KBest: Efficient Vector Search on Kunpeng CPU

Kaihao Ma[†], Meiling Wang[†], Senkevich Oleg[†], Zijian Li[†], Daihao Xue[†], Malyshev Dmitriy[¶],
Yangming Lv[†], Shihai Xiao[†], Xiao Yan[‡], Radionov Alexander[†], Weidi Zeng[†], Yuanzhan Gao[†], Zhiyu
Zou[†], Xin Yao[†], Lin Liu[†], Junhao Wu[†], Yiding Liu[†], Yaoyao Fu[†], Gongyi Wang[†], Gong Zhang[†], Fei
Yi[†], Yingfan Liu[§]

[†]Huawei Technologies Ltd.   [‡] Wuhan University   [¶] Higher School of Economics   [§] Xidian University

Abstract

Vector search, which returns the vectors most similar to a given query vector from a large vector dataset, underlies many important applications such as search, recommendation, and LLMs. To be economic, vector search needs to be efficient to reduce the resources required by a given query workload. However, existing vector search libraries (e.g., Faiss and DiskANN) are optimized for x86 CPU architectures (i.e., Intel and AMD CPUs) while Huawei Kunpeng CPUs are based on the ARM architecture and competitive in compute power. In this paper, we present KBest as a vector search library tailored for the latest Kunpeng 920 CPUs. To be efficient, KBest incorporates extensive hardware-aware and algorithmic optimizations, which include single-instruction-multiple-data (SIMD) accelerated distance computation, data prefetch, index refinement, early termination, and vector quantization. Experiment results show that KBest outperforms SOTA vector search libraries running on x86 CPUs, and our optimizations can improve the query throughput by over 2x. Currently, KBest serves applications from both our internal business and external enterprise clients with tens of millions of queries on a daily basis.

1 Introduction

With the development of machine learning, many embedding models [4, 6, 35, 49] are proposed to map data objects (e.g., texts, images, videos, and molecules) to vectors that encode their semantics. On these embedding vectors, similarity is a key notion. For instance, two images look similar if their embeddings are similar (e.g., as measured by Euclidean distance), a text description matches a video if their have similar embeddings, and a user may like a product if their embeddings are similar. Therefore, vector search, which returns the vectors most similar to a given query vector from a vector dataset, is a basic operation on embeddings [27]. As shown in Figure 1, vector search underlies many important applications such as content search (e.g., for images and videos) [21, 24, 48], recommendation (e.g., for e-commerce or contents) [5, 17, 26, 43], medicine [7, 15, 39], finance [19], and even LLM chat-bots [29, 37]. Vector search usually needs to handle large vector datasets (e.g., with millions or even trillions of high-dimension vectors) [10] and
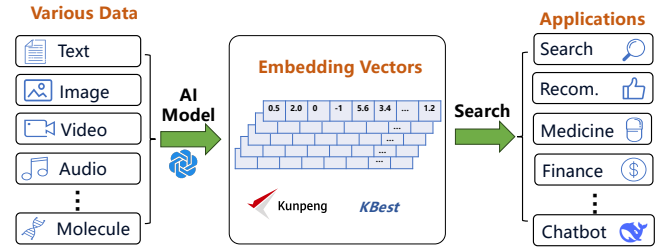
Figure 1: Vector embeddings and applications of vector search

meet stringent performance requirements (e.g., returning search results within 10ms and serving millions of queries per second) [28]. As such, vector search should be efficient (i.e., achieving a high query throughput) to reduce the required resources for serving a given workload and thus monetary cost.

Due to the importance of efficiency, there are several highly optimized libraries for vector search. For instance, HNSWlib [30] provides an efficient implementation for HNSW [31], which is a proximity graph index for vector search and shown to achieve good performance. Developed by Meta, FAISS [23] supports both inverted file (IVF) [21] and proximity graph as the indexes and allows to use vector quantization for efficient distance computation. DiskANN [41] is proposed by Microsoft and supports both memory-based and disk-based vector search with a novel proximity graph index called Vamana. Powered by Google, ScaNN [36] uses registers to compute vector distance efficiently via table lookup.

However, these vector search libraries target x86 (i.e., Intel and AMD) CPUs while ARM CPUs are becoming competitive. In particular, Huawei started the ARM-based Kunpeng CPU series [16, 18] in 2013 and the production lines maintains an annual capacity of hundreds of thousands of CPUs. With 80 cores at 2.9GHz, the latest Kunpeng 920 CPU matches AMD 9654 (with 96 cores and 2.4GHz) in compute power. Currently, Kunpeng 920 CPUs are widely deployed for both our internal business and external cloud services. As such, it is crucial to develop an efficient vector search library for Kunpeng CPUs in order to support the applications that rely on vector search. However, developing such a library is challenging because (i) ARM CPUs have different hardware characteristics and instruction sets from x86 CPUs, and thus a deep understanding of ARM CPUs and extensive engineering efforts are required; (ii) vector search has a long research history, and thus an extensive survey is required to understand the best practices and integrate them into our library.

In this paper, we present KBest (Kunpeng Blazing-fast embedding similarity search thruster) as an efficient vector search library tailored for ARM-based Huawei Kunpeng CPUs. KBest targets
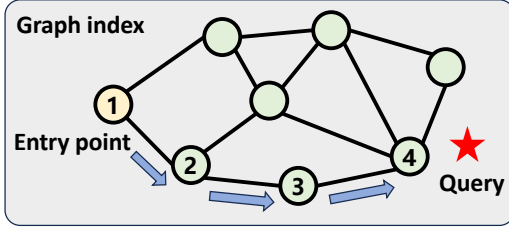
Figure 2: An illustration of vector search on proximity graph indexes, which traverses the graph to identify neighbors

---

Algorithm 1 Graph Traversal for Vector Search

1: **Input:** Graph G, query q, result count k, queue size L
2: **Output:** k similar vectors to q
3: Initialize a size-L priority queue Q with $(v_1, \|v_1 - q\|)$
4: **while** Q has unvisited node **do**
5:     Read the most similar but unvisited node v in Q
6:     **for** each neighbor u of v in G **do**
7:         **if** distance $\|q - u\|$ is not computed **then**
8:             Compute $\|q - u\|$
9:             Try to insert $(u, \|q - u\|)$ into Q
10: **return** The k vectors with the smallest distances in Q

---

memory-based vector search, which assumes that the vectors fit in the main memory of a machine and is the most common scenario, and adopts proximity graph indexes, which achieve the best performance for vector search. To improve efficiency, KBest incorporates comprehensive hardware-aware and algorithmic optimizations. In particular, we leverage the single-instruction-multiple-data (SIMD) instructions of ARM to implement efficient vector distance computation, which is the basic operation in vector search, and conducts software prefetch to reduce the cache miss when accessing vectors following the edges of proximity graph indexes. We also use huge memory pages and align the vectors with cache lines to reduce memory management overheads. From the algorithm perspective, we introduce a refinement step, which checks the 2-hop neighbors in an existing proximity graph index to improve its quality, and propose a lightweight graph reordering algorithm to renumber the vectors for improved data access locality. Besides, we also design a method to terminate the processing of a query early when its neighbors have been identified and allow users to flexibly configure their vector quantization algorithms, which reduce distance computation complexity by using compressed vectors and are crucial for efficiency.

We evaluate KBest on 4 real-world vector datasets and compare with 3 SOTA x86-based vector search libraries. The results show that KBest running on Kunpeng 920 CPU outperforms existing vector search libraries running on AMD 9654 CPU, highlighting the huge performance potential of ARM-based platforms. Ablation studies also suggest that our optimizations are effective by improving the query throughput by over 2x. Currently, KBest is widely used from both our internal business and external users, serving tens of millions of queries in a daily basis.

To summarize, we make the following contributions:

- We design and implement KBest as the first efficient vector search library for ARM CPUs, incorporating both ARM-specific hardware optimizations and general algorithmic improvements.
- We design user-friendly API to allows users to easily use KBest and integrate with existing vector databases.
- Our optimizations encompass the best practices for CPU-based vector search and can guide followup works.

2 Preliminaries

In this part, we introduce the basics of vector search and Huawei Kunpeng CPUs to facilitate subsequent discussions.

2.1 Vector Search and Proximity Graph Index

Vector search, also known as nearest neighbor search (NNS), is usually defines as follows.

Definition 2.1. Given a query vector $\mathbf{q} \in \mathbb{R}^d$ and a vector dataset $\mathcal{X} = \{\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_n\} \subset \mathbb{R}^d$, find the set $\mathcal{N}_q \subset \mathcal{X}$ of the top-k nearest neighbors for $\mathbf{q}$ such that

$$|\mathcal{N}_q| = k \text{ and } \text{dist}(\mathbf{q}, \mathbf{x}) \leq \text{dist}(\mathbf{q}, \mathbf{x}') \ \forall \mathbf{x} \in \mathcal{N}_q, \mathbf{x}' \in \mathcal{X} - \mathcal{N}_q,$$

Here, $\text{dist}(\mathbf{q}, \mathbf{x})$ is a distance function, such as Euclidean distance (i.e., $\|\mathbf{q} - \mathbf{x}\|_2$) or negative inner product (i.e, $-\langle \mathbf{q}, \mathbf{x} \rangle$). As the dimension of embedding vectors are usually high (e.g., at hundreds), exact NNS requires a linear scan due to the curse of dimensionality [38]. To trade for efficiency, approximate NNS (ANNS) is usually used in practice, which returns most rather than all of the top-k nearest neighbors for each query. The quality of an approximate result set $\mathcal{N}'_q$ is typically measured by *recall*, which is defined as $|\mathcal{N}'_q \cap \mathcal{N}_q|/k$. Applications usually require a high recall (e.g., 0.9 or 0.95) for good result quality and a low query latency (e.g., <10ms) for good QoS. The performance of vector search is commonly measured by the QPS (query processing throughput) at specific recall levels.

Many algorithms and indexes have been designed for vector search including locality sensitive hashing (LSH) [13], tree-based data structures [3, 40], inverted file (IVF) [21], and proximity graph [20]. Proximity graph is the most efficient for high dimension embedding vectors in that it requires the fewest distance computations to reach the same recall. As shown in Figure 2, proximity graph organizes the vector dataset as a graph, where the nodes are vectors and edges connects similar vectors. The number of neighbors for each vector is usually limited by a small number M (e.g., 64). Vector search is conducted by the graph traversal procedure in algorithm 1. *candidate queue Q* is a minimum priority queue to manage the distances and return the unvisited node with the smallest distance. Search starts with a random or fixed entry node (i.e., v_1) and checks the most similar but unvisited node v by computing distances for v's neighbors. Vector search terminates when Q can no longer be updated and the finally line 10 returns the top-k results with small distances. A larger queue size L improves recall buy computing more distances but consumes longer search time. There are many variants of proximity graph, e.g., HNSW [31], NSG [12], Vamana [41], SSG [11], each with different edge selection rules during index building. They usually perform well for different datasets.

Table 1: Hardware specifications for Huawei Kunpeng 920 CPU and representative x86 CPUs

	Intel 8558p	AMD 9654	Kunpeng 920
Cores	48C	96C	80C
Threads	96T	192T	160T
Frequency	2.7GHz	2.4 GHz	2.9GHz
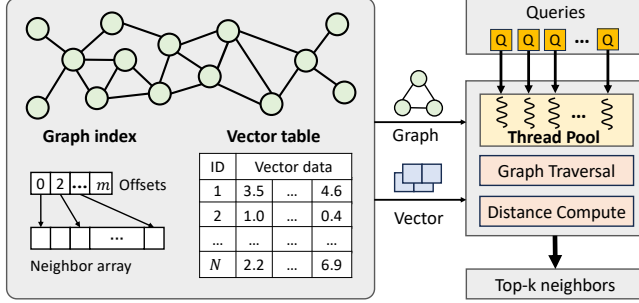
**Figure 3: The memory layout and workflow of KBest**

2.2 Huawei Kunpeng CPUs

Huawei's Kunpeng CPU series are based on the ARM architecture and have more than ten years of development history. The architectural evolution spans from the first-generation Hi1610 in 2013 to the current fourth-generation Kunpeng 920 in 2019, fabricated using 7nm process technology. As shown in Table 1, Kunpeng 920 CPU processors features up to 80-core configuration with 2.9GHz clock frequency, demonstrating competitive performance against SOTA X86 platform Intel Xeon and AMD EPYC processors. Currently Kunpeng 920 CPU processors have been widely adopted in many fields such as cloud computing infrastructures and database systems.

Instruction set of ARM CPUs. ARM-based Kunpeng CPUs and x86 processors exhibit similar functionalities but in different forms. For SIMD acceleration, Kunpeng leverages 128-bit NEON (ARM Advanced SIMD) and scalable bit-length SVE (Scalable Vector Extension) instructions, offering adaptive vectorization for irregular dimensions, contrasting with 256-bit AVX2 (Advanced Vector Extensions 2) and 512-bit AVX-512 (Advanced Vector Extensions 512) on x86 platforms. Both architectures employ hardware and software prefetching: hardware prefetching is automatically enforced without explicit instructions, while software prefetching can be implemented using intrinsics like *PLDL1KEEP* on Kunpeng 920 CPUs or *_mm_prefetch* on x86 platforms. Memory optimizations like huge pages and NUMA are similar, where huge pages can help improve translation lookaside buffer (TLB) coverage for large datasets and NUMA-aware allocation can help improve memory locality.

3 The KBest Library

Overview. KBest is an optimized graph-based ANNS algorithm specifically tuned for Kunpeng 920 CPUs, leveraging unique hardware features and algorithmic enhancements to achieve efficient search. As shown in Figure 1, the system employs an in-memory architecture with two key components: (1) a CSR-formatted graph index with fixed out-degree M (null-padded for uniform access),
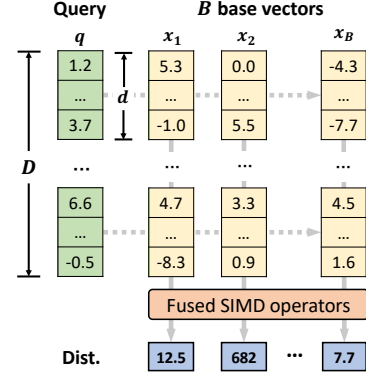
Figure 4: The abstraction of SIMD accelerated operators of fused 1-to-B distance computation

and (2) vector data stored in flattened, dimension-padded 1D arrays for memory alignment. KBest utilizes a dynamic thread pool to automatically distribute incoming queries to idle worker threads, enabling concurrent processing while maintaining the Kunpeng architecture's full computational potential through both hardware-aware adaptations and general algorithmic optimizations.

3.1 Kunpeng-aware Optimizations

SIMD accelerated distance computation. Distance computation is the basic operation for graph-based ANNS, primarily executed when evaluating neighbors against query vectors to update candidate sets. While developers typically leverage architecture-specific SIMD extensions (namely AVX/AVX512 on x86 and NEON/SVE on ARM), KBest introduces several Kunpeng hardware-specific optimizations. As showed in Figure 4, we optimize SIMD instruction parallelism by exploiting the Kunpeng 920 CPU's multi-issue architecture, which enables concurrent execution of independent SIMD operations. In particular, we transform scalar 1-to-1 distance calculations into batched 1-to-B vectorized operations, enabling up to 16 parallel distance computations per cycle when data dependencies permit. This approach fully saturates 128-bit NEON registers to maximize CPU utilization while amortizing memory access latency through query vector reuse across B database items. For smaller workloads where batched processing is inefficient, we implement pipelined segmented accumulation for 1-to-1 distance calculations through carefully scheduled instruction streams. Second, we leverage some effective built-in fused operators to combine some fundamental operations into single SIMD intrinsics. For example, with float-type vectors, we utilize the `vmlaq_f32` NEON instruction to fuse multiply-accumulate operations, reducing instruction count and improving pipeline efficiency.

Data prefetch. Modern CPUs employ a hierarchical memory architecture where data must traverse multiple cache levels before reaching registers for computation: from main memory to L3 cache, then L2, and finally L1. While modern processors implement hardware prefetch that predicatively load memory blocks based on access patterns, these mechanisms prove inadequate for graph-based ANNS algorithms. The irregular access patterns inherent to graph traversal due to the random access following the outgoing edges of the graph result in significant memory bottlenecks.

To address this challenge, KBest introduces a pipeline software prefetching strategy that loads data ahead of computation.

Figure 5 shows our prefetch strategy in the search process, where KBest prefetches the adjacent list and vector data of the top priority node in the candidate set. This is because these nodes represent the immediate traversal targets in subsequent iterations. In each search iteration upon extracting the nearest node from the priority queue, the system prefetches a batch of B neighbor nodes (where B is determined by cache constraints) while concurrently processing the current node's neighbors. This batch prefetching continues until accumulating B neighbors, at which point KBest performs a batched one-to-many distance computation and inserts qualifying neighbors back into the candidate set. The prefetch batch size B is determined by the cache-aware formulation:

$$B = \left\lfloor \frac{\alpha \cdot C_{\text{L1d}}}{d \cdot s} \right\rfloor \qquad (1)$$

where C_{L1d} denotes the per-thread L1 data cache size, d represents the vector dimensionality, s is the element size in bytes (e.g., 4 bytes for float32), and α is the cache allocation ratio for prefetching (typically 0.5 for optimal compute-prefetch overlap).

Our implementation leverages ARM's low-level prefetch instruction through inline assembly assembly *asm volatile("prfm PLD1KEEP, [%0]" :: "r"(address))*. Here *prfm* is an ARMv8 assembly instruction for cache prefetching, *PLD1KEEP* operand specifies a long-term prefetch policy, instructing the memory subsystem to retain the prefetched data in cache hierarchy rather than treating it as transient and *address* is the aligned memory location targeted for prefetching. This approach effectively bridges the latency gap between unpredictable memory accesses and computational pipelines.

Memory management. KBest implements dual-layer memory optimization to address the bandwidth-bound nature of graph-based ANNS on Kunpeng processors: At the virtual memory level, the system enforces 2MB huge page allocation through explicit `madvise(MADV_HUGEPAGE)` directives and the system-level configure `/sys/kernel/mm/transparent_hugepage/enabled`. This addresses the performance penalty caused by conventional 4KB page sizes, which induces significant Translation Lookaside Buffer (TLB) misses during random graph traversal, which is a critical bottleneck where adjacency lists and feature vectors may span hundreds of memory pages. Through contiguous physical mappings of huge pages, KBest can reduce TLB miss rates while improving row buffer hit rates on Kunpeng's 8-channel DDR4 memory subsystem.

At the cache level, KBest guarantees 64-byte aligned memory allocation for all critical data structures (including graph edges and feature vectors) via *std::align_alloc*, with each vector dimension padded to cache line boundaries. This alignment serves two purposes: (i) eliminating cross-cache-line access penalties during SVE/NEON vectorized distance computations, and (ii) minimizing cache coherence overhead through natural partition alignment.

3.2 Algorithmic Optimizations

Index refinement. KBest builds upon state-of-the-art graph construction strategies (e.g., NSG, SSG and Vamana) through a three-phase pipeline that enhances both construction efficiency and search performance. First, a k-nearest neighbors (kNN) graph is
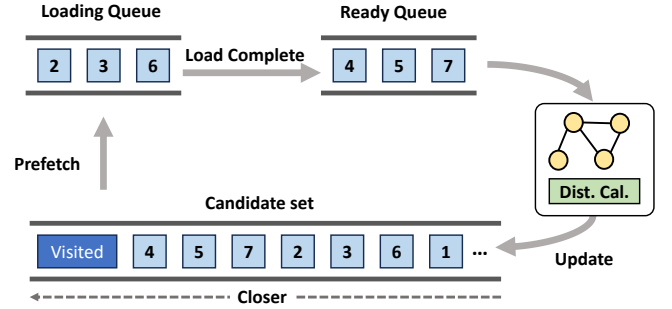

Figure 5: The workflow of KBest's prefetch strategy

constructed where each vertex's neighborhood contains its nearest vectors from the dataset. We employ the advanced RNNDescent [32] algorithm to generate the initial kNN graph for its high efficiency and quality guarantee. Second, we refine each vertex's neighborhood based on search results from the initial kNN graph, ensuring the edges cover diverse directions in the vector space. To accommodate different dataset characteristics, we support several edge selection strategies in SOTA graph index. For instance, HNSW, NSG with their distance-based strategies and SSG with its angle-based selection rules. Additionally, we introduce a novel iterative refinement strategy, during each iteration, we expand candidate to include all 2-hop neighbors for each node and reapply our edge selection rules. This process continues for F iterations until the graph stabilizes or the construction time budget is exhausted. This optimization can help shorten the search path on graph and improve efficiency.

Graph reordering. Graph reordering is a cache optimization technique that improves memory locality by placing neighboring nodes in consecutive or near-consecutive memory locations. When a node and its associated vector data are loaded into memory, modern CPUs often prefetch adjacent memory blocks. By reordering nodes so that likely traversal paths correspond to spatially adjacent memory, the algorithm benefits from higher cache hit rates.

To formally define the optimization goal of graph reordering, we adopt the well-known graph bandwidth minimization problem. Given a graph $G = (V, E)$ with vertices $V = \{v_1, v_2, \ldots, v_n\}$, the objective is to find a bijection $\pi : V \rightarrow \{1, 2, \ldots, n\}$ that maps each node to a unique memory position such that the maximum distance between connected nodes is minimized:

$$\min_{\pi} \max_{(v_i, v_j) \in E} |\pi(v_i) - \pi(v_j)| \qquad (2)$$

Here, $\pi(v_i)$ and $\pi(v_j)$ denote the memory positions of nodes v_i and v_j. Minimizing this objective places connected nodes closer in memory, improving spatial locality and reducing cache misses during graph traversal.

Unfortunately, the graph bandwidth minimization problem is known to be NP-hard, making it unlikely to admit a polynomial-time exact solution. While heuristic methods such as Cuthill-McKee [8] and Gorder [46] have been proposed, we observe that these remain suboptimal for graph index traversal in ANNS. This limitation stems from fundamental incompatibilities: ANNS graphs exhibit small-world properties characterized by densely clustered local connections and sparse long-range shortcuts, forming heterogeneous

---

Algorithm 2 Graph Node Reordering for Memory Locality

1: **Input:** Original graph G, entry node e
2: **Output:** Reordered node sequence S
3: Generate minimum spanning tree (MST) T from G
4: Select the entry node e as the root of T
5: Initialize $\text{met}(v) \leftarrow 1, \forall v \in T$
6: Initialize empty stack and visited set
7: Push (e, False) onto stack
8: **while** stack is not empty **do**
9:     $(u, processed) \leftarrow \text{stack.pop}()$
10:     **if** processed **then**
11:         **for** each child v of u in T **do**
12:             $\text{met}(u) \leftarrow \text{met}(u) + \text{met}(v)$
13:     **else**
14:         Push (u, True) onto stack
15:         **for** each child v of u in T (in reverse order) **do**
16:             Push (v, False) onto stack   ▷ DFS visit order
17: Initialize empty priority queue Q and list S
18: Push root node into Q
19: **while** Q is not empty **do**
20:     $u \leftarrow Q.\text{pop}()$ ▷ Dequeue max $\text{met}(v)$
21:     $S.\text{append}(u)$
22:     **for** each child v of u in T **do**
23:         $Q.\text{push}(v, \text{key} = \text{met}(v))$
24: **return** S

---

topologies that conventional algorithms fail to preserve. Specifically, Cuthill-McKee is designed for matrix bandwidth minimization—disrupts critical ANNS shortcuts during BFS traversal through long-range label jumps, artificially elongating search paths. Similarly, Gorder's cache-locality optimization may forcibly co-locate topologically connected but geometrically distant nodes in high-dimensional space, violating the underlying data geometry essential for ANNS performance.

Considering this important long-range shortcut edges, we propose a specialized graph reordering algorithm tailored for ANNS in Algorithm 2. In algorithm 2 we first constructs a minimum spanning tree (MST) T from the original graph G to establish connectivity and select the entry node e. Second, line 5-16 computes subtree sizes (met(v)) for all nodes via an iterative depth-first search (DFS), where each node's metric aggregates its subtree cardinality. Finally, line 17-23 performs a prioritized traversal that processes nodes in descending order of their subtree sizes, effectively clustering densely connected regions together in memory. This approach reduces cache misses by ensuring frequently co-accessed nodes (those in large subtrees) are stored contiguously.

Early termination. Graph-based ANNS algorithms utilize Best-First Search (BFS) to traverse the graph structure while maintaining a fixed-size candidate list L to track potential nearest neighbors. The search terminates when all nodes in L have been visited, with the parameter L critically determining both search scope and recall accuracy. We find that the candidate list exhibits low utilization rates that may cause numerous invalid search paths, resulting in substantial computational overhead. To address this inefficiency,

we propose an early termination algorithm that dynamically determines when to halt the search. The algorithm tracks two critical metrics: The insertion position $p \in \mathbb{Z}^+$ of each new candidate in the candidate set and the number of consecutive insertions occurring beyond a threshold position t. The search terminates when consecutive insertions beyond t exceed $\tau_{\max}$:

$$\text{EarlyTerm.}(t, \tau_{\max}) := \left\lceil \sum_{i=k-\tau_{\max}}^{k} \mathbb{I}(p_i > t) \right\rceil \geq \tau_{\max} \quad (3)$$

The early-termination heuristic relies on the observation that when consecutive unvisited nodes consistently rank near the end of the candidate list (i.e., farthest from the query), the search is likely diverging from the query's neighborhood. Here the threshold position t and patience $\tau_{\max}$ are tunable parameters and their optimal values depend heavily on the dataset characteristics and recall requirements. To determine the best configuration, we perform a two-stage search with dry-run queries. In practice, we initialize with t as about 60% of L, the total search list size in BFS, and then conduct binary search for $\tau_{\max}$ under the given recall constraint. To further optimize search performance while maintaining recall, we explore reducing t from 60% down to 30% of L, identifying the setting that maximizes search speed without compromising recall.

Vector quantization. This technique compresses high-dimensional data by mapping vectors to discrete codewords from a learned codebook. The process dramatically reduces storage and computation costs while preserving relative similarity, making it ideal for large-scale search systems that operate on compressed representations instead of raw data. Two widely-used variants include Product Quantization (PQ) and Scalar Quantization (SQ), which PQ divides the vector space into orthogonal subspaces for independent quantization, and SQ operates by independently quantizing each vector component to a fixed scalar range.

Notably, KBest's quantization module is implemented as a standalone component with well-defined interfaces, enabling seamless integration of vector quantization algorithms to further enhance retrieval efficiency and accuracy without architectural modifications. This modular design ensures forward compatibility with emerging quantization techniques while maintaining the system's core optimization pipeline.

4 API and Use Cases of KBest

In this section, we present KBest's API design and demonstrate its deployment options, including usage as a standalone library and integration with Milvus [44] as an ANNS algorithm component.

4.1 The API of KBest

KBest is implemented in C++ with approximately 9K lines of code, and provides user-friendly interfaces in both C++ and Python. Like other mainstream libraries, it organizes the ANNS workflow into three stages: parameter preparation, index construction, and query processing. In the first stage, users initialize a KBest instance with specified parameters. During index construction, base vectors and graph-building parameters are provided to build the index. In the final stage, KBest answers queries using the constructed graph and vector data. Table 2 summarizes the key APIs supporting these steps.

Table 2: Key APIs of KBest

Step	KBest API	Interpretation
Parameter prepare	*KBest(config)*	Initalizea KBest with detailed configuration
Index construct	*Add(n, x)*	build the graph index with n input index
Query process	*Search(nq, q, k, nt)*	search top-k NN of *nq* queries with *nt* threads

To avoid redundant index building, KBest also supports saving and loading pre-built graph indices.

KBest seamlessly integrates with vector databases like Milvus [44] through the Knowhere computation layer, which orchestrates the entire workflow by calling KBest's specific interfaces. During the build phase, Knowhere invokes KBest's graph construction API to incrementally build the index, which is then serialized along with the user data for storage. When executing queries, Knowhere manages the complete search pipeline by loading indices through KBest's deserialization interface, performing efficient top-k search via its optimized query interface, and delivering results to Milvus' distributed query coordinator. Through this tight integration, Knowhere ensures KBest maintains its native performance while fully leveraging Milvus' distributed architecture and hardware acceleration capabilities.

4.2 Use Cases of KBest

KBest has been widely adopted in large-scale industrial applications, demonstrating exceptional scalability and efficiency in real-world scenarios. Notably, it has been successfully deployed in top-tier social media, e-commerce, and food delivery platforms, processing tens of millions of daily queries across massive server clusters while maintaining millisecond-level latency. In social media applications, user-generated content including images and short videos is automatically encoded into dense vector representations through advanced AI models, while user preferences are simultaneously converted into query vectors, enabling real-time personalized content recommendations through high-performance similarity search.

For e-commerce platforms, the system efficiently transforms product information including titles, descriptions and images into item vectors, while converting user search queries (such as "wireless headphones") into corresponding query vectors. This allows KBest to instantly retrieve the most relevant products from its large-scale indexes, delivering low-latency recommendations even during peak traffic periods, handling millions of concurrent requests. In the food delivery sector, restaurant menus, dish images and location-based user queries are intelligently mapped to a unified vector space. KBest's optimized architecture enables instantaneous search and matching of dishes with consistent response times less than 5ms.

Additionally, KBest has been integrated into mainstream vector databases including Milvus [44] and OpenGauss [33], enhancing their vector search capabilities. Beyond commercial platforms, KBest serves national infrastructure projects by enabling rapid similarity search across satellite imagery and sensor data vectors, while telecom companies leverage it to match network patterns and

Table 3: Statistics of the vector datasets used for experiments.

Dataset	Num	Dim	Similarity
Glove	1M	100	Angular
Deep	10M	96	Angular
Text-to-Image	10M	200	Inner-Product
BigANN	100M	128	L2

troubleshoot issues from encoded operational data. These mission-critical deployments utilize thousands of servers processing tens of millions of queries daily while maintaining high availability, security, and performance standards.

5 Experimental Evaluation

We conduct comprehensive experiments to evaluate KBest's performance on Huawei Kunpeng 920 CPUs against state-of-the-art (SOTA) graph-based ANNS algorithms on x86 platforms. We further conduct an ablation study that quantifies the individual contribution of each design to the performance gains.

5.1 Experiment Settings

Datasets. We evaluate KBest on four standard benchmark datasets: Glove [34], Deep [2], Text-to-Image [9], and BigANN [1, 25]. We summarize the dataset characteristic in Table 3. To comprehensively show the performance of varying scales and cases, we use a sampled 1M subset of dataset of Glove and a 10M subset of dataset of Deep. In Text-to-Image dataset, the distribution of queries is different from the input data and we verify the capability of KBest of handling out-of-distribution dataset. Finally, to show the scalability of KBest, we evaluate the performance on the BIGANN of 100M scale.

Baselines. We compare our KBest approach with three representative graph index types: HNSW, NSG, and Vamana, using their respective state-of-the-art implementations which includes:

- **HNSW** [23]: We use the Faiss library for HNSW. Faiss is developed by Meta's Fundamental AI Research (FAIR) team and provides highly optimized implementations for most useful SOTA vector search algorithms. The implementation leverages multiple acceleration libraries including MKL and OpenBLAS, combined with low-level optimizations of graph structures and search procedures to achieve outstanding query performance.

- **NSG** [45]: For NSG implementation, we employ Zilliz's Pyglass library, a lightweight solution that implements NSG and other graph indexes without any third-party dependencies. The implementation demonstrates sophisticated memory management and optimized data structure design that significantly reduces memory footprint while maintaining high search accuracy.

- **Vamana** [41]: We utilize the in-memory version from Microsoft's DiskANN project, which pioneered an innovative two-pass edge selection strategy with adjustable parameter α to optimize graph density. DiskANN's in-memory implementation preserves all Vamana's algorithmic advantages while optimizing for RAM-based operation through compressed data representation and efficient memory access patterns, achieving excellent query performance in memory-resident environments.
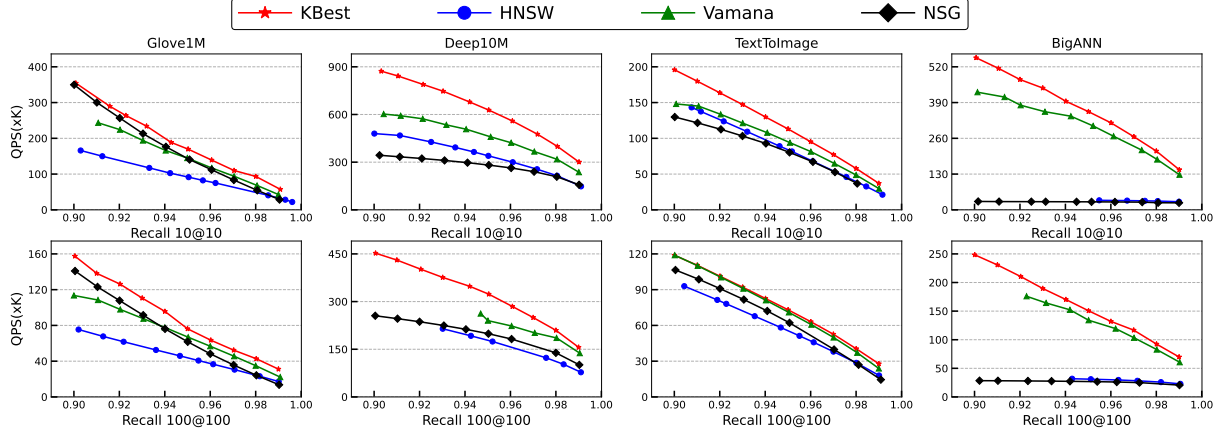
Figure 6: QPS vs. Recall of KBest on Kunpeng 920 compared with baselines on AMD 9654

Table 4: The QPS of KBest on Kunpeng 920 and baselines on AMD 9654 at recall 0.95

Datasets	QPS (xK) Recall@10=0.95				QPS (xK) Recall@100=0.95			
	Glove	Deep	T2Img	B-ANN	Glove	Deep	T2Img	B-ANN
NSG	140	282	80	29	62	199	62	27
HNSW	91	340	82	35	41	174	51	31
Vamana	145	459	94	305	67	240	71	134
KBest	**170**	**628**	**113**	**357**	**76**	**323**	**73**	**151**
Comparison	1.17x	1.37x	1.20x	1.17x	1.14x	1.35x	1.03x	1.12x

For each method, we adopt the default graph construction parameters recommended by their authors, including the number of neighbors per node M and the search list size L during index building. After constructing the graph indices in advance for each dataset, we vary the search list size efs to explore different throughput−recall trade-offs.

Platform and performance metrics. We evaluate KBest on an ARM-based platform and compare it against other baselines running on x86 platforms. The primary ARM testbed is equipped with 2.9GHz Huawei Kunpeng 920 processors, running OpenEuler 22.04. For the SOTA baselines, we use a powerful x86 platform with an AMD EPYC 9654 processor overclocked to 3.7GHz, running Ubuntu 22.04. To ensure a fair comparison, we enable hyper-threading and utilize all available threads to handle queries in parallel.

We focus on evaluating both the efficiency and accuracy of each ANNS algorithm. Specifically, we use full-machine query-per-second (QPS) as the performance metric, and recall@10 and recall@100 (i.e., 10@10 and 100@100 recall) as the accuracy metrics.

5.2 Main Results

Figure 6 shows the QPS-recall curves comparing KBest with other x86-based baselines, while Table 4 provides detailed QPS measurements under a 0.95 recall requirement. The results demonstrate that in the high-recall range of 0.90-1.0, KBest achieves 1.04x−1.34x higher performance than the best baseline (Vamana) and up to 12.6x improvement over other baselines across all four datasets.

This performance advantage stems from Kunpeng-specific hardware optimizations combined with algorithmic enhancements to graph index quality and search efficiency.

We first examine the a million-scale Glove dataset. As showed in Figure 6, under low recall requirements, KBest and other baselines retrieve correct results efficiently, resulting in only modest performance gains: KBest achieves 1.17× and 1.14× higher QPS than the best baseline Vamana at recall@10=0.95 and recall@100=0.95, respectively in Table 4. However, as the recall target increases toward near-exact, the QPS of all methods drops significantly. In this high-recall regime, KBest's advantage becomes more pronounced, reaching up to 1.4× the performance of Vamana. This is attributed to the dense distribution of the Glove dataset that achieving near-exact recall requires searching a large number of vectors around the query, which introduces many redundant paths. KBest's early termination mechanism effectively prunes these redundant paths, leading to improved efficiency.

Next, we evaluate the significantly larger Deep dataset. Despite its size, the overall QPS is notably higher. This is because Deep better reflects the distribution of real-world datasets, leading to shorter search paths in the graph index across all methods. Additionally, the dataset has relatively low dimensionality, which amplifies the impact of memory access bottlenecks commonly seen in graph-based ANNS. To address this, KBest incorporates several memory efficiency optimizations, including prefetch pipelines and an optimized memory layout to reduce cache misses, as well as graph reordering techniques that convert random access patterns into
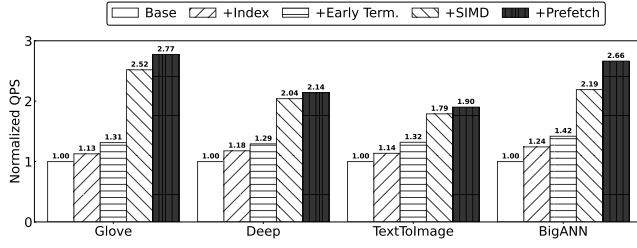
Figure 7: Ablation study of KBest

relatively sequential ones. These strategies significantly improve memory locality and utilization. As a result, KBest achieves up to 1.45× times QPS compared to the best baseline Vamana.

To evaluate the robustness of KBest, we assess its performance on an out-of-distribution (OOD) scenario using the Text-to-Image dataset, where the query vector distribution differs from that of the training data used to construct the graph index. As shown in Table 4, KBest maintains superior performance at recall@10=0.95 and recall@100=0.95, achieving 1.20× and 1.03× QPS respectively compared to the best baseline Vamana. Figure 6 demonstrates that across the high-recall range of 0.9-1.0, KBest delivers consistently better performance, with average QPS of 1.22× for recall@10 and 1.05× for recall@100 over Vamana, indicating its strong generalization capability under distribution shift.

To evaluate the scalability of KBest and other baselines, we conduct experiments on the 100M-scale BigANN dataset. Compared to previous small datasets, BigANN presents more severe challenges for graph traversal due to its larger size and the increased bottleneck of random memory access. As shown in Figure 6 and Table 4, baselines HNSW by Faiss and NSG by PyGlass exhibit significantly degraded performance, with only marginal improvements in recall as the search parameter efs increases. This limitation arises because these methods are primarily optimized for smaller datasets. At the billion-scale, their performance is heavily constrained by memory access overhead, dominating the total search time. In contrast, Vamana by DiskANN is optimized for disk-based large-scale search, achieving much higher QPS on BigANN. Notably, KBest still outperforms Vamana, delivering average QPS of 1.22× and 1.24× at recall@10 and recall@100.

5.3 Ablation Study

To assess the impact of individual optimizations, we conduct an ablation study by progressively enhancing the base version of KBest: starting with the unoptimized baseline(Base), we first incorporate graph index refinements (+Index), then introduce early termination (+Early Term.), followed by SIMD acceleration (+SIMD), and finally integrate prefetching (+Prefetch) to arrive at the fully optimized KBest.

As shown in Figure 7, all four optimizations demonstrate consistent performance gains across these 4 datasets. The graph index optimization yields improvements ranging from 12.8% (Glove) to 24.3% (BigANN), with greater benefits observed at larger scales due to increased optimization potential in the index structure. Early termination achieves performance gains of up to 16% on datasets like Glove and Text-To-Image. This improvement stems from its ability to prune redundant search paths, particularly effective for

non-standard data distributions (e.g., Glove) and out-of-distribution cases (e.g., Text-to-Image). The most significant improvements come from SIMD optimization, delivering average 60% and up to 92% acceleration by addressing the computational bottleneck in distance calculations through our proposed fused SIMD operators and 1-to-B vectorized operations, which maximize parallelization efficiency on Kunpeng CPUs. Finally, prefetching optimization specifically targets memory access bottlenecks in large graphs, achieving more than 20% performance boost on the billion-scale BigANN dataset.

6 Related Work

Vector indexes. Over the past few decades, various vector indexing methods have been developed to efficiently solve the Approximate Nearest Neighbor Search (ANNS) problem: the hashing-based methods (e.g., Local Sensitive Hashing (LSH) [13] and Spectral Hashing [47]) split the dataset and index vectors via hash tables using distance-preserving hash functions. Tree-based methods (e.g., KD-tree [40] and R-tree [3]) recursively organize vectors into hierarchical tree structures with spatial partitioning. Inverted File (IVF) methods, including IVFPQ [22] and Scann [36], first partition the dataset via clustering and build inverted indices for coarse-to-fine search. Our KBest algorithm is a graph-based ANNS method that constructs optimized proximity graphs for greedy traversal. Through custom graph refinement techniques, it achieves superior search efficiency and accuracy compared to state-of-the-art graph indexes including HNSW [31], Vamana [41], and SSG [11].

Vector search libraries. The exponential growth of vector data has driven significant advances in modern search systems. These libraries combine innovative indexing structures with hardware-conscious optimizations to deliver high performance under rigorous recall requirements. FAISS [23], developed by Meta, implements IVF, PQ, and HNSW indices along with clustering algorithms for efficient vector search, while leveraging GPU acceleration for enhanced throughput. Microsoft's DiskANN [41] employs SSD-optimized Vamana graphs with PQ compression, reducing memory overhead through intelligent disk-memory hierarchy management. Google's Scalable Nearest Neighbors (ScaNN) [36] utilizes anisotropic vector quantization [14] and SOAR [42] techniques to optimize both inner-product and distance-based searches without compromising recall accuracy. While these state-of-the-art systems are primarily optimized for x86 architectures, our KBest is specifically designed for Kunpeng 920 CPUs, incorporating both hardware-aware and algorithmic optimization to outperform other libraries on x86 CPUs.

7 Conclusions

We present KBest, an efficient graph-based vector search library specifically optimized for Huawei Kunpeng 920 CPUs. We find that existing state-of-the-art vector search libraries primarily target x86 platforms and exhibit suboptimal performance on ARM architectures. To bridge this performance gap, we implement ARM-specific hardware optimizations, including accelerated SIMD operators, software prefetching, and memory-related enhancements. Additionally, we introduce general algorithmic improvements such as graph index refinement and early termination during the search process. KBest offers a user-friendly API and has been widely adopted, both within our internal business and by external enterprises.

References

[1] Big ann-benchmarks Team. 2021. BigANN Benchmark: NeurIPS'21 Track. https://big-ann-benchmarks.com/neurips21.html. Accessed: 2023-11-15.

[2] Artem Babenko and Victor Lempitsky. 2016. Efficient indexing of billion-scale datasets of deep descriptors. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2055–2063.

[3] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, USA, May 23-25, 1990*, Hector Garcia-Molina and H. V. Jagadish (Eds.). ACM Press, 322–331. doi:10.1145/93597.98741

[4] Jianlv Chen, Shitao Xiao, Peitian Zhang, Kun Luo, Defu Lian, and Zheng Liu. 2024. BGE M3-Embedding: Multi-Lingual, Multi-Functionality, Multi-Granularity Text Embeddings Through Self-Knowledge Distillation. *CoRR* abs/2402.03216 (2024). arXiv:2402.03216 doi:10.48550/ARXIV.2402.03216

[5] 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 Neighborhood Search. In *Advances in Neural Information Processing Systems*, M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan (Eds.), Vol. 34. Curran Associates, Inc., 5199–5212. https://proceedings.neurips.cc/paper_files/paper/2021/file/299ddc35e747eb77177d9cea10a802da2-Paper.pdf

[6] Yiqun Chen and James Zou. 2024. GenePT: a simple but effective foundation model for genes and cells built from ChatGPT. *bioRxiv* (2024), 2023–10.

[7] Juan Carlos Cuevas-Tello, Daniel Hernández-Ramírez, and Christian Alberto Garcia-Sepulveda. 2013. Support vector machine algorithms in the search of KIR gene associations with disease. *Comput. Biol. Medicine* 43, 12 (2013), 2053–2062. doi:10.1016/J.COMPBIOMED.2013.09.027

[8] Elizabeth H. Cuthill and J. McKee. 1969. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 24th national conference, ACM 1969, USA, 1969*, Solomon L. Pollack, Thomas R. Dines, Ward C. Sangren, Norman R. Nielsen, William G. Gerkin, Alfred E. Corduan, Len Nowak, James L. Mueller, Joseph Horner III, Pasteur S. T. Yuen, Jeffery Stein, and Margaret M. Mueller (Eds.). ACM, 157–172. doi:10.1145/800195.805928

[9] Artem Babenko Dmitry Baranchuk. 2021. Text-to-Image dataset for billion-scale similarity search. Retrieved April 13, 2025 from https://research.yandex.com/datasets/text-to-image-dataset-for-billion-scale-similarity-search

[10] Magdalen Dobson, Zheqi Shen, Guy E. Blelloch, Laxman Dhulipala, Yan Gu, Harsha Vardhan Simhadri, and Yihan Sun. 2023. Scaling Graph-Based ANNS Algorithms to Billion-Size Datasets: A Comparative Analysis. *CoRR* abs/2305.04359 (2023). arXiv:2305.04359 doi:10.48550/ARXIV.2305.04359

[11] Cong Fu, Changxu Wang, and Deng Cai. 2022. High Dimensional Similarity Search With Satellite System Graph: Efficiency, Scalability, and Unindexed Query Compatibility. *IEEE Trans. Pattern Anal. Mach. Intell.* 44, 8 (2022), 4139–4150. doi:10.1109/TPAMI.2021.3067706

[12] 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 (2019), 461–474. doi:10.14778/3303753.3303754

[13] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. 1999. Similarity Search in High Dimensions via Hashing. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie (Eds.). Morgan Kaufmann, 518–529. http://www.vldb.org/conf/1999/P49.pdf

[14] Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. 2020. Accelerating Large-Scale Inference with Anisotropic Vector Quantization. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event (Proceedings of Machine Learning Research, Vol. 119)*. PMLR, 3887–3896. http://proceedings.mlr.press/v119/guo20h.html

[15] Lee Harris, Philippe De Wilde, and James Bentham. 2025. Comparing Lexical and Semantic Vector Search Methods When Classifying Medical Documents. *CoRR* abs/2505.11582 (2025). arXiv:2505.11582 doi:10.48550/ARXIV.2505.11582

[16] HiSilicon. 2024. Kunpeng 920 Chipset. https://www.hisilicon.com/en/products/kunpeng/huawei-kunpeng/huawei-kunpeng-920. Accessed: 2025-07-29.

[17] 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*. 2553–2561.

[18] Huawei Technologies Co., Ltd. 2023. Kunpeng Computing. https://www.hikunpeng.com/zh. Accessed: 2023-12-01.

[19] Giorgos Iacovides, Wuyang Zhou, and Danilo Mandic. 2025. FinDPO: Financial Sentiment Analysis for Algorithmic Trading through Preference Optimization of LLMs. arXiv:2507.18417 [cs.CL] https://arxiv.org/abs/2507.18417

[20] Piotr Indyk and Rajeev Motwani. 1998. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing* (Dallas, Texas, USA) *(STOC '98)*. Association for Computing Machinery, New York, NY, USA, 604–613. doi:10.1145/276698.276876

[21] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. 2011. Product Quantization for Nearest Neighbor Search. *IEEE Trans. Pattern Anal. Mach. Intell.* 33, 1 (2011), 117–128. doi:10.1109/TPAMI.2010.57

[22] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2021. Billion-Scale Similarity Search with GPUs. *IEEE Trans. Big Data* 7, 3 (2021), 535–547. doi:10.1109/TBDATA.2019.2921572

[23] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2021. Billion-Scale Similarity Search with GPUs. *IEEE Transactions on Big Data* 7, 3 (2021), 535–547. doi:10.1109/TBDATA.2019.2921572

[24] Yannis Kalantidis and Yannis Avrithis. 2014. Locally optimized product quantization for approximate nearest neighbor search. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2321–2328.

[25] Hervé Jégou Laurent Amsaleg. 2010. Datasets for approximate nearest neighbor search. http://corpus-texmex.irisa.fr/ Accessed: 2025-04-17.

[26] Yejin Lee, Hyunji Choi, Sunhong Min, Hyunseung Lee, Sangwon Beak, Dawoon Jeong, Jae W. Lee, and Tae Jun Ham. 2022. ANNA: Specialized Architecture for Approximate Nearest Neighbor Search. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 169–183. doi:10.1109/HPCA53966.2022.00021

[27] Wen Li, Ying Zhang, Yifang Sun, Wei Wang, Mingjie Li, Wenjie Zhang, and Xuemin Lin. 2019. Approximate nearest neighbor search on high dimensional data—experiments, analyses, and improvement. *IEEE Transactions on Knowledge and Data Engineering* 32, 8 (2019), 1475–1488.

[28] Wen Li, Ying Zhang, Yifang Sun, Wei Wang, Mingjie Li, Wenjie Zhang, and Xuemin Lin. 2020. Approximate Nearest Neighbor Search on High Dimensional Data - Experiments, Analyses, and Improvement. *IEEE Trans. Knowl. Data Eng.* 32, 8 (2020), 1475–1488. doi:10.1109/TKDE.2019.2909204

[29] Shige Liu, Zhifang Zeng, Li Chen, Adil Ainihaer, Arun Ramasami, Songting Chen, Yu Xu, Mingxi Wu, and Jianguo Wang. 2025. TigerVector: Supporting Vector Search in Graph Databases for Advanced RAGs. arXiv:2501.11216 [cs.DB] https://arxiv.org/abs/2501.11216

[30] Yu. A. Malkov and D. A. Yashunin. 2018. hnswlib: Header-only C++ library for fast approximate nearest neighbors search. https://github.com/nmslib/hnswlib.

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

[32] Naoki Ono and Yusuke Matsui. 2023. Relative NN-Descent: A Fast Index Construction for Graph-Based Approximate Nearest Neighbor Search. In *Proceedings of the 31st ACM International Conference on Multimedia, MM 2023, Ottawa, ON, Canada, 29 October 2023- 3 November 2023*, Abdulmotaleb El-Saddik, Tao Mei, Rita Cucchiara, Marco Bertini, Diana Patricia Tobon Vallejo, Pradeep K. Atrey, and M. Shamim Hossain (Eds.). ACM, 1659–1667. doi:10.1145/3581783.3612290

[33] openGauss Community. 2023. openGauss: An Open-Source Enterprise-Class Relational Database. Official Website. https://opengauss.org/en/ An open-source RDBMS supported by Huawei.

[34] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. GloVe: Global Vectors for Word Representation. https://nlp.stanford.edu/projects/glove/. Accessed: 2025-07-31.

[35] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. 2021. Learning transferable visual models from natural language supervision. In *International conference on machine learning*. PmLR, 8748–8763.

[36] Google Research. 2020. ScaNN: Efficient Vector Similarity Search.

[37] Bhaskarjit Sarmah, Benika Hall, Rohan Rao, Sunil Patel, Stefano Pasquali, and Dhagash Mehta. 2024. HybridRAG: Integrating Knowledge Graphs and Vector Retrieval Augmented Generation for Efficient Information Extraction. arXiv:2408.04948 [cs.CL] https://arxiv.org/abs/2408.04948

[38] David W. Scott. 1992. *Multivariate Density Estimation: Theory, Practice, and Visualization*. Wiley. doi:10.1002/9780470316849

[39] Wen Shi, Jianling Liu, Jingyu Zhang, Yuran Men, Hongwei Chen, Deke Wang, and Yang Cao. 2022. Feature Selection and Parameter Optimization of Support Vector Machines Based on a Local Search Based Firefly Algorithm for Classification of Formulas in Traditional Chinese Medicine. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.* 105-A, 5 (2022), 882–886. doi:10.1587/TRANSFUN.2021EAL2075

[40] Chanop Silpa-Anan and Richard I. Hartley. 2008. Optimised KD-trees for fast image descriptor matching. In *2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2008), 24-26 June 2008, Anchorage, Alaska, USA*. IEEE Computer Society. doi:10.1109/CVPR.2008.4587638

[41] Suhas Jayaram Subramanya, Devvrit, Rohan Kadekodi, Ravishankar Krishaswamy, and Harsha Vardhan Simhadri. 2019. *DiskANN: fast accurate billion-point nearest neighbor search on a single node*. Curran Associates Inc., Red Hook, NY, USA.

[42] Philip Sun, David Simcha, Dave Dopson, Ruiqi Guo, and Sanjiv Kumar. 2023. SOAR: Improved Indexing for Approximate Nearest Neighbor Search. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural*

Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023, Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (Eds.). http://papers.nips.cc/paper_files/paper/2023/hash/0973524e02a712af33325d0688ae6f49-Abstract-Conference.html

[43] Christophe Van Gysel, Maarten de Rijke, and Evangelos Kanoulas. 2016. Learning latent vector spaces for product search. In *Proceedings of the 25th ACM international on conference on information and knowledge management.* 165–174.

[44] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, Kun Yu, Yuxing Yuan, Yinghao Zou, Jiquan Long, Yudong Cai, Zhenxiang Li, Zhifeng Zhang, Yihua Mo, Jun Gu, Ruiyi Jiang, Yi Wei, and Charles Xie. 2021. Milvus: A Purpose-Built Vector Data Management System. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21).* Association for Computing Machinery, New York, NY, USA, 2614–2627. doi:10.1145/3448016.3457550

[45] Zihao Wang. 2025. Graph Library for Approximate Similarity Search. https://github.com/zilliztech/pyglass

[46] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. 2016. Speedup Graph Processing by Graph Ordering. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16).* Association for Computing Machinery, New York, NY, USA, 1813–1828. doi:10.1145/2882903.2915220

[47] Yair Weiss, Antonio Torralba, and Robert Fergus. 2008. Spectral Hashing. In *Advances in Neural Information Processing Systems 21, Proceedings of the Twenty-Second Annual Conference on Neural Information Processing Systems, Vancouver, British Columbia, Canada, December 8-11, 2008*, Daphne Koller, Dale Schuurmans, Yoshua Bengio, and Léon Bottou (Eds.). Curran Associates, Inc., 1753–1760. https://proceedings.neurips.cc/paper/2008/hash/d58072be2820e8682c0a27c0518e805e-Abstract.html

[48] Yanhao Zhang, Pan Pan, Yun Zheng, Kang Zhao, Yingya Zhang, Xiaofeng Ren, and Rong Jin. 2018. Visual search at alibaba. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining.* 993–1001.

[49] Yifan Zhao, Huiyu Cai, Zuobai Zhang, Jian Tang, and Yue Li. 2021. Learning interpretable cellular and gene signature embeddings from single-cell transcriptomic data. *Nature communications* 12, 1 (2021), 5261.