which contains an HBM of 8 GB capacity, 32 memory channels, and 460 GB/s bandwidth. We implement the FAERY pipeline described in Figure 5 using the hardware programming language SystemVerilog. In the following part, we discuss several implementation details using this FPGA with a typical setting: One embedding contains 128 elements of 2 bytes each (i.e., the embedding size is 256 bytes), k is 1024, and the prototype runs at a clock frequency of 400 MHz, which matches the HBM bandwidth. An ASIC implementation of FAERY with the same HBM bandwidth but higher clock frequency (e.g., 1 GHz), could not provide significant performance improvement, as the end-to-end performance is mainly determined by the HBM bandwidth.

Corpus manager. Since the embedding size is 256 bytes, the burst size can be set to 256 bytes based on the embedding compaction strategy, resulting in no waste on both storage space and read bandwidth. Based on the measurement, the achievable HBM bandwidth is 414 GB/s, with 90% utilization of the theoretical upper bound of 460 GB/s. Given that the HBM has 32 memory channels of 32-byte width, the corpus scanning reads 1024 ($32*32=1024$) bytes from HBM every clock cycle, almost catching up with the HBM bandwidth at 400 MHz ($1024*400/1000=409.6$ GB/s), and outputs 4 ($1024/256=4$) embeddings per clock cycle on average. To support online corpus update, horizontal division keeps half of the 8 GB HBM space (i.e., 4 GB) for the runtime corpus, which supports up to 16M item embeddings in a single FPGA.

Similarity calculation. To match the throughput of 32 parallel HBM channels, similarity calculation is implemented with 32-channel SUs in parallel. Each SU performs inner product calculation, which consists of three stages. The first stage performs element-wise multiplications between the item and the query. Given that an HBM channel width (32 bytes) contains 16 elements (each 2 bytes) of an embedding, it requires 16 parallel multipliers in this stage to sustain the HBM channel bandwidth. In the second stage, it conducts a summation of the results in the first stage with an accumulation tree, which has $log_2 16=4$ layers. The summation result is finally added to the computing score in the last stage. Therefore, the latency of similarity calculation is 6 ($1+4+1=6$) cycles, and the throughput of similarity calculation with 32 parallel SUs (i.e., 4 scores per clock cycle) matches exactly the throughput of corpus scanning (i.e., 4 item embeddings per clock cycle).

K-selection. K-selection is implemented based on an existing pipeline [29], whose latency is $k+log_2 k$ clock cycles. For $k=1024$, the latency is 1034 cycles. This fully pipelined K-selection can process one score per clock cycle. Different from the ideal architecture, a single K-selection pipeline is required in FAERY, with the filter to bridge the throughput mismatch between similarity calculation and K-selection.

Filter. Since similarity calculation generates four scores per cycle, while K-selection only processes one score per cycle, the filter must drop at least 3/4 of the scores on average to bridge their speed gap. Based on the analysis in §3.4, the filtering efficiency in this setting is higher than 98% and thus greater than 3/4. To absorb bursts, the filter buffer is set to store at most 8192 ($2*k*x$, where $x=4$ and $k=1024$) scores,

	Per-query resources	Common resources
LUT	7.31%	11.05%
FF	6.98%	14.78%
BRAM	13.05%	10.66%
DSP	8.6%	0.07%

Table 2: Breakdown of FAERY resource consumption (batch size = 1). Per-query resources increase linearly with the batch size, while common resources remain unchanged.

slightly larger than the initial burst size ($k * x$) derived in §3.4 to reduce approximation error in the analysis. This buffer is implemented with only 8 Block RAMs (BRAMs), consuming less than 0.2% of the total FPGA memory resources. With both the high filtering efficiency and the sufficient buffer, the filter works well to bridge the throughput mismatch and to absorb temporal bursts, and we observe no performance loss with the above setting. Compared with the ideal architecture shown in Figure 2, FAERY with the filter and a single K-selection pipeline, can save 32% on-chip memories and 27% compute resources by eliminating the other three K-selection pipelines and a four-port merger [35] per query compute pipeline.

Batch support. FAERY supports batch queries as described in Figure 3. Despite the performance advantages, the resource requirements of batch queries increase with the batch size. As a result, the maximum batch size supported in our prototype is determined by the available resources in the Xilinx VU35P FPGA. The resources are consumed by two types of components: per-query compute pipelines (similarity calculation, K-selection, and filter) exclusive for each query, and common modules (corpus manager and PCIe DMA) shared among batch queries. Table 2 breaks down the resource consumption of a FAERY accelerator with the batch size of 1 into per-query resources and common resources. Based on this result, the upper bound of the batch size is 6 in the Xilinx VU35P FPGA. However, this FPGA chip is composed of multiple dies, so that timing closure is challenging when the resource utilization is high or cross-die routing is congested. We end up with an implementation with a batch size of 3, to balance good batch performance and easy timing closure.

6 Evaluation

We evaluate the performance of the FAERY implementation, and compare it with CPU- and GPU-based EBR, respectively. Our results reveal that:

- FAERY approaches the optimal query latency, and achieves 98.09×-118.99× and 1.85×-18.81× lower latency than CPU- and GPU-based EBR, respectively. The degraded FAERY with the same memory bandwidth of GPU still achieves 1.21×-12.27× lower latency than GPU-based EBR.
- In terms of latency-bounded ($\leq$ 10 ms) throughput, FAERY and the degraded FAERY outperform GPU-based EBR by 1.33×-6.58× and 0.87×-4.29×, respectively, while CPU-based EBR fails to meet the 10 ms latency target.
- FAERY achieves 1.66×-8.20× higher energy efficiency and 1.31×-6.46× higher cost efficiency than GPU-based EBR.
- A FAERY server with two accelerators provides 2× higher query throughput in the replication mode, and 2× higher corpus capacity in the sharding mode with less than 1.1% increase in latency.

6.1 Experiment Setup

Baseline. We compare FAERY with Faiss [23], an open-source similarity search library that supports both CPU and GPU. The K-selection implementation in Faiss GPU is `WarpSelect`, a heap-based algorithm using on-chip memory, as shown in Figure 4a, denoted as GPU-o. We further replace the Faiss K-selection implementation with an algorithm using external memory, as shown in Figure 4b, denoted as GPU-e. We choose `RadixSelect` implemented in [33], which reports the best performance when k is greater than 512, compared to other algorithms. Both GPU-o and GPU-e use `fp16` for embeddings and `fp32` for scores.

Platforms. FAERY is evaluated on a server with two 8-core Intel Xeon Silver 4110 CPUs. CPU-based EBR is evaluated on a server with two 16-core Xeon Gold 5218 CPUs and 192 GB memory. We choose Nvidia Tesla T4 GPU [2] in GPU-based EBR, as the T4 GPU shares a similar cost to the Xilinx VU35P FPGA (cost comparison will be discussed in §6.2.4). The CUDA version is 11.2 and the Tensor Core acceleration is enabled. T4 GPU is equipped with 16 GB GDDR6 of 300 GB/s bandwidth. To bridge the difference of memory bandwidth between FPGA (460 GB/s) and GPU (300 GB/s), we also evaluate a degraded FAERY, denoted as FAERY-d, by throttling its HBM bandwidth to 300 GB/s.

Corpus. We use the synthetic corpus, with randomly generated 128-dimensional item embeddings of 2 bytes each dimension, and retrieve $k = 1024$ items for each query. We use synthetic random corpora to verify the generality of FAERY, which by design, is not sensitive to any specific workload.

In the following, we first evaluate the performance of a single accelerator (§6.2). Many important applications contain a moderate corpus. For example, the YouTube video corpus contains tens of millions of items [40], and the Google play application corpus contains one million items [11]. The corpus of these applications could fit into the HBM of a single card based on the current FAERY implementation (§5). Then, we show the performance of a FAERY server with two accelerators (§6.3) to demonstrate FAERY's capability in supporting either higher query throughput or a larger corpus by adding cards.

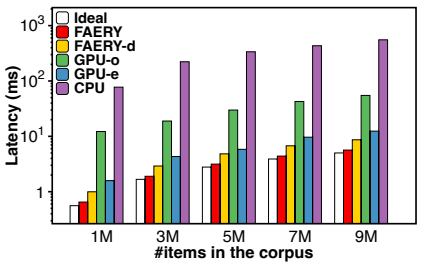


Figure 12: Query latency comparison among different EBR architectures (batch size = 1, latency is in log scale).

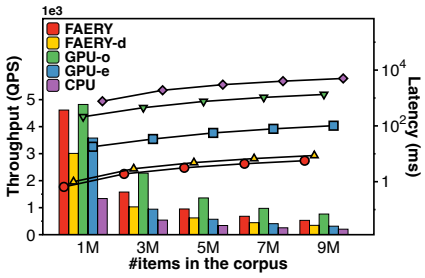


Figure 13: Query throughput comparison among different EBR architectures (Corresponding latency is also shown).

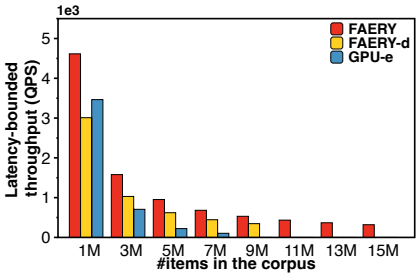


Figure 14: Comparison of latency-bounded throughput, where CPU and GPU-o fail to meet the latency target ($\leq$ 10 ms), and thus are not shown.

6.2 Single-accelerator Performance

6.2.1 Latency

We compare the query latency among different EBR architectures in Figure 12. Average latency is used as the metric, as latency distribution in each of these architectures doesn't show significant variance due to the deterministic execution flow of KNN. The query latency of the ideal architecture is calculated based on Equation 1, where S is $N * 256$ bytes, N is the number of items in the corpus, B is the maximum HBM bandwidth 460 GB/s, and C is the FAERY pipeline latency 2.6 us. The query latency of FAERY approaches the optimal latency of the ideal architecture, with only 1.13×-1.16× increases, which results from non-full ($\sim$ 90%) memory bandwidth utilization as measured in Figure 6. Both FAERY and FAERY-d consistently outperform CPU and GPU in query latency with different corpus sizes. Compared with CPU, FAERY significantly reduces the average latency (98.09×-118.99× lower) due to its high memory bandwidth and appropriate parallelism paradigms for different operators. Compared with GPU, FAERY achieves 9.48×-18.81× and 1.85×-2.44× lower latency than GPU-o and GPU-e, respectively. Even if we degrade the FAERY memory bandwidth to that of GPU T4 (300 GB/s), FAERY-d also achieves 6.18×-12.27× and 1.21×-1.59× lower latency than GPU-o and GPU-e, respectively. This verifies that even with the same memory bandwidth, FAERY-d still outperforms GPU-based EBR, because the poor pipeline support of GPU leads to a significant increase of the second part (C) in Equation 1, as detailed in §2.3.2.

6.2.2 Throughput

We compare the maximum throughput and its corresponding latency among different EBR architectures in Figure 13. Batch queries are used in all architectures to achieve the maximum throughput. Both FAERY and FAERY-d are evaluated with the batch size of 3, the same as that in the implementation. Although the throughput of CPU- and GPU-based EBR systems can be improved by increasing the batch size, we only show the results with the batch size up to 1024, because further increasing the batch size leads to marginal improvement. GPU-o consistently outperforms FAERY in throughput by 1.04×-1.44×, and FAERY-d by 1.60×-2.21×, with a large batch size but a much higher query latency (ranging from 212 ms to 1339 ms with different corpus sizes). In contrast, both FAERY and FAERY-d keep low query latency as that of batch size 1 when increasing the batch size. The throughput of GPU-e does not increase significantly with larger batch sizes, due to heavy contention on external memory bandwidth in K-selection among multiple queries. As a result, GPU-e achieves only 59%-78% (91%-119%) of the FAERY (FAERY-d) throughput, but has a much higher latency (ranging from 18 ms to 102 ms). FAERY outperforms CPU in throughput by 2.60×-3.45× even when the CPU-based EBR runs with a large batch size. Moreover, CPU suffers form the worst latency.

6.2.3 Latency-bounded Throughput

Latency-bounded throughput is a critical metric for EBR, as retrieval is a typical real-time service with strict requirements on the response time. For example, the response time is within 10 ms in the Taobao production retrieval [15, 25], and the query serving time of the entire recommendation pipeline (retrieval + ranking) is on the order of 10 ms in the Google application recommendation [11]. In this paper, we set the upper bound of the retrieval latency to 10 ms, and compare the latency-bounded throughput among different EBR architectures.

Since CPU and GPU-o fail to meet the latency target in any condition, we only compare FAERY and GPU-e in Figure 14. The latency target prevents GPU-e from using a large batch size, which increases per-query latency significantly due to the contention on memory bandwidth. In contrast, FAERY follows the ideal architecture for batch queries, maintaining constantly low latency when increasing the batch size, as discussed in §2.2. When the number of items in the corpus ranges from 1M to 7M, FAERY achieves 1.33×-6.58×

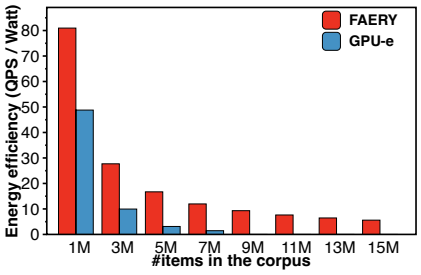


Figure 15: Comparison of energy efficiency among different EBR accelerators.

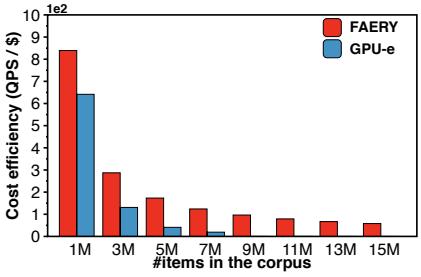


Figure 16: Comparison of cost efficiency among different EBR accelerators.

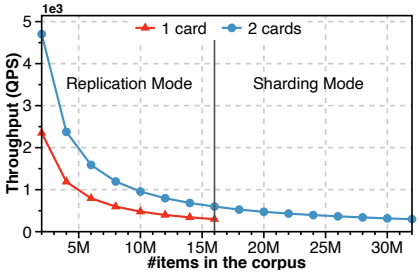


Figure 17: Throughput of a FAERY server with two cards.

higher latency-bounded throughput than GPU-e. However, FAERY-d achieves only 87% of the GPU-e latency-bounded throughput with the small corpus size of 1M, as GPU-e can leverage a large batch size (64 in GPU-e vs. 3 in FAERY-d) to boost the throughput with moderate memory bandwidth contention when the corpus size is small. As the corpus size increases from 3M to 7M items, FAERY-d exhibits its advantages in latency-bounded throughput and achieves 1.46×-4.29× higher latency-bounded throughput than GPU-e. When the number of items is larger than 7M, GPU-e fails to meet the latency target in any batch size, while FAERY-d can increase the corpus size until 9M items under latency target, and FAERY supports up to 15M items.

6.2.4 Energy & Cost Efficiency

The GPU and FPGA used in the evaluation have different architectural advantages and disadvantages, e.g., the GPU has lower memory bandwidth (300 GB/s vs. 460 GB/s), but much higher computing power (130 TOPS vs. 18.6 TOPS for INT8) than the FPGA. In addition to using the degraded FAERY with 300 GB/s memory bandwidth in a direct comparison between FPGA and GPU in terms of latency and throughput, we consider both energy efficiency (performance per watt) and cost efficiency (performance per dollar), as yet another fair metrics to compare the efficiency between totally different hardware architectures. We use the latency-bounded throughput measured in Figure 14 as the performance reference.

Energy efficiency. Based on the measurement, FAERY is 57 Watt and GPU-e is 71 Watt during serving. The above power consumption does not vary significantly with different corpus sizes. Given these power consumption and throughput data, Figure 15 shows the result of energy efficiency (QPS/Watt), where FAERY consistently outperforms GPU-e with 1.66×-8.20× higher energy efficiency.

Cost efficiency. As the concrete cost numbers are confidential, we normalize the costs of GPU, FPGA, and server used in the evaluation to 1, 1.1, and 4.4, respectively. With these cost units, the normalized costs of the FAERY and GPU servers are 5.5 (=1.1+4.4) and 5.4 (=1+4.4), respectively. Based on these normalized costs and the latency-bounded throughput data, Figure 16 shows the result of cost efficiency (i.e., QPS/(cost unit)), where FAERY provides 1.31×-6.46× higher cost efficiency than GPU-e.

6.2.5 Summary

Table 3 summarizes the EBR performance comparison among different processors, and reveals that each processor has its unique advantages for EBR. FAERY, an FPGA-based EBR, achieves the lowest latency, the highest latency-bounded throughput, and the highest energy and cost efficiency compared with CPU and GPU. Compared with CPU, FPGA's performance gain results from the high memory bandwidth provided by HBM and massive programmable compute elements to enable appropriate parallelism paradigms and batch processing. FPGA outperforms GPU due to the fully pipelined design with perfectly overlapping communications with computations of operators, and a programmable architecture that supports efficient K-selection. All these advantages make FAERY not only approach the optimal latency, but also achieve linear-scaling throughput when increasing the batch size. CPU-based EBR supports the largest corpus size, thanks to the large capacity of CPU DDR memory. GPU-based EBR achieves the highest raw throughput without latency bound with a very large batch size, thanks to its massive compute cores.

6.3 Multi-accelerator Performance

We evaluate a FAERY server with two accelerators. Figure 17 shows the aggregate query throughput with different corpus sizes. When the corpus can fit into a single card (i.e., the number of items is not larger than 16M), we replicate the corpus in the two cards to double the query throughput, as shown in the left part of Figure 17. When the corpus size is larger than the memory capacity of a single card, we evenly shard the corpus between the two cards, and thus the supported corpus size is extended up to 32M items, i.e., 2× the HBM capacity of a single card, as shown in the right part of Figure 17. In the

	Corpus size in bytes	Normalized latency	Normalized throughput	Normalized latency-bounded throughput (< 10 ms)	Normalized energy efficiency	Normalized cost efficiency
CPU	> 100 GB	98.09-118.99	0.290-0.385	-	-	-
GPU	16-80 GB	1.85-18.81	0.593-1.440	0.152-0.752	0.122-0.602	0.155-0.763
FPGA (FAERY-d)	8-32 GB	1.53	0.652	0.652	-	-
FPGA (FAERY)	8-32 GB	1	1	1	1	1

Table 3: Summary of performance comparison among different EBR processors.

sharding mode, the software front-end in CPU has to merge the two Top-K results from the two cards and yield the final Top-K, introducing an extra latency of less than 15 us, i.e., 1.1% of the total query latency.

7 Discussion

System lessons. While we focus on FPGA-accelerated EBR in this paper, we believe FPGA is a promising choice for not only EBR acceleration in specific, but also domain specific accelerator (DSA) in general. First, FPGAs are readily available for DSA in several hyper-scale cloud providers [10,14,41,45]. Second, FPGAs are inherently capable of faithfully implementing DSA systems such as FAERY, MicroRec [22], and Tiara [41]. These systems are memory and compute bounded, so they can benefit from customized parallelism and pipelining with optimized memory accesses provided by FPGAs.

Most FPGA-based architectures can be baked into custom ASICs for higher performance and efficiency. In FAERY, the query latency and throughput are mainly limited by the memory bandwidth, so an ASIC implementation with the same memory bandwidth would not significantly improve the performance. However, an ASIC version of FAERY can achieve higher energy efficiency. Nonetheless, it will require a significant volume to amortize the high non-recurring engineering (NRE) cost for higher cost efficiency.

Online update. The online update approach described in §3.1 minimizes the degradation of the total query throughput (QPS) during the update, by taking half of the HBM memory in each card as update buffer. We further note that there are other ways for online update from a distributed system perspective. In a typical production EBR system, there are multiple corpus replicas distributed across multiple FAERY servers for reliability and load balancing purposes. The online update in this case can be performed by taking off one replica at a time for updating while keeping the others online. This approach may achieve higher memory utilization, but experience higher update time and lower QPS than our update approach during the update process.

Support new models. In addition to the online corpus update, FAERY is able to change the pipeline structure on the fly to adapt to new models. Given the relatively stable EBR pipeline structure, including corpus scanning, similarity calculation, and K-selection, we are able to use the same hardware code to support different EBR pipeline variants with just different parameters (e.g., embedding size, data type, k). When a new model requires a change of the pipeline structure, we can simply change parameters in the code, generate a hardware image, and then load the image into FPGA on the fly.

Accelerate ANN-based EBR. Although FAERY is designed to accelerate KNN-based EBR, it can be extended to accelerate ANN to achieve higher throughput by sacrificing retrieval accuracy. Indexing-based ANN algorithms, e.g., IVF [34] and HNSW [28], leverage an index layer before corpus scanning to reduce the number of accessed items per query. Quantization-based ANN algorithms, e.g., PQ [21] and OPQ [17], leverage a codebook to compact the corpus. FAERY can support both ANN variants by maintaining the index layer or the cookbook in FPGA on-chip memory. Most of the other operators are the same, and their designs can be shared among KNN- and ANN-based FAERY.

Use FAERY for other services. Although FAERY is a DSA for retrieval in recommendation systems, we believe a similar idea can be applied to vector search in general, which is a fundamental part of many applications [13, 20, 25] that use semantic embedding vectors to represent contents (articles, images, audios, videos, etc.) and perform searches. These applications share a similar data flow to that described in this paper, but their characteristics vary. Interesting future work is to extend FAERY to accelerate a generic vector search service (such as Microsoft Vector search [5] and Google Vertex AI Matching Engine [6]).

8 Related Work

CPU- and GPU-based EBR systems have been discussed in §2. Existing FPGA-based similarity searches [27,44] were not designed for EBR, and thus not suitable. They leveraged massive parallel comparators to perform K-selection, whose resource consumption is unbearable for k being a few thousand in EBR. Moreover, they did not optimize the efficiency of corpus scanning, as they either did not leverage the high bandwidth of HBM [44] or failed to achieve high bandwidth utilization [27]. There are other kinds of work that accelerated specific ANN

algorithms, e.g., HPQ [7] for quantization-based ANN and QuickNN [32] for indexing-based ANN. They are orthogonal to FAERY that focuses on optimizing the entire EBR pipeline as a whole, including corpus scanning, similarity calculation, and K-selection.

9 Conclusion

FAERY is a domain specific accelerator (DSA) for embedding-based retrieval (EBR). The components of FAERY: corpus scanning, similarity calculation, and K-selection are arranged using the appropriate parallel techniques as required by an ideal EBR architecture. As a result, FAERY does not have the shortcomings and performance penalties of existing CPU- and GPU-based EBR approaches. FAERY not only provides both low latency and high throughput compared with CPU-based EBR, but also outperforms GPU-based EBR in terms of latency-bounded throughput.

Acknowledgments

We thank our anonymous reviewers and shepherd Christopher Rossbach for their insightful comments. We also thank Hong Zhang and Lixin Zheng for all technical discussions and valuable comments. This work is supported in part by the Key-Area Research and Development Program of Guangdong Province (2021B0101400001), an HKUST-ByteDance Research Project, and the Hong Kong RGC TRS T41-603/20-R, GRF 16213621 and GRF 16215119.

References

[1] Intel memory latency checker (mlc). https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html.

[2] T4 tensor core datasheet. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-t4/t4-tensor-core-datasheet-951643.pdf.

[3] Theoretical maximum memory bandwidth for intel® core™ x-series processors. https://www.intel.com/content/www/us/en/support/articles/000056722/processors/intel-core-processors.html.

[4] Ultrascale+ fpga product tables and product selection guide. https://www.xilinx.com/support/documentation/selection-guides/ultrascale-plus-fpga-product-selection-guide.pdf.

[5] Vector search - microsoft ai lab. https://www.microsoft.com/en-us/ai/ai-lab-vector-search.

[6] Vertex ai matching engine overview. https://cloud.google.com/vertex-ai/docs/matching-engine/overview.

[7] Ameer MS Abdelhadi, Christos-Savvas Bouganis, and George A Constantinides. Accelerated approximate nearest neighbors search through hierarchical product quantization. In *2019 International Conference on Field-Programmable Technology (ICFPT)*, 2019.

[8] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. In *International conference on similarity search and applications*, 2017.

[9] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 2013.

[10] Adrian M Caulfield, Eric S Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, et al. A cloud-scale acceleration architecture. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[11] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, et al. Wide & deep learning for recommender systems. In *Proceedings of the 1st workshop on deep learning for recommender systems*, 2016.

[12] Paul Covington, Jay Adams, and Emre Sargin. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM conference on recommender systems*, 2016.

[13] Miao Fan, Jiacheng Guo, Shuai Zhu, Shuo Miao, Mingming Sun, and Ping Li. Mobius: towards the next generation of query-ad matching in baidu's sponsored search. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019.

[14] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. Azure accelerated networking: Smartnics in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018.

[15] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. Fast approximate nearest neighbor search with the navigating spreading-out graph. In *Proceedings of the VLDB Endowment*, 2019.

[16] Weihao Gao, Xiangjun Fan, Chong Wang, Jiankai Sun, Kai Jia, Wenzhi Xiao, Ruofan Ding, Xingyan Bin, Hui Yang, and Xiaobing Liu. Deep retrieval: Learning a retrievable structure for large-scale recommendations. In *arXiv preprint arXiv:2007.07203*, 2021.

[17] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. Optimized product quantization. In *IEEE transactions on pattern analysis and machine intelligence*, 2013.

[18] Mihajlo Grbovic and Haibin Cheng. Real-time personalization using embeddings for search ranking at airbnb. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2018.

[19] Udit Gupta, Carole-Jean Wu, Xiaodong Wang, Maxim Naumov, Brandon Reagen, David Brooks, Bradford Cottel, Kim Hazelwood, Mark Hempstead, Bill Jia, et al. The architectural implications of facebook's dnn-based personalized recommendation. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020.

[20] Jui-Ting Huang, Ashish Sharma, Shuying Sun, Li Xia, David Zhang, Philip Pronin, Janani Padmanabhan, Giuseppe Ottaviano, and Linjun Yang. Embedding-based retrieval in facebook search. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2020.

[21] Herve Jegou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. In *IEEE transactions on pattern analysis and machine intelligence*, 2010.

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

[23] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with gpus. In *arXiv preprint arXiv:1702.08734*, 2017.

[24] Gwangsun Kim, Jiyun Jeong, John Kim, and Mark Stephenson. Automatically exploiting implicit pipeline parallelism from multiple dependent kernels for gpus. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, 2016.

[25] Sen Li, Fuyu Lv, Taiwei Jin, Guli Lin, Keping Yang, Xiaoyi Zeng, Xiao-Ming Wu, and Qianli Ma. Embedding-based product retrieval in taobao search. In *Proceedings of the 27th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2021.

[26] Jianxun Lian, Fuzheng Zhang, Xing Xie, and Guangzhong Sun. Towards better representation learning for personalized news recommendation: a multi-channel deep fusion approach. In *IJCAI*, 2018.

[27] Alec Lu, Zhenman Fang, Nazanin Farahpour, and Lesley Shannon. Chip-knn: A configurable and high-performance k-nearest neighbors accelerator on cloud fpgas. In *2020 International Conference on Field-Programmable Technology (ICFPT)*, 2020.

[28] Yu A Malkov and Dmitry A Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. In *IEEE transactions on pattern analysis and machine intelligence*, 2018.

[29] Naoyuki Matsumoto, Koji Nakano, and Yasuaki Ito. Optimal parallel hardware k-sorter and top k-sorter, with fpga implementations. In *2015 14th International Symposium on Parallel and Distributed Computing*, 2015.

[30] Yuriy Mishchenko, Yusuf Goren, Ming Sun, Chris Beauchene, Spyros Matsoukas, Oleg Rybakov, and Shiv Naga Prasad Vitaladevuni. Low-bit quantization and quantization-aware training for small-footprint keyword spotting. In *2019 18th IEEE International Conference On Machine Learning And Applications (ICMLA)*, 2019.

[31] Hieu Duy Nguyen, Anastasios Alexandridis, and Athanasios Mouchtaris. Quantization aware training with absolute-cosine regularization for automatic speech recognition. In *Interspeech*, 2020.

[32] Reid Pinkham, Shuqing Zeng, and Zhengya Zhang. Quicknn: Memory and performance optimization of kd tree based nearest neighbor search for 3d point clouds. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020.

[33] Anil Shanbhag, Holger Pirk, and Samuel Madden. Efficient top-k query processing on massively parallel hardware. In *Proceedings of the 2018 International Conference on Management of Data*, 2018.

[34] Josef Sivic and Andrew Zisserman. Video google: A text retrieval approach to object matching in videos. In *Proceedings Ninth IEEE International Conference on Computer Vision*, 2003.

[35] Wei Song, Dirk Koch, Mikel Luján, and Jim Garside. Parallel hardware merge sorter. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2016.

[36] Elias Stehle and Hans-Arno Jacobsen. A memory bandwidth-efficient hybrid radix sort on gpus. In *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017.

[37] Shyam A Tailor, Javier Fernandez-Marques, and Nicholas D Lane. Degree-quant: Quantization-aware training for graph neural networks. In *arXiv preprint arXiv:2008.05000*, 2020.

[38] Jizhe Wang, Pipei Huang, Huan Zhao, Zhibo Zhang, Binqiang Zhao, and Dik Lun Lee. Billion-scale commodity embedding for e-commerce recommendation in alibaba. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2018.

[39] Ruobing Xie, Zhijie Qiu, Jun Rao, Yi Liu, Bo Zhang, and Leyu Lin. Internal and contextual attention network for cold-start multi-channel matching in recommendation. In *IJCAI*, 2020.

[40] Xinyang Yi, Ji Yang, Lichan Hong, Derek Zhiyuan Cheng, Lukasz Heldt, Aditee Kumthekar, Zhe Zhao, Li Wei, and Ed Chi. Sampling-bias-corrected neural modeling for large corpus item recommendations. In *Proceedings of the 13th ACM Conference on Recommender Systems*, 2019.

[41] Chaoliang Zeng, Layong Luo, Teng Zhang, Zilong Wang, Luyang Li, Wenchen Han, Nan Chen, Lebing Wan, Lichao Liu, Zhipeng Ding, et al. Tiara: A scalable and efficient hardware acceleration architecture for stateful layer-4 load balancing. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022.

[42] Han Zhang, Songlin Wang, Kang Zhang, Zhiling Tang, Yunjiang Jiang, Yun Xiao, Weipeng Yan, and Wen-Yun Yang. Towards personalized and semantic retrieval: An end-to-end solution for e-commerce search via embedding learning. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2020.

[43] Heng-Ru Zhang, Fan Min, Zhi-Heng Zhang, and Song Wang. Efficient collaborative filtering recommendations with multi-channel feature vectors. In *International Journal of Machine Learning and Cybernetics*, 2019.

[44] Jialiang Zhang, Soroosh Khoram, and Jing Li. Efficient large-scale approximate nearest neighbor search on opencl fpga. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018.

[45] Teng Zhang, Jianying Wang, Xuntao Cheng, Hao Xu, Nanlong Yu, Gui Huang, Tieying Zhang, Dengcheng He, Feifei Li, Wei Cao, et al. Fpga-accelerated compactions for lsm-based key-value store. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, 2020.

[46] Weijie Zhao, Shulong Tan, and Ping Li. Song: Approximate nearest neighbor search on gpu. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, 2020.

[47] Zhen Zheng, Chanyoung Oh, Jidong Zhai, Xipeng Shen, Youngmin Yi, and Wenguang Chen. Versapipe: a versatile programming framework for pipelined computing on gpu. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2017.