Published: 05 December 2025

RESEARCH-ARTICLE

High-Throughput, Cost-Effective Billion-Scale Vector Search with a Single GPU

HAODI JIANG, Tsinghua University, Beijing, China

HAO GUO, Tsinghua University, Beijing, China

MINHUI XIE, Renmin University of China, Beijing, China

JIWU SHU, Tsinghua University, Beijing, China

YOUYOU LU, Tsinghua University, Beijing, China

High-Throughput, Cost-Effective Billion-Scale Vector Search with a Single GPU

HAODI JIANG, Tsinghua University, China
HAO GUO, Tsinghua University, China
MINHUI XIE, Renmin University of China, China
JIWU SHU, Tsinghua University, China
YOUYOU LU*, Tsinghua University, China

Approximate nearest neighbor search (ANNS) is broadly adopted in numerous scenarios. Real-world applications seek efficient ways to search billion-scale vectors in high throughput. On-SSD graph-based ANNS systems have the opportunity to achieve this goal, but the limited CPU computing power becomes a bottleneck. In this paper, we propose a GPU-centric, CPU-assisted ANNS architecture and design GustANN, a billion-scale graph-based vector search system for high throughput and cost-effectiveness. We achieve these goals with three techniques: (1) memory-efficient GPU kernels optimized to minimize the GPU memory usage in the graph search, which allows higher concurrency for GPU and SSD; (2) CPU-assisted transfer to address the PCIe bandwidth bottleneck on the GPU-side; (3) pivot search for inter-SSD load balancing. Compared to existing ANNS systems, GustANN achieves at least 2.50× higher throughput, and is 2.62× more cost-effective (measured in \$/QPS).

CCS Concepts: • **Information systems** → **Storage management**; *Information retrieval*; • **Computing methodologies** → *Parallel algorithms.*

Additional Key Words and Phrases: Approximate nearest neighbor search; GPU; SSD

ACM Reference Format:
Haodi Jiang, Hao Guo, Minhui Xie, Jiwu Shu, and Youyou Lu. 2025. High-Throughput, Cost-Effective Billion-Scale Vector Search with a Single GPU . *Proc. ACM Manag. Data* 3, 6 (SIGMOD), Article 334 (December 2025), 27 pages. https://doi.org/10.1145/3769799

1 Introduction

Approximate nearest neighbor search (ANNS) is widely used in various scenarios, such as web search [29, 36], recommendation systems [45], and retrieval augmented generation (RAG) [4, 35]. Given a query vector, ANNS finds its k approximate nearest neighbors in an ANNS index, which is pre-built using the vector dataset. In real-world systems, vector datasets can contain billions of vectors (e.g., in Microsoft [43] and Alibaba [21]), and vector search throughput can reach ~100K QPS (e.g., in Google search [3]). Thus, both academia and industry are actively pursuing efficient methods for billion-scale vector search with high throughput [18, 19, 21, 30, 43, 48].

In this paper, we explore the possibility of high-throughput billion-scale vector search in a *cost-effective* (i.e., fewer \$/QPS) manner. To this end, we focus on on-SSD graph-based ANNS indexes [30, 49], because of two reasons: first, billion-scale vectors require terabytes (TBs) of

*Youyou Lu is the corresponding author (luyouyou@tsinghua.edu.cn).

Authors' Contact Information: Haodi Jiang, Tsinghua University, Beijing, China; Hao Guo, Tsinghua University, Beijing, China; Minhui Xie, Renmin University of China, Beijing, China; Jiwu Shu, Tsinghua University, Beijing, China; Youyou Lu, Tsinghua University, Beijing, China.

ACM 2836-6573/2025/12-ART334
https://doi.org/10.1145/3769799

space [21, 43], which often exceeds the memory capacity of a single server. It is more cost-effective to use solid-state drives (SSDs) than memory, because they offer a lower cost per gigabyte and larger capacity. Second, graph-based ANNS indexes are more cost-effective than other ANNS indexes (e.g., cluster-based ones [17, 18, 47]) for high-throughput ANNS. They organize vectors as a fine-grained graph, thus achieving superior throughput under the same hardware configuration [21, 49].

However, we find that existing CPU on-SSD graph-based ANNS indexes (named *CPU-only architecture*) fail to achieve high throughput, due to CPU bottlenecks. With two CPUs, a typical CPU-based system [30] can only achieve 6 GB/s of SSD bandwidth (60% of a single SSD's maximum bandwidth), insufficient to saturate 6 – 24 SSDs in a typical storage server [9, 14]. This is because, in high-throughput scenarios, the overall computing power of the CPU is insufficient. Specifically, we find that even when utilizing the theoretical maximum computing power of the CPU, such an ANNS system can only saturate the bandwidth of 3.6 SSDs (see §3.1 for more details).

To increase throughput, a possible way is to leverage the computing power of GPU. However, existing GPU-based ANNS show limitations, which can be categorized into:

- *CPU-centric, GPU-assisted architecture* [47, 56], which uses CPU for graph traversal (by accessing SSDs) and offloads vector computation (e.g., distance computation) to GPU, still suffers from the performance bottleneck of CPU search for graph-based ANNS.
- *GPU-only architecture* [41, 48, 57] solves the aforementioned problem by offloading both graph traversal and vector computing to GPU and enjoys massive parallelism. However, it still suffers from the limited GPU memory and inbound data transfer bandwidth, causing only 1.7 SSDs saturated. First, a large batch size is required to saturate the performance of both GPU and SSDs. However, existing systems [30, 48] require a large amount of GPU memory to store query-related data structures like visited table (see §4.2.1), which limits the maximum achievable batch size to only 50%. Second, GPU-only architecture initiates SSD reads directly from GPU kernels (referred to as *GPU direct-access*[42]), which causes GPU to utilize only 24GB/s of SSD bandwidth (~2.4 SSDs saturated) via PCIe 4.0 bus in the ideal scenario.

To address these issues of existing architectures, we propose GustANN[1], a high-throughput and cost-effective graph-based ANNS system. It follows a *GPU-centric, CPU-assisted architecture*, which uses GPU for graph traversal but leverages CPU to assist data transfer. The performance of this architecture is not limited by CPU search. It is also possible to mitigate the inbound bandwidth bottleneck by CPU-side I/O management.

We start with optimizing the memory footprint of GPU-based search to support larger batch sizes and ensure high concurrency for both the GPU and SSD. Specifically, we design a *memory-efficient graph traversal GPU kernel* by trading redundant computation for less memory space. While typical CPU-based ANNS algorithms [21, 30, 40] rely on space-consuming data structures that trade memory usage to eliminate redundant computation, it is inapplicable on GPUs. Our key observation is that the SIMT architecture of GPUs can natively tolerate some degree of redundant computation. We design a GPU-friendly parallel algorithm for the graph search to reduce the overhead for redundant computation. Instead of maintaining a space-consuming data structure (e.g., a visited table) to avoid redundant computation, this algorithm first applies all computation tasks (which may contain redundant computation) in parallel. Then it performs a parallel deduplication procedure to ensure the correctness of the algorithm.

Second, we design *CPU-assisted transfer* to mitigate the inbound bandwidth bottleneck of the GPU-side PCIe link. GPU direct-access (such as BaM [42]) reads data in page granularity. However, we observe that, for current real-world datasets [18, 31, 44, 52], a graph node is typically smaller than one SSD page, namely less than 1/4 of an SSD page. Therefore, instead of directly accessing

[1]GustANN is open-sourced at https://github.com/thustorage/GustANN.

data, we should selectively transfer data to the GPU. To achieve this, GustANN uses CPU to assist the selection and transfer. Specifically, when accessing a graph node, we first transfer its corresponding page from the SSD to CPU DRAM, which exploits the aggregated SSD bandwidth. Then, we only transfer the target node from DRAM to GPU, which saves the limited GPU inbound bandwidth.

Third, we also observe that placing the graph index across multiple SSDs introduces significant load imbalance in tail latency (2.17× divergence across SSDs for 99% latency), hindering ANNS performance. We propose *pivot search* to tackle this load imbalance. We identify that the load imbalance issue originates from the single-entry design of the existing graph index. The single-entry design generates *micro-scale load imbalance*. By the nature of single-entry design, the first few traversal rounds only visit a nearby region around the entry node. However, these nodes in the region are not well-balanced across these SSDs. Such an imbalance is magnified as all queries need to visit this region in the single-entry design. To tackle the load imbalance, we employ the pivot search to scatter queries across different regions of the graph. This way can avoid the micro-scale load imbalance of a single region. To scatter the queries, the pivot search leverages a small pivot graph on the GPU memory. The pivot graph is built upon a random subset of the whole vector dataset, representing different regions of the graph. Vector search starts with the pivot search, which finds the nearest neighbor node to the query vector in the pivot graph, then continues with an on-disk search starting from this node.

We evaluate GustANN with different public datasets, and compare it against three CPU-based systems (DiskANN [30], SPANN [18], Starling [49]) and two GPU-based systems (BANG [48], RUMMY [56]). Using a single GPU and 6 SSDs, GustANN achieves 247K billion-scale vector search per second and a maximum of 86.7% SSD utilization. Compared to CPU-based systems, GustANN has 6.98× higher throughput on billion-scale datasets. Compared to GPU-based systems, GustANN has at least 2.50× higher throughput. In addition, GustANN is at least 2.62× more cost-effective (measured in $/QPS) than other systems.

To summarize, we make the following contributions:

- We show that the computing power of CPU is the bottleneck to realize high-throughput on-SSD graph-based ANNS.
- We identify the challenges for utilizing GPU and SSDs for high-throughput and cost-effective graph-based ANNS.
- We design GustANN, which uses GPU and multiple SSDs to achieve high-throughput and cost-effective ANNS. It has three main designs: efficient GPU kernels for graph search, CPU-assisted transfer, and pivot search.
- We implement and evaluate GustANN, proving its efficiency for high-throughput ANNS and its cost-effectiveness against other ANNS systems.

2 Background

2.1 Approximate Nearest Neighbor Search

Vector search is a critical step in retrieval augmented generation (RAG) [35], which empowers the ability of large language models (LLM) with domain-specific knowledge. In RAG, knowledge, essentially multi-modal data (e.g., text, image), is embedded and stored as high-dimensional vectors. A vector search in the RAG uses vectors that embed questions about the knowledge as input, and the output vectors (considered as related knowledge) will be sent to LLM for further analysis. Other fields, such as web search [29, 36], recommendation systems [45], and scientific computing [58], also adopt vector search.

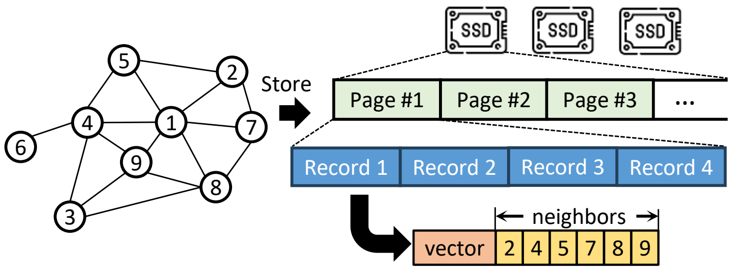

Fig. 1. Layout of an on-disk graph. Each SSD page stores multiple records representing nodes on the graph. Each record consists of the vector and the neighbors of the node.

The vector search is realized through the k-nearest neighbor search (KNNS), which finds the top-k nearest vectors in a dataset for a set of query vectors. In practice, approximate nearest neighbor search (ANNS), which allows errors in KNNS, is adopted because KNNS algorithms for low dimensions degrade to brute force in high-dimensional scenarios [20]. The accuracy of the ANNS is typically measured as *recall*. Let the set of the exact k-nearest vectors be X, and the vectors returned by an ANNS algorithm be Y, where $|X| = |Y| = k$. Then recall@k is defined as $\frac{|X \cap Y|}{k}$, meaning the proportion of exact k-nearest vectors the algorithm finds.

Graph-based ANNS outperforms in search correctness and performance compared to other ANNS methods [16, 21, 28, 39, 40]. In graph-based ANNS, the vectors in the dataset are organized as a directed graph where each node of the graph represents a vector, and the edges of the graph represent the neighboring relationship of the vectors. The search is performed through a graph traversal, which will be described in detail in §2.2.

2.2 On-Disk Graph-Based ANNS

Real-world applications typically include billion-scale vectors [21, 43], requiring hundreds of GB to several TB of storage for the graph index. This often exceeds the DRAM capacity of a server. Therefore, these indexes are often stored in SSDs, due to their larger capacity and cost-effectiveness compared to DRAM [27, 30, 49].

Figure 1 shows the on-disk graph layout. The graph is stored as multiple *records* on SSD pages, each representing one graph *node*. A record includes the full-precision *vector* and a *neighbor list*, which contains the IDs of *neighbors*. The number of neighbors in a node is limited to a global constant R, ensuring fixed-size records. No record overlaps between two pages, making sure each record can be fetched in one SSD page read.

Algorithm 1 illustrates the on-disk search procedure. The algorithm is composed of multiple *iterations* (Lines 8 to 17). Several structures are maintained throughout iterations: ❶ The *candidate set S* contains the nearest L (*search length*) candidates found so far. ❷ The *result set E* contains the exact distances of the candidates.

The search process starts from a fixed *entry node p*, followed by several iterations. An iteration consists of the following steps: ❶ Read the SSD page of the current *exploring node u* (Line 9), and retrieve the full-precision vector and the neighbor list from the corresponding record. ❷ Mark u as explored (Line 10) to make sure each node is explored only once. ❸ Insert u into the result set by calculating the distance to the query vector with the full-precision vector from SSD (Line 11). ❹ Visit all neighbors of u and insert them into the candidate set with PQ distance (see below) (Lines 12 to 15). ❺ Resize the candidate set to L to limit the number of iterations (Line 14). ❻ Find the new exploring node with the closest distance to the query vector (Line 16). If all nodes are explored, the

Algorithm 1 Search Procedure on an SSD-based Graph

```
1: procedure SEARCH($G$)
2:     $q \leftarrow$ query vector, $p \leftarrow$ entry node
3:     $L \leftarrow$ search length, $k \leftarrow$ top-$k$ vectors to search
4:     $S \leftarrow \{p\}$                                          ▹ candidate set
5:     $E \leftarrow \emptyset$                                             ▹ result set
6:     $u \leftarrow p$                                             ▹ exploring node
7:     Initialize PQ lookup table for $q$.
8:     while $u \neq$ null do
9:         $(x, N) \leftarrow$ Read (vector, neighbor) of $u$                ▹ On SSD
10:        set $u$ as explored in $S$
11:        $E$.insert($u$, dist($x, q$))
12:        for $v \in N$ do
13:            $S$.insert($v$, PQ-dist($v, q$))
14:            $S$.resize($L$)                        ▹ Remove farthest candidates
15:        end for
16:        $u \leftarrow$ the nearest unexplored node in $S$
17:    end while
18:    return top-$k$ nearest neighbor in $E$.
19: end procedure
```

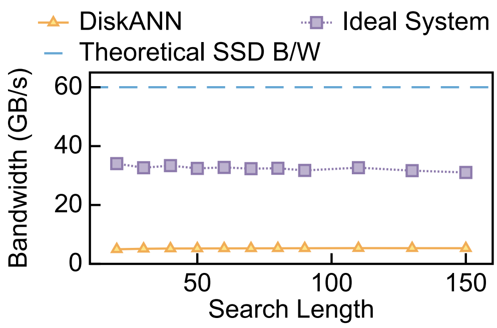

Fig. 2. Limited bandwidth utilization of DiskANN. We measure the SSD read bandwidth of DiskANN. We also compute its ideal bandwidth, assuming that all disk read latency can be hidden with computation (Ideal System). The raw bandwidth of our SSDs is shown in the blue dashed line.

search process terminates, and the top-k nearest nodes in E would be returned as the approximate nearest neighbors.

To reduce the number of fetching full-precision vectors, many on-disk graph-based ANNS systems keep a copy of in-memory compressed vectors using product quantization (PQ) [30, 34, 49], and use PQ distance as an approximation when inserting neighbors to the candidate set (Line 13).

3 Motivation

3.1 CPU Limits ANNS Cost-Effectiveness

Current disk-based graph ANNS systems fail to fully exploit the performance of SSDs. DiskANN [30] is a typical graph-based ANNS on SSDs. We measure its SSD read bandwidth during the search (§5.1

details the configuration of the experiments; for a more accurate estimation of the computing power, hyper-threading is disabled in this experiment to prevent interference between hyper-threads). Figure 2 shows that DiskANN achieves only 6 GB/s of SSD bandwidth, only 60% of the theoretical maximum bandwidth (10 GB/s) of the SSD.

Although DiskANN spends time on I/O waiting, we find that overlapping all of the I/Os still shows undesirable performance. In our experiment, a DiskANN worker thread spends approximately 1/6 of its time computing and 5/6 reading from the SSD. This is because DiskANN traverses the graph serially. Even with perfect I/O-computation overlap, where no CPU time is wasted waiting for I/O, the increased computation still cannot fully utilize the SSD bandwidth (*Ideal System* in Figure 2). This is because, as the Ideal System spends 6× of the CPU time on computing compared to DiskANN, it can only achieve 6× of the SSD bandwidth, which equals ~36 GB/s. The bandwidth of the ideal system still cannot reach the bandwidth upper bound of 6 SSDs (our configuration, 60 GB/s) because of the limited computing power of the CPU.

Our analysis shows that **CPU limits the high-throughput performance of ANNS.** This seems contrary to previous thoughts that the SSD is the dominating factor in disk-based ANNS [19, 30, 49]. This difference arises because previous works mainly focus on latency-critical scenarios. In these scenarios, assigning one CPU core per SSD is sufficient, because the latency of synchronous SSD reads is longer than the computation time, thus limiting the overall latency. However, to achieve high throughput, the number of CPU cores must match the SSD I/O bandwidth. For example, storage servers may have 6 – 24 SSDs [9, 14]. As a result, fully saturating the bandwidth requires approximately 94 – 375 CPU cores (Our 56-core server can serve the bandwidth of ~3.6 SSDs), which is difficult to achieve in a single server.

The insufficient CPU computing power limits the cost-effectiveness of ANNS. As CPUs on one server cannot saturate the bandwidth of the SSDs, more servers are needed to reach the same QPS, thus increasing the cost of deploying the ANNS service (§5.5).

3.2 Opportunity: GPU for ANNS

Based on the observation above, we use GPU instead of CPU for high-throughput and cost-effective ANNS, because of the strong computing power of the GPU. Due to its highly parallelized hardware design, GPU outperforms CPU in both FLOPS and memory bandwidth. For example, NVIDIA A100 can achieve 19.5 TFLOPS for 32-bit floating numbers, and 1935 GB/s memory bandwidth [6]. In contrast, CPUs have much lower FLOPS (1.25 TFLOPS for the 28-core CPU used in our evaluation [2]) and DRAM read bandwidth (typically ~30 GB/s for each DRAM).

Existing systems using GPU for ANNS show limitations for exploiting multiple SSDs. They can be categorized into two types:

(1) *CPU-centric, GPU-assisted architecture* [47, 56]. Only vector computation (e.g., distance) is offloaded to GPU; the main search process is still executed on CPU. Although this is feasible for cluster-based [56] indexes due to their high compute density of intra-cluster vectors, this architecture suffers from the performance bottleneck in graph-based indexes, which is not that compute-intensive compared to cluster indexes.

(2) *GPU-only architecture* [41, 48, 57]. All the search tasks are executed on GPU, including graph traversal and vector computation. It eliminates the bottleneck of CPU search. However, it suffers from limited GPU memory and the inbound transfer bandwidth of GPU.

To handle these limitations, we propose a *GPU-centric, CPU-assisted architecture*, where the search task is executed on GPU, but CPU is utilized to assist data transfer, which is possible to eliminate the bottleneck of limited in-bound transfer bandwidth. However, this architecture still faces several challenges:

3.3 Challenges

Challenge 1. *The memory-consuming search process limits the concurrency of the system, which is required by both GPU and SSD.*

Both GPU and SSD are highly concurrent in their designs. GPUs achieve high computing power with a large number of computing threads. For example, the NVIDIA A100 GPU utilizes 108 streaming multiprocessors (SMs), and each SM supports 2048 threads running concurrently [6]. In addition, modern SSDs achieve high bandwidth by processing multiple requests simultaneously. As a result, combining GPU and SSD requires a highly concurrent system design.

However, existing graph-based ANNS systems cannot support high concurrency because of their high memory consumption. GPU memory is limited (e.g., 40 GB memory in our setup), and most of its memory (~75%) is used to store the compressed dataset (PQ codes). Therefore, the memory left for each query is scarce. However, most current implementations for graph-based ANNS are CPU-oriented [21, 30, 40] and use memory-consuming per-query data structures. Current GPU-based ANNS systems inherit this design [48, 57]. Consequently, these structures consume a significant amount of memory (28 – 45% of the memory usage for different implementations, see §4.2 and §5.3). As a result, the limited GPU memory restricts the maximum number of concurrent queries (a maximum of 50% reduction), further preventing the full utilization of the GPU and SSD.

Challenge 2. *To mitigate the GPU-side bandwidth bottleneck, how to effectively use CPU to assist data transfer is under-exploited.*

GPU-only methods use *GPU direct-access* such as BaM [42], GPUDirect Storage (GDS) [7], to direct transfer from SSDs to the GPU. However, applying GPU direct-access cannot achieve desirable performance for graph-based ANNS. As Figure 3(a) indicates, a baseline system with GPU direct-access can only achieve $\sim$ 40% of GustANN's maximum performance.

By analysis, we find that the CPU-centric PCIe topology limits efficient GPU access to SSDs. All PCIe devices are first connected to the root complex, which connects to the CPU and DRAM through high-bandwidth links. Typically, devices lack direct connections. Such a layout results in asymmetric transfer bandwidth between SSDs and the GPU. As the number of SSDs grows, their aggregated bandwidth may exceed the inbound bandwidth to the GPU, which becomes a bottleneck. Figure 6(a) gives a detailed example.

GPU direct-access suffers from this asymmetric bandwidth. Take BaM [42] as an example. As shown in Figure 3(b), when adopting BaM as the I/O engine of the ANNS with 6 SSDs, the effective bandwidth of SSD only reaches 27% of its theoretical maximum. This is because GPU direct-access methods need to read full SSD pages through the PCIe link, whose bandwidth is only $\sim$ 40% of the maximum bandwidth as indicated in the figure.

Challenge 3. *Load imbalance exists among SSDs with a RAID-0 layout, which degrades the overall search performance.*

To maximize the bandwidth of multiple SSDs, a natural approach is to distribute SSD pages in a RAID 0 layout. Although this layout distributes reads evenly (and so is the bandwidth) in our datasets, it still results in load imbalance in terms of SSD access latency during the search. We measure the SSD read latency with the RAID 0 layout in the search. As shown in Figure 3(c), the maximum 99% latency is 2.17× higher than the minimum. Such high tail latency divergence is unacceptable for search on the GPU due to the bucket effect: the GPU prefers batching for high concurrency, but the long tail latency of a single query slows down the entire batch. As a result, both the SSD bandwidth utilization and the search performance drop because of the load imbalance (See §5.3.3).

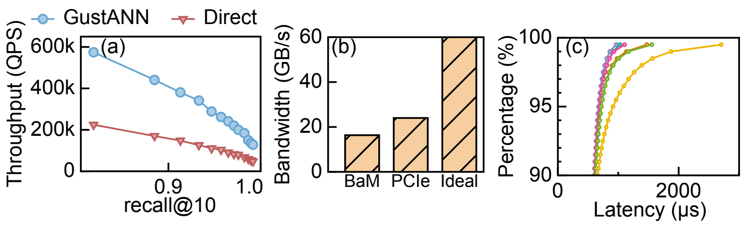

Fig. 3. Design challenges. (a) Throughput comparison of GustANN and the *GPU direct-access* solution (*Direct* in the figure). (b) The SSD bandwidth comparison of BaM (a recent GPU direct-access solution), and the theoretical maximum (*Ideal* in the figure). *PCIe* is the maximum PCIe transfer bandwidth of the GPU. (c) The distribution of the SSDs' read latency. Each line indicates an SSD.

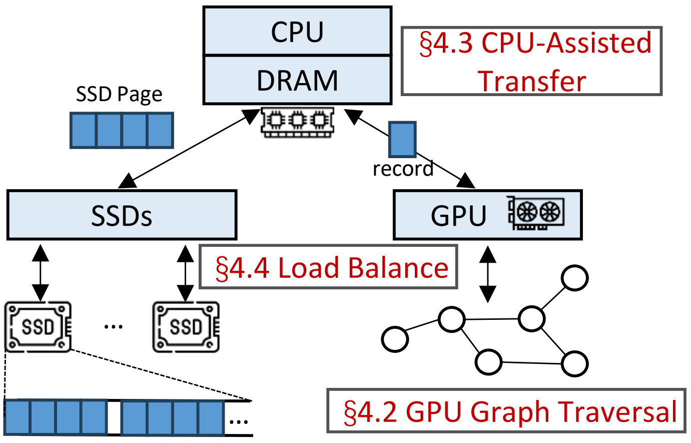

Fig. 4. GustANN architecture.

4 Design

4.1 Overview

GustANN uses a *GPU-centric, CPU-assisted architecture* to achieve high-throughput and cost-effective ANNS. Its overall architecture is shown in Figure 4.

4.1.1 Data Placement. The data placement on disks of GustANN is similar to DiskANN [30] (§2.2), which stores vectors and neighbors on disks. In addition, GustANN uses multiple SSDs, and the data is stored in a RAID 0 scheme. To tackle the load balance issue (Challenge 3), we leverage a pivot search (§4.4) and store a pivot graph in GPU memory. Also, most data structures (e.g., compressed vectors) are moved from DRAM to GPU memory.

The main usage of DRAM in GustANN is the bouncing buffers in the CPU-assisted transfer to enable selective transfer from SSD to GPU. Instead of GPU direct-access, this method can mitigate the bottleneck of the PCIe bandwidth for the GPU (§4.3). GustANN mainly uses DRAM for its bandwidth, and its capacity demand is much smaller compared to other DRAM-based billion-scale ANNS systems, thus reducing costs for servers (§5.5).

4.1.2 Search Process: Single Vector. GustANN leverages GPU for graph traversal tasks to increase the bandwidth utilization of SSDs, ensuring search throughput.

Initialization. GustANN initializes a query similarly to DiskANN. In addition, the pivot search for load balancing is performed (more detail in §4.4) to decide the entry node of the main search.

Search. The search process is led by the CPU, which coordinates reading on SSDs and searching on the GPU. To overcome the inbound bandwidth limit of the GPU, we leverage the CPU to assist in initiating the SSD read process (§4.3).

The searching process follows a similar scheme to Algorithm 1 (§2.2), except that most computation is moved to the GPU. It consists of repeated iterations until all vectors in the candidate set are explored. In each iteration, there are two main phases:

(1) *SSD phase*, where we fetch the corresponding SSD page of a selected exploring node u from SSD to DRAM, and then transfer the desired record in the page to GPU (§4.3).
(2) *GPU phase*, which performs distance calculations and data structure updates on GPU. The GPU phase also selects the exploring node for the next SSD phase (§4.2).

4.1.3 Search Process: Multiple Vectors. To support high throughput ANNS, GustANN puts queries into batches to further exploit the computing power of GPU and read bandwidth of SSDs. Note that we optimize the memory usage of each query (§4.2) to ensure a large batch size and thus a high concurrency level.

However, batched queries still cannot fully utilize GPU and SSDs. The graph traversal is sequential, with strong dependencies between stages, such as the exploring node and the record from SSD. As a result, two phases cannot run simultaneously, and half of the hardware resources are wasted.

To further extract hardware potentials, GustANN introduces *inter-query pipelining*. Instead of a single large batch for all queries, we split them into several mini-batches. These batches are independent of other batches, so both stages can be filled with different mini-batches. Therefore, all hardware resources can be utilized.

GustANN does not require strong CPU for assistance. During the search process, the main task of the CPU is only to schedule and dispatch different batches. Therefore, there is no need for CPU with strong computing power or special hardware features (e.g., advanced SIMD instructions). In addition, CPU with a low number of cores (e.g., 8 cores) can also serve GustANN efficiently. In our configuration (§5.1), we only need 2 threads for sending requests to SSDs and the GPU, and 6 threads for SSD I/O queue management (each sending ~2M IOPS requests to a designated SSD).

4.2 Memory-Efficient Graph Traversal Kernel

In this section, we address the high memory consumption issue in §3.3 (Challenge 1) by redesigning the GPU kernels for graph ANNS. We first analyze the memory footprint of the ANNS searching procedure. Then, we point out the limitations of the current implementation and our observations. Finally, we present our search algorithm to optimize the memory usage.

4.2.1 Memory Footprint Analysis. We analyze the memory footprint of one iteration in the search process shown in Algorithm 1 (procedure with full-precision vectors is omitted for its negligible time and memory overhead; Only per-query data structures are included below):

(1) Read all neighbor IDs from the *record* (on SSD).
(2) Calculate the PQ distance with the *PQ lookup table*.
(3) Insert the (neighbor ID, PQ distance) pair into the *candidate set*, which should be (partially) sorted.

Note that to keep the correctness of the algorithm, all element IDs in the candidate set should be distinct. For the CPU-based algorithm, it is natural to maintain an auxiliary data structure (denoted as the *visited table*), such as a hash table or bloom filter [30, 48], to identify whether a node has been calculated *before* step (2), and thus reduce redundant computation.

The visited table holds up a large proportion of memory. This is because, for each iteration, all R neighbors of the current exploring node should be inserted into the visited table. As the search needs at least L iterations to explore all nodes in the candidate set, space for at least $L \times R$ nodes should be reserved. For example, when $L = 300$ and $R = 128$, and each neighbor uses 4 bytes, the memory consumption of the visited table is at least 150 KB. In practice, such data structures need to be half-full to prevent performance degradation, meaning their real size should typically be doubled ($>$ 300KB). In comparison, the candidate set only stores L nodes ($1-2$ KB). Other structures (i.e., the record and PQ lookup table) have fixed sizes (~40 KB) throughout the search. In addition, as GustANN needs high concurrency (~40000 concurrent queries), $>$ 11.5 GB of memory is needed for that single data structure, which is a large memory consumption given the limited GPU memory space (40 GB in our setup) and the memory need of other global data structures (such as PQ code, which needs $>$ 30 GB of space for billion-scale datasets).

4.2.2 Sort First, Then Deduplicate. Memory-consuming visited table restricts the batching capability of GPU due to the limited GPU memory, thus further reducing the search throughput. To save space, some approach on GPU uses linear lookup and insertion [26]. However, the time consumption is usually unacceptable.

To save memory space while not incurring much time overhead, we adopt the following observation: **Redundant computation is acceptable on GPU**. For ANNS on the CPU, it is natural to eliminate the duplicate neighbors before distance calculation, so that there is no redundant computation for PQ distances. However, GPU threads are executed in the granularity of warps (32 threads each). Threads in the same warp always execute the same instruction or are waiting (even if they should execute other branches). Therefore, there is no need to eliminate the redundant computation in one thread, as it will be hidden by other threads.

Based on this observation, we first perform the distance calculation and sorting procedure, before we deduplicate the sorted array. As a sorted array can be deduplicated with a linear scan (detailed in §4.2.3), no extra data structures such as the visited table are needed. In addition, the warp structure of the GPU ensures that the redundant computation in distance calculation and sorting can be hidden by other threads, and does not cause much time overhead.

4.2.3 Algorithm. We present our search algorithm, which first sorts and then deduplicates. In the search, we organize the *candidate set* as a sorted array according to the PQ distance. Each item in the array contains the candidate node's ID and the PQ distance. In addition, the information of the explored node (see Algorithm 1) is stored as an extra bit for each item.

For each search iteration, we first read all neighbors for the current exploring node (SSD phase). Then we add these neighbors to the candidate set with the following steps (GPU phase), as shown in Figure 5(a):

PQ distance calculation. We first calculate the PQ distances of the current exploring node's neighbors (Step ❶). We use R (number of neighbors) GPU threads, each of which calculates one neighbor's PQ distance. By assigning distance calculation for each neighbor to one thread, the redundant computation of duplicated neighbors can be hidden by the computation of other neighbors due to the warp property of the GPU.

Sorting. After that, all neighbors are appended to the candidate set, and the candidate set is sorted according to the PQ distances (Step ❷). Sorting consists of two substeps. First, we sort all neighbors

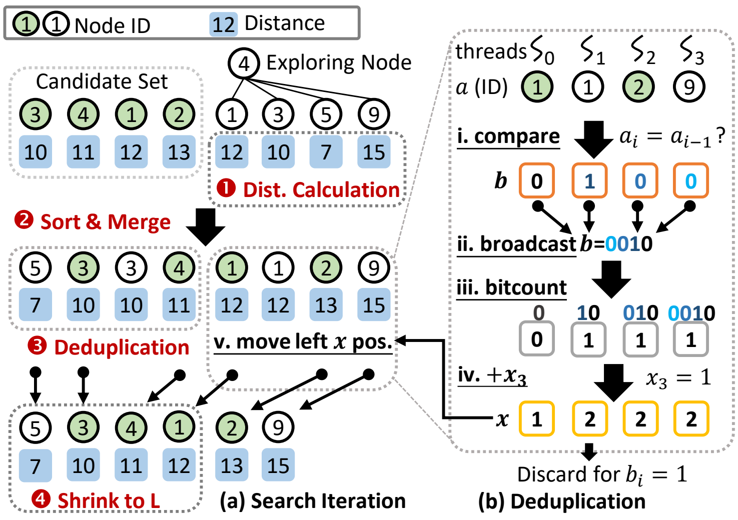

Fig. 5. The search algorithm on GPU. (a) One search iteration. (b) Parallel deduplication in the iteration. We show the round $k = 1$, which processes $a_4 \sim a_7$. For simplicity, we set $B = 4$ (the number of elements to be deduplicated in parallel) and $L = 4$ (size of the candidate set) in the figure.

retrieved from the last step. After that, we merge these neighbors with the remaining part of the candidate set (i.e., sorted nodes from previous iterations). We adopt parallel merge sort [25] for these two substeps. We also use R GPU threads in this step, so that the sorting procedure can be fused with the PQ distance calculation kernel, making better use of the high-bandwidth shared memory in the GPU.

Deduplication. After sorting, the candidate set is updated with new neighbors and sorted according to the distance, but duplicate items remain. To ensure the correctness of the algorithm, deduplication must be performed (Step ❸), before selecting the first L elements as the new candidate set (Step ❹). The deduplication of a sorted array is a traditional algorithm with a linear scan (for example, `std::unique` in C++ STL). However, a parallelized deduplication on the GPU is non-trivial, which needs to handle data dependency carefully.

We first present a non-parallelized version of the deduplication procedure. It is done with a linear scan of all node IDs in the candidate set, denoted as array a. During the scan, we maintain x_i, which indicates the number of duplicated elements from a_1 to a_i. We use the following formula to compute x_i:

$$\begin{cases} x_i = x_{i-1} + \delta(a_i, a_{i-1}), & i \geq 1 \\ x_0 = 0, \end{cases} \tag{1}$$

where $\delta(m, n)$ returns 1 if $m = n$ and 0 otherwise. After that, we move a_i (and other elements for that node) to a_{i-x_i}, for all i satisfying $a_i \neq a_{i-1}$. This step removes all duplicates.

To utilize the parallel ability, we use $B = 32$ GPU threads (a warp) for the scan. However, the scan must be redesigned to fit the parallel scheme, as illustrated in Figure 5(b). We divide the scan into multiple rounds, with each round scanning B elements in a simultaneously. In the k-th round of scan, the i-th thread $(0 \leq i < B)$ processes the $(Bk + i)$-th element. It first calculates $b_i = \delta(a_{Bk+i}, a_{Bk+i-1})$ (Step **i**). After that, each thread broadcasts b_i to other threads, which can be done with the warp primitives of CUDA [37] (Step **ii**). The result can be represented as a B-bit

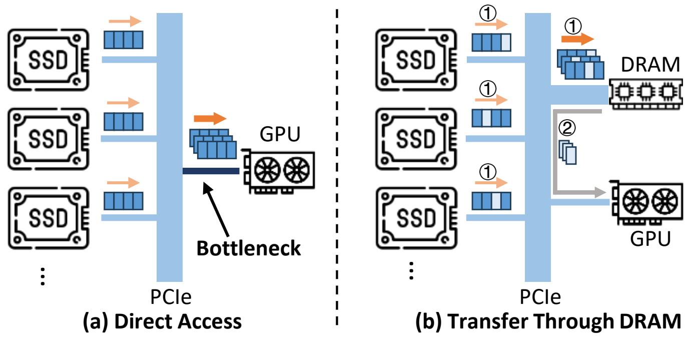

Fig. 6. Comparison of (a) GPU direct-access to SSD, which causes bandwidth bottleneck, and (b) first transfer the whole page to the bouncing buffer in DRAM, and then transfer the selected record to GPU.

integer **b**, where the i-th bit of **b** is b_i. After synchronization, the i-th thread computes:

$$x_{Bk+i} = x_{Bk-1} + bitcount(\mathbf{b}_{0..i}) \tag{2}$$

where $\mathbf{b}_{0..i}$ is the first i bits in **b** (Step **iii** and **iv**). Note that x_{Bk-1} has been computed in the previous round, and we assume $x_{-1} = 0$ for $k = 0$ (first round). Next, each thread performs the data move mentioned above (Step **v**).

A further observation is that each duplication appears only twice in the list (once in the neighbor list and the other in the previous candidate set). As a result, only the first $\min(L + R, 2L)$ elements before deduplication may become the new candidate. Therefore, we only perform scans on these items.

Note that the exploring node for the next iteration can also be found during the scan, by selecting the first element that has not been explored.

4.3 CPU-Assisted Transfer

Bandwidth between SSDs and the GPU is asymmetric, which degrades SSD utilization, as shown in Figure 6(a). On the SSD side, as SSDs do not share PCIe lanes, their bandwidth can be aggregated to around 60 GB/s with 6 SSDs (our setup in §5.1) with PCIe 5.0 x4. However, on the GPU side, the bandwidth is limited to 24 GB/s with PCIe 4.0 x16 for NVIDIA A100. Therefore, GPU direct-access to SSDs (such as BaM [42], GDS [7]) cannot fully exploit the bandwidth potential of multiple SSDs.

To address this challenge, we find that only one record is used for each iteration, which typically accounts for less than 25% of the whole SSD page. During the GPU phase of the search, the GPU threads read one SSD-resident record, which is typically less than 1KB. For example, the vectors in the DEEP1B [52] dataset have 96 dimensions, and each dimension of the vector is a 32-bit floating number. With graph degree $R = 128$, the total record size would be `sizeof(float)` $\times 96+$ `sizeof(int)` $\times 128 = 896$ bytes. Meanwhile, SSDs have a read granularity of 4 KB, meaning less than 25% of the data may be used by the search algorithm.

This observation inspires us that we can selectively transfer data to the GPU. GPU direct-access cannot achieve selective transfer. Therefore, we leverage CPU to assist this transfer process, with CPU DRAM used as a bouncing buffer, as shown in Figure 6(b)

Step 1: The whole SSD page is transferred from SSDs to DRAM. As the transfer bandwidth from the PCIe root complex to DRAM is higher, the full transfer speed of SSDs can be achieved. In our 6-SSD setup, the bandwidth of 4 DRAM chips meets the need for the aggregated SSD bandwidth of ~60 GB/s. (§5.5.1 further discusses the DRAM usage.)

Step 2: Only the graph record of the exploring node is sent from DRAM to the GPU. Compared to GPU direct-access, this method eliminates the transfer of unneeded data to the GPU, thus reducing $> 75\%$ of the GPU inbound bandwidth demand. For example, a maximum of $60 \times 25\% = 15$ GB/s of GPU inbound bandwidth for the DEEP1B dataset is required, which is within the maximum bandwidth of the GPU.

Therefore, as GPU inbound bandwidth is no longer a bottleneck in the data transfer process, the bandwidth strength of multiple SSDs can be fully utilized, thus providing an opportunity for higher search throughput. In contrast, GPU direct-access methods only support reading in page granularity, which causes a waste of GPU inbound bandwidth.

Note that some datasets with high dimensions (e.g. OpenAI Embeddings [5] have 1536 dimensions) may exceed the SSD page size. However, these high-dimensional embeddings can be compressed to a smaller number of dimensions without losing much precision [23, 34, 51, 52]. Therefore, we can first reduce their dimensions before storing them.

4.4 Load Balance with Pivot Search

In this section, we first analyze the cause of load imbalance (Challenge 3). After that, we present our solution with a GPU-resident pivot graph.

4.4.1 Single Entry is not Efficient for Multi-SSDs. Many graph-based methods [21, 30, 39] adopt a single-entry design, where queries start the graph traversal from the same entry node.

However, such a single-entry design is inefficient as it causes a phenomenon we refer to as *micro-scale load imbalance*. This is because all queries only visit the nearby region around the entry node of the graph in the first few iterations, and only a few nodes are possible to be visited. For example, only 103 nodes for the first iteration, and 2783 nodes for the second iteration are visited in a billion-scale dataset [31]. Only nodes around 10^{-6} of the whole dataset are visited.

Such a small number of nodes are vulnerable to the load imbalance issue. For example, for a 10000-vector query set, while the median access count for the first iteration is 48, the maximum access count is 657 (13.7×). Such a divergence in access counts results in varied performance of SSDs, especially in terms of tail latency, as shown in Figure 3(c). This imbalance is further magnified because all queries need to visit this region in the single-entry design.

4.4.2 Solution: GPU-resident Pivot Graph. Based on the analysis above, we argue that these queries should be scattered to *different regions* of the graph instead of the single region around the entry node. This method has two advantages: first, it mitigates load imbalance because each region now has fewer nodes, so the divergence will be less significant. In addition, as the target for the first few iterations in a single-entry graph-based search is to locate the top-1 nearest neighbor of the query [55], a region closer to the nearest neighbor of the query will result in better search speed.

To find a closer region, we introduce the *pivot search* technique: we randomly select *pivot vectors* from datasets (e.g., 0.1% of a billion-scale dataset) to represent different regions, and build a *GPU-resident pivot graph G'* on the GPU memory. For a query, we first perform a top-1 search on G' (the pivot search). The result (the nearest vector of the sampled set) will be used as the entry node of the SSD-resident graph search.

Pivot search has a similar search scheme to §4.2. Here, we mention some key differences: (1) To reduce the overall search overhead, we use a smaller *search length* (i.e., the length of the candidate set) for the pivot search (default value is 4). This is because pivot search does not need high recall for the nearest neighbor. (2) We use full-precision vectors instead of compressed PQ codes, as the pivot graph does not consume much memory. (3) We use the selection sort instead of the merge sort. This is because the search length is small, and the selection sort can be better parallelized under such circumstances.

It is worth mentioning that Starling [49] also uses a similar technique called the navigation graph to reduce the number of SSD I/O. However, GustANN differs from Starling in two aspects: (1) GustANN introduces the GPU-resident pivot graph to mitigate the load imbalance of SSDs, caused by the one-entry design. (2) GustANN uses GPU, instead of CPU, to perform the pivot search. Therefore, a different GPU-based search scheme is adopted, as shown above.

4.5 Discussion

4.5.1 Scalability of Data Size. In this section, we explore the possibility for scaling the data size by analyzing the storage usage of GustANN. GustANN mainly uses SSD and GPU memory, with a low DRAM usage (~200 MB for the bouncing buffers in the search).

On the SSD side, we store the vectors and the graph index as described in Figure 1. Billion-scale dataset (e.g., DEEP1B [52]) takes up ~950 GB of SSD space. Given that the capacity of modern SSDs can reach several TBs (e.g., 3.84 TB in our setup), the SSD will not be the bottleneck for scaling.

The GPU memory is mainly used for two purposes in GustANN. The first part is compressed vectors for distance approximation. Each compressed PQ vector is 32 bytes, and ~30 GB is required for the billion-scale dataset. To support larger datasets, recent quantization techniques such as LVQ [15], RabitQ [22, 24] attempt to further reduce the memory consumption of compressed vectors (e.g., from 32 B to 16 B). Such methods could double the dataset size, and are orthogonal to GustANN.

The GPU memory is also used to manage the search state of the queries. GustANN achieves a low memory budget for the search (~40 KB per query) by making the deduplication procedure memory-efficient. This provides more space for larger datasets.

Another approach for scalability is to increase the GPU memory capacity. For example, with 80 GB GPU memory (2× of our experiment setup), GustANN could support larger datasets (i.e., 2 billion vectors) without performance loss.

Combining larger memory (2×) and better quantization, GustANN could support 4 billion vectors with a single GPU.

4.5.2 Generalization of the Techniques. While GustANN targets at optimizing on-SSD ANNS with GPU, some ideas can be generalized to other problems:

- §4.2.3 provides a way to deduplicate with a low memory budget on the GPU. Other deduplication tasks on GPU could be done similarly (e.g., performing `DISTINCT` constraint). In the deduplication algorithm, two main techniques can be generalized: ❶ parallel predicates evaluation for on-GPU databases (Step **i** in Figure 5), and ❷ prefix sum on GPU with warp instructions (Step **ii** to **iv**).
- §4.3 introduces CPU-assisted selective transfer, which can be generalized to mitigate read amplification from SSD to GPU, potentially including on-SSD databases that utilize GPU for processing acceleration (e.g., when data processing on GPU is performed on a filtered subset of on-SSD data).

4.5.3 Limitation. To exploit the concurrent ability of GPU and SSD, GustANN trades off the search latency (~140 ms latency to reach the maximum throughput, see §5.6).

However, emerging applications, such as RAG, are not latency-critical because the latency of the whole process is mainly on the LLM side (e.g., prefill stage of LLM takes 100ms to 1s) [32, 38]. On the other hand, given the growing need for LLM and RAG applications, the ANNS in such scenarios are under much higher throughput demand, and we believe GustANN achieves a good throughput-latency tradeoff.

Some traditional ANNS applications, such as web search and recommendation systems, are typically considered latency-critical (e.g., 10 ms scale) [21, 30]. In these scenarios, GustANN may not achieve the maximum throughput due to the strict latency requirement.

5 Evaluation

In this section, we evaluate GustANN, seeking to answer the following questions:

- How does GustANN compare to other ANNS systems on both CPU and GPU for search throughput? (§5.2)
- How do designs of GustANN contribute to the search efficiency? (§5.3)
- Is GustANN sensitive to different workloads? (§5.4)
- How does GustANN compare to other ANNS systems in terms of cost-effectiveness? (§5.5)
- How does GustANN perform in terms of latency? (§5.6)

5.1 Experimental Setup

Basic configuration. We use a server with the following configuration for evaluation:

- **CPU**: 2× 28-core Intel Xeon Gold 5420+;
- **RAM**: 256 GB (8× 32 GB DDR5 4800MT/s);
- **SSD**: 6× Samsung PM1743 3.84 TB;
- **GPU**: NVIDIA A100-40GB;
- **OS**: Ubuntu 22.04 LTS with Linux kernel 6.2.0;
- **CUDA**: Version 12.3.

Datasets. We use 4 widely used public datasets for the evaluation:

- SIFT1B [31] is a dataset with 1 billion vectors of 128 `uint8` dimensions. The query set size is 10000.
- DEEP1B [52] is a dataset with 1 billion vectors of 96 `float` dimensions. The query set size is 10000.
- SIFT100M [31] is a subset of SIFT1B with 100 million vectors, and shares the same query set with SIFT1B.
- DEEP100M [52] is a subset of DEEP1B with 100 million vectors, and shares the same query set with DEEP1B.

Compared systems. We compare GustANN with two types of ANNS systems: (1) ANNS on CPU with SSD, and (2) ANNS on GPU with DRAM.

For ANNS on CPU with SSD, we use DiskANN, Starling, and SPANN. These systems store indexes in the SSD and use the CPU for computation. DiskANN [30] is a graph-based SSD-resident ANNS system on the CPU. Starling [49] improves DiskANN by reshuffling the indexing layout of SSDs and searching procedures. In addition, Starling uses a navigation graph to reduce the SSD I/O counts. SPANN [18] is a cluster-based ANNS system on the CPU. Cluster-based methods partition vectors into multiple clusters and search for the query in some selected clusters. SPANN leverages SSDs to store clustered vectors.

To store the vector index on the SSD, GustANN, DiskANN, and Starling have similar index structures and need ~65/~95 GB of storage space for SIFT100M/DEEP100M respectively. SPANN needs ~87/~235 GB for these two datasets. For the billion-scale dataset, the storage requirement is roughly 10× of the 100M-scale datasets (e.g., GustANN needs ~650/~950 GB for both datasets).

For ANNS on GPU with DRAM, we use BANG and RUMMY in the evaluation. BANG [48] is a recent graph-based ANNS system, and RUMMY [56] is a cluster-based ANNS system. They store indexes in DRAM and use the GPU for computation. BANG uses the DiskANN-style graph index and needs similar DRAM space (i.e., ~65/~95 GB for SIFT100M/DEEP100M), and RUMMY uses ~36 GB of memory for DEEP100M, mainly for the dataset.

Parameter Settings. We use the same graph SSD index for GustANN, DiskANN, and Starling with the Vamana [30] algorithm. The main indexing parameters are $R = 128$ and $L = 200$. We reshuffle the index for Starling. The PQ code length is 32 bytes so it can fit in the GPU memory for billion-scale datasets. DiskANN and Starling use beam search (i.e., read SSD pages of W exploring nodes simultaneously), and W is set to 4. The implementation of BANG limits the maximum number of neighbors to 64. Therefore, we select $R = 64$ to build the index for BANG and set the PQ code length to 64 bytes as suggested in their open-source code.

For the pivot graph used by GustANN, we build the navigation graph by randomly selecting 1M vectors with building parameters $R = 32$ and $L = 50$. It is also used as the navigation graph for Starling, as they share a similar graph structure. For fair comparison, we cache 1M graph nodes for DiskANN.

For SPANN, we adopt the configuration presented in [50] for index building and searching. For RUMMY, we modify the configuration in the open-source repository to produce higher throughput. We set the training data ratio to be fixed at 1% (1M vectors).

To store the SSD indexes for systems with CPU and SSD, as they use file systems for index storage, we set up an `ext4` file system on the SSDs, with a RAID 0 configuration to support multi-SSD. For GustANN, we use SPDK [13] for high SSD read performance.

During the search, we use all 112 threads (including hyper-thread) of the experiment server for all systems except GustANN. Note that GustANN only uses 8 CPU cores in the search, with 2 of them processing queries and 6 of them managing SSD I/O queues. GustANN uses multiple minibatches (§4.1.3). Unless otherwise stated, we run 40 concurrent minibatches on 2 CPU threads, and one minibatch processes 1120 vector queries.

Metrics. We mainly compare the following three metrics during the evaluation:

- **Search throughput and accuracy**. The search throughput is measured in queries per second (QPS), while search accuracy is measured in recall@k. We repeatedly send the query vectors to search for a more precise measurement.
- **SSD Bandwidth**, the actual SSD pages a system reads per second, measured in GB/s. Note that for BANG and RUMMY, we do not compare this metric, as they use DRAM instead of SSDs for the search.
- **Cost-Effectiveness**, the cost a system needs to set up the server divided by the search throughput the system can provide, measured in \$/QPS.

5.2 Overall Performance

In this section, we show the overall performance of GustANN and the compared systems. We send queries in the query set for ANNS, and measure the search throughput and the SSD bandwidth under different recalls.

5.2.1 100M-Scale Datasets. The results of SIFT100M and DEEP100M datasets are shown in Figure 7 and 8 respectively. We make the following observations:

First, GustANN shows high throughput compared to other methods on CPU (DiskANN, Starling and SPANN) **and cluster-based method on GPU** (RUMMY).

Compared to DiskANN/Starling, the graph-based ANNS on CPU, GustANN has 6.70×/8.03× higher search throughput at recall@10=0.9 on SIFT100M. The reason is that GPU can fully utilize the bandwidth of multiple SSDs. Overlapping computing and I/O on the CPU cannot achieve this. Namely, DiskANN does not overlap CPU computation and SSD reading in the same thread, but

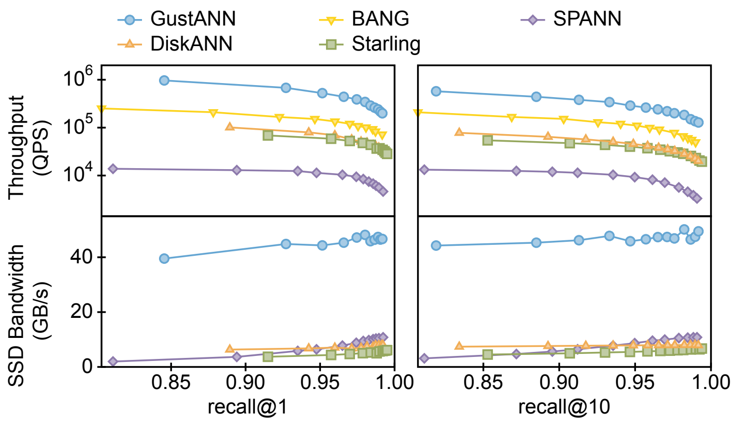

Fig. 7. Search results on the SIFT100M dataset. Note that RUMMY does not support `uint8` index of SIFT100M, so we do not compare with it.

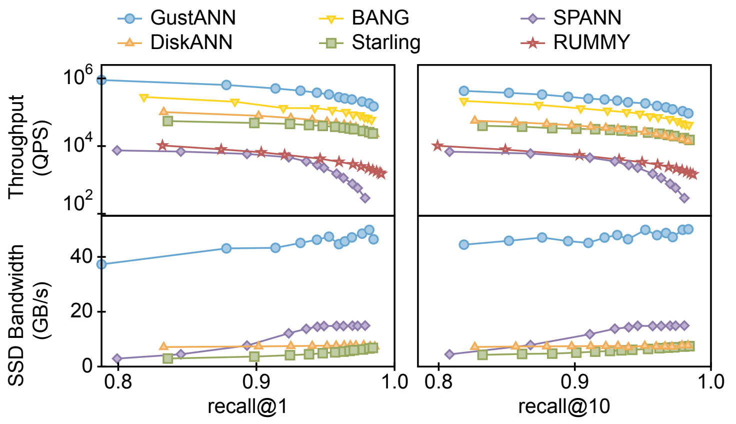

Fig. 8. Search results on the DEEP100M dataset.

as §3.1 suggests, CPUs still could not fully consume SSD bandwidth even when fully parallelized. Starling introduces a pipelining technique, by calculating other records on the same page in the current iteration while reading the page for the next iteration. However, it does not improve much throughput for two reasons. First, it introduces redundant computation. Second, the pipelining does not fully hide I/O latency.

GustANN also outperforms cluster-based ANNS, on both CPU and GPU. The search throughput of GustANN is 55.69× higher than that of SPANN, and 46.86× higher than that of RUMMY, when recall@10=0.9 on DEEP100M. This is because cluster-based ANNS needs to scan more vectors per query than graph-based ANNS. For example, to reach 0.9 of recall@10 in DEEP100M, graph-based ANNS typically need to check ~50 nodes for full-precision vectors. However, cluster-based ANNS needs to scan ~6000 vectors. The extra computation and I/O induced prevent cluster-based ANNS from reaching high search throughput.

Second, GustANN outperforms other graph-based algorithm on GPU. GustANN achieves 2.50× higher throughput than BANG at recall@10 = 0.9 on SIFT100M. Several factors contribute to

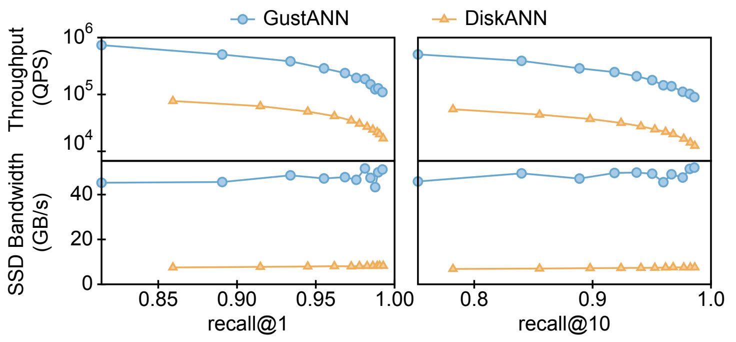

Fig. 9. Search results on the SIFT1B dataset.

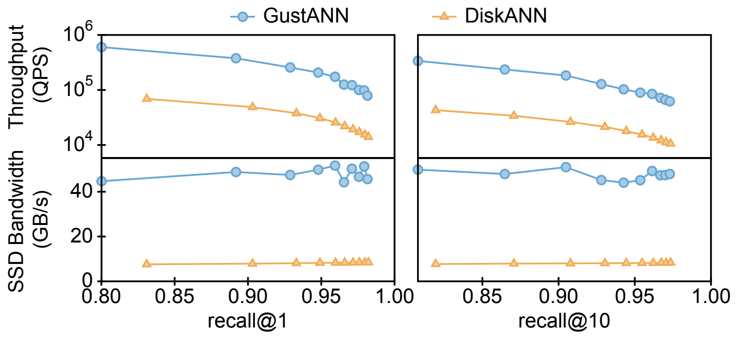

Fig. 10. Search results on the DEEP1B dataset.

this: (1) The high read bandwidth of DRAM compared to SSDs does not help when reading from the GPU, as PCIe inbound bandwidth limits the overall read bandwidth towards the GPU. (2) BANG has poor coordination between CPU and GPU. BANG frequently copies data between CPU and GPU, which does not fully overlap with other computations on the GPU. As a result, BANG does not have a high GPU utilization (43.0% compared to 68.6% of GustANN) and cannot search as efficiently as GustANN.

Third, GustANN achieves high SSD utilization. GustANN has significantly higher multi-SSD utilization than other SSD-based ANNS systems. GustANN reaches a maximum of 83.6% SSD bandwidth utilization (with each SSD offering 10 GB/s read bandwidth) on two datasets. For DiskANN and Starling, the maximum bandwidth utilization is 13.4% and 11.1% respectively, due to the inadequate pipelining of compute and I/O, and more importantly, the insufficient computing power of CPUs.

For SPANN, the maximum SSD bandwidth utilization is higher than DiskANN and Starling (18.0%) because of a friendlier pattern of I/O (reading several pages in parallel instead of reading only one page). However, it is at the expense of more SSD reads than graph-based methods (on average 147.9 pages compared to 31.8 pages of GustANN on each query for recall@10=0.9 in SIFT100M). As a result, cluster-based ANNS cannot reach the same search throughput even with a high SSD utilization, compared to graph-based ANNS.

5.2.2 Billion-Scale Datasets. For the two billion-scale datasets, we compare GustANN with DiskANN. Other methods need a large amount of memory (which exceeds the total memory of our setup) when either building or searching indexes. Therefore, we do not compare them on the billion-scale datasets.

Figures 9 and 10 show the results of SIFT1B and DEEP1B respectively. GustANN is scalable for large datasets, providing similar throughput on billion-scale datasets. Compared to SIFT100M, GustANN reaches 65% search throughput for the same recall@10 = 0.9 on SIFT1B. The reduction of the throughput is mainly due to the increase in search length. GustANN achieves almost the same search throughput in SIFT1B with the same search length as in SIFT100M.

GustANN also outperforms DiskANN by 7.81× when recall@10=0.9 on SIFT1B, and 6.98× on DEEP1B. For SSD bandwidth, GustANN maintains a similar SSD utilization, a maximum of 86.7%, compared to 13.7% of DiskANN on the SIFT1B dataset.

5.3 Breakdown Analysis

In this section, we analyze the efficiency of the three main designs proposed by GustANN.

5.3.1 Memory Usage of the GPU search. To demonstrate the GPU memory efficiency of GustANN by deduplicating after sorting (§4.2), we compare GustANN with BANG, which leverages a bloom filter for duplication detection. We use SIFT100M with $k = 10$ and $L = 100$. We measure the GPU global memory used for each query (memory shared for all queries such as PQ codes is excluded) of both systems. For BANG, the memory consumption for each query is 480 KB, while GustANN uses 40.22 KB of memory. GustANN is 11.93× more space efficient than BANG.

The main reduction of memory consumption is from the visited table, as BANG uses ~390 KB of memory for this data structure. Even though BANG uses bloom filter to reduce the space consumption compared to hash table, it still has two disadvantages: (1) bloom filter generates false positives, which degrades the search efficiency; (2) the bucket size should still be large (proportional to the number of visited neighbors) to reduce the false positive rate. In comparison, GustANN addresses these issues because (1) GustANN uses an exact deduplication algorithm without any approximation; (2) GustANN needs no extra memory during deduplication.

As a result, to support 40000 concurrent queries (which is the minimal number to fully utilize the pipeline and reach maximum SSD bandwidth), GustANN uses 1.53 GB memory (3.8% of the 40GB GPU memory), while BANG uses 18.3 GB memory (45% of the GPU memory). Considering the memory consumption of other global data structures (e.g., PQ code needs ~30 GB of GPU memory for billion-scale datasets), the large memory consumption of BANG degrades its concurrent capability (50% reduction of the maximum batch size).

Additionally, GustANN is more cache-friendly for query-specific data structures, which are frequently accessed by threads in the same streaming multiprocessor.

5.3.2 Effect of the CPU-Assisted Transfer. To demonstrate the sufficiency of the CPU-assisted transfer (§4.3), we compare GustANN with directly accessing SSDs from the GPU. We implement a variant of GustANN, which uses BaM [42], the state-of-the-art GPU direct-access system, to read SSDs from GPU directly. We call this variant BaM-ANN. We do not use NVIDIA GPUDirect Storage (GDS) [7] because BaM shows better 4KB random access performance (the typical workload of on-disk ANNS) than GDS [42]. Note that BaM uses the GPU hardware scheduler to hide the latency of direct I/O, so we do not use CPU pipelining in the I/O path.

Figure 11(a) shows the results for SIFT100M. GustANN consistently outperforms BaM-ANN by 2.50 − 2.71×. Figure 11(b) shows SSD and PCIe bandwidth usage. Due to the selective transfer enabled by the bouncing buffer on DRAM, GustANN can transfer much fewer records to GPU compared to the number of records read from SSDs. Therefore, the PCIe bandwidth requirement towards GPU is less than the SSD bandwidth (approximately 28.6%).

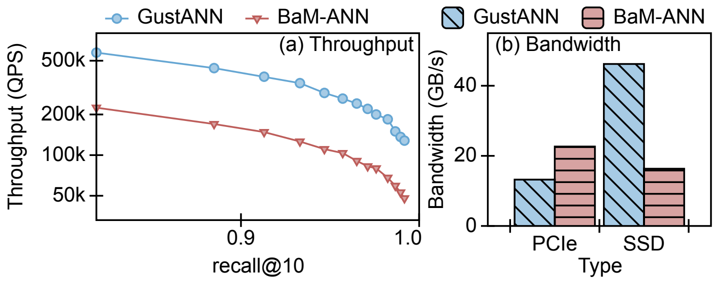

Fig. 11. Comparison of GustANN and BaM-ANN, which reads SSDs from GPU directly. "PCIe" means the PCIe inbound bandwidth to GPU.

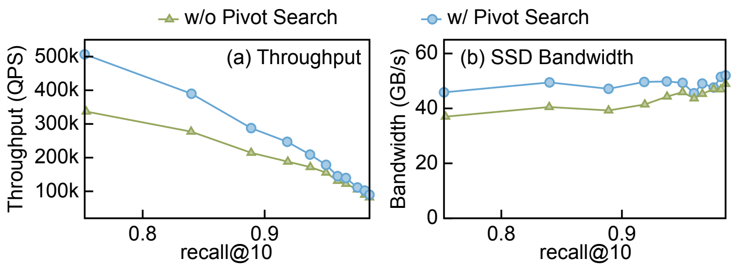

Fig. 12. Comparison of search throughput and SSD bandwidth of GustANN with and without pivot search.

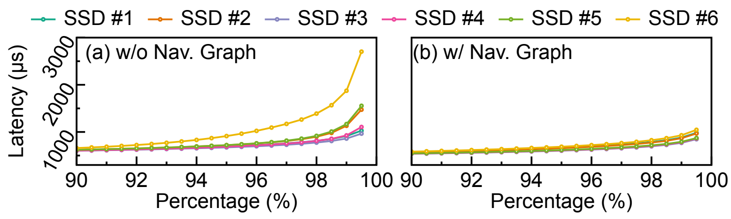

Fig. 13. SSD read latency comparison with and without pivot search.

On the contrary, BaM-ANN needs to read the whole SSD page to the GPU, which includes unneeded records for this iteration. As a result, the bandwidth requirement of PCIe towards GPU is identical to the SSD bandwidth (more on the figure because other transferred data is included). As the bandwidth limit of PCIe is smaller than the aggregated bandwidth of SSDs, the transfer process is bottlenecked by the GPU inbound PCIe bandwidth.

5.3.3 *Effect of the Pivot Search.* To illustrate the effect of SSD bandwidth utilization on the pivot search (§4.4), we remove the pivot graph of GustANN, and compare it with GustANN.

Figure 12 shows the result for search throughput and SSD bandwidth with SIFT1B and $k = 10$. We find that the pivot search improves the search throughput by at most 1.50×, and SSD bandwidth by 1.23×. The pivot graph improves the SSD utilization by load balancing. Figure 13 shows the SSD access latency distribution when $L = 30$. With the introduction of the pivot search, the divergence of access latency is much reduced. Therefore, the load imbalance in batches improves. We also find

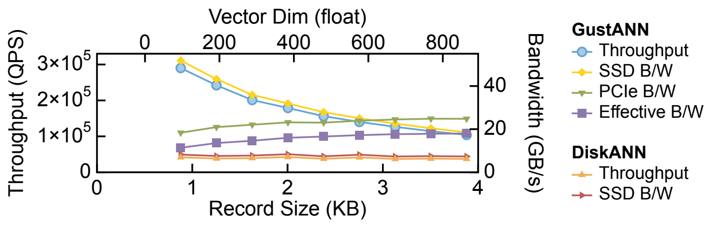

Fig. 14. Performance comparison with different vector dimensions. Note that the record size includes both vectors and neighbors (512 Bytes). "PCIe B/W" means the actual PCIe inbound bandwidth to GPU. "Effective B/W" indicates the effective bandwidth for transferring records to GPU with CPU-assisted selective transfer (§4.3).

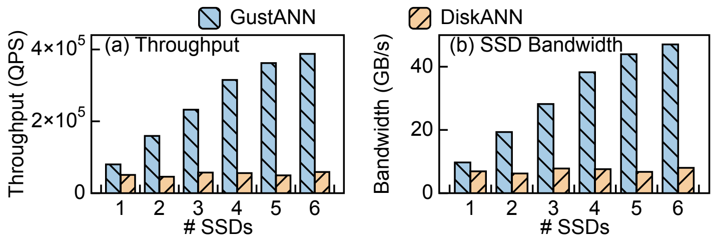

Fig. 15. Performance comparison with different numbers of SSDs.

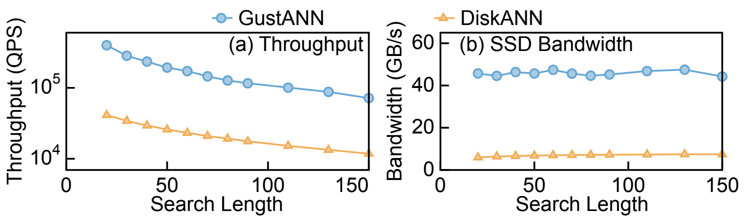

Fig. 16. Performance comparison on larger query set.

that the improvement is more significant in the lower recall case. This is because the search length in this configuration is small. Thus it is more vulnerable to the load imbalance around the entry node.

5.4 Sensitivity Analysis

Vector Dimension. When the vector dimension increases, the benefit from the CPU-assisted transfer (§4.3) decreases. To understand its impact, we replicate vectors from DEEP100M dataset to higher dimensions (e.g., concatenating a 96-dimensional vector to itself to get a 192-dimensional vector). We vary the record size (vector size plus neighbor size) from its original size to a full page (4 KB). Figure 14 shows its result when recall@10=0.9. The throughput and the SSD bandwidth drop for two reasons: (1) the data transferred as a proportion of the SSD page is increasing, which reduces the effect of selective transfer and causes the bandwidth to be bounded by the PCIe bandwidth; (2) more dimensions introduce extra computation, which also reduces the throughput. The throughput and bandwidth of DiskANN do not change with vector dimensions, as it always reads the whole page from the SSD to the CPU.

Table 1. Comparison of cost effectiveness [1, 8, 10–12].

Setup	GPU+CPU+SSD	CPU+SSD	CPU+GPU
CPU / Price (Incl. chassis)	1× 4410Y / \$17230	2× 5420+ / \$21360	2× 5420+ / \$21360
GPU / Price	1× A100 / \$12999	-	1× A100 / \$12999
DRAM / Price	128 GB / \$560	128 GB / \$560	1 TB / \$4480
SSD / Price	6× PM1743 / \$6150	1× PM1743 / \$1025 (Measured) 4× PM1743 / \$4100 (Ideal)	-
Representative System	GustANN	DiskANN	BANG
QPS	380793	56782 (Measured) / 165788 (Ideal)	152262
Cost Eff. (\$/QPS)	0.097	0.404 (Measured) / 0.156 (Ideal)	0.255

Number of SSDs. Figure 15 compares GustANN and DiskANN on different numbers of SSDs with SIFT100M and recall@10=0.9. GustANN shows stable scalability in both throughput and SSD bandwidth as the number of SSDs increases. In contrast, DiskANN can only saturate 60% – 70% of an SSD's bandwidth, regardless of the number of SSDs.

Query Set Size. GustANN shows stable performance regardless of the query set size. All datasets we use only provide 10000 query vectors. However, GustANN can serve > 40000 queries concurrently. To eliminate any potential cache effect during the whole search process, we create a query set with 10^6 query vectors based on the query vectors in SIFT1B, and search these vectors on the SIFT1B dataset.

The result is shown in Figure 16. Even with a larger query vector scale, GustANN can have a similar search throughput to the original query set, and still outperforms DiskANN by 6.08 – 9.65× with the same search length set as §5.2.2. The maximum SSD utilization for GustANN is 79.1%.

5.5 Cost Analysis

In this section, we first analyze the hardware resource usage of GustANN. Then we compare the cost-effectiveness with other ANNS systems on billion-scale ANNS.

5.5.1 Hardware Resource Usage. GustANN can use less hardware resources, namely CPU and DRAM, than specified in §5.1, without dropping the search performance.

CPU. As mentioned in §4.1.3, GustANN uses very few CPU cores (i.e., 8 cores in the experiment). Therefore, a weak CPU (e.g., Intel Xeon Silver 4410Y, which has the same frequency but fewer cores compared to the CPU in §5.1 [1]) will meet the need.

DRAM. GustANN uses a maximum of 33 GB memory space (mostly for data transfer to GPU during initialization; during the search, only ~200 MB will be used as the bouncing buffer in §4.3), 61 GB/s DRAM write, and 13 GB/s DRAM read bandwidth. In addition, memory from only one NUMA node is used for GustANN during the search. Therefore, 4 memory chips (128 GB) on a single NUMA node suffice.

Apart from these, other hardware for GustANN can be used as described in §5.1.

5.5.2 Cost Comparison. We split all the compared systems into three categories, based on their hardware requirements, and compare their cost-effectiveness using the system with the highest throughput (the first system in the list below) when recall@10 = 0.9 on SIFT100M:

(1) **GPU+CPU+SSD** (*GustANN*). The hardware specification is mentioned above.
(2) **CPU+SSD** (*DiskANN*, Starling, SPANN). We use hardware specification as mentioned in §5.1, except that no GPU is needed and DRAM usage can also be 128 GB. These systems can only

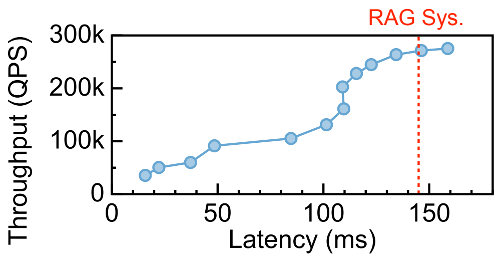

Fig. 17. Search latency on different throughputs. The search latency of RAG systems [53] is also shown on the figure (red dashed line).

saturate the bandwidth of less than 1 SSD, so we will use 1 SSD for a fair comparison. We also show the cost-effectiveness of the ideal system mentioned in §3.1 ("Ideal" in the table). According to the estimation, the ideal system can saturate 3.6 SSDs, so 4 SSDs will be used.

(3) **CPU+GPU** (*BANG*, RUMMY). To support a billion-scale dataset, larger DRAM is needed. A maximum of 1 TB DRAM is needed for DEEP1B. No SSDs are required.

The comparison of the three setups is shown in Table 1. GustANN (GPU+CPU+SSD) is 5.09× more cost-effective than CPU+SSD system (1.74× than the ideal CPU+SSD system), and 2.62× more cost-effective than CPU+GPU system. By introducing GPU, GustANN achieves a much higher search throughput compared to the CPU+SSD system. GustANN also prevents the non-scalable DRAM in CPU+GPU systems by using high-throughput SSDs.

5.6 Latency Analysis

GustANN trades latency off for throughput, because of the large-scale batched query. We measure the search latency against search throughput by adjusting the number of minibatches and number of queries processed per minibatch. We use SIFT1B dataset and show the results for recall@10=0.9. As shown in Figure 17, to reach the maximum search throughput, the search latency is approximately 140 ms. On the other hand, to ensure an average latency of < 25 ms, about 1/5 of the maximum search throughput can still be achieved. Such latency is acceptable in RAG workloads because these applications typically require 100ms-level to 1s-level latency for model processing [32, 38]. Similar latency is also observed in the RAG systems [53].

6 Related Work

SSD-resident ANNS. To support ANNS for billion-scale datasets, previous works have used SSDs to avoid storing terabytes of indexes in memory [18, 30, 46, 49, 54]. DiskANN [30] stores the graph index in the SSD, and reduces the search latency by modifying both index-building and searching algorithms. Starling [49] further reduces the number of on-disk I/Os by reshuffling the on-disk index and positioning nearby nodes on the same SSD page. SPANN [18] explores efficient on-disk indexing for cluster-based ANNS by allowing replication of vectors in its nearby clusters. These works focus on latency-critical ANNS scenarios, which fail to provide 100K-scale high throughput due to the limited computing power of CPUs. Instead, we use GPU to exploit multiple SSDs in graph-based ANNS, achieving high I/O efficiency and throughput. FusionANNS [47] leverages GPU for on-disk cluster-based ANNS. However, cluster-based methods are coarse-grained and thus have lower I/O efficiency than graph-based GustANN.

ANNS on GPU. Numerous works have explored how to execute graph-based ANNS [26, 41, 48, 57] and cluster-based ANNS [33, 56] on GPU. Recent works have also studied GPU search on larger datasets. BANG [48] supports graph index for large-scale datasets by coordinating computation on CPU and GPU. RUMMY [56] enables cluster-based FAISS [33] to support large-scale datasets by pipelining the transfers between GPU and DRAM. These works store the index in either DRAM or GPU memory, reducing cost-effectiveness. In addition, GustANN optimizes the GPU-based graph search algorithm to address the limited GPU memory and achieve better concurrency.

7 Conclusion

In this paper, we propose a GPU-centric, CPU-assisted architecture for ANNS and present GustANN, a billion-scale ANNS system targeted for high throughput and cost-effectiveness. Experiments show that GustANN achieves better throughput and is more cost-effective than current systems.

Acknowledgments

We sincerely thank all anonymous reviewers for their valuable feedback. This work is supported by Beijing Natural Science Foundation (Grant No. L242016) and the National Natural Science Foundation of China (Grant No. 62332011).

References

[1] 2024. 4th Gen Intel® Xeon® Scalable Processors. https://ark.intel.com/content/www/us/en/ark/products/series/228622/4th-gen-intel-xeon-scalable-processors.html.
[2] 2024. APP Metrics for Intel® Microprocessors. https://www.intel.com/content/dam/support/us/en/documents/processors/APP-for-Intel-Xeon-Processors.pdf.
[3] 2024. Google search statistics 2024 (no. of searches per day). https://www.demandsage.com/google-search-statistics/.
[4] 2024. Introducing OpenAI o1. https://openai.com/o1/.
[5] 2024. New embedding models and API updates. https://openai.com/index/new-embedding-models-and-api-updates/.
[6] 2024. NVIDIA A100 | NVIDIA. https://www.nvidia.com/en-us/data-center/a100/.
[7] 2024. NVIDIA GPUDirect Storage. https://docs.nvidia.com/gpudirect-storage/index.html.
[8] 2024. NVIDIA Tesla A100 Ampere 40 GB Graphics Processor Accelerator - PCIe 4.0 x16 - Dual Slot. https://www.amazon.com/NVIDIA-Ampere-Graphics-Processor-Accelerator/dp/B08X13X6HF/.
[9] 2024. PowerEdge R760 Rack Server. https://www.dell.com/en-us/shop/dell-poweredge-servers/new-poweredge-r760-rack-server/spd/poweredge-r760/pe_r760_15724_vi_vp.
[10] 2024. PowerEdge R760xa Rack Server. https://www.dell.com/en-us/shop/cty/pdp/spd/poweredge-r760xa/.
[11] 2024. Samsung 32GB DDR5 4800MHz PC5-38400 ECC RDIMM 1Rx4 (EC8 10x4) Single Rank 1.1V Registered DIMM 288-Pin Server RAM Memory M321R4GA0BB0-CQK. https://www.amazon.com/Samsung-4800MHz-PC5-38400-Registered-M321R4GA0BB0-CQK/dp/B0C35566S9.
[12] 2024. Samsung PM1743 3.84TB NVMe GEN5 E3.S 1T 12000MBps/12000MBps - MZ3LO3T8HCJR-00A07. https://www.newegg.com/p/2U3-0005-000N6?Item=9SIA12KKA53814.
[13] 2024. Storage Performance Development Kit. https://spdk.io.
[14] 2024. Supermicro 1U SuperStorage Server. https://store.supermicro.com/us_en/1u-superstorage-ssg-121e-ne316r.html.
[15] Cecilia Aguerrebere, Ishwar Singh Bhati, Mark Hildebrand, Mariano Tepper, and Theodore Willke. 2023. Similarity Search in the Blink of an Eye with Compressed Indices. *Proc. VLDB Endow.* 16, 11 (July 2023), 3433–3446. doi:10.14778/3611479.3611537
[16] Akhil Arora, Sakshi Sinha, Piyush Kumar, and Arnab Bhattacharya. 2018. HD-index: pushing the scalability-accuracy boundary for approximate kNN search in high-dimensional spaces. *Proc. VLDB Endow.* 11, 8 (apr 2018), 906–919. doi:10.14778/3204028.3204034
[17] Dmitry Baranchuk, Artem Babenko, and Yury Malkov. 2018. Revisiting the inverted indices for billion-scale approximate nearest neighbors. In *Proceedings of the European Conference on Computer Vision (ECCV).* 202–216.
[18] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. 2021. SPANN: Highly-efficient Billion-scale Approximate Nearest Neighbor Search. In *35th Conference on Neural Information Processing Systems (NeurIPS 2021).*
[19] Rongxin Chen, Yifan Peng, Xingda Wei, Hongrui Xie, Rong Chen, Sijie Shen, and Haibo Chen. 2024. Characterizing the Dilemma of Performance and Index Size in Billion-Scale Vector Search and Breaking It with Second-Tier Memory. *CoRR* abs/2405.03267 (2024). arXiv:2405.03267 doi:10.48550/ARXIV.2405.03267
[20] Kenneth L. Clarkson. 1994. An Algorithm for Approximate Closest-Point Queries. In *Proceedings of the Tenth Annual Symposium on Computational Geometry, Stony Brook, New York, USA, June 6-8, 1994,* Kurt Mehlhorn (Ed.). ACM, 160–164.

doi:10.1145/177424.177609

[21] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2019. Fast approximate nearest neighbor search with the navigating spreading-out graph. *Proc. VLDB Endow.* 12, 5 (jan 2019), 461–474. doi:10.14778/3303753.3303754

[22] Jianyang Gao and Cheng Long. 2024. RaBitQ: Quantizing High-Dimensional Vectors with a Theoretical Error Bound for Approximate Nearest Neighbor Search. *Proc. ACM Manag. Data* 2, 3, Article 167 (May 2024), 27 pages. doi:10.1145/3654970

[23] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. 2013. Optimized Product Quantization for Approximate Nearest Neighbor Search. In *2013 IEEE Conference on Computer Vision and Pattern Recognition.* 2946–2953. doi:10.1109/CVPR.2013.379

[24] Yutong Gou, Jianyang Gao, Yuexuan Xu, and Cheng Long. 2025. SymphonyQG: Towards Symphonious Integration of Quantization and Graph for Approximate Nearest Neighbor Search. *Proc. ACM Manag. Data* 3, 1, Article 80 (Feb. 2025), 26 pages. doi:10.1145/3709730

[25] Oded Green, Robert McColl, and David A. Bader. 2012. GPU merge path: a GPU merging algorithm. In *Proceedings of the 26th ACM International Conference on Supercomputing* (San Servolo Island, Venice, Italy) *(ICS '12).* Association for Computing Machinery, New York, NY, USA, 331–340. doi:10.1145/2304576.2304621

[26] Fabian Groh, Lukas Ruppert, Patrick Wieschollek, and Hendrik P. A. Lensch. 2023. GGNN: Graph-Based GPU Nearest Neighbor Search. *IEEE Transactions on Big Data* 9, 1 (2023), 267–279. doi:10.1109/TBDATA.2022.3161156

[27] Hao Guo and Youyou Lu. 2025. Achieving Low-Latency Graph-Based Vector Search via Aligning Best-First Search Algorithm with SSD. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25).* USENIX Association, Boston, MA, 171–186.

[28] Ben Harwood and Tom Drummond. 2016. FANNG: Fast Approximate Nearest Neighbour Graphs. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR).* 5713–5722. doi:10.1109/CVPR.2016.616

[29] Jui-Ting Huang, Ashish Sharma, Shuying Sun, Li Xia, David Zhang, Philip Pronin, Janani Padmanabhan, Giuseppe Ottaviano, and Linjun Yang. 2020. Embedding-based Retrieval in Facebook Search. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Virtual Event, CA, USA) *(KDD '20).* Association for Computing Machinery, New York, NY, USA, 2553–2561. doi:10.1145/3394486.3403305

[30] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. 2019. DiskANN: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2019/file/09853c7fb1d3f8ee67a61b6bf4a7f8e6-Paper.pdf

[31] Hervé Jégou, Romain Tavenard, Matthijs Douze, and Laurent Amsaleg. 2011. Searching in one billion vectors: re-rank with source coding. In *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP).* IEEE, 861–864.

[32] Chao Jin, Zili Zhang, Xuanlin Jiang, Fangyue Liu, Xin Liu, Xuanzhe Liu, and Xin Jin. 2024. RAGCache: Efficient Knowledge Caching for Retrieval-Augmented Generation. *CoRR* abs/2404.12457 (2024). arXiv:2404.12457 doi:10.48550/ARXIV.2404.12457

[33] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2019. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data* 7, 3 (2019), 535–547.

[34] Herve Jégou, Matthijs Douze, and Cordelia Schmid. 2011. Product Quantization for Nearest Neighbor Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 33, 1 (2011), 117–128. doi:10.1109/TPAMI.2010.57

[35] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS '20).* Curran Associates Inc., Red Hook, NY, USA, Article 793, 16 pages.

[36] Sen Li, Fuyu Lv, Taiwei Jin, Guli Lin, Keping Yang, Xiaoyi Zeng, Xiao-Ming Wu, and Qianli Ma. 2021. Embedding-based Product Retrieval in Taobao Search. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining* (Virtual Event, Singapore) *(KDD '21).* Association for Computing Machinery, New York, NY, USA, 3181–3189. doi:10.1145/3447548.3467101

[37] Yuan Lin and Vinod Grover. 2025. Using CUDA Warp-Level Primitives. https://developer.nvidia.com/blog/using-cuda-warp-level-primitives/.

[38] Yuhan Liu, Hanchen Li, Yihua Cheng, Siddhant Ray, Yuyang Huang, Qizheng Zhang, Kuntai Du, Jiayi Yao, Shan Lu, Ganesh Ananthanarayanan, Michael Maire, Henry Hoffmann, Ari Holtzman, and Junchen Jiang. 2024. CacheGen: KV Cache Compression and Streaming for Fast Large Language Model Serving. In *Proceedings of the ACM SIGCOMM 2024 Conference* (Sydney, NSW, Australia) *(ACM SIGCOMM '24).* Association for Computing Machinery, New York, NY, USA, 38–56. doi:10.1145/3651890.3672274

[39] Yury Malkov, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. 2014. Approximate nearest neighbor algorithm based on navigable small world graphs. *Information Systems* 45 (2014), 61–68. doi:10.1016/j.is.2013.10.006

[40] Yu A. Malkov and D. A. Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Trans. Pattern Anal. Mach. Intell.* 42, 4 (apr 2020), 824–836. doi:10.1109/TPAMI.2018.2889473

[41] Hiroyuki Ootomo, Akira Naruse, Corey Nolet, Ray Wang, Tamas Feher, and Yong Wang. 2024. CAGRA: Highly Parallel Graph Construction and Approximate Nearest Neighbor Search for GPUs. In *2024 IEEE 40th International Conference on Data Engineering (ICDE).* 4236–4247. doi:10.1109/ICDE60146.2024.00323

[42] Zaid Qureshi, Vikram Sharma Mailthody, Isaac Gelado, Seungwon Min, Amna Masood, Jeongmin Park, Jinjun Xiong, C. J. Newburn, Dmitri Vainbrand, I-Hsin Chung, Michael Garland, William Dally, and Wen-mei Hwu. 2023. GPU-Initiated On-Demand High-Throughput Storage Access in the BaM System Architecture. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023).* Association for Computing Machinery, New York, NY, USA, 325–339. doi:10.1145/3575693.3575748

[43] Harsha Simhadri. 2024. Research talk: Approximate nearest neighbor search systems at scale. https://www.youtube.com/watch?v=BnYNdSIKibQ&list=PLD7HFcN7LXReJTWFKYqwMcCc1nZKIXBo9&index=9.

[44] Harsha Vardhan Simhadri, George Williams, Martin Aumüller, Matthijs Douze, Artem Babenko, Dmitry Baranchuk, Qi Chen, Lucas Hosseini, Ravishankar Krishnaswamny, Gopal Srinivasa, et al. 2022. Results of the NeurIPS'21 challenge on billion-scale approximate nearest neighbor search. In *NeurIPS 2021 Competitions and Demonstrations Track.* PMLR, 177–189.

[45] Ján Suchal and Pavol Návrat. 2010. Full Text Search Engine as Scalable k-Nearest Neighbor Recommendation System. In *Artificial Intelligence in Theory and Practice III*, Max Bramer (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 165–173.

[46] Bing Tian, Haikun Liu, Zhuohui Duan, Xiaofei Liao, Hai Jin, and Yu Zhang. 2024. Scalable Billion-point Approximate Nearest Neighbor Search Using SmartSSDs. In *2024 USENIX Annual Technical Conference (USENIX ATC 24).* USENIX Association, Santa Clara, CA, 1135–1150. https://www.usenix.org/conference/atc24/presentation/tian

[47] Bing Tian, Haikun Liu, Yuhang Tang, Shihai Xiao, Zhuohui Duan, Xiaofei Liao, Hai Jin, Xuecang Zhang, Junhua Zhu, and Yu Zhang. 2025. Towards High-throughput and Low-latency Billion-scale Vector Search via CPU/GPU Collaborative Filtering and Re-ranking. In *23rd USENIX Conference on File and Storage Technologies (FAST 25).* USENIX Association, Santa Clara, CA, 171–185. https://www.usenix.org/conference/fast25/presentation/tian-bing

[48] Karthik V., Saim Khan, Somesh Singh, Harsha Vardhan Simhadri, and Jyothi Vedurada. 2024. BANG: Billion-Scale Approximate Nearest Neighbor Search using a Single GPU. *CoRR* abs/2401.11324 (2024). arXiv:2401.11324 doi:10.48550/ARXIV.2401.11324

[49] Mengzhao Wang, Weizhi Xu, Xiaomeng Yi, Songlin Wu, Zhangyang Peng, Xiangyu Ke, Yunjun Gao, Xiaoliang Xu, Rentong Guo, and Charles Xie. 2024. Starling: An I/O-Efficient Disk-Resident Graph Index Framework for High-Dimensional Vector Similarity Search on Data Segment. *Proc. ACM Manag. Data* 2, 1, Article 14 (March 2024), 27 pages. doi:10.1145/3639269

[50] Yuming Xu, Hengyu Liang, Jin Li, Shuotao Xu, Qi Chen, Qianxi Zhang, Cheng Li, Ziyue Yang, Fan Yang, Yuqing Yang, Peng Cheng, and Mao Yang. 2023. SPFresh: Incremental In-Place Update for Billion-Scale Vector Search. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23).* Association for Computing Machinery, New York, NY, USA, 545–561. doi:10.1145/3600006.3613166

[51] Jintang Xue, Yun-Cheng Wang, Chengwei Wei, and C. C. Jay Kuo. 2024. Word Embedding Dimension Reduction via Weakly-Supervised Feature Selection. arXiv:2407.12342 [cs.CL] https://arxiv.org/abs/2407.12342

[52] Artem Babenko Yandex and Victor Lempitsky. 2016. Efficient Indexing of Billion-Scale Datasets of Deep Descriptors. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR).* 2055–2063. doi:10.1109/CVPR.2016.226

[53] Runjie Yu, Weizhou Huang, Shuhan Bai, Jian Zhou, and Fei Wu. 2025. AquaPipe: A Quality-Aware Pipeline for Knowledge Retrieval and Large Language Models. *Proc. ACM Manag. Data* 3, 1, Article 11 (Feb. 2025), 26 pages. doi:10.1145/3709661

[54] Minjia Zhang and Yuxiong He. 2019. GRIP: Multi-Store Capacity-Optimized High-Performance Nearest Neighbor Search for Vector Search Engine. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management* (Beijing, China) *(CIKM '19).* Association for Computing Machinery, New York, NY, USA, 1673–1682. doi:10.1145/3357384.3357938

[55] Qianxi Zhang, Shuotao Xu, Qi Chen, Guoxin Sui, Jiadong Xie, Zhizhen Cai, Yaoqi Chen, Yinxuan He, Yuqing Yang, Fan Yang, Mao Yang, and Lidong Zhou. 2023. VBASE: Unifying Online Vector Similarity Search and Relational Queries via Relaxed Monotonicity. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23).* USENIX Association, Boston, MA, 377–395. https://www.usenix.org/conference/osdi23/presentation/zhang-qianxi

[56] Zili Zhang, Fangyue Liu, Gang Huang, Xuanzhe Liu, and Xin Jin. 2024. Fast Vector Query Processing for Large Datasets Beyond GPU Memory with Reordered Pipelining. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 23–40. https://www.usenix.org/conference/nsdi24/presentation/zhang-zili-pipelining
[57] Weijie Zhao, Shulong Tan, and Ping Li. 2020. SONG: Approximate Nearest Neighbor Search on GPU. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. 1033–1044. doi:10.1109/ICDE48307.2020.00094
[58] Chun Jiang Zhu, Minghu Song, Qinqing Liu, Chloé Becquey, and Jinbo Bi. 2020. Benchmark on Indexing Algorithms for Accelerating Molecular Similarity Search. *J. Chem. Inf. Model.* 60, 12 (2020), 6167–6184. doi:10.1021/ACS.JCIM.0C00393

Received April 2025; revised July 2025; accepted August 2025