Scalable Billion-point Approximate Nearest Neighbor Search Using SmartSSDs

Bing Tian, Haikun Liu, Zhuohui Duan, Xiaofei Liao, Hai Jin, and Yu Zhang,
Huazhong University of Science and Technology

https://www.usenix.org/conference/atc24/presentation/tian

This paper is included in the Proceedings of the
2024 USENIX Annual Technical Conference.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-41-0

Scalable Billion-point Approximate Nearest Neighbor Search Using SmartSSDs

Bing Tian, Haikun Liu[*], Zhuohui Duan, Xiaofei Liao, Hai Jin, Yu Zhang
National Engineering Research Center for Big Data Technology and System,
Service Computing Technology and System Lab/Cluster and Grid Computing Lab,
School of Computer Science and Technology, Huazhong University of Science and Technology, China

Abstract

Approximate nearest neighbor search (ANNS) in high-dimensional vector spaces has become increasingly crucial in database and machine learning applications. Most previous ANNS algorithms require TB-scale memory to store indices of billion-scale datasets, making their deployment extremely expensive for high-performance search. The emerging SmartSSD technology offers an opportunity to achieve scalable ANNS via *near data processing* (NDP). However, there remain challenges to directly adopt existing ANNS algorithms on multiple SmartSSDs.

In this paper, we present SmartANNS, a SmartSSD-empowered billion-scale ANNS solution based on a hierarchical indexing methodology. We propose several novel designs to achieve near-linear scaling with multiple SmartSSDs. First, we propose a "host CPUs + SmartSSDs" cooperative architecture incorporated with hierarchical indices to significantly reduce data accesses and computations on SmartSSDs. Second, we propose dynamic task scheduling based on optimized data layout to achieve both load balancing and data reusing for multiple SmartSSDs. Third, we further propose a learning-based shard pruning algorithm to eliminate unnecessary computations on SmartSSDs. We implement SmartANNS using Samsung's commercial SmartSSDs. Experimental results show that SmartANNS can improve *query per second* (QPS) by up to 10.7× compared with the state-of-the-art SmartSSD-based ANNS solution–CSDANNS. Moreover, SmartANNS can achieve near-linear performance scalability for large-scale datasets using multiple SmartSSDs.

1 Introduction

Approximate nearest neighbor search (ANNS) in high-dimensional spaces refers to the process of searching objects that are most similar to a given vector. It is a fundamental problem in algorithms research, and has a broad range of applications in many fields such as data mining [1], information

[*]Corresponding author: Haikun Liu (hkliu@hust.edu.cn)

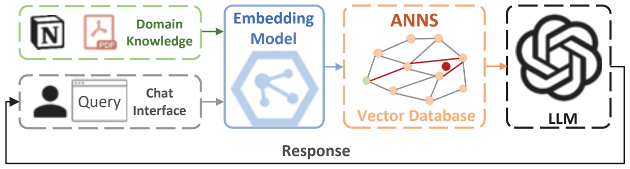


Figure 1: Retrieval augmented generation for LLMs

retrieval [2], and AI-powered recommendation systems [3, 4]. Particularly, driven by the recent research boom in *large language models* (LLMs) [5–8], ANNS services backed with vector databases have become a fundamental building block of modern AI infrastructure. As shown in Figure 1, the domain knowledge in various data formats (such as documents, images, and speeches) are embedded into high-dimensional vector spaces, and are stored in a vector database in the form of feature vectors. Upon a user query from the chatbot, the ANNS engine finds similar objects based on the query's semantics, and delivers the most relevant and context-aware results to the LLM for further processing.

There have been numerous algorithms designed for the ANNS problem, such as graph-based [9–11], hash-based [12, 13], tree-based [14], and quantization-based [15–17], mainly focusing on the methodologies of indexing. Most ANNS algorithms rely on in-memory indices to support fast and accurate search, but significantly increase the memory resource requirement. For example, many commercial recommendation systems such as Alibaba [18] usually require TB-scale memory space to accommodate billion-scale vectors. However, the huge resource requirement significantly increases the *total cost of ownership* (TCO) (including purchasing and operating costs), making the ANNS service impractical to scale to large-scale datasets with hundreds of billions vectors. The scalability of these algorithms is mainly limited by the large amount of indices maintained in main memory.

To reduce the cost of ANNS services, a practical approach is to store the majority of vector indices on SSDs while using a moderately-sized DRAM as working memory [19, 20]. However, recent studies [21] have demonstrated that I/O operations can account for about 70% of total execution time in SSD-based ANNS approaches. The performance of batch queries also can not scale due to limited PCIe bandwidth between SSDs and host memory. Moreover, ANNS engines are often integrated with other software to provide collaborative services, such as ANNS supported recommendation systems, ANNS empowered *retrieval augmented generation* (RAG) for LLMs. These scenarios may aggravate the PCIe bandwidth contention between ANNS engines and other programs, and thus significantly degrade the application performance.

The emergence of SmartSSDs [22] (i.e., computational storage devices) offers vast optimization opportunities to balance the performance and cost for ANNS services. Unlike traditional SSD-based approaches which often incur frequent data movement between host memory and SSDs [21, 23, 24], SmartSSDs leverage the *near data processing* (NDP) paradigm to process data locally with their on-board DRAM and *Field Programmable Gate Arrays* (FPGAs). In this solution, the host can offload ANNS queries to SmartSSDs for near data processing. SmartSSDs perform ANNS queries on SSD-resident indices locally, and then return partial results to the host for aggregation. More importantly, since each SmartSSD uses its internal PCIe bus for local data transfer, multiple SmartSSDs can achieve near-linear acceleration for ANNS queries. This approach also significantly reduces bandwidth contention of host PCIe bus and host CPU/memory resource consumption.

Recently, a proof-of-concept study CSDANNS [23] has explored SmartSSDs to offload large-scale ANNS for batch queries. CSDANNS implements a classic graph-based ANNS algorithm—*Hierarchical Navigable Small World* (HNSW) [9] with FPGA in each SmartSSD. It splits the dataset sequentially to ensure that the HNSW indices of each segment can be fully accommodated by the on-board DRAM. However, this simple approach has to scan all indices for each query on all SmartSSDs, and thus results in significant computational overhead on resource-limited SmartSSDs. Furthermore, since CSDANNS offloads all ANNS queries to SmartSSDs entirely, it only achieves sub-optimal performance. Hierarchical indexing, such as SPANN [20], is a promising way to reduce the search space of ANNS without compromising accuracy. It partitions the billion-scale dataset into a number of shards based on a clustering algorithm. Upon a query, the centroids of all shards are first consulted to find shards that most likely contain the nearest neighbors. This approach can significantly reduce unnecessary data accesses and computations, and thus is applicable to SmartSSDs for ANNS offloading.

However, there still remain several challenges to directly apply the hierarchical indexing approach to multiple SmartSSDs. **Challenge 1:** Due to lack of communication channels among multiple SmartSSDs, each SmartSSD has to search more shards to achieve the required level of accuracy, leading to additional computation overhead. Thus, a global coordinator is required to reduce the search space via consulting the centroids of all shards. **Challenge 2:** Using the hierarchical indexing approach, each ANNS query is delivered to a subset of shards, resulting in an uneven (skewed) distribution of accesses across different shards. A batch of queries, if not properly scheduled, may lead to load imbalance across SmartSSDs, which eventually harms the system scalability. **Challenge 3:** Since the search scope of queries may be significantly different, it is usually difficult to determine the minimum number of shards for each query. A static configuration may result in either unnecessary computations or accuracy loss.

In this paper, we present SmartANNS, a scalable ANNS solution using SmartSSDs for billion-scale datasets, based on the hierarchical indexing methodology [20]. We explore several optimization technologies to tackle the above challenges. 1) We propose a "host CPUs + SmartSSDs" cooperative processing architecture for ANNS. SmartANNS maintains the centroid of each shard in host main memory while storing only HNSW indices of each shard in SmartSSDs. During the search stage, the host CPU first traverses the in-memory centroids, and then dispatches query tasks to different SmartSSDs for more fine-gained search. In this way, the host CPU is used as a global coordinator to reduce the search space in all SmartSSDs. 2) We propose a dynamic task scheduling mechanism to balance the load across SmartSSDs and fully exploit data reuse among queries. We first identify the hotness of each shard via offline sampling, and evenly distribute hot shards among different SmartSSDs to optimize the data layout. Then, we schedule query tasks with overlapped shards by considering both data locality and the load of SmartSSDs. 3) We further exploit a learning-based shard pruning algorithm to avoid unnecessary computations on SmartSSDs. We first construct the mapping between each query and its search scope via offline training, and then the host CPU determines the minimum number of shards required by each query at runtime. Overall, we make the following contributions:

- For **Challenge 1**, we propose a "host CPU + SmartSSDs" cooperative processing architecture for scalable ANNS services. It exploits hierarchical indices to reduce massive data accesses and computations on SmartSSDs.
- For **Challenge 2**, we propose a dynamic task scheduling mechanism based on the optimized data layout to achieve both load balancing and data reusing.
- For **Challenge 3**, we propose a lightweight learning-based shard pruning algorithm to eliminate unnecessary computations on SmartSSDs.
- We implement SmartANNS using Samsung's commercial SmartSSDs, with an optimized implementation of the HNSW search kernel using on-board FPGAs. Experimental results show that SmartANNS can improve *query per second* (QPS) by up to 10.7$\times$ compared with the state-of-the-art SmartSSD-based ANNS solution–

CSDANNS [23]. SmartANNS also outperforms the state-of-the-art SSD-based solution–SPANN [20], and GPU-based solutions–CUhnsw [25] and GGNN [26]. More importantly, SmartANNS can achieve near-linear scalability for large-scale datasets using multiple SmartSSDs.

2 Background and Motivation

In this section, we first introduce the fundamental background for ANNS and SmartSSDs. Then, we present SmartSSD-empowered optimization opportunities for ANNS, and motivate the design of SmartANNS based on our observations.

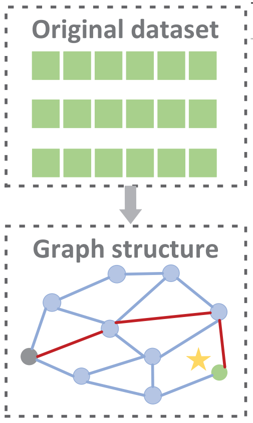
2.1 Approximate Nearest Neighbor Search

The *nearest neighbor search* (NNS) problem refers to finding the closest point to a given point within a dataset. In this context, let us consider a dataset D consisting of n points, where each point is represented by a d-dimensional vector $(x_1, x_2, ..., x_d)$. Given a query point q represented by a d-dimensional vector $(q_1, q_2, ..., q_d)$, the goal of NNS is to identify a point p in the dataset D such that the distance $d(p,q)$ is minimized. The distance that measures the similarity between two points, could be Euclidean distance, Hamming distance, and so on. For a given dataset, the distance metric is often determined by the embedding model that generates vectors.

In high-dimensional vector spaces, the NNS becomes a computationally hard problem because it is impossible to retrieve the exact neighbors in a large-scale dataset via a linear scan [27]. This is a phenomenon known as the curse of dimensionality [28]. To circumvent this challenge, many *approximate nearest neighbor search* (ANNS) algorithms have been proposed. The goal is to retrieve a given number of neighbors which are close to the optimal candidate. These algorithms can significantly enhance search efficiency by utilizing various indexing techniques, which effectively prune regions of the dataset that are unlikely to contain the nearest neighbors. Among numerous ANNS algorithms, graph-based approaches have gained significant prominence in recent years. For example, HNSW [9], NSG [10], and FANNG [11] have demonstrated promising performance and accuracy for ANNS services. As illustrated in Figure 2, the graph-based approach constructs a proximity graph. Feature vectors are abstracted as nodes in the graph, and edges are deemed as the distance between two nodes. For each query vector, the algorithm navigates the graph from a given node and sequentially traverse neighboring nodes that are likely candidates of approximate nearest neighbors.

2.2 Computational Storage

Computational storage device (CSD) [29] follows a design principle that "computing closer to the data source is more efficient than transferring data for remote processing." By



```
Algorithm1: Graph-based Vector Search
Input: query $q$, result $k$, search list $L > k$
Output: $k$ nearest neighbor of a query $q$
 1: $C.add(Enterpoint)$
 2: while $|C| > 0$ do
 3:     $j$ = first index vertex in C
 4:     $VisitList.add(j)$
 5:     for each $e \in neighbor(j)$ do
 6:         if $e \notin VisitList$
 7:         $dist = getdist(q, e)$
 8:         $VisitList.add(e); C.add(e)$
 9:     end for
10:     Sort C in ascending order of $dist$
11:     $C.resize(L)$ when $C.size() > L$
12: return $C[:k]$
```

Figure 2: An example for graph-based ANNS

placing computing units within storage devices, CSDs support near data processing naturally. These devices not only reduce the amount of data movement between storage device and host CPUs, but also reduce the usage of host resources. Figure 3 illustrates a traditional computing architecture and a CSD-empowered NDP architecture. In traditional architectures, all PCIe devices compete for the PCIe bus, and thus the system performance is constrained by the bandwidth of the host PCIe bus. In the CSD-empowered NDP architecture, since each CSD accesses data locally using its internal PCIe switch, the system can achieve a near-linear scaling with more CSDs.

Commercial CSDs, such as Samsung's SmartSSDs, have been available recently in the marketplace. Figure 3 illustrates the inner architecture of the Samsung SmartSSD. The on-board DRAM serves as a cache for the NAND Flash, and can be accessed by both the FPGA and the host CPU. The host CPU can access data via standard I/O interfaces, and can also offload computational instructions directly to the SmartSSD. However, data movement between the NAND Flash and the FPGA's DRAM is achieved by a *peer-to-peer* (P2P) mechanism via the internal PCIe switch, as shown in Figure 3. At present, Samsung SmartSSD's internal bandwidth between the NAND Flash and the FPGA is only 3 GB/s.

2.3 Motivations

Advantages of using SmartSSDs for ANNS. Billion-scale ANNS services usually require TB-scale memory space to achieve high-performance queries. In the traditional computing architecture, since SSD-based ANNS algorithms often incur frequent and costly data movement between disk and main memory, the ANNS performance is mainly bounded by the host PCIe bandwidth [21, 23]. We implement the state-of-the-art SSD-based SPANN [20] and experiments show that I/O operations account for about 67% of total execution time. Although faster PCIe 6.0 is beneficial for performance

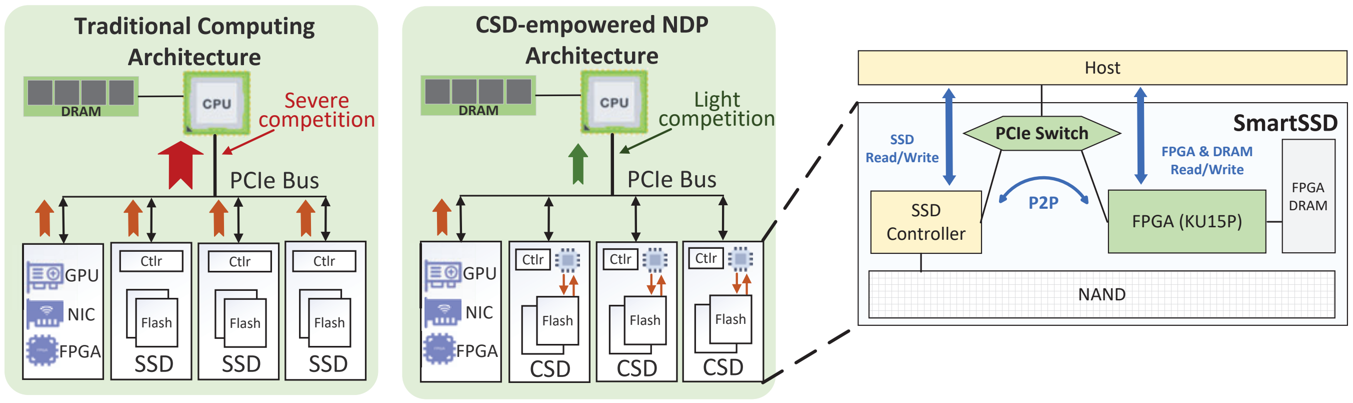
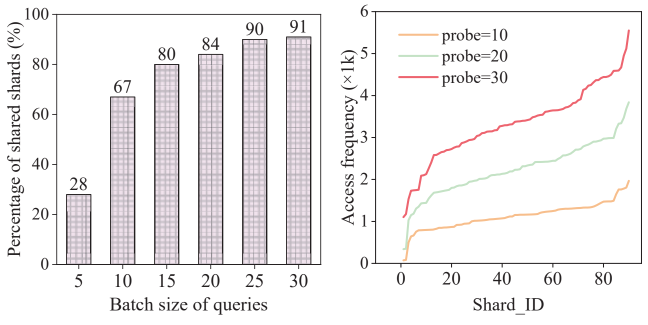


Figure 3: Traditional computing architecture vs. CSD-empowered NDP architecture

improvement, SPANN may still spend more than 20% of total time in I/O operations. SmartSSDs can relax this constraint by using their internal channels for data access. Thus, SmartSSDs offer several significant benefits for deploying large-scale ANNS service:

- The large storage capacity of SmartSSDs is sufficient to accommodate ANNS indexes of billion-scale datasets.
- Benefiting from the near data processing paradigm, SmartSSDs can significantly reduce data movement between NAND flash and host main memory. Moreover, on-board FPGAs are also efficient for massive vector distance calculations in ANNS algorithms.
- The in-storage computing capability of SmartSSDs offers more opportunities to mitigate host resource contention. When ANNS engines are co-deployed with other software such as recommendation systems and RAG-based LLMs, the host can allocate more CPU, memory, and I/O resources to serve these interactive applications.
- Since each SmartSSD can work as a SoC to process data individually, multiple SmartSSDs can achieve near-linear performance scalability if there is not data dependency among different SmartSSDs.

Limitations of an existing CSD-based ANNS solution. CSDANNS [23] is so far the only proof-of-concept work that implements a graph-based ANNS algorithm–HNSW [9] using on-board FPGAs of CSDs. By splitting the original dataset into multiple shards, CSDANNS constructs graph-based indices for each shard. These shards and their indices are evenly stored in multiple SmartSSDs. However, for each query, CSDANNS has to scan all shards one by one using on-board FPGA kernels, incurring significant computational overhead. This magnifies the disadvantage of SmartSSDs in terms of limited computational capability, and even counteracts the benefits of NDP. Additionally, because CSDANNS completely offloads the graph-based ANNS to SmartSSDs,



(a) Data locality among queries (b) Access frequency of shards

Figure 4: The data access pattern among different queries

it overlooks the possibilities of utilizing host CPU for hardware/software cooperative processing. As a result, CSDANNS achieves sub-optimal performance. This motivates us to explore a more sophisticated approach for deploying ANNS services on SmartSSDs.

Optimization opportunities of hierarchical indexing. ANNS algorithms using clustering-based hierarchical indexes [20, 30] have been demonstrated as an efficient way to decrease the computational cost of ANNS without compromising accuracy. This approach partitions a dataset into a number of shards where high-similarity vectors are clustered together. Then, graph-based indices are constructed for each shard to accelerate the traversal. Before a search task is dispatched to a SmartSSD, we can prune irrelevant shards by comparing the centroid of each shard with the query vector, and only traverse a few shards whose centroids have a low distance with the query vector. Thus, the hierarchical indexing approach can significantly reduce the computations within SmartSSDs.

To handle ANNS queries on SmartSSDs more efficiently, we further explore the data access pattern of hierarchical indices through two experiments. We exploit a *hierarchical balanced clustering* (HBC) algorithm [31] to partition a SIFT dataset [32] containing 100M vectors into 90 shards of the

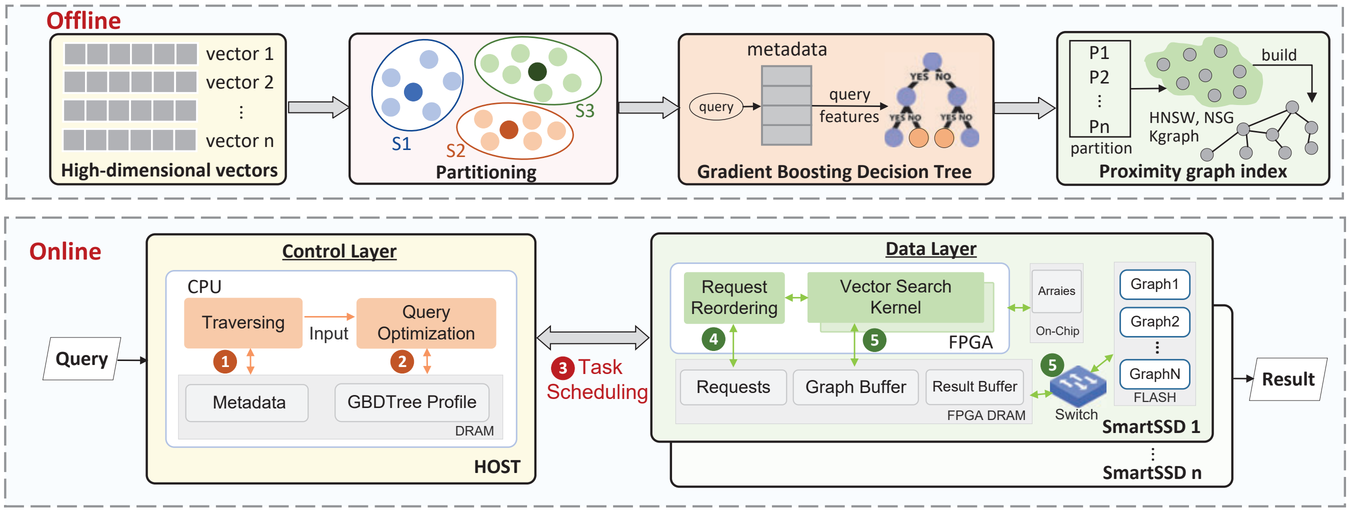


Figure 5: SmartANNS architecture

same size, and generate 10K queries to find the most similar vectors. By default, we configure each query to search top-10 nearest shards. First, we vary the batch size of queries to measure the percentage of shards touched by different queries, as shown in Figure 4a. Second, we configure three probes, i.e., the top-K nearest shards that should be searched by a query, and then measure the access frequency of each shard, as shown in Figure 4b. We have two observations as follows.

Observation 1: A large portion of shards are accessed by different queries over a period of time, implying a good data locality. Since different queries may search the same shard, and some popular shards may be accessed frequently by a batch of queries. As shown in Figure 4a, when the batch size exceeds 25, 90% of total shards are accessed more than once by different queries. **This implies that a task scheduling scheme should fully exploit the data locality among queries to reuse in-memory shards as much as possible.**

Observation 2: The access distribution of different shards are highly skewed. As shown in Figure 4b, all shards are ordered according to their access frequencies. About 10% shards are extremely hot, while about 10% shards are infrequent accessed. Since most similar vectors are clustered together and represent a similar degree of hotness, **we should carefully place hot shards on different SmartSSDs to achieve load balancing**.

3 SmartANNS Overview

SmartANNS exploits hierarchical indices to achieve collaborative ANNS processing with “host CPUs + SmartSSDs”. Figure 5 shows an overview of SmartANNS architecture.

Offline processing: Like conventional ANNS algorithms, SmartANNS also has to construct indices in an offline manner. At first, SmartANNS employs a HBC algorithm [31] to partition a dataset into multiple equally-sized shards in which vectors have a high similarity. Then, an HNSW index is constructed for each shard independently. The centroids of these shards are maintained in host main memory while shards are stored in SmartSSDs. After that, SmartANNS samples a subset of training queries to train *gradient boosting decision trees* (GBDT) [33]. This learned model can be used to determine the minimum search scope for each query at runtime. In addition, SmartANNS also collects the hotness of each shard to calibrate the data layout on multiple SmartSSDs.

Online processing: Upon an ANNS query, SmartANNS calculates the distance between the query vector and the centroid of each shard. With these distances, gradient boosting decision trees predict the number of shards that should be searched to find near neighbors. After that, the host distributes query tasks to different SmartSSDs using a task scheduling mechanism elaborated in Section 4.3.

When a SmartSSD receives a query task, it prioritizes this task if it can reuse shards that have been loaded into the on-board DRAM with other tasks. For each task, the SmartSSD utilizes the internal PCIe switch to transfer indices of the shard from the SSD to the FPGA’s onboard DRAM. Then, the ANNS engine within the SmartSSD begins its iterative search and sends results back to the host. Finally, the host aggregates all results returned from different SmartSSDs.

4 Design and Implementation

In this section, we present the design of SmartANNS. We elaborate the construction of hierarchical indices, the learning-based shard pruning, and the task scheduling for multiple SmartSSDs based on optimized data layout. At last, we present the FPGA implementation of the vector search engine within SmartSSDs.

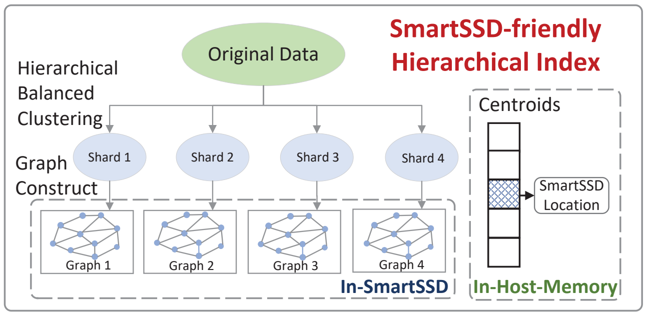


Figure 6: Hierarchical indices in SmartANNS

4.1 Hierarchical Indexing

Figure 6 illustrates how those hierarchical indices are constructed in both SmartSSDs and the host. To reduce the computational cost of SmartSSDs, SmartANNS exploits the HBC algorithm [31] to partition a vector dataset into multiple shards as evenly as possible. We first configure the maximum size of shards that can be accommodated by the SmartSSD's onboard DRAM. Then, the dataset is partitioned iteratively in a top-down manner till all shards become smaller than the configured maximum size. To circumvent boundary concerns during clustering [20], we also adopt a flexible vector assignment strategy. If a vector lies on the boundary of multiple shards S_i, we place this vector according to the following formula:

$$v \in S_i \Leftrightarrow Dist(v, S_i) \leq (1+\varepsilon) \times Dist(v, S_1)$$

where v denotes the vector to be placed, S_1 represents the shard that is nearest to vector v, S_i denotes other shards, and ε determines the distance whether a vector should be simultaneously placed to other shards. Each vector can be placed in two shards at most. After the partitioning phase, we employ the HNSW algorithm to generate a graph-based index for all vectors within each shard individually. The proximity graph is constructed by continuously adding new vectors to an empty graph. When a vector is added as a new vertex, its top-k (typically 64) nearest neighbors are searched in the current graph, and new edges between the newly-added vertex and its nearest neighbors are constructed. Then, these neighboring vertices should update their nearest neighbors to keep the maximum number of edges.

However, it is costly to directly place shards along with their centroids on different SmartSSDs because this data layout would result in additional computational cost. Due to a lack of inter-SmartSSD communication channels, a query on one SmartSSD is not aware of distances between this query and shards on other SmartSSDs. For a given query, the closest shard within a SmartSSD may be not the closest shard globally. Therefore, each SmartSSD usually should scan more shards locally to meet the desired accuracy.

To address this issue, SmartANNS exploits the host CPU as a central coordinator. Given that all SmartSSDs can be accessed by the host CPU, SmartANNS retains centroids of all shards within the host memory. It manages a key-value pair to record the address of each shard in SmartSSDs. When a number of shards associated with a query are determined, the host CPU retrieves the corresponding shards from SmartSSDs according to these key-value pairs. Then, tasks are assigned to SmartSSDs based on the task scheduling mechanism. With these hierarchical indices, each query is aware of the most closest shards that should be further searched on SmartSSDs, and thus can eliminate unnecessary computations within SmartSSDs.

Table 1: GBDT input features

Features	Description
F0: query	The query vector
F1: relative_distance	The ratio of D_k to D_1
F2: num_shard	The number of all shards

4.2 Shard Pruning

Because the difficulty of queries may be significantly different, the search scope of each query may change dynamically [34, 35]. Inverted indexing approaches [15–17] often construct much smaller shards and a large amount of centroids. Thus, the majority of computing overhead stems from centroid traversal. In contrast, SmartANNS focuses on minimizing the host resource consumption while fully realizing the potential of SmartSSDs. To this end, SmartANNS constructs larger shards and correspondingly fewer centroids. The cost of traversing centroids is relative low, and the most computational cost lies in searching shards. Thus, it is not reasonable to configure a fixed number of shards for all queries since irrelevant shards may be unnecessarily scanned.

To circumvent this issue, we exploit a learning-based approach to determine the optimal number of shards for each query. We choose the *gradient boosting decision trees* (GBDT) model due to its lightweight and high performance. GBDT is an ensemble learning algorithm that combines multiple simple decision trees (called "weak learners") to create a strong predictive model. It trains the learned model iteratively by initially predicting the mean, computing residuals, and fitting weak decision trees to these negative gradients. During the inference stage, the model generates predictions by assimilating weighted contributions of all individual weak models.

The input parameters of this model include the query vector, its spatial relationship with shards, and the total number of shards. These input features are illustrated in Table 1. We use relative distances as features for model training. Let D_1 denote the distance between the query and the top-1 nearest shard, and D_k denote the distance between the query and the top-k nearest shards, we use the ratio of D_k to D_1 as key features to train the decision tree. These abstracted features can

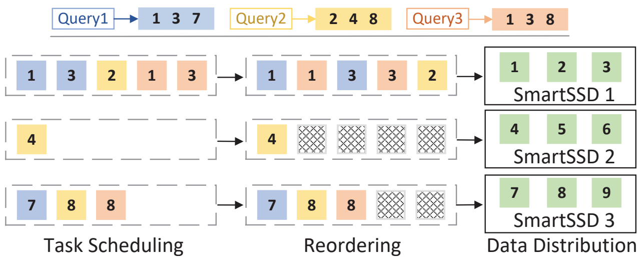


Figure 7: Data parallelism among multiple SmartSSDs

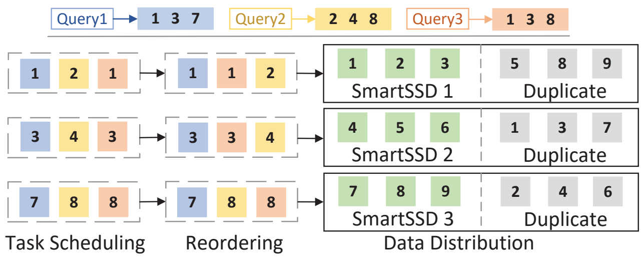


Figure 8: Optimized data layout of shards for task scheduling

reflect the inherent correlations between queries and data, and improve the model's adaptability and robustness. To prepare training data, we sample one million training vectors from each billion-scale dataset. For output values, we perform an exhaustive search for the ground truth nearest neighbors and then identify the minimum number of shards that cover them. This process takes less than an hour using an NVIDIA V100 GPU. Then, we set the learning rate to 0.05 and begin training the GBDT model over 500 iterations, which takes only 3 minutes using a CPU.

The trained model is very lightweight, with a memory footprint just about 1 MB, and an inference takes just a few tens of microseconds. To integrate this model into SmartANNS, we deploy this model in the host using a CPU core. Upon each query, we calculate its distance to each shard and then sort these distances in an ascending order. The model can determine the ideal number of shards for each query, and then these shards are scanned by FPGA kernels in SmartSSDs.

4.3 Task Scheduling

SmartANNS enables the functionality of NDP using multiple SmartSSDs in parallel. A straightforward approach is to exploit task parallelism, i.e., the dataset is replicated to each SmartSSD and queries are evenly dispatched to all SmartSSDs. However, this simple approach cannot fully exploit data locality among queries and may lead to a significant waste of storage capacity. In contrast, we address these issues by fully exploiting data parallelism among multiple SmartSSDs. In this approach, the dataset is evenly distributed across all SmartSSDs, and queries are dispatched to appropriate SmartSSDs based on the data distribution. However, this strategy may result in load imbalance among SmartSSDs because the data hotness of shards may be different, as illustrated in Figure 7.

To improve the system scalability with multiple SmartSSDs, we first optimize the data layout according to the hotness of shards and their duplicates. Then, we design a task scheduler based on optimized data layout. When we construct the training data for GBDT, we occasionally record the hotness of each shard. When we place shards to SmartSSDs, we sort shards based on the hotness. By iteratively placing the shard with the highest hotness to the SmartSSD with the lowest cumulative hotness, we can ultimately achieve approximately the same hotness values for all SmartSSDs. Furthermore, we replicate shards from one SmartSSD to another SmartSSD once, as shown in Figure 8. This data replication mechanism offers more flexibility for load balancing by dynamically migrating tasks from an overloaded SmartSSD to another SmartSSD.

The key idea behind the task scheduler is to dynamically balance the load of SmartSSDs by taking into account data reuse among queries. Algorithm 1 outlines the pseudo code of task scheduling. Its input includes the mappings between SmartSSD devices and shards, and tasks to be assigned. The output is a two-dimensional array recording tasks assigned to each SmartSSD. We propose a simple model to estimate the total load of a task list. The execution of a task includes loading shards to the FPGA DRAM and vector searching. Since the size of each shard is almost the same, the searching latencies for all tasks are similar. Thus, we can estimate the searching latency of a task, and the latency of shard loading according to the shard size and the SmartSSD's internal bandwidth. Our model only counts the latency of shard loading once if duplicated shards are accessed by multiple tasks.

During scheduling, we first identify SmartSSDs that store the shard corresponding to the task, and deem them as base devices (line 2). Then, we check whether there are already assigned tasks on these devices that have the same shard (line 3). If these base devices either have or do not have such task (line 4), the additional processing overhead associated with these devices is the same. Therefore, we select the device with the least load among these base devices to assign the task (line 4-line 6). In cases where some base devices have such task while others do not have (line 8), we need to consider the case where the additional processing overhead differs due to data reuse. For this situation, we create a temporary task list for each base device and insert the given task into the list. Subsequently, we use the aforementioned model to estimate the total load for the temporary task list. Finally, we select the device with the minimum estimated load as the assignment target (line 8-line 16). We record the load of each device and update the target device after each task assignment. Figure 8 shows a simple example. Our approach can fully leverage data reusing among queries while guaranteeing load balancing among SmartSSDs.

Algorithm 1: Task Scheduling

```
Input: $Devices, Tasks(query, shard\_id)$
Output: $Device\_to\_Tasks[\,][\,]$
for $i = 0; i < Tasks.size(); i{+}{+}$ do
    /* Find devices that store task-relevant shards. */
    $Base[\,] \leftarrow FindBase(Devices, Tasks[i])$;
    /* Check whether there is data reuse on device for
       this task. */
    $Assign[\,] \leftarrow Check(Device\_to\_Tasks, Tasks[i])$;
    if $(Assign.size = 0) \vee (Assign.size = Base.size)$
     then
        $Target \leftarrow MinWork(DeviceLoad, Base)$;
        $Device\_to\_Tasks[Target].insert(Tasks[i])$;
        $DeviceLoad.update(Target)$;
    else
        $TimeInfo[\,] \leftarrow NULL$;
        for $j = 0; j < Base.size(); j{+}{+}$ do
            $Temp[\,] \leftarrow Device\_to\_Tasks[Base[j]]$;
            $Temp.insert(Tasks[i])$;
            $TimeInfo[j] \leftarrow Estimate(Temp)$;
        end
        $Target \leftarrow Min(TimeInfo)$;
        $Device\_to\_Tasks[Target].insert(tasks[i])$;
        $DeviceLoad.update(Target)$;
    end
end
return $Device\_to\_Tasks$;
```

4.4 Vector Search Engine

We implement a fully-optimized HLS-based vector search engine within SmartSSDs based on the HNSW algorithm. As shown in Figure 9, this engine is composed of two key modules: task reordering and the search kernel. As illustrated in Figure 4a, a shard may be repeatedly accessed by different tasks. To eliminate this redundancy in SmartSSDs, we design the task reordering module. When SmartSSD receives tasks from the host, it first rearranges the execution order of tasks so that tasks associated with the same shard can be handled sequentially. Then, SmartSSD loads the graph-based index of the shard from the NAND Flash into the FPGA DRAM for the subsequent searching. In this way, the data can be reused among queries, effectively amortizing the cost of data accesses.

The key components of the search kernel include a layer monitor, distance calculation modules, sorting modules, and array updating modules. Upon searching on the loaded graph-based index, the kernel maintains three distinct lists: the visited list, the sorted candidate list, and the sorted final list. When the first vector in the candidate list is traversed at each layer, the distance calculation module is invoked to calculate the distance between the neighbor of the vector and the query

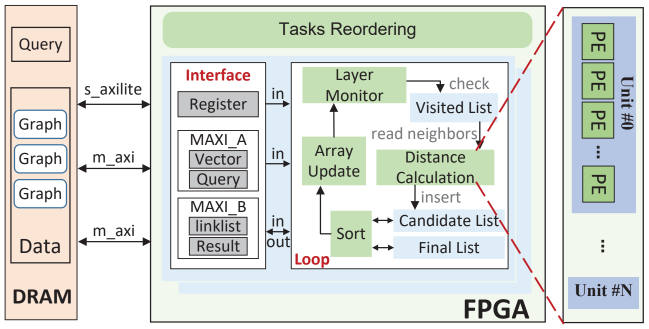


Figure 9: An overview of vector search engine

vector. Subsequently, neighbors with closer distances are inserted into both candidate and final lists. This loop continues till the candidate list is empty or no closer neighbor is found.

To improve the efficiency of FPGA, we design several optimizations for the search kernel. First, when the kernel traverses a candidate vector, it reads all neighbors in parallel, and configures separate interfaces to load the vector data and the corresponding linklist in parallel. As shown in Figure 9, we use the m_axi adapter to encapsulate queries and the vector data in MAXI-A, and encapsulate the linklist and search results in MAXI-B. Second, to save on-chip memory space, we replace the boolean array used by the original HNSW algorithm with bitmaps to implement the visited list. Third, to update both candidate and final lists efficiently, we implement the bitonic sort algorithm that is highly optimized for FPGA. Fourth, in the distance calculation module, we exploit loop unrolling and pipelining optimizations of HLS, facilitating parallel distance calculations between high-dimensional vectors with multiple *processing elements* (PE) of FPGA. Fifth, since each shard can be processed independently, we establish a data pool in the on-board DRAM and a kernel pool within the FPGA to improve the task parallelism. Once a shard is loaded into the data pool, the kernels in the kernel pool can process tasks associated with this shard in parallel. Also, the SmartSSD can load a shard to the data pool while another shard is being searched by kernels. This pooling mechanism can effectively overlap shard loading and vector searching, and thus hide the latency of data accesses.

4.5 System Implementation

We implement the prototype of SmartANNS in a heterogeneous computing architecture composed of host CPUs and SmartSSDs. The host side includes the first-tier index searching, shard pruning, and task scheduling modules programmed with C++. Since these modules are rather lightweight in terms of computing resource, we use only one CPU core to execute them, and its average utilization is even lower than 10%. The SmartSSD side contains HLS-based search kernels deployed on FPGA. We extend the functionalities of the BBANN [31] and hnswlib [36] libraries to construct the hierarchical in-

Table 2: FPGA resource utilization

	LUT	FF	BRAM	URAM	DSP
Count	323008	471442	895	50	796
Percentage	(61.80%)	(45.10%)	(91.54%)	(39.38%)	(40.56%)

dex. For the GBDT model, we exploit the LightGBM [37] framework and follow LAET [34] to configure the model parameters. The host-SmartSSD communication is achieved through the *Xilinx Run Time* (XRT) [38], which manages data preparation, kernel activation, and the collection of results from search kernels. We synthesize and implement the kernels using Xilinx Vitis 2021. Due to limited FPGA resource, we instantiate two kernels within each SmartSSD. Table 2 shows the resource utilization of the search kernels on FPGA.

5 Evaluation

In this section, we evaluate the effectiveness of SmartANNS with respect to: (*i*) end-to-end performance improvement in comparison to a state-of-the-art SmartSSD-based solution–CSDANNS [23]; (*ii*) performance scalability using multiple SmartSSDs; (*iii*) effectiveness of individual technologies; and (*iv*) comparison with SSD-based and GPU-based ANNS solutions. Our experimental results are summarized as follows:

- Compared to CSDANNS, SmartANNS achieves up to a 10.7× improvement in *Query Per Second* (QPS) under the same accuracy.
- SmartANNS demonstrates better scalability over a state-of-the-art SSD-based solution, achieving nearly linear performance improvement when the number of SmartSSDs increases.
- SmartANNS achieves higher throughput than SSD-based and GPU-based solutions.

5.1 Experiment Setup

Our experiments are conducted on a server equipped with Intel Xeon Gold 5220 2.20GHz 72-core processor and 128 GB DRAM. The server runs Ubuntu 20.04.4 LTS operating system. We use Samsung's first-generation SmartSSDs as computational storage devices, which contain Xilinx's UltraScale+ FPGA, 4 GB DDR4, and 4 TB NAND Flash. We use a number of SmartSSDs to evaluate the performance scalability for ANNS. We also use a Nvidia Tesla V100 to evaluate GPU-based ANNS solutions. The detailed hardware configurations in our experiments are shown in Table 3. We use the s-tui tool and the Vitis analyzer [39] to monitor the static and dynamic power consumption of host CPUs and SmartSSDs.

Datasets. In our experiments, we evaluate four billion-scale datasets using 10 K~100 K queries. All datasets are sourced

Table 3: Hardware platform configurations

	SmartSSD	**CPU-based**	**GPU-based**
Compute Units	Xilinx Kintex UltraScale+ KU15P FPGA	2 Intel Xeon Gold 5220 CPUs	Nvidia Tesla V100
DRAM	4 GB DDR4	128 GB DDR4	32 GB HBM
SSD	4 TB, 4 GB/s	2 TB, 4 GB/s	2 TB, 4 GB/s
Interface	PCIe 3.0×4	PCIe 3.0×4	PCIe 3.0×4

Table 4: Datasets (one billion)

Dataset	**Dimension**	**Base Size**	**Summary**	**Data Type**
SIFT1B	128	119 GB	58.51 KB	Image
SPACEV1B	100	93 GB	43.76 KB	Web Search
DEEP1B	96	358 GB	136.88 KB	Image
Turing1B	100	373 GB	134.38 KB	Web Search

from the BIGANN benchmark [40], including SIFT1B [32], SPACEV1B [41], DEEP1B [42] and Turing1B [43]. The details of these datasets are shown in Table 4.

ANNS solutions for comparison. We first compare SmartANNS with CSDANNS, which is the state-of-the-art solution for offloading ANNS to SmartSSD. The HNSW search engine of CSDANNS is also implemented with HLS. We adopt the parameters specified in CXL-ANNS [44] to construct the HNSW graph-based indices for all experiments. We also compare SmartANNS with the state-of-the-art SSD-based solution–SPANN, which searches the graph-based index composed of centroids in host memory, and then load the relevant data from SSDs. Moreover, we compare CUhnsw [25] and GGNN [26], both of them are GPU accelerated graph-based ANNS solutions. In these setups, the CPU is responsible for data movement, while the GPU handles distance calculations.

Metrics. For a fair comparison, we evaluate the ANNS performance by measuring QPS at the same level of search accuracy, which is quantified by the recall rate, i.e., *Recall@k*. This metric represents the proportion of queries in which the top-*k* nearest neighbors retrieved during the search encompasses (at least one of) the ground-truth nearest neighbors. According to the BIGANN benchmark standard, we present our experimental results at an accuracy of 90% under Recall@10, unless specified otherwise. The search accuracy is determined by the number of shards searched and the length of the search queue within each shard. When the shard pruning is enabled, we guarantee the same search accuracy by increasing the length of the search queue.

5.2 Performance

We evaluate the QPS of SmartANNS and CSDANNS under the same search accuracy. For these experiments, we use a single SmartSSD device to demonstrate the efficiency of SmartANNS against CSDANNS. For different datasets and different levels of search accuracy, SmartANNS outperforms

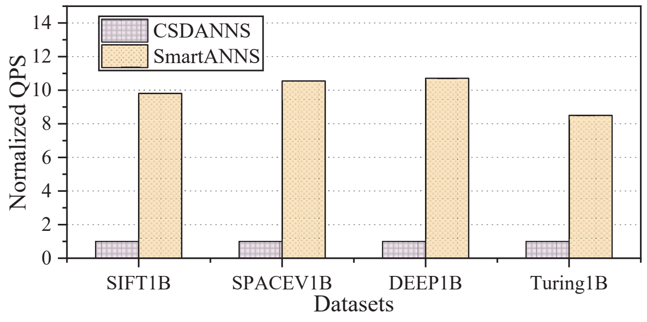


Figure 10: Throughput

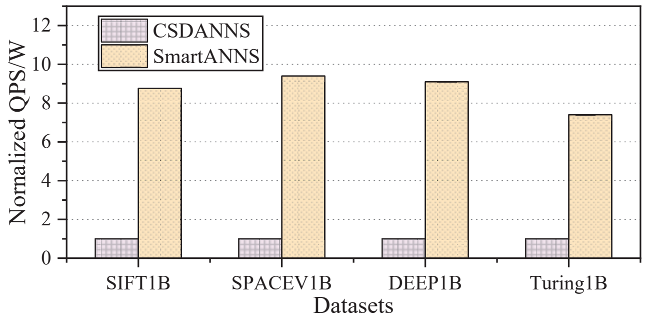


Figure 12: Energy efficiency

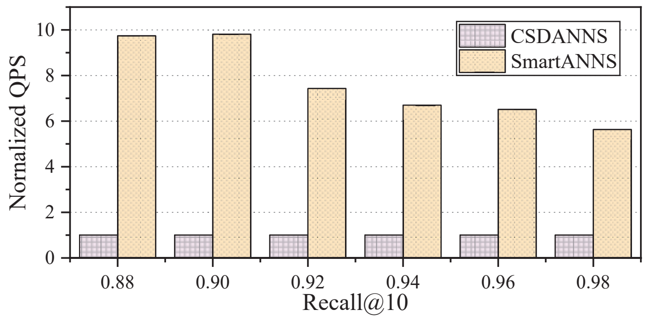


Figure 11: Recall vs. QPS (using SIFT1B)

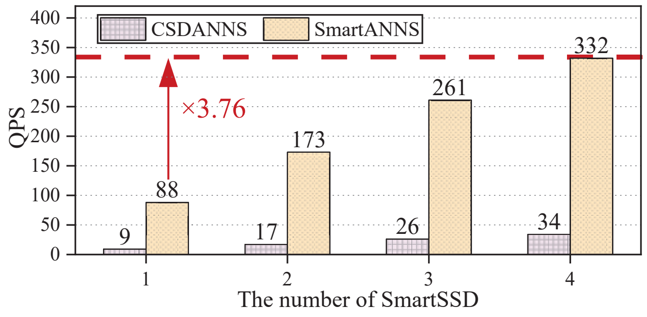


Figure 13: Performance scalability (using SIFT1B)

CSDANNS significantly in all cases.

Performance under different datasets. We compare SmartANNS with CSDANNS using SIFT1B, SPACEV1B, DEEP1B, and Turing1B datasets, respectively. Figure 10 shows the QPS at a search accuracy of 90% under Recall@10, all normalized to CSDANNS. SmartANNS improves the QPS by 8.5 to 10.7 × compared with CSDANNS, demonstrating a notably performance improvement for all billion-scale datasets with different data sizes, types, characteristics, and application scenarios. This performance improvement mainly stems from the hierarchical indexing and the promising NDP function of SmartSSDs. Since SmartANNS can significantly reduce the search space of shards on SmartSSDs by using the first-tier indices in the host, it avoids most unnecessary data movement between the NAND Flush and the FPGA DRAM, and also eliminates unnecessary distance computations on the FPGA by using the shard pruning strategy. SmartANNS shows a slight decrease of performance speedup for Turing1B dataset. The reason is that the out-of-distribution characteristic of this dataset has a negative effect on the vector clustering. Thus, SmartANNS should search more shards to achieve the desired accuracy level.

Performance vs. accuracy. To demonstrate the superiority of SmartANNS over CSDANNS at different accuracy levels, we change the search accuracy from 88% to 98% using the SIFT1B dataset. Figure 11 shows the normalized QPS under different levels of accuracy. SmartANNS achieves about 5.6-9.8 × performance improvement compared with CSDANNS. We find that the performance speedup declines when the accuracy level becomes higher. The root cause is that higher accuracy levels imply a significant expansion of the search space. SmartANNS should retrieve more shards to find increasingly-elusive nearest neighbors, losing more opportunities to stop the search process.

Energy efficiency. To evaluate the energy efficiency of SmartSSD based ANNS solutions, we measure the dynamic power consumption of host CPUs and SmartSSDs when ANNS tasks are being processed. Figure 12 shows the QPS per watt of SmartANNS, all normalized to CSDANNS. SmartANNS achieves about 8.4-9.4 × higher energy efficiency than CSDANNS. Since a SmartSSD's dynamic power consumption remains stable (about 25 watt) when the FPGA is working, the energy efficiency of SmartANNS over CSDANNS mainly stems from the significant performance improvement in QPS.

5.3 Scalability

We evaluate the scalability of SmartANNS using multiple SmartSSDs with the SIFT1B dataset. The standard U.2 form factor of the SmartSSD allows for simple scaling of devices. Figure 13 shows the QPS of SmartANNS when the number of SmartSSDs increases from 1 to 4. When a single SmartSSD is used, the QPS of SmartANNS is 88. When SmartANNS uses four SmartSSDs, its QPS increases to 332, achieving 3.76 × performance improvement. Thus, SmartANNS achieves near-linear performance scalability with the increase of SmartSSDs. Since our task scheduling mechanism can evenly distribute

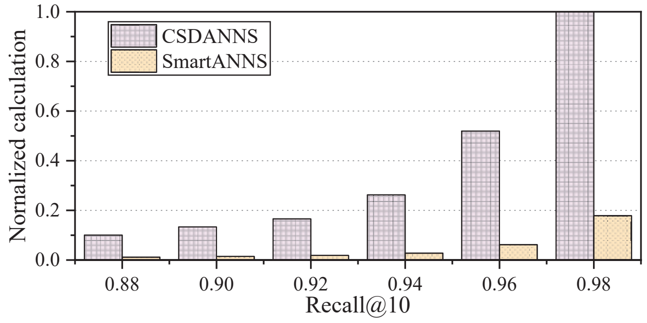


Figure 14: Distance calculation reduction, all normalized to CSDANNS with an accuracy of 0.98

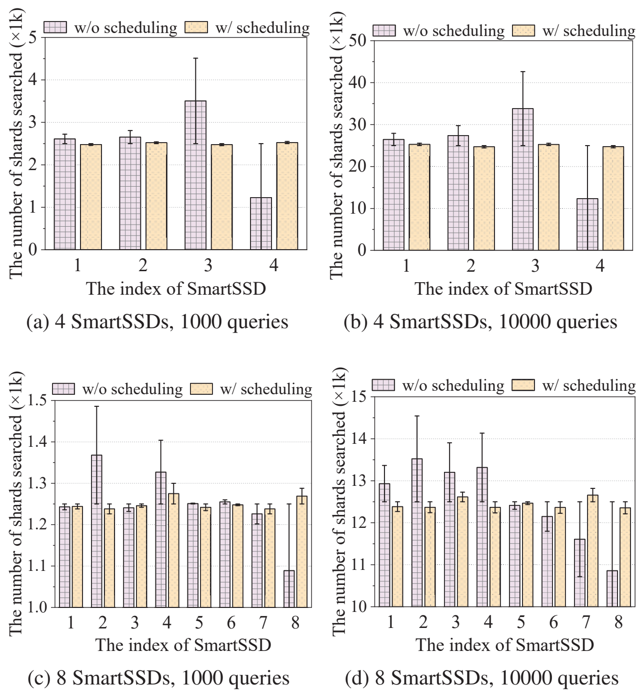


Figure 15: The workload distribution in each SmartSSD

tasks across multiple SmartSSDs, and thus minimizes the wait time caused by overloaded SmartSSDs.

5.4 Effectiveness of Individual Technologies

In this subsection, we validate the effectiveness of individual technologies in SmartANNS and evaluate their contributions to the overall performance improvement. Using the SIFT1B dataset, we first conduct experiments to assess the effect of hierarchical indexing, task scheduling, and shard pruning separately, and then we incrementally add these techniques to evaluate their impacts on the overall performance.

Effectiveness of hierarchical indexing. Figure 14 shows the normalized numbers of distance calculations that are required by CSDANNS and SmartANNS. On average, SmartANNS necessitates 80% fewer distance calculations compared with CSDANNS. The reason is that SmartANNS exploits the host CPU as a global coordinator to narrow the search space based on the hierarchical indexing, thus avoiding costly global searches. As a result, SmartANNS significantly reduces the computational load within each SmartSSD.

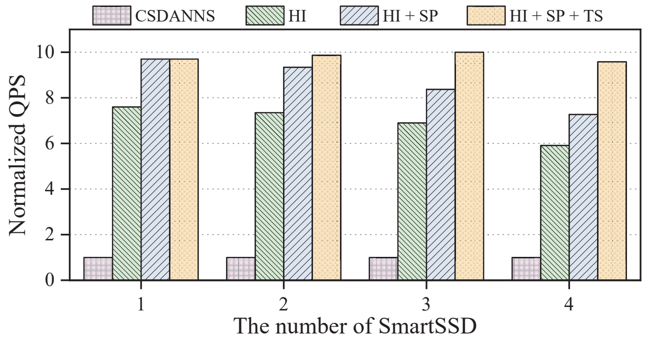


Figure 16: Performance improvement due to different technologies. HI, SP and TS denote hierarchical indexing, shard pruning, and task scheduling, respectively.

Effectiveness of task scheduling. We record the number of touched shards in each SmartSSD when multiple SmartSSDs are used to serve a batch of queries. Figure 15 shows the distribution of workload (with a deviation from the average) on each SmartSSD when we enable and disable the task scheduling mechanism. Without task scheduling, the workload across SmartSSD exhibits a significant imbalance for all configurations. This unbalanced distribution usually causes overloaded and underutilized SmartSSDs during parallel processing, and thus has a negative impact on the performance scalability. When our task scheduling mechanism is enabled, workloads are distributed to different SmartSSDs evenly. Thus, multiple SmartSSDs can achieve scalable performance via load balancing.

Effectiveness of shard pruning. We count the average number of shards searched per query using our learning-based shard pruning scheme. For a comparison, we also evaluate a simple method that searches a fixed number of shards for each query. Under a constraint of the same level of accuracy, our learning-based approach and the simple method have to search 9 and 13 shards from a total of 114 shards on average, respectively. By searching fewer shards, our shard pruning scheme substantially reduces the cost of data accessing and distance calculations, especially for batch queries. Thus, it can achieve considerable performance improvement, as shown in Figure 16.

The performance breakdown of individual technologies. To evaluate the impact of each individual technique on the overall performance, we adopt these techniques incrementally. As shown in Figure 16, the hierarchical indexing significantly improves the QPS by 7.6 × compared with CSDANNS for a single SmartSSD. However, when the number of SmartSSD

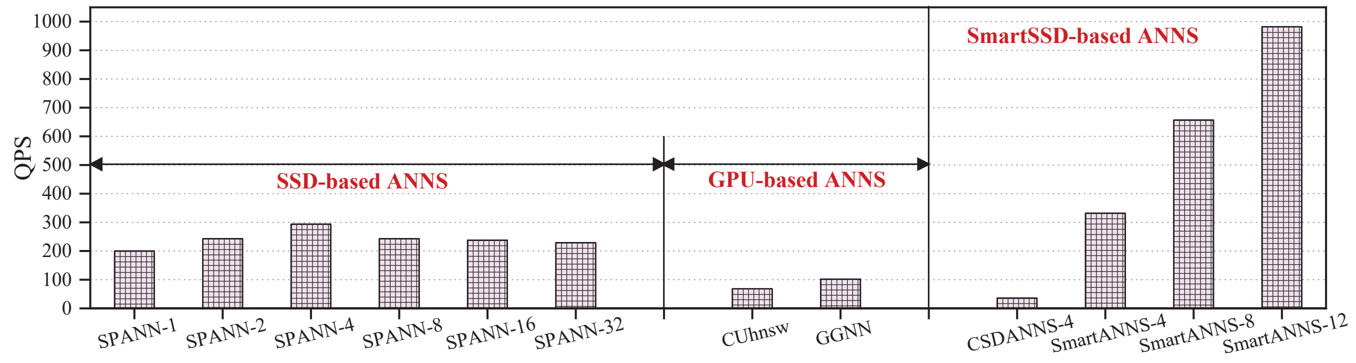


Figure 17: Comparison with state-of-the-art SSD-based ANNS system–SPANN and GPU accelerated solutions–CUhnsw, GGNN. SPANN-*n* denotes SPANN using *n* threads, and CSDANNS-*n*/SmartANNS-*n* denote CSDANNS/SmartANNS using *n* SmartSSDs.

increases, the performance gains achieved by the hierarchical indexing tend to decline. Similarly, the shard pruning scheme exhibits a similar trend of performance improvement. In contrast, the task scheduling scheme delivers more performance gains than other schemes when the number of SmartSSDs increases, because it can significantly mitigate the load imbalance among multiple SmartSSDs. This implies the load balancing has a significant impact on the performance scalability of SmartANNS when a large number of SmartSSDs are deployed to achieve high throughput for large-scale datasets.

5.5 Comparison with SSD/GPU-based ANNS

We compare SmartANNS with a SSD-based ANNS solution–SPANN, and two GPU-based ANNS solutions–CUhnsw and GGNN using the SIFT1B dataset. As shown in Figure 17, the QPS of SmartANNS using four SmartSSDs is much higher than that of SPANN, CUhnsw, and GGNN. Moreover, the QPS of SPANN approaches saturation when the number of threads increases to 4. The reason is attributed to the inherent constraint of PCIe bus in the traditional computing architecture, as illustrated in Figure 3. Since the graph-based index and all I/O requests initiated by queries should traverse the same PCIe bus, they cause intensive bandwidth contention on the host PCIe. Thus, the limited PCIe bandwidth becomes a performance bottleneck, and I/O operations account for about 67% of total execution time in SPANN. In contrast, SmartANNS benefits from the internal PCIe bus of each SmartSSD and proposed search optimizations, enabling its performance to scale almost linearly with the number of SmartSSDs. Due to the limited GPU memory, the billion-scale datasets and their corresponding graph-based indexes cannot be loaded into the GPU memory entirely. As a result, the data is partitioned and loaded from SSDs into the GPU sequentially for queries. This extensive data movement and frequent memory swapping result in poor performance for these two GPU-based ANNS solutions.

Certainly, as a new storage device, a SmartSSD (about $2000) is more expensive than a traditional SSD (about $400). However, SmartSSDs can eliminate the performance bottleneck of the host PCIe, and achieve near-linear scalability by using their internal PCIe buses. Since ANNS is often cooperatively used with recommendation systems and LLMs, which often incur intensive data movement between host memory and GPUs through PCIe. To eliminate the PCIe bandwidth contention, we may have to use more expensive dedicated servers to deploy the ANNS service. However, its scalability is still constrained to the host PCIe bandwidth. With the growing prevalence of SmartSSDs and their anticipated decrease in cost, ANNS is expected to gain tremendous benefits from cost-efficiency SmartSSDs in the future.

6 Related Work

We present the most related work in the following categories.

Memory extension for ANNS. For billion-scale datasets, existing systems often struggle to accommodate these datasets entirely in memory. A number of SSD-based approaches [19, 20, 30, 31] design different index structures for main memory and disks to index and query efficiently, and achieve a good balance between the query latency and the accuracy. Although these proposals can alleviate the memory resource requirement, ANNS queries often incur intensive I/O requests to disks, resulting in extensive data swapping between disks and main memory. HM-ANN [45] leverages a hybrid memory system including DRAM and Intel Optane persistent memory to extend the memory capacity for ANNS. It stores simplified graphs and detailed graphs in DRAM and Optane memory, respectively, and thus reduces the number of accesses to Optane memory during the ANNS process. CXL-ANNS [44] expands host memory based on *Compute Express Link* (CXL) and designs caching and prefetching techniques for the CXL memory, minimizing the impact of CXL memory's high latency on the system performance. However, these memory extension approaches substantially increases the hardware cost in data centers. In contrast, SmartANNS can significantly reduce the memory resource requirement by offloading ANNS queries to SmartSSDs, and achieve near-linear performance scalability for large-scale datasets using multiple SmartSSDs.

Computation acceleration for ANNS. A few recent studies [46–49] exploit GPUs and FPGAs to accelerate the vector search based on the IVF-PQ algorithm [46]. Since this algorithm often incurs intensive distance calculations, the high-parallel computing capabilities of GPUs and FPGAs can be fully exploited to speed up the vector search. Moreover, a few efforts have been made to accelerate graph-based ANNS algorithms using GPUs [26,50,51] and FPGAs [52]. However, most of these approaches can only achieve promising performance for small-scale datasets, and cannot scale to large datasets due to limited memory capacity in GPUs and FPGAs. Unlike these approaches that focus on computation acceleration using GPUs and high-end FPGAs, SmartANNS fully exploits the near-data-processing paradigm in a "CPUs + SmartSSDs" cooperative architecture to optimize ANNS services for billion-scale datasets.

In-Storage processing for ANNS. To provide cost-efficient and energy-efficient ANNS services, a few recent proposals offload ANNS to computational storage devices based on the NDP paradigm. Vstore [21] is a graph-based ANNS accelerator integrated into SSDs. It fully exploits the SSD's multiple-channel feature to search the graph-based index in parallel, and also explores data reuse between queries to mitigate the cost of data accesses and distance calculations. Pyramid [53] exploits a *near-memory-computing* (NMC) accelerator and an *in-storage-processing* (ISP) accelerator for ANNS acceleration. However, these proposals are based on architectural simulations and none of them have considered the scalability issue when using multiple computational storage devices. In contrast, SmartANNS is implemented on real commercial SmartSSDs and achieves near-linear performance scalability by using multiple SmartSSDs.

CSDANNS [23] is so far the first work that demonstrates the feasibility of offloading ANNS to computational storage devices. It sequentially splits the dataset and constructs an HNSW index for each partition. This simple approach necessitates each query to scan all partitions, and thus results in high computational cost on SmartSSDs. As a result, CSDANNS achieves sub-optimal performance. SmartANNS reduces the computational cost on SmartSSDs based on a hierarchical indexing methodology, and addresses the challenges of adopting this method to SmartSSDs in a software-hardware cooperative manner. Moreover, SmartANNS exploits data reusing, load balancing, and learning-based shard pruning to achieve scalable ANNS performance on multiple SmartSSDs.

7 Conclusion

In this work, we present SmartANNS, a hardware/software co-design architecture using SmartSSDs to support the large-scale ANNS service. SmartANNS is built on a hierarchical indexing approach, addressing a series of challenges on the "host CPU + SmartSSDs" platform through several optimizations that range from architecture, data layout, and algorithm levels. Compared with an existing state-of-the-art solution–CSDANNS, SmartANNS significantly improves the system's QPS. Moreover, the performance of SmartANNS increases almost linearly with the number of SmartSSDs, implying that SmartANNS can scale well for extremely large datasets.

Acknowledgments

We appreciate our shepherd and anonymous reviewers' constructive comments for improving the quality of this paper. This work is supported jointly by National Key Research and Development Program of China under grant No. 2022YFB4500303, National Natural Science Foundation of China (NSFC) under grants No.62072198, 62332011, 62302178, and Natural Science Foundation of Hubei Province under grant No.2021CFA037.

References

[1] Yukihiro Tagami. AnnexML: Approximate Nearest Neighbor Search for Extreme Multi-Label Classification. In *Proceedings of the ACM KDD International Conference on Knowledge Discovery and Data Mining*, page 455–464, 2017.

[2] Jianjin Zhang, Zheng Liu, Weihao Han, Shitao Xiao, Ruicheng Zheng, Yingxia Shao, Hao Sun, Hanqing Zhu, Premkumar Srinivasan, Weiwei Deng, Qi Zhang, and Xing Xie. Uni-retriever: Towards learning the unified embedding based retriever in bing sponsored search. In *Proceedings of the ACM KDD International Conference on Knowledge Discovery and Data Mining*, pages 4493–4501, 2022.

[3] Yanhao Zhang, Pan Pan, Yun Zheng, Kang Zhao, Yingya Zhang, Xiaofeng Ren, and Rong Jin. Visual Search at Alibaba. In *Proceedings of the ACM KDD International Conference on Knowledge Discovery and Data Mining*, pages 993–1001, 2018.

[4] 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 ACM KDD International Conference on Knowledge Discovery and Data Mining*, pages 2553–2561, 2020.

[5] Jiaxi Cui, Zongjian Li, Yang Yan, Bohua Chen, and Li Yuan. ChatLaw: Open-source legal large language model with integrated external knowledge bases. arxiv:2306.16092, 2023.

[6] Cheonsu Jeong. A Study on the Implementation of Generative AI Services Using an Enterprise Data-Based LLM Application Architecture. arxiv:2309.01105, 2023.

[7] Cheng Wen, Xianghui Sun, Shuaijiang Zhao, Xiaoquan Fang, Liangyu Chen, and Wei Zou. Chathome: Development and evaluation of a domain-specific language model for home renovation. arxiv:2307.15290, 2023.

[8] Xuanyu Zhang, Qing Yang, and Dongliang Xu. Xuanyuan 2.0: A large chinese financial chat model with hundreds of billions parameters. arxiv:2305.12002, 2023.

[9] Yury A. Malkov and Dmitry A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 42(4):824–836, 2020.

[10] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. Fast approximate nearest neighbor search with the navigating spreading-out graph. *Proceedings of the VLDB Endowment*, 12(5):461–474, 2019.

[11] Ben Harwood and Tom Drummond. FANNG: fast approximate nearest neighbour graphs. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5713–5722, 2016.

[12] Long Gong, Huayi Wang, Mitsunori Ogihara, and Jun Xu. iDEC: Indexable Distance Estimating Codes for Approximate Nearest Neighbor Search. *Proceedings of the VLDB Endowment*, 13(9):1483–1497, 2020.

[13] Qiang Huang, Jianlin Feng, Yikai Zhang, Qiong Fang, and Wilfred Ng. Query-aware locality-sensitive hashing for approximate nearest neighbor search. *Proceedings of the VLDB Endowment*, 9(1):1–12, 2015.

[14] Chanop Silpa-Anan and Richard I. Hartley. Optimised kd-trees for fast image descriptor matching. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8, 2008.

[15] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(1):117–128, 2011.

[16] Zhibin Pan, Liangzhuang Wang, Yang Wang, and Yuchen Liu. Product quantization with dual codebooks for approximate nearest neighbor search. *Neurocomputing*, 401:59–68, 2020.

[17] Artem Babenko and Victor Lempitsky. The inverted multi-index. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3069–3076, 2012.

[18] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. Fast approximate nearest neighbor search with the navigating spreading-out graph. *Proceedings of the VLDB Endowment*, 12(5):461–474, 2019.

[19] Suhas Jayaram Subramanya, Devvrit, Rohan Kadekodi, Ravishankar Krishaswamy, and Harsha Vardhan Simhadri. DiskANN: Fast accurate billion-point nearest neighbor search on a single node. In *Proceedings of the International Conference on Neural Information Processing Systems*, 2019.

[20] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. SPANN: highly-efficient billion-scale approximate nearest neighborhood search. In *Proceedings of the International Conference on Neural Information Processing Systems*, pages 5199–5212, 2021.

[21] Shengwen Liang, Ying Wang, Ziming Yuan, Cheng Liu, Huawei Li, and Xiaowei Li. Vstore: in-storage graph based vector search accelerator. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 997–1002, 2022.

[22] SmartSSD - Samsung Semiconductor. `https://semiconductor.samsung.com/ssd/smart-ssd/`.

[23] Ji-Hoon Kim, Yeo-Reum Park, Jaeyoung Do, Soo-Young Ji, and Joo-Young Kim. Accelerating large-scale graph-based nearest neighbor search on a computational storage platform. *IEEE Transactions on Computers*, 72(1):278–290, 2023.

[24] Fuping Niu, Jianhui Yue, Jiangqiu Shen, Xiaofei Liao, and Hai Jin. FlashGNN: An In-SSD Accelerator for GNN Training. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture*, pages 361–378, 2024.

[25] CUhnsw. `https://github.com/js1010/cuhnsw`.

[26] Fabian Groh, Lukas Ruppert, Patrick Wieschollek, and Hendrik P. A. Lensch. GGNN: Graph-Based GPU Nearest Neighbor Search. *IEEE Transactions on Big Data*, 9(1):267–279, 2023.

[27] Mengzhao Wang, Xiaoliang Xu, Qiang Yue, and Yuxiang Wang. A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search. *Proceedings of the VLDB Endowment*, 14(11):1964–1978, 2021.

[28] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the ACM Thirtieth Annual Symposium on Theory of Computing*, page 604–613, 1998.

[29] Computaional Storage | SNIA. https://semiconductor.samsung.com/ssd/smart-ssd/.

[30] Minjia Zhang and Yuxiong He. GRIP: Multi-store capacity-optimized high-performance nearest neighbor search for vector search engine. In *Proceedings of the ACM International Conference on Information and Knowledge Management*, page 1673–1682, 2019.

[31] Harsha Vardhan Simhadri, George Williams, Martin Aumüller, Matthijs Douze, Artem Babenko, Dmitry Baranchuk, Qi Chen, Lucas Hosseini, Ravishankar Krishnaswamny, Gopal Srinivasa, Suhas Jayaram Subramanya, and Jingdong Wang. Results of the NeurIPS'21 Challenge on Billion-Scale Approximate Nearest Neighbor Search. In *Proceedings of the NeurIPS 2021 Competitions and Demonstrations Track*, volume 176, pages 177–189, 2022.

[32] Hervé Jégou, Romain Tavenard, Matthijs Douze, and Laurent Amsaleg. Searching in one billion vectors: Rerank with source coding. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, pages 861–864, 2011.

[33] Jerome H. Friedman. Greedy function approximation: A gradient boosting machine. *The Annals of Statistics*, 29(5):1189 – 1232, 2001.

[34] Conglong Li, Minjia Zhang, David G. Andersen, and Yuxiong He. Improving approximate nearest neighbor search through learned adaptive early termination. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 2539–2554, 2020.

[35] Pengcheng Zhang, Bin Yao, Chao Gao, Bin Wu, Xiao He, Feifei Li, Yuanfei Lu, Chaoqun Zhan, and Feilong Tang. Learning-based query optimization for multi-probe approximate nearest neighbor search. *The VLDB Journal*, 32(3):623–645, 2022.

[36] HNSWlib. https://github.com/nmslib/hnswlib.

[37] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. LightGBM: A highly efficient gradient boosting decision tree. In *Proceedings of the International Conference on Neural Information Processing Systems*, pages 3146–3154, 2017.

[38] XRT - Xilinx Runtime Library. https://www.xilinx.com/products/design-tools/vitis/xrt.html.

[39] Xilinx - Vitis Unified Software Platform. https://www.xilinx.com/products/design-tools/vitis.html.

[40] BigANN benchmark. https://big-ann-benchmarks.com/neurips21.html.

[41] Microsoft. 2021. SpaceV1B. https://github.com/microsoft/SPTAG/tree/main/datasets/SPACEV1B.

[42] Yandex Billion-Scale Datasets. https://research.yandex.com/datasets/biganns.

[43] Hongfei Zhang, Xia Song, Chenyan Xiong, Corby Rosset, Paul Bennett, Nick Craswell, and Saurabh Tiwary. Generic intent representation in web search. In *Proceedings of the International ACM SIGIR Conference on Research and Development in Information Retrieval*, page 65–74, 2019.

[44] Junhyeok Jang, Hanjin Choi, Hanyeoreum Bae, Seungjun Lee, Miryeong Kwon, and Myoungsoo Jung. CXL-ANNS: software-hardware collaborative memory disaggregation and computation for billion-scale approximate nearest neighbor search. In *Proceedings of the USENIX Annual Technical Conference*, pages 585–600, 2023.

[45] Jie Ren, Minjia Zhang, and Dong Li. HM-ANN: efficient billion-point nearest neighbor search on heterogeneous memory. In *Proceedings of the International Conference on Neural Information Processing Systems*, 2020.

[46] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with gpus. *IEEE Transactions on Big Data*, 7(3):535–547, 2021.

[47] Wenqi Jiang, Shigang Li, Yu Zhu, Johannes De Fine Licht, Zhenhao He, Runbin Shi, Cedric Renggli, Shuai Zhang, Theodoros Rekatsinas, Torsten Hoefler, and Gustavo Alonso. Co-design hardware and algorithm for vector search. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2023.

[48] Yejin Lee, Hyunji Choi, Sunhong Min, Hyunseung Lee, Sangwon Beak, Dawoon Jeong, Jae W. Lee, and Tae Jun Ham. ANNA: specialized architecture for approximate nearest neighbor search. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture*, pages 169–183, 2022.

[49] 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*, pages 4924–4932, 2018.

[50] Yuanhang Yu, Dong Wen, Ying Zhang, Lu Qin, Wenjie Zhang, and Xuemin Lin. GPU-accelerated proximity graph approximate nearest neighbor search and construction. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 552–564, 2022.

[51] Weijie Zhao, Shulong Tan, and Ping Li. SONG: approximate nearest neighbor search on GPU. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 1033–1044, 2020.

[52] Shulin Zeng, Zhenhua Zhu, Jun Liu, Haoyu Zhang, Guohao Dai, Zixuan Zhou, Shuangchen Li, Xuefei Ning, Yuan Xie, Huazhong Yang, and Yu Wang. DF-GAS: a distributed FPGA-as-a-service architecture towards billion-scale graph-based approximate nearest neighbor search. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*, page 283–296, 2023.

[53] Zhenhua Zhu, Jun Liu, Guohao Dai, Shulin Zeng, Bing Li, Huazhong Yang, and Yu Wang. Processing-in-hierarchical-memory architecture for billion-scale approximate nearest neighbor search. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 1–6, 2023.