Introduction

With the rapid advancement of artificial intelligence and big data, vector retrieval has become a core technology in areas such as image search, recommendation systems, and natural language processing. In medical pathology, each digital pathology slide contains vast amounts of microscopic information, and each pathology report includes high-level semantic content. Modern AI technologies (e.g., UNI encoders[1], MPNet[2]) can encode these images or textual information into high-dimensional vectors (e.g., 1024 dimensions), creating a computable semantic space for semantic-level searches through distance calculations. As a key technology in this process, **Vector Retrieval** enables quick identification of similar information in databases based on similarity, providing support for diagnostic decisions and treatment recommendations. However, when data scales reach tens or hundreds of millions, traditional 'brute force search' methods (calculating distances between all vectors one by one) are too time-consuming to meet clinical real-time requirements.

Therefore, **how to efficiently index and retrieve large-scale vector data** becomes a critical issue. The following sections will introduce vector retrieval technology from basic to advanced concepts and discuss the core value of GPU acceleration in this domain.

Fundamentals of Vector Retrieval

1. Data Vectorization: From Unstructured Data to Semantic Space

Core Idea: Enabling computers to "understand" unstructured data

In the medical pathology field, whether it's digital pathology slides or diagnostic report texts, they are essentially **unstructured data—**lacking predefined formats but rich in semantic information. To make them computable, we need to map this data into a high-dimensional vector space using **embedding models**, thereby constructing quantifiable semantic representations.

Technical Implementation:

- **Images**: Use deep neural networks (such as ResNet[3], ViT[4]) to extract morphological features from pathology slides, outputting fixed-length vectors (e.g., 1024 dimensions). For example, characteristics like glandular structures and nuclear atypia in breast cancer are encoded as specific dimension activation values within vectors.
- **Texts**: Utilize Transformer models (such as BERT[5], PubMedBERT[6]) to convert pathology reports into vectors. Key terms in reports like "moderately differentiated adenocarcinoma" and "lymphatic metastasis" are encoded into vectors with semantically similar directions.

A good embedding model should preserve the semantics of the original data, meaning that **vectors representing similar pathological features are closer in space**. For instance, the similarity score between two lung adenocarcinoma slides might be 0.92, whereas the similarity between lung adenocarcinoma and squamous cell carcinoma might only be 0.65. Common measures of similarity include Euclidean distance, cosine similarity, dot product, etc.

2. Similarity Search: Evolution from KNNS to ANNS

Basic Problem: Given a query vector, how to quickly find the top-K most similar results in the database?

(1) Limitations of Traditional KNNS Methods

Early similarity searches mainly relied on **Exact Nearest Neighbor Search (K-Nearest Neighbors Search, KNNS)**, which aimed to accelerate computations through building indexing structures, including:

- **KD-Tree[7]**: Recursively divides space based on data dimensions (e.g., median or average division along each dimension of n-dimensional vectors) to construct binary trees for accelerating region queries.
- **Ball-Tree[8]**: Divides space using hyperspheres, utilizing sphere containment relationships to quickly exclude irrelevant regions.

However, KNNS faces significant bottlenecks in large-scale medical data scenarios:

- **Dimensionality Curse**: KD-Trees and Ball-Trees suffer from performance degradation in high-dimensional spaces (e.g., hundreds of dimensions or more), where recursive divisions and retrieval efficiency plummet.
- **Dynamic Update Deficiencies**: KD-Trees or Ball-Trees require static data; however, medical databases continuously add new cases (e.g., thousands of new pathology slides daily). Due to the inability to dynamically insert data, rebuilding KD-Trees after each update is required, failing to meet real-time needs.

(2) Breakthroughs in ANNS Approximation

To adapt to efficient retrieval of large-scale high-dimensional vector data, **Approximate Nearest Neighbor Search (ANNS)** achieves efficiency gains by tolerating controlled precision loss.

The following sections will detail some mainstream vector indexing techniques, including those based on space partitioning and quantization methods (IVFPQ[10]), graph-based retrieval methods (NN-Descent[11], HNSW[12]), and GPU-accelerated retrieval methods (SONG[13], CAGRA[14]).

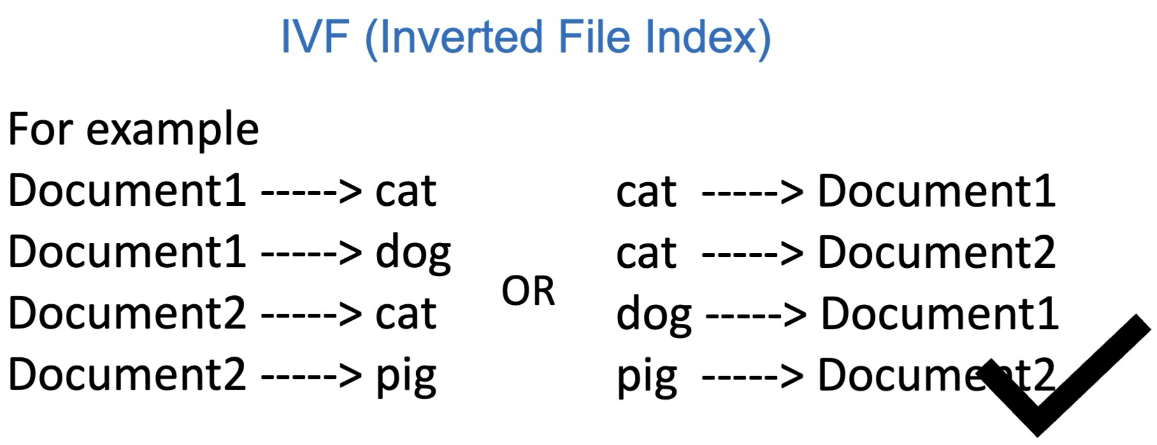
Analysis of Mainstream Indexing Techniques

1. IVFPQ

Construction Process

IVFPQ (Inverted File Index with Product Quantization) [10] is an efficient indexing method that combines space partitioning and vector compression. Its core process is divided into two phases: inverted index construction (IVF) and product quantization (PQ):

(1) Inverted Index Construction (IVF)



IVF (Inverted File Index)

For example

Document1 -----> cat

Document1 -----> dog

Document2 -----> cat

Document2 -----> pig

OR

cat -----> Document1

cat -----> Document2

dog -----> Document1

pig -----> Document2

✓

Figure 1: Example of an inverted file index

To better understand, consider a document keyword search example as shown in the left part of Figure 1. Suppose we have a set of documents (Document1, Document2, Document3, Document4), each containing different words. If we want to find which documents contain a specific word like "dog," a straightforward approach would involve iterating through every word in every document, checking if it matches "dog." This method is obviously time-consuming, potentially requiring a full scan of all documents.

IVF takes a different approach by pre-calculating which words are stored in which documents, as shown in the right part of Figure 1. By recording which documents each word corresponds to, searching for a word only requires finding its location to know where it appears. Since the length of the vocabulary is often shorter than the total number of documents, this can save search time.

In the context of vector retrieval, the basic process is as follows:

1. Cluster Center Generation: Use the K-Means algorithm to cluster the entire vector dataset $\mathcal{D} = \{v_1, v_2, \ldots, v_n\}$, generating n_{list} cluster centers $C = \{c_1, c_2, \ldots, c_{n_{\text{list}}}\}$.

2. Inverted List Construction: For each cluster center c_i, record the set of member vectors $\mathcal{L}_i = \{v_j | \arg\min_k \|v_j - c_k\|_2 = i\}$, forming an inverted list structure.

3. Parameter Significance: n_{list} controls the number of coarse-grained clusters; larger values increase retrieval accuracy but also computational cost (default value is typically 1024).

(2) Product Quantization (PQ)

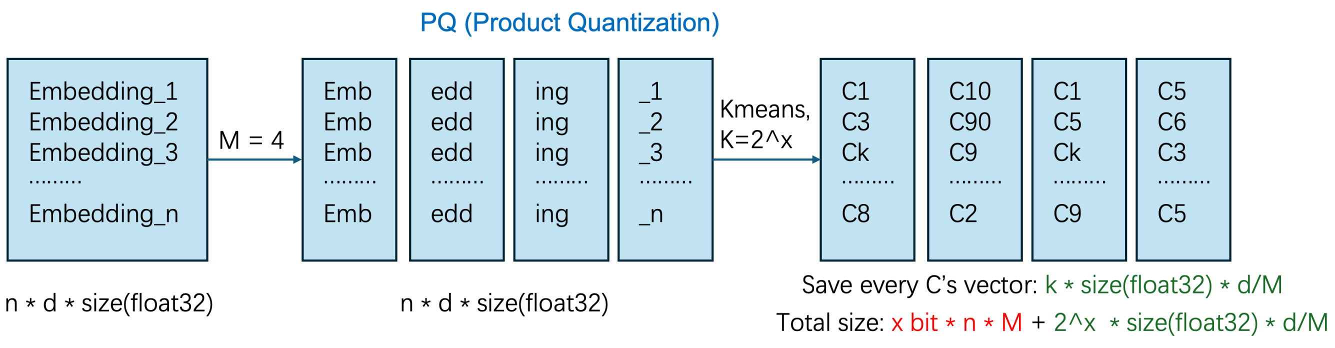


Figure 2: Example of product quantization

The main goal of product quantization is to reduce memory usage. With a large number of vectors, memory consumption can be significant. The method is designed to reduce memory usage, as illustrated in Figure 2. The steps are as follows:

1. Vector Partitioning: Divide the original vector $v \in \mathbb{R}^d$ into M sub-vector blocks $v = [v^{(1)}, v^{(2)}, \dots, v^{(M)}]$, each of dimension d/M.

2. Subspace Quantization: Perform K-Means clustering uniformly across all subspaces $\mathbb{R}^{d/M}$ to generate codebooks $\mathcal{B}_m = \{b_{m,1}, b_{m,2}, \dots, b_{m,2^x}\}$, where x is the bit number of the codebook (typically $x = 8$, $2^x = 256$).

3. Encoding Compression: Replace the original vector v with the cluster ID codes corresponding to each sub-block $\text{code}(v) = [k_1, k_2, \dots, k_M]$, where $k_m = \arg\min_j \|v^{(m)} - b_{m,j}\|_2$.

Memory Optimization Analysis

The total storage after quantization consists of two parts:

- **Codebook Storage**: $M \times 2^x \times \frac{d}{M} \times \text{float32} = 2^x d \times \text{float32}$.
- **Encoded Storage**: $n \times M \times x$ bits.

For example, with $d = 1024$, $M = 8$, and $x = 8$, the original storage requires $1024 \times 4 = 4096$ bytes per vector, while PQ compression reduces this to $8 \times 8 = 64$ bits (8 bytes), achieving a compression ratio of up to 512 times.

Retrieval Mechanism

1. Coarse-Grained Filtering (IVF Phase): Calculate the distance between the query vector q and all cluster centers c_i, selecting the nearest n_{probe} clusters.

2. Fine-Grained Calculation (PQ Phase): For each vector v_j in the candidate clusters $\mathcal{L}_i$, use precomputed codebooks for fast distance estimation:

$$\hat{d}(q, v_j) = \sum_{m=1}^{M} \|q^{(m)} - b_{m,k_j^{(m)}}\|_2^2 \tag{1}$$

where $q^{(m)}$ is the m-th sub-block of the query vector, and $b_{m,k_j^{(m)}}$ is the codeword center corresponding to the m-th sub-block of v_j.

3. Top-K Sorting: Sort by calculated distances in ascending order, returning the top K nearest neighbor vectors globally.

2. NN-Descent

Construction Process

NN-Descent (Nearest Neighbor Descent) [11] is an approximate nearest neighbor search algorithm based on local graph structure optimization. Its core idea is to dynamically adjust neighbor relationships through iterative optimization, gradually approaching the true K nearest neighbor graph. The construction process can be formally described as follows:

(1) Graph Initialization

1. Random Neighbor Sampling: For each data point $v_i \in \mathcal{D}$, randomly select K initial neighbors $N^{(0)}(v_i)$ to construct the initial directed neighborhood graph G_0.

2. Reverse Neighbor Indexing: Simultaneously record reverse neighbor relationships $R(v_i) = \{v_j | v_i \in N^{(0)}(v_j)\}$ to form a bidirectional connection structure.

(2) Iterative Optimization

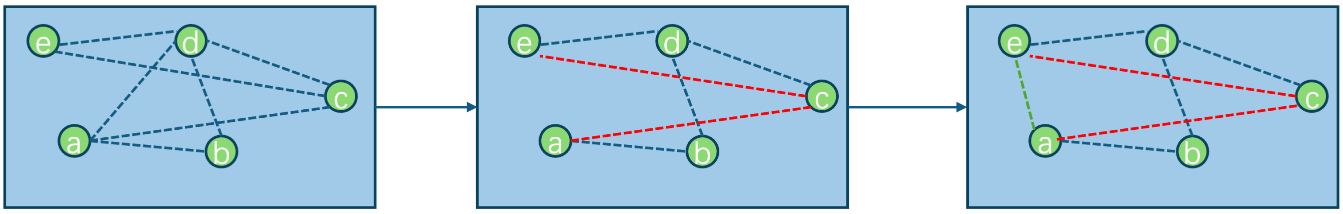


Figure 3: Schematic of Iterative Optimization

During the t-th iteration, optimize neighbor relationships using reverse neighbor propagation and triangle closure properties. As shown in Figure 3, during one iteration, for point a, find all its neighbors and reverse neighbors (e.g., b, c, and d). For point c, find all its neighbors and reverse neighbors (e.g., e and d). Calculate the distance between a and e. If the distance from a to e is shorter than the path from a to c to e, update the graph by connecting a and e. The detailed algorithm process is as follows:

1. Candidate Set Generation:

For point v_i, merge its neighbors and reverse neighbors to generate a candidate set:

$$\mathcal{C}(v_i) = N^{(t)}(v_i) \cup \bigcup_{v_j \in N^{(t)}(v_i)} R(v_j) \tag{2}$$

2. Local Distance Calculation:

Compute distances between v_i and all candidate points in $\mathcal{C}(v_i)$, and filter out closer potential neighbors.

3. Neighbor Update:

For each $v_j \in \mathcal{C}(v_i)$, if $d(v_i, v_j) < \max_{v_k \in N^{(t)}(v_i)} d(v_i, v_k)$, add v_j to the new neighbor set $N^{(t+1)}(v_i)$.

4. Degree Constraint:

Retain the top K nearest neighbors for each point v_i, pruning the rest of the connections.

(3) Convergence Conditions

Iteration terminates when one of the following conditions is met:

- The neighbor update ratio falls below a threshold ϵ (e.g., $\epsilon = 0.01$).
- The maximum number of iterations $T_{\max}$ is reached (e.g., $T_{\max} = 20$).

Retrieval Mechanism

On the constructed neighborhood graph, efficient search is achieved with the following strategies:

1. Multi-start Parallel Search:

Start from L randomly selected starting points (e.g., $L = 50$) and execute parallel greedy hill-climbing algorithms.

2. Dynamic Candidate List Maintenance:

For each search path, maintain a dynamic candidate queue $\mathcal{Q}$, retaining the top K candidates sorted by distance.

3. Neighbor Expansion Strategy:

For the current nearest point v_{curr}, add its neighbor set $N(v_{\text{curr}})$ to $\mathcal{Q}$ and update the distance sorting.

4. Termination Condition:

Terminate the search when the minimum distance in the candidate queue $\mathcal{Q}$ does not update for τ consecutive iterations (e.g., $\tau = 3$).

Mathematical Modeling

Let the distance between any two points in dataset $\mathcal{D}$ be $d(v_i, v_j)$. The algorithm approximates the true K-nearest neighbor graph by minimizing the following objective function:

$$\mathcal{L}(G) = \sum_{v_i \in \mathcal{D}} \sum_{v_j \in N(v_i)} d(v_i, v_j) \tag{3}$$

The iterative optimization process can be seen as performing gradient descent within local regions, gradually reducing $\mathcal{L}(G)$.

3. HNSW

Construction Process

HNSW (Hierarchical Navigable Small World) [12] is an efficient indexing method that combines hierarchical structures with navigable small-world graphs. Its core design inspiration comes from natural beehive networks and the multi-layer topological characteristics of human social networks. The algorithm achieves exponential search path reduction by constructing a graph structure with decreasing layers. The specific process is as follows:

(1) Generation of Hierarchical Graph Structure

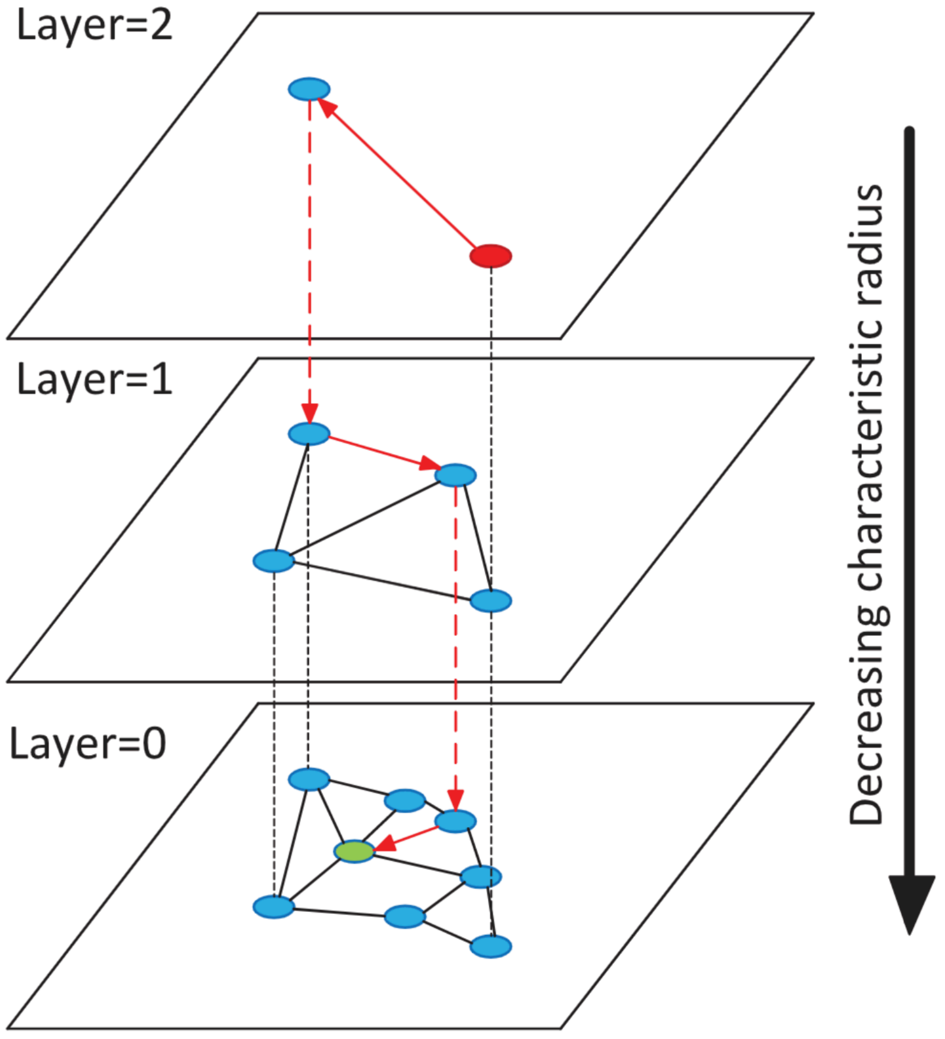


Figure 4: Schematic of HNSW Hierarchical Structure (Assuming the highest layer L=3, counting layers from 0)

1. Probabilistic Layer Assignment: For each data point $v_i \in \mathcal{D}$, randomly assign the maximum layer $l_{\max}(v_i)$, satisfying:

$$l_{\max}(v_i) = \lfloor -\ln(\text{rand}(0,1)) \cdot m_L \rfloor \tag{4}$$

where m_L is the inter-layer decay coefficient (typically set to $1/\ln(M)$, where M is the maximum number of connections per layer). This assignment strategy ensures that the number of nodes in higher layers decreases exponentially.

2. Layer-by-Layer Graph Construction:

- **Top Layer Construction (Layer L)**: Only includes nodes satisfying $l_{\max}(v_i) \geq L$, forming a sparse connected small-world graph.
- **Layer-by-Layer Expansion**: In layer l, nodes include all those satisfying $l_{\max}(v_i) \geq l$, and each node maintains up to M bidirectional edges.

(2) Dynamic Insertion Strategy

When inserting a new node v_{new}, update the graph structure layer by layer according to the following rules (taking layer l as an example):

1. Search for Current Layer Entry Point: Start from the top layer L, use a greedy algorithm to find the entry point e_l in layer l that is closest to v_{new}.

2. Local Neighborhood Exploration: In layer l, starting from e_l, expand the candidate set using a priority queue, retaining the $ef_{\text{Construction}}$ nearest nodes to v_{new}.

3. Connection Optimization: From the candidate set, select up to M neighbors, satisfying:

- **Nearest Neighbor Principle**: Preferentially connect to the nearest nodes.
- **Diversity Constraint**: Avoid local clustering by selecting edges with different directions (avoiding island phenomena) through heuristic algorithms.

Retrieval Mechanism

The search process of HNSW implements efficient navigation through a **skip-list-like** layer-by-layer refinement strategy. The steps are as follows:

1. Top Layer Entry Point Positioning: Start from a random node in the top layer L, execute a greedy algorithm to find the node e_L closest to the query q in this layer.

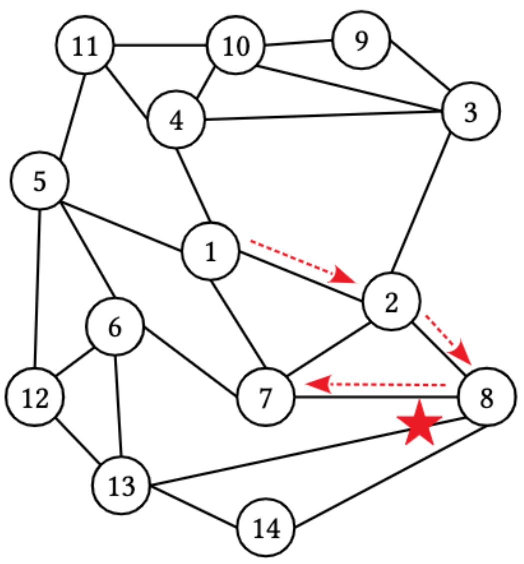
2. Layer-by-Layer Descent Search: Starting from e_L, based on the skip-list relationships between layers, find the neighbor closest to q among the neighbors of e_{L-1} in the next lower layer, and use it as the input for the next layer.

3. Bottom Layer Precise Retrieval: In layer 0, starting from the entry point e_0, use a dynamic candidate list algorithm similar to NN-Descent:

1. Initialize the dynamic list $\mathcal{C}$, containing e_0 and its neighbors.
2. Iteratively expand $\mathcal{C}$, maintaining the top ef_{Search} nearest neighbors.
3. Terminate when there is no update in the nearest neighbors for τ consecutive iterations, and return the Top-K results.

4. SONG

Design Motivation



Initialization	*q:*	1
	topk:	∅
	visited:	1
Iteration 1	*q:*	2 7 4 5
	topk:	1
	visited:	1 2 4 5 7
Iteration 2	*q:*	8 7 3 4 5
	topk:	1 2
	visited:	1 2 3 4 5 7 8
Iteration 3	*q:*	7 3 4 5 14 13
	topk:	1 2 8
	visited:	1 2 3 4 5 7 8 13 14
Iteration 4	*q:*	3 4 5 6 14 13
	topk:	7 2 8
	visited:	1 2 3 4 5 6 7 8 13 14

Figure 5: Traditional Graph-Based ANNS Search Process

Approximate Nearest Neighbor Search (ANNS) based on graphs typically relies on three core data structures: **priority queue, result heap, and visited record table**. Its iterative search process is serial on a CPU:

1. Priority Queue (q): Maintains candidate nodes to be expanded (sorted by distance).

2. Result Heap (TopK): Stores the current K nearest results.

3. Visited Record Table (Visited): Marks visited nodes to avoid redundant calculations.

Taking the search process in Figure 5 as an example, traditional methods require iterative rounds: extracting the nearest node from the queue, calculating distances to its neighbors, and updating the result heap. However, **95% of the computational time is spent on high-dimensional vector distance calculations** (e.g., 1024-dimensional Euclidean distance). The single-threaded architecture of CPUs cannot parallelize this process efficiently. Even if query tasks are parallelized, a single complex query can still block the entire pipeline.

Overall Architecture

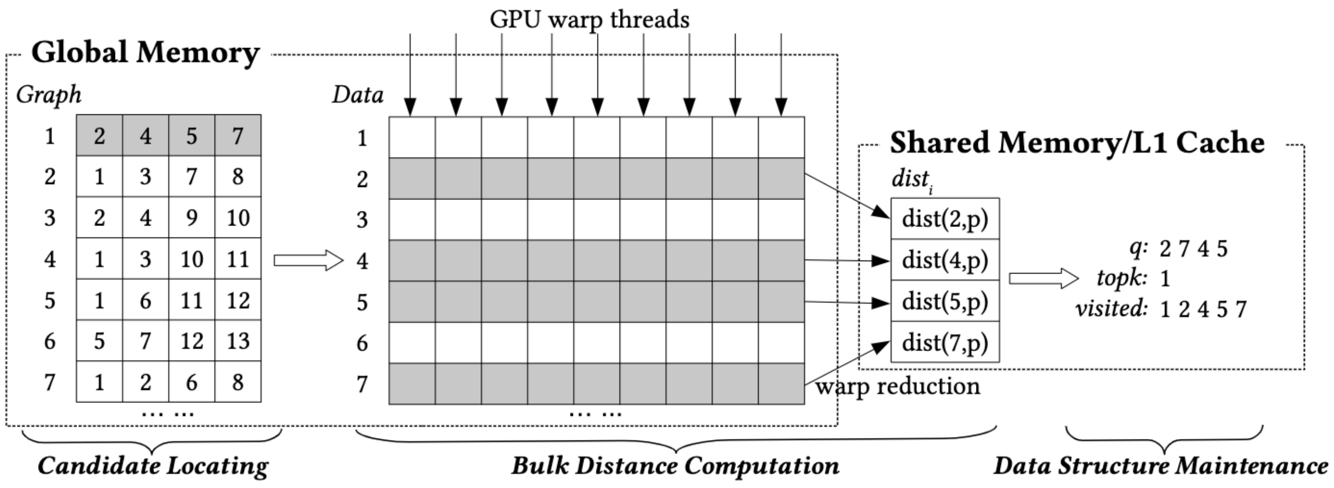


Figure 6: SONG Overall Architecture

The core innovation of SONG [13] lies in **reconstructing the search pipeline**, decoupling serial steps into GPU-friendly three-stage parallel tasks, as shown in Figure 6:

Optimization Design

1. Candidate Locating

- **Objective**: Load the neighbor lists of candidate nodes from global memory.
- **Optimization Strategies**:
    - **Fixed Degree Graph Storage**: Limit the number of neighbors per node (e.g., up to 64), pre-allocate contiguous space in global memory to avoid dynamic memory overhead.
    - **Multi-query Batch Loading**: Each GPU warp processes multiple queries' candidate nodes simultaneously.

2. Bulk Distance Computation

- **Parallelization Core**: Decompose high-dimensional vector calculations into GPU thread-level tasks.
  - **Formula**: The Euclidean distance $d(q,v) = \sqrt{\sum_{i=1}^{d}(q_i - v_i)^2}$ is broken down into d parallel subtraction-square operations, with Warp-level thread cooperation for reduction and summation.
  - **Memory Aligned Access**: Vector data for the same node is stored contiguously, allowing single reads of 128 bytes (GPU cache line).

3. Data Structure Update

- **Lightweight Atomic Operations**: A single thread is responsible for updating the queue, result heap, and visited records.
  - **TopK Heap Compression**: Retain only the K nearest results, with a fixed heap capacity of $K_{\max}$ (e.g., K=100).
  - **Cuckoo Filter [15] Replaces Hash Tables**: Supports dynamic insertion and deletion.

5. CAGRA

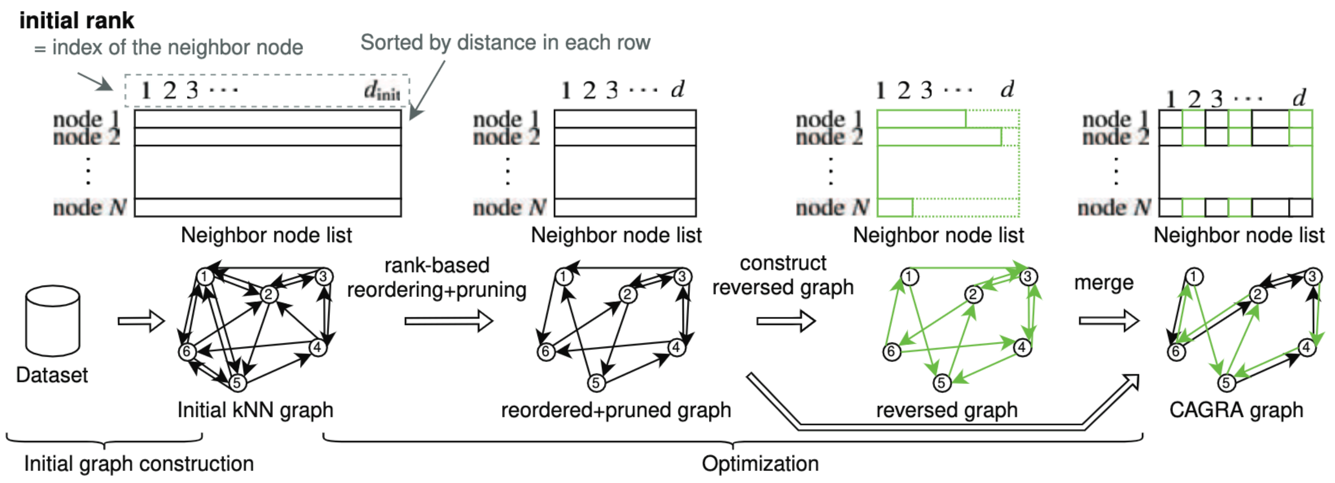


Figure 7: Schematic of CAGRA Graph Optimization Process (Left: Initial k-NN Graph; Middle: Rank Reordering; Right: Reverse Edge Enhancement)

(1) Initial k-NN Graph Construction

Using a GPU-accelerated NN-Descent algorithm [11], the basic k-NN graph (k=2d~3d) is constructed. The process differs significantly from the CPU version of NN-Descent:

- **Batch Reverse Neighbor Calculation**:

  Convert the calculation of reverse neighbor relationships $R(v_i)$ into a GPU-friendly matrix transpose operation, utilizing shared memory to accelerate atomic operations. For example, with d=64, a single iteration can handle reverse relationship updates for 10^6 nodes in parallel.

- **Dynamic Memory Pre-allocation**:

  Pre-allocate global memory to eliminate dynamic expansion overhead. Each node maintains a fixed-capacity candidate queue (e.g., 128), with excess entries truncated.

(2) Graph Structure Optimization

Perform two levels of optimization on the initial k-NN graph to enhance search efficiency:

1. Rank-Based Reordering:

For each node v_i's neighbor list $N(v_i)$, generate ranks based on original distances and move lower-rank edges (important connections) forward. This operation is fully parallelized on GPUs, allowing rank reordering for millions of nodes within one second.

2. Reverse Edge Enhancement:

To improve graph connectivity, add reverse edges $v_j \rightarrow v_i$ for each edge $v_i \rightarrow v_j$, but limit the out-degree of each node to d (default d=64). Reverse edges are dynamically truncated based on the source node's rank to ensure continuous global memory access.

Retrieval Mechanism

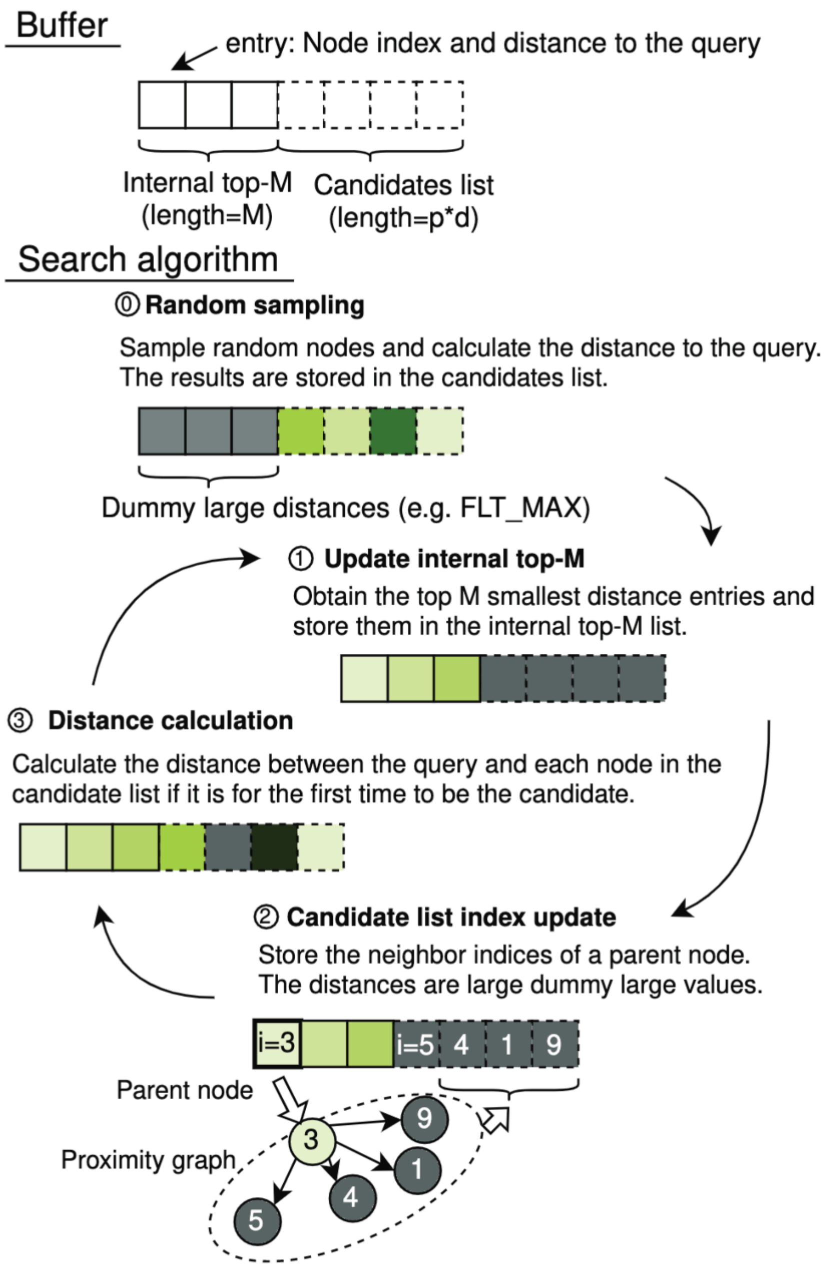

Buffer
entry: Node index and distance to the query
Internal top-M (length=M)
Candidates list (length=p*d)
Search algorithm
⓪ Random sampling
Sample random nodes and calculate the distance to the query. The results are stored in the candidates list.
Dummy large distances (e.g. FLT_MAX)
① Update internal top-M
Obtain the top M smallest distance entries and store them in the internal top-M list.
③ Distance calculation
Calculate the distance between the query and each node in the candidate list if it is for the first time to be the candidate.
② Candidate list index update
Store the neighbor indices of a parent node. The distances are large dummy large values.
i=3
i=5
4
1
9
Parent node
Proximity graph
3
9
1
4
5

Figure 8: CAGRA Search Process

CAGRA [14]'s retrieval process has been systematically restructured for GPU architecture, with key innovations in **full pipeline parallelization** and **dynamic candidate buffering mechanisms**. Unlike hierarchical graph indexes such as HNSW, CAGRA abandons the serial path optimization strategy of layer-by-layer progression, instead enhancing GPU thread utilization through **breadth-first parallel expansion**. The specific implementation is as follows:

(1) Dual Buffer Architecture Design

CAGRA employs a dual dynamic buffering mechanism with **Top-M Buffer** and **Candidate Buffer** (as shown in Figure 8, Top-M Buffer stores the current best results in order; Candidate Buffer stores potential candidates unordered), achieving deep decoupling between computation and memory access:

- **Top-M Buffer** (Ordered Zone):
    - Fixed capacity of M (e.g., $M = 200$), storing the currently best candidate nodes in ascending order of distance.
    - Uses a **min-heap** data structure, with insertion complexity $O(\log M)$, supporting dynamic replacement.
- **Candidate Buffer** (Unordered Zone):
    - Capacity of E (expansion factor, e.g., $E = 64$), storing newly expanded candidate nodes.
    - Unordered design avoids sorting overhead, implementing parallel writes using atomic operations.

(2) Parallel Search Strategy

At the GPU warp level, batch expand and compute distances for candidate nodes. The specific process is as follows:

1. Random Seed Sampling: At query start, randomly select E nodes (e.g., $E = 64$) from the global graph as initial candidates, storing them in the **Candidate Buffer**.

2. Update Top_M: Merge the **Top-M Buffer** and **Candidate Buffer**, retaining the top M nearest neighbors, and store them in the **Top-M Buffer**.

3. Update Candidate Buffer: From the **Top-M Buffer**, select unvisited nearest nodes to the query and add their neighbors to the **Candidate Buffer**.

4. Distance Calculation: Compute distances between nodes in the **Candidate Buffer** and the query. Return to step 2 until convergence (when all nodes in the **Top-M Buffer** are visited and are the nearest to the query).

Why Use GPU Acceleration for Vector Retrieval?

In large-scale vector retrieval scenarios, traditional CPU architectures are increasingly showing performance bottlenecks. With the exponential growth of data scale and the continuous increase in vector dimensions, computational complexity has exploded, especially with high-dimensional vector distance calculations becoming a major time-consuming point. GPUs, with their powerful parallel computing capabilities, offer a new solution for accelerating vector retrieval.

1. Parallel Computing Advantages of GPUs

GPUs have thousands of cores capable of handling a large number of thread tasks simultaneously. For high-dimensional distance calculations in vector retrieval (such as Euclidean distance, cosine similarity, etc.), GPUs can decompose these calculations into fine-grained parallel tasks, where each thread is responsible for calculating the difference and square operations of one or more dimensions, followed by efficient reduction operations to complete the final summation. This highly parallelized computing approach gives GPUs a significant speed advantage over CPU-centric single-threaded processing when handling large-scale vector retrieval.

2. Memory Bandwidth and Data Throughput Capability

Vector retrieval involves frequent vector loading and storage operations, which place extremely high demands on memory bandwidth. GPUs are equipped with high-bandwidth video memory that can quickly read and write large-scale vector data. Additionally, GPU memory access optimization mechanisms (such as shared memory, cache-aligned access) further enhance data throughput efficiency, reducing performance losses due to memory latency.

3. Hardware Acceleration for Algorithms

Modern GPUs support specialized mathematical operation instruction sets (such as CUDA Core, Tensor Core), enabling efficient execution of matrix operations and vector operations. For example, during the bulk distance calculation phase, GPUs can leverage warp-level collaboration mechanisms to transform high-dimensional vector distance calculations into parallel subtraction, squaring, and reduction operations, achieving acceleration effects of tens or even hundreds of times. Moreover, for specific algorithms (such as Product Quantization PQ, graph index search), GPUs can further enhance performance through customized optimizations (such as fixed-degree graph storage, using Cuckoo Filters instead of hash tables).

4. Support for Real-Time Requirements

In practical application scenarios such as medical pathology and recommendation systems, vector retrieval often needs to meet real-time response requirements at the millisecond or even microsecond level. The high throughput and low latency characteristics of GPUs enable them to easily handle real-time query tasks for vector databases ranging from millions to billions of entries. For instance, GPU-based indexing methods (such as SONG, CAGRA) can process hundreds of candidate nodes in parallel during a single query, significantly reducing retrieval time.

5. Flexibility and Scalability

GPUs are not only suitable for accelerating single tasks but can also be further enhanced through multi-GPU parallel expansion. For ultra-large-scale vector datasets, distributed GPU clusters can achieve cross-node collaborative computing to meet higher concurrency demands. Furthermore, GPU programming frameworks (such as CUDA) provide flexible development interfaces, allowing researchers to optimize algorithm designs for specific scenarios.

Summary

In conclusion, GPU acceleration has become one of the core technologies in the field of vector retrieval. Its outstanding parallel computing capabilities, efficient memory management mechanisms, and hardware optimizations for algorithms ensure that vector retrieval remains efficient and real-time even when dealing with massive data and high-dimensional spaces. Whether in academic research or industrial applications, GPU acceleration is driving the rapid development of vector retrieval technology, providing strong support for artificial intelligence and big data fields.

[1] Chen, Richard J., et al. "Towards a general-purpose foundation model for computational pathology." *Nature Medicine* 30.3 (2024): 850-862.

[2] Song, Kaitao, et al. "Mpnet: Masked and permuted pre-training for language understanding." *Advances in neural information processing systems* 33 (2020): 16857-16867.

[3] He, Kaiming, et al. "Deep residual learning for image recognition." *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016.

[4] Dosovitskiy, Alexey, et al. "An image is worth 16x16 words: Transformers for image recognition at scale." *arXiv preprint arXiv:2010.11929* (2020).

[5] Devlin, Jacob, et al. "Bert: Pre-training of deep bidirectional transformers for language understanding." *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*. 2019.

[6] Gu, Yu, et al. "Domain-specific language model pretraining for biomedical natural language processing." *ACM Transactions on Computing for Healthcare (HEALTH)* 3.1 (2021): 1-23.

[7] Bentley, Jon Louis. "Multidimensional binary search trees used for associative searching." *Communications of the ACM* 18.9 (1975): 509-517.

[8] Omohundro, Stephen M. "Five balltree construction algorithms." (1989).

[9] Han, Yikun, Chunjiang Liu, and Pengfei Wang. "A comprehensive survey on vector database: Storage and retrieval technique, challenge." *arXiv preprint arXiv:2310.11703* (2023).

[10] Jegou, Herve, Matthijs Douze, and Cordelia Schmid. "Product quantization for nearest neighbor search." *IEEE transactions on pattern analysis and machine intelligence* 33.1 (2010): 117-128.

[11] Dong, Wei, Charikar Moses, and Kai Li. "Efficient k-nearest neighbor graph construction for generic similarity measures." *Proceedings of the 20th international conference on World wide web*. 2011.

[12] Malkov, Yu A., and Dmitry A. Yashunin. "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs." *IEEE transactions on pattern analysis and machine intelligence* 42.4 (2018): 824-836.

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

[14] Ootomo, Hiroyuki, et al. "Cagra: Highly parallel graph construction and approximate nearest neighbor search for gpus." *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 2024.

[15] Fan, Bin, et al. "Cuckoo filter: Practically better than bloom." *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*. 2014.