CMANNS: GPU-Accelerated Graph Index Construction for ANNS via Compute–Memory Disaggregation

CHENGYING HUAN, State Key Laboratory for Novel Software Technology, Nanjing University, China
RENJIE YAO, State Key Laboratory for Novel Software Technology, Nanjing University, China
SHAONAN MA, Qiyuan Lab, China
RONG GU*, State Key Laboratory for Novel Software Technology, Nanjing University, China
ZHENGYI YANG, University of New South Wales, Australia
LIZHENG CHEN, State Key Laboratory for Novel Software Technology, Nanjing University, China
ZHIBIN WANG, State Key Laboratory for Novel Software Technology, Nanjing University, China
MINGXING ZHANG, Tsinghua University, China
FANG XI, Qiyuan Lab, China
GUIHAI CHEN, State Key Laboratory for Novel Software Technology, Nanjing University, China
CHEN TIAN, State Key Laboratory for Novel Software Technology, Nanjing University, China

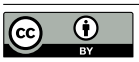
Graph-based approximate nearest neighbor search (ANNS) delivers state-of-the-art accuracy–latency tradeoffs, yet *index construction* remains the bottleneck: fusing dense distance evaluation with irregular traversal and pruning collapses GPU throughput, and limited device memory forces costly data movement at scale.

To address these problems, in this paper, we present CMANNS, a GPU-accelerated graph index construction framework that preserves the algorithmic rules of target graph (e.g., NSG and HNSW) and its query procedure. The core idea is *compute–memory (CM) disaggregation*: distance evaluation is reformulated as high–arithmetic-intensity GEMMs on Tensor Core accelerators with fused epilogues, while memory-intensive phases employ hot-set–aware on-chip locality (e.g., shared-memory staging, warp-cooperative gathers and scatters) to maximize effective bandwidth. To scale beyond the HBM capacity, we stream device-sized shards through a double-buffered pipeline and write back only compact adjacency. Data transfers and kernel execution overlap, so each shard completes in roughly the time of the slower step, keeping the GPU highly utilized even with irregular access. Across seven benchmarks, CMANNS reduces end-to-end index build time by up to **13.05**× (vs. FAISS) and **2.20**× (vs. FLASH), increases the cache hit rate by up to **58.7%**, and preserves vector query latency and recall.

CCS Concepts: • **Information systems → Nearest-neighbor search**; • **Computing methodologies → Parallel algorithms**.

*Rong Gu is the corresponding author of this paper.

Authors' Contact Information: Chengying Huan, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China, huanchengying@nju.edu.cn; Renjie Yao, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China, 522024330111@smail.nju.edu.cn; Shaonan Ma, Qiyuan Lab, Beijing, China, mashaonan@qiyuanlab.com; Rong Gu, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China, gurong@nju.edu.cn; Zhengyi Yang, University of New South Wales, Sydney, Australia, zhengyi.yang@unsw.edu.au; Lizheng Chen, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China, 221240093@smail.nju.edu.cn; Zhibin Wang, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China, wzbwangzhibin@gmail.com; Mingxing Zhang, Tsinghua University, Beijing, China, zhang_mingxing@mail.tsinghua.edu.cn; Fang Xi, Qiyuan Lab, Beijing, China, xifang@qiyuanlab.com; Guihai Chen, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China, gchen@nju.edu.cn; Chen Tian, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China, tianchen@nju.edu.cn.



ACM 2836-6573/2026/6-ART150
https://doi.org/10.1145/3802027

Additional Key Words and Phrases: ANNS, Graph Index, GPU Acceleration.

ACM Reference Format:
Chengying Huan, Renjie Yao, Shaonan Ma, Rong Gu, Zhengyi Yang, Lizheng Chen, Zhibin Wang, Mingxing Zhang, Fang Xi, Guihai Chen, and Chen Tian. 2026. CMANNS: GPU-Accelerated Graph Index Construction for ANNS via Compute–Memory Disaggregation. *Proc. ACM Manag. Data* 4, 3 (SIGMOD), Article 150 (June 2026), 27 pages. https://doi.org/10.1145/3802027

1 Introduction

Approximate nearest neighbor search (ANNS) retrieves the k most similar vectors to a query under a chosen measure (e.g., Euclidean, cosine, or inner product) and underpins web search [57], recommender systems [6, 20, 37, 58], RAG [25, 30, 40], and large-scale data mining [42]. To avoid the $O(Nd)$ cost of brute-force scanning, systems build *indexes* that prune the search space while preserving high recall. Among the major index families, quantization (e.g., IVF and IVF-PQ) [23], tree and space-partitioning [3, 50], and *graph-based* methods [1, 4, 11, 24, 28, 33–35, 39, 45, 54] have emerged as leaders on the recall–latency Pareto frontier in high dimensions [2]. They build a sparse proximity graph over the dataset and answered queries through greedy (or beam) navigation, often with sublinear work in N thanks to the locality in the graph.

Among graph index families, two designs dominate practice for their fast queries and strong recall: *NSG* (Navigating Spreading-out Graph) [11] and *HNSW* (Hierarchical NSW) [34]. NSG builds a single-layer, degree-bounded proximity graph that approximates a monotone relative-neighborhood structure. It combines (i) a navigational node-selection rule, (ii) diversification/pruning to enforce bounded out-degree (compact memory, predictable traversal), and (iii) carefully placed long-range "shortcut" edges connecting distant regions, which yield a sparse yet highly navigable index with near–logarithmic search behavior in practice, well-suited for latency-sensitive deployments. HNSW augments NSW with multiple layers: the sparse upper layers enable long-range routing, while the dense base layer refines locally. Queries start at the top, descend greedily through the hierarchy, and finish with a best-first search at the bottom. This organization reduces the number of visited nodes and, critically, supports *incremental construction*, which is a key operational advantage. HNSW's robustness across datasets and straightforward tuning have made it a widely-used index [31].

Despite their excellent *query* performance [22, 29, 46], popular graph indexes such as NSG and HNSW exhibit long *construction* times on CPUs [46], particularly at large-scales. In production, embeddings drift as models evolve and content changes; many systems, therefore, refresh indices to meet tight service-level objectives (SLOs), ranging from hourly to daily. CPU-only builders make refresh cycles costly: on a 200M-vector dataset, CPU-only NSG rebuilds exceed *ten hours*, monopolize many-core servers, and often force dual-serving—keeping one index online while another rebuilds—which inflates the hardware footprint and reduces query throughput due to contention [22].

Modern GPUs provide an attractive substrate for accelerating ANNS construction [18, 28, 38, 53, 55, 59, 60]. Three synergistic properties are particularly relevant: *(1) Massive SIMT parallelism*: tens of thousands of lightweight threads scheduled in warps will hide latency via rapid context switching, turning bulk regular work (e.g., batched distance evaluation) into high-throughput kernels [12]; *(2) Matrix–multiply accelerators*: Tensor Core–class units [5, 10, 48, 49] execute mixed-precision GEMMs (FP16/BF16 inputs with FP32 accumulation), delivering order-of-magnitude higher peak arithmetic throughput than scalar CUDA cores once distances are cast as matrix products; and *(3) High on-device bandwidth*: GPU HBM provides substantially greater bandwidth than CPU DRAM [27], and on-chip SRAM (e.g., shared memory) is even higher; when accesses are coalesced and locality is exploited, this bandwidth sustains compute. On our platform (Table 1), HBM bandwidth is roughly 8× CPU DDR4, while on-chip SRAM (shared memory) offers an additional

Table 1. CPU vs. GPU: Key performance metrics.

Performance Metric	GPU (NVIDIA A100)	CPU (Xeon Gold 6426Y)
FP32 TFLOPS (CUDA Cores)	19.5	2.56
FP16/BF16 TFLOPS (Tensor Cores)	312	N/A
Memory Bandwidth	~1,600 GB/s (HBM2)	~200 GB/s (8 channels DDR4)
SRAM Bandwidth	~19 TB/s (Shared Memory)	~ 0.5 TB/s (L2 Cache)

> 10× over HBM. Harnessed together via hardware-conscious algorithm design, these features can reduce build times from hours to minutes while preserving recall.

However, when directly implementing traditional graph indices such as NSG and HNSW on a GPU platform, **simply "moving code to the GPU" is ineffective**. Graph construction interleaves irregular, pointer-chasing phases (expansion, ordering, and pruning) with dense floating-point distance evaluation. This *mixed* workload collapses under SIMT execution due to warp divergence and de-coalesced memory traffic. In addition, GPU high-bandwidth memory (HBM) is more capacity-limited than CPU DRAM, so large-scale datasets (often exceeding GPU memory size) must be streamed from CPU DRAM; without careful orchestration, CPU–GPU data transfer I/O dominates wall-clock time. Furthermore, advanced methods optimized for CPUs, such as FLASH [46], accelerate construction by trading arithmetic operations for memory lookups to maximize AVX512 utilization. Such designs are architecturally incompatible with GPUs. Porting approaches relying on Look-Up Tables to SIMT architectures induces severe warp divergence, effectively underutilizing the greatest strength of the GPU: its massive arithmetic throughput (e.g., Tensor Cores). These obstacles motivate a hardware-conscious redesign of the construction pipeline.

When considering recent GPU-based graph index construction methods, they span a spectrum from *structure-changing* designs to *structure-preserving* builders. **CAGRA** [38] pursues a search-centric, GPU-friendly graph by fixing the out-degree and refining connectivity from an initial kNN scaffold through heuristic pruning and augmentation—prioritizing regular work patterns and high throughput on accelerators while deviating from classic NSG or HNSW invariants. **GGNN** [18] restructures graph construction and search for SIMT efficiency (e.g., batching expansions, GPU-cooperative updates), emphasizing parallel regularity over adherence to a specific legacy index. **GANNS** [55] adopts a divide-and-conquer approach (parallel local subgraphs followed by controlled merging) tailored for GPUs.

However, these GPU-based design choices limit their efficiency and generality. **First**, many approaches *modify the index structure* to expose uniform parallelism (e.g., fixed out-degree, flat layouts, bespoke merging rules). While GPU-friendly, these modifications decouple the builder from the established graph index, making them non–drop-in and often introducing quality or parameter trade-offs tied to the bespoke structure. **Second**, kernels frequently *fuse* traversal and pruning (irregular, latency-bound) with distance evaluation (dense, compute-bound), which induces warp divergence and de-coalesced accesses on SIMT hardware; few systems *disaggregate* compute vs. memory phases or map distance to high–arithmetic-intensity GEMMs, leaving Tensor Core–class accelerators underutilized. **Third**, many designs assume *in-core* execution; i.e., datasets must reside entirely in HBM with minimal buffering, leading transfers and kernels to run largely *serialized*. At

~100 GB scale, host–device PCIe I/O becomes a major stall source without an explicit out-of-core pipeline, e.g., the serialized baseline spends **17%** of wall-clock time on data transfers.

Taken together, naïve GPU ports and prior GPU-based methods suffer from (i) tight coupling to bespoke graph structures, (ii) fused compute–memory kernels that collapse under SIMT, and (iii) inadequate out-of-core I/O orchestration. These shortcomings leave substantial performance on the table and limit applicability to standard high-quality graph indexes.

Contributions. We observe that GPU-accelerated graph index construction is essential for index freshness at scale and show that it is practically feasible with the right abstractions: disaggregating compute- and memory-intensive work, mapping each to stage-specific kernels (Tensor Core–accelerated distance computation; hot-set–aware locality for irregular phases), and efficiently executing out-of-core with double-buffered host–device streaming. Concretely, the paper makes the following contributions:

- **CM-Disaggregated Framework for Graph Index Construction.** We propose the *first Compute–Memory (CM) disaggregated framework* for graph index construction, which decomposes the builder into several stages and implements a persistent, multi-stage GPU kernel that dynamically reconfigures the on-chip memory across these stages. This avoids cross-stage interference that collapses SIMT throughput, preserves graph quality, and exposes predictable parallelism.
- **Tensor Core–Accelerated Distance Computation.** We reformulate vector–vector distances as high–arithmetic-intensity *GEMMs* ($QC^\top$) with fused epilogues (bias addition, in-kernel top-k), executed on matrix-multiply accelerators (Tensor Core) with FP16/BF16 inputs and FP32 accumulation. We also use Micro-GEMMs to accelerate distance checking during pruning, further enhancing performance.
- **Hot-Set–Aware Dynamic Locality for Irregular Stages.** For each memory-bound phase, we semantically identify the small hot working set (e.g., visited IDs, candidate arrays, reverse-edge staging) and place it in user-managed on-chip storage (shared memory). The on-chip layout is reconfigured at stage boundaries and accessed warp-cooperatively to preserve coalescing, maximize effective bandwidth, and reduce global-memory traffic and long-latency stalls.
- **Out-of-GPU-Memory Streaming Pipeline.** For datasets larger than device memory, we split the corpus into device-fit shards and run a double-buffered pipeline with separate I/O and compute streams. While kernels build the graph for shard D_i, the next shard D_{i+1} is prefetched and the compact CSR output from D_i is drained. This overlap drives per-shard latency down to the slower of transfer or compute, cuts out-of-core overhead, and sustains high GPU utilization.
- **Extensive Evaluation.** Across seven benchmarks, including a 95 GB dataset, CMANNS reduces end-to-end build time by up to **13.05**× (vs. FAISS) and **2.20**× (vs. FLASH) at matched recall, raises the cache hit rate by up to **58.7%**, maintains peak construction memory within a few percent of CPU baselines, and preserves search quality (recall and P95 latency).

2 Background

2.1 Graph-based Index for ANNS

The approximate nearest neighbor search (ANNS) problem asks for the k closest vectors to a query under a chosen distance or similarity (e.g., Euclidean, cosine, inner product). To avoid brute-force scanning, systems build *indexes* that guide the search over large vector collections and deliver high recall at low latency.

Graph-based Indexes. Graph-based methods [33, 34] offer SOTA accuracy–latency tradeoffs in high dimensions. They construct a sparse, navigable graph whose vertices are data points and whose edges connect promising neighbors; queries then perform greedy (or beam) search from an entry point to high-quality neighborhoods.

HNSW. Hierarchical Navigable Small World (HNSW) extends NSW with multiple layers: upper layers contain progressively fewer representative nodes, enabling long-range routing; the search descends to denser layers for refinement. This skiplist-like hierarchy reduces visited nodes and yields near logarithmic scaling. HNSW has seen broad deployment in production retrieval systems [13].
NSG. Navigating Spreading-out Graph (NSG) approximates a monotone relative neighborhood structure while enforcing explicit out-degree bounds and a navigational pruning rule. The result is a sparse yet highly navigable graph that achieves high recall and competitive query times with substantially fewer edges than MRNG [11]. However, NSG construction remains costly: expansion, ordering, and diversified pruning are irregular and challenging to parallelize efficiently on GPUs.
Vamana and Vanilla NSW. As the core index of DiskANN, **Vamana** [22] generates a directed graph approximating the *Relative Neighborhood Graph (RNG)*. It employs *robust pruning* (parameter $\alpha \geq 1$) to preserve long-range edges for faster greedy routing while maintaining bounded degree. However, constructing Vamana is expensive due to multiple passes of iterative refinement, each expanding a large search list to identify pruning candidates under aggressive geometric criteria, incurring high computational overhead and random memory access. **Vanilla NSW** [33] serves as the fundamental building block of the widely used HNSW index (the bottom layer). Without hierarchical skip-pointers, insertion suffers from longer paths and frequent local minima, causing substantial latency from distance computations and irregular memory accesses.

Many graph indexes (including NSG) start from an approximate kNN graph built by *NN-Descent* [8], which iteratively improves neighborhoods via the "neighbor-of-a-neighbor" heuristic. In our pipeline, we also employ NN-Descent to obtain the initial kNN graph that seeds subsequent refinement.

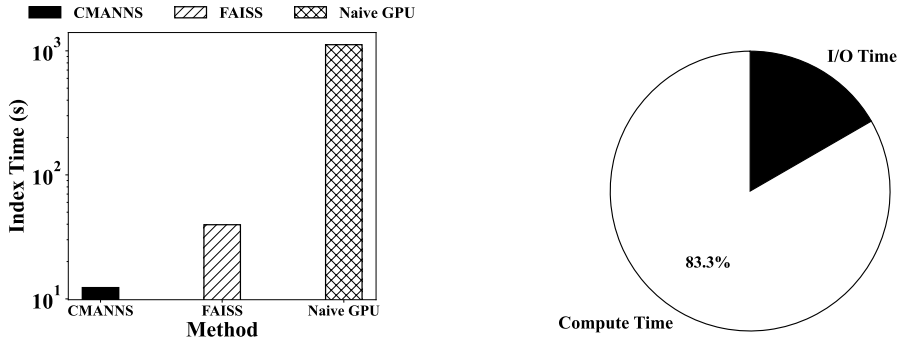
Overall, graph-based ANNS offer excellent query performance, but the *index construction* phase remains the scalability bottleneck, motivating the design choices developed in this paper.

2.2 GPU Architecture

Processing Units and Thread Hierarchy. Modern NVIDIA GPUs expose massive lightweight parallelism under the SIMT (Single Instruction, Multiple Threads) model. Threads are grouped into *warps* (typically 32) that execute in lockstep on a streaming multiprocessor (SM), and warps are scheduled from *cooperative thread arrays* (CTAs, or *blocks*) launched as a grid. Each SM issues instructions for ready warps to hide long-latency operations (e.g., memory accesses) via rapid context switching. Intra-warp exchange uses shuffle operations, while block-level collaboration uses shared memory and `__syncthreads()`. Divergence within a warp reduces effective parallelism, so branch-uniform kernels are preferred.
Memory Hierarchy. A modern GPU (e.g., NVIDIA A100) provides a layered memory system: (i) *global memory* with up to 40 GB HBM2; (ii) a chip-wide 40 MB *L2 cache*; (iii) per-SM 192 KB on-chip memory configurable as *L1* and *software-managed shared memory*; and (iv) per-thread *register files*. Staging reused data (e.g., neighbor slices) in shared memory reduces global memory traffic.
Tensor Core Matrix–Multiply Accelerators. Modern NVIDIA GPUs incorporate *Tensor Cores*, specialized units that execute small-tile matrix multiply–accumulate (MMA) operations. Exposed via WMMA intrinsics and CUTLASS, they support mixed-precision modes (e.g., FP16/BF16/TF32 with FP32 accumulation), delivering higher throughput than CUDA Cores for GEMM-class workloads.

3 Motivation and Challenges

Motivation. In production, the *construction cost* of graph indexes is widely recognized as a primary barrier to large-scale graph-based ANNS adoption [22, 46], especially because many systems refresh accuracy by *periodically rebuilding* the index from scratch. For example, AnalyticDB-V [51] maintains two index versions in tandem: one serves online traffic while the other is rebuilt in the



(a) Performance comparison. (b) I/O overhead for a subtask.

Fig. 1. Motivation Experiments: (a) NSG construction time: CPU version vs. naïve GPU port (lower is better) vs. Ours GPU-optimized version; (b) End-to-end time breakdown for a single NSG build subtask, separating computation and I/O.

background; upon completion, the system switches over. But this incurs substantial compute/memory overhead and can contend with online traffic. At scale, the cost becomes prohibitive: building a high-quality HNSW index on 200M vectors can take over ten hours on a traditional server, which is unacceptable for latency-sensitive applications (e.g., recommendation and real-time search).

Opportunity. These observations expose a gap between the excellent *query* performance of graph-based ANNS and the inefficiency of *index construction*. Bridging it requires exploiting GPU parallelism and bandwidth. By *tailoring* construction algorithms to GPU execution (warp-uniform control flow, coalesced access, shared-memory tiling) and system support (batching, overlapped I/O, out-of-core staging), build times can be reduced from hours to minutes, enabling frequent refresh for real-time, large-scale deployments.

In the remainder of this section, we characterize graph index construction on GPUs and distill the execution patterns that surface the core challenges addressed by our work.

3.1 Characteristics of Graph Index Construction

To understand where time and bandwidth are spent during graph index *construction*, we profile a standard build decomposed into five stages. The results reveal a sharp dichotomy between memory-intensive and compute-intensive behavior, explaining why monolithic (non-disaggregated) implementations underutilize GPUs.

- **Stage 1: Graph Traversal (Candidate Expansion) (Memory-Intensive).** Starting from current neighbors, the builder explores adjacency lists to discover new candidates (pointer chasing). Accesses are highly irregular and defeat caches: L2 hit rate is **8.7%**, with *long scoreboard* accounting for **78%** of warp stalls, indicating a bandwidth/latency-limited stage.
- **Stage 2: Batched Distance Computation (Compute-Intensive).** Distances from a source point to its expanded candidates are evaluated densely. This stage exhibits high arithmetic intensity and maps well to matrix units. Profiling shows that SM FP32 throughput reaches **84.8%** of peak and the kernel is compute bound.
- **Stage 3: Candidate Ordering (Memory-Intensive).** Candidates are ranked by distance via parallel sort/selection. The work is dominated by data movement rather than arithmetic: SM FP32 utilization is below **5%**, indicating a memory-centric bottleneck.

- **Stage 4: Diversified Pruning (Compute-Intensive).** The sorted list is scanned to retain a degree-bounded, diverse neighborhood (e.g., NSG rules). We measure average SM utilization of **62.2%**, so S4 behaves as compute bound.
- **Stage 5: Symmetrization (Reverse-Edge Insertion) (Memory-Intensive).** For each retained edge $u \to v$, we materialize the reverse edge $v \to u$ to ensure navigability. This stage is dominated by append-style adjacency writes, so global-store bandwidth is the primary limiter (low SM FP utilization) and stalls are mainly memory throttling or long-scoreboard. Shared-memory staging with warp-aggregated atomics improves coalescing and reduces contention, but the stage remains fundamentally memory-bound.

Overall, S1/S3/S5 are constrained by irregular access and bandwidth, whereas S2/S4 are compute-intensive. This heterogeneity explains the poor efficiency of fused kernels and motivates our CM-disaggregated design (Section 4), which applies stage-specific kernels and optimizations to match hardware strengths.

3.2 Challenges

Given the characteristics in Section 3.1, directly porting CPU builders to GPUs is ineffective: irregular pointer chasing and branch-heavy control flow misalign with SIMT hardware, causing warp divergence and de-coalesced accesses. We highlight two challenges: (1) *mixed compute–memory workload collapse*, and (2) *I/O and capacity bottlenecks* in out-of-core execution.

Challenge I: Performance Collapse from Mixed Workloads and Inefficient Memory Access. Graph construction forces GPUs to interleave compute- and memory-intensive phases within a monolithic control flow: distance evaluation is tightly coupled with candidate expansion and pruning. While CPUs can tolerate this fusion, it conflicts with SIMT execution. In CPU code, a thread fetches a candidate ID and immediately computes its distance; on GPUs, this induces scattered and divergent per-warp reads that break memory coalescing and trigger long-latency stalls, causing threads to idle. Empirically, a direct GPU port of NSG construction on SIFT1M [23] is **28× slower** than an optimized CPU version (**1121 s** vs. **39.6 s**). Profiling further shows the pruning stage suffers highly random accesses, with an L2 hit rate of only **6.18%** and sharply reduced effective bandwidth. Therefore, monolithic fusion is untenable; we must *separate* conflicting phases to restore warp-uniform, coalesced execution.

Challenge II: Limited GPU Memory and High I/O Overhead. HBM capacity is far smaller than CPU DRAM, so large datasets require out-of-core construction: vectors stay in host memory while the GPU builds the index. This inevitably incurs frequent host–device transfers; without optimization like pipelining, interconnect traffic (e.g., PCIe) becomes a dominant stall source. Figure 1b shows that for a representative NSG subtask on SIFT, data transfer accounts for **16.7%** (≈1/6) of end-to-end time. Thus, I/O is already a first-order cost and can overwhelm computation at scale, making capacity- and I/O-aware scheduling indispensable.

To scale GPU construction, we must (i) disentangle compute and irregular phases with warp-cooperative, coalesced kernels, and (ii) hide transfers via disaggregation-aware I/O orchestration (asynchronous staging, double buffering); our framework implements both with adaptive parallelism and compute–I/O overlap.

4 Framework Overview

Key Insight. The core observation is that *compute-intensive distance evaluation* and *memory-intensive graph traversal, candidate ordering, and pruning* should be *decoupled* and executed with stage-specific parallelism. This separation restores warp-uniform control flow, reduces irregular memory access and cache misses, and improves load balance across the GPU.

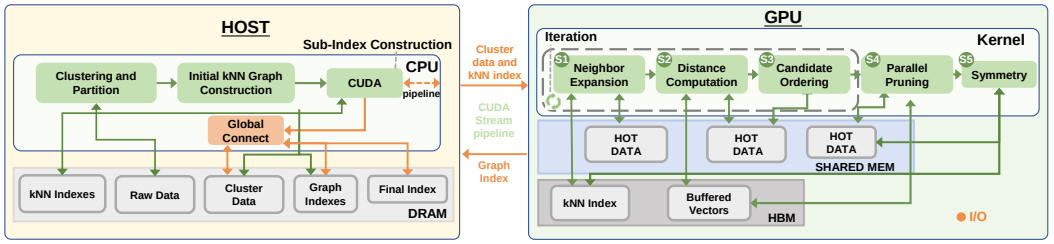


Fig. 2. Workflow of CMANNS including initialization, partition, Stage 1-5 and global connect.

Algorithm 1: Graph Index Construction Workflow.

Input: Initial kNN graph G, dataset $X \in \mathbb{R}^{n \times d}$, expansion rounds T, expansion parallelism s, entry point ep
Output: Refined graph index G^*

Initialize G^* with G;
foreach *point* $p_i \in X$ *in parallel* **do**
  Initialize hash table H_i with neighbors of ep;
  Initialize expansion list E_i for p_i;
  for $t \leftarrow 1$ ***to*** T **do**
    `// Stage 1:Fetch neighbors of expansion points`
    $C_i \leftarrow Expand(p_i, G^*, s)$;
    `// Stage2:Batched computation and sorting`
    Compute distances $\{d(p_i, c) \mid c \in C_i\}$ using warp-granularity kernels;
    `// Stage3:Parallel sorting`
    Sort C_i by distance using parallel sorting algorithms (BitonicSort & RadixSort);
  end
  `// Stage4:Parallel pruning`
  $R_i \leftarrow Prune(p_i, C_i, X)$;
  Update neighbors of p_i in G^* with R_i;
end
foreach *point* $p_i \in X$ *in parallel* **do**
  `// Stage5:optimize connectivity`
  add p_i to each of its neighbor pn_j if pn_j's neighbors is not full;
end
return G^*;

Design Principles. Guided by this insight, our approach rests on two ideas: (1) *Compute–Memory (CM) disaggregated adaptive parallelism*: tailoring kernels and work granularity to the nature of each stage, including graph traversal, distance computation, candidate sorting, pruning, and symmetry. In that case, the GPU's arithmetic and memory subsystems are utilized efficiently, without cross-stage interference; (2) *Pipeline orchestration*: overlap stage execution and explicitly *hide host–device I/O* under out-of-core operation when device memory is constrained, using asynchronous staging and double-buffering.

Building on these principles, we present CMANNS, a GPU optimized graph index construction framework that scales to large-scale vector datasets via a highly parallel, out-of-core pipeline with a novel CM-disaggregated execution.

Workflow. CMANNS transforms graph index construction into a CM-disaggregated, multi-stage pipeline that is highly parallel and out-of-core. Figure 2 illustrates the dataflow: the **host** (left)

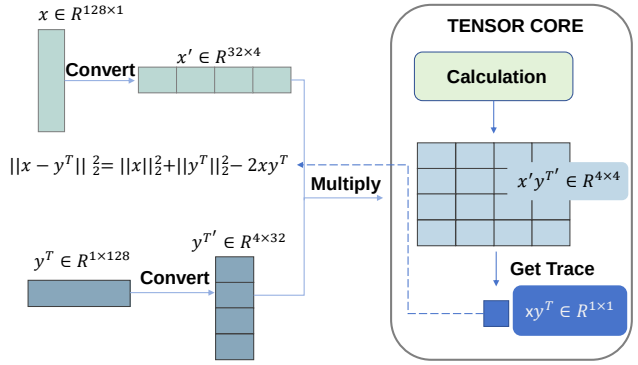


Fig. 3. Vector-to-Matrix reformulation for high-throughput distance calculation.

partitions the corpus into balanced shards, stages them in pinned memory, and manages double-buffered transfers—prefetching shard D_{i+1} while the GPU processes D_i. The **GPU** (right) runs a persistent kernel executing stages **S1–S5**, with hot data staged in shared memory to mitigate irregular accesses. After **S5**, only compact outputs (CSR adjacency slices) are returned to the host for stitching. The end-to-end process comprises three parts:

(1) Partitioning and Streaming. The corpus is split into device-fit shards $D_1, \ldots, D_n$ via clustering (e.g., Faiss-IVF [26] or k-means) and a double-buffered pipeline overlaps data transfers with GPU computation (Section 5.3).

(2) Sub-Index Construction ($D_i \rightarrow G_i$). Each shard D_i is refined to achieve G_i through iterative stages **S1–S5**: neighbor expansion, distance computation, candidate ordering, pruning, and symmetrization (Algorithm 1).

(3) Global Stitching. Sub-indices are merged via a meta-graph G_{meta} built from boundary points using the *same* pruning rules, ensuring a unified, navigable index.

5 CMANNS's System Design

Optimization Overview. With CM disaggregation in place, we employ three complementary families of optimizations: (i) *Tensor Core-accelerated distance computation for compute-intensive tasks.* CMANNS lifts distance evaluation and inner-product calculations to high-intensity GEMMs on matrix-multiply accelerators (Tensor Cores) with fused epilogs (bias addition, in-kernel top-k), approaching peak arithmetic throughput; (ii) *Hot-set aware dynamic locality for memory-intensive tasks.* CMANNS maximizes effective bandwidth via structure-of-arrays layouts, degree-bucketed adjacency, warp-cooperative or coalesced accesses, and cache-friendly compaction to increase L1/L2 residency and reduce cache thrashing; and (iii) *Out-of-GPU-memory streaming pipeline under device-memory constraints.* We reduce and hide host–device I/O using asynchronous staging, double buffering, stream concurrency, and compact adjacency deltas. Together, these techniques accelerate computation, saturate memory bandwidth, and minimize the I/O overhead under tight device-memory budgets.

5.1 Tensor Core-Accelerated Computation

Under CM-disaggregated execution (Section 4), compute-heavy work is isolated into two hotspots: **Stage 2** (batched distance computation) and **Stage 4** (distance computation and checking within pruning). Our objective is to maximize arithmetic throughput in these stages while preserving accuracy. We achieve this via (i) algebraic reformulation of vector distances into high–arithmetic-intensity matrix multiplications, (ii) aggressive tiling to increase data reuse, (iii) mixed-precision execution with error-bounded refinement, and (iv) micro-batched matrix products within pruning.

Algebraic Reformulation to GEMM. Let $Q \in \mathbb{R}^{b\times d}$ be a batch of query vectors and $C \in \mathbb{R}^{m\times d}$ a candidate block. For the squared Euclidean distance,

$$\|\mathbf{q}_i - \mathbf{c}_j\|_2^2 = \|\mathbf{q}_i\|_2^2 + \|\mathbf{c}_j\|_2^2 - 2\,\mathbf{q}_i\mathbf{c}_j^\top.$$

Define $n_Q \in \mathbb{R}^b$ with $[n_Q]_i = \|\mathbf{q}_i\|_2^2$ and $n_C \in \mathbb{R}^m$ with $[n_C]_j = \|\mathbf{c}_j\|_2^2$. Then, the distance matrix satisfies

$$D \;=\; \mathbf{1}\, n_C^\top \;+\; n_Q\, \mathbf{1}^\top \;-\; 2\,QC^\top,$$

i.e., a rank-1 row bias, a rank-1 column bias, and a single *GEMM*, as shown in figure 3. For cosine similarity, we normalize the rows of Q and C, and use $D_{\cos} = 2 - 2\,QC^\top$. This conversion lifts Stage 2 from vector–vector kernels to high-intensity matrix multiplication, where GPUs reach near-peak throughput.

Matrix-Multiply Accelerators and Fused Epilogues. We dispatch $QC^\top$ via Tensor-Core matrix-multiply accelerators using mixed precision (FP16/BF16 inputs, FP32 accumulation). Two epilogue fusions further reduce memory traffic: (i) *Bias fusion*: incorporate $\alpha = -2$ into the GEMM and add n_Q / n_C via broadcast in the epilogue so that only final distances are written; (ii) *Top-k fusion*: within each output tile, maintain a warp-local selection structure (e.g., a fixed-size min-heap or selection network) and emit only the per-row top-k candidates, limiting global writes by $O(m)$. Both fusions convert what would be GEMM→store→scan into a single, write-minimal kernel.

Tiling, Blocking, and Data Layout. We adopt a 3-level tiling strategy: (a) *thread block tiles* sized to accelerator fragments, (b) *warp tiles* for register blocking, and (c) *MMA fragments* for on-core pipelines. Vectors are stored in a structure-of-arrays (SoA) layout with 128-bit alignment; tiles prefetch rows of Q and C into shared memory and registers. This design maximizes reuse along $K = d$ and raises arithmetic intensity beyond the device roofline threshold, keeping S2 compute-bound.

Then, to preserve ranking accuracy under mixed precision, we use: (i) *calibrated scaling* per dimension (or per block) to avoid overflow/underflow; (ii) FP32 accumulation in GEMM; (iii) *refinement on a small set*: re-evaluate the distances of each row's top-k' ($k' \approx 2k$) in FP32 vector code. Empirically, $k' = 2k$ keeps recall unchanged while retaining $> 90\%$ of the mixed-precision speedup. The refinement is bandwidth-light (tiny working set) and does not disturb steady-state GEMM throughput.

Micro-GEMMs for Pruning (S4). Diversified pruning (e.g., NSG-style) requires repeated distance checks between a candidate v and the currently accepted neighbor set $S(u)$. Rather than scalar loops, we batch these checks as micro-GEMMs. Let $V \in \mathbb{R}^{t\times d}$ be a tile of candidates and $N \in \mathbb{R}^{r\times d}$ the stacked neighbors (with $r \le M$). We compute $G = VN^\top$ using the same accelerator path (mixed precision, FP32 accumulation), and apply the pruning test via vectorized epilogues with precomputed norms. This yields two benefits as following: (i) reuse of N's rows across many candidates (higher intensity), and (ii) warp-uniform control for the comparison logic. When r is small, we switch to register-blocked WMMA fragments to avoid under-utilization (an *auto-tuner* chooses cutovers by d, t, r).

After that, we launch Stage 2 (S2) and Stage 4 (S4) as **persistent kernels** with device-side work queues. Then, batches for GEMMs (rows of Q) and micro-GEMMs (tiles of V) are sized to maintain high occupancy while respecting shared memory limits, together with warp-cooperative loading and warp-synchronous reductions (e.g., `shfl_xor`) to keep control flow uniform.

Complexity and Throughput Implications. Let n be the shard size and d be the dimensionality. S2's cost becomes one GEMM of shape $b \times d$ by $m \times d$ (transposed), leading to $O(bmd)$ MACs with accelerator throughput. Then, the bias adds are $O(b + m)$. As for S4, it processes t-by-r micro tiles

with $O(trd)$ MACs but amortizes N's loads across t candidates. Note that MAC denotes Multiply-Accumulate, computing $a \leftarrow a + b \times c$ in one cycle—the standard unit for GEMM workload and GPU arithmetic throughput.

Putting It Together. In **S2**, after neighbor expansion assembles candidate pools, we compute distances via a single large GEMM with fused bias and fused top-k epilogues, followed by FP32 refinement on a narrow set. In **S4**, pruning evaluates candidate–neighbor distances via micro-GEMMs and vectorized acceptance tests with early-elision.These techniques elevate both stages to accelerator-class throughput, converting them from bottlenecks into steady, compute-bound kernels within the CM-disaggregated pipeline.

Remark. The above optimizations are orthogonal to memory-centric techniques (Section §5.2) and out-of-GPU-memory orchestration (Section §5.3); together, they enable minute-scale construction at a large-scale while matching baseline recall.

5.2 Hot-Set–Aware Dynamic Locality

Memory-intensive stages (e.g., neighbor expansion and candidate ordering) are dominated by irregular, high-latency accesses to GPU global memory, where hardware-managed caches (L1/L2) provide limited benefit under weak spatial and temporal locality. Under CM-disaggregated execution (Section 4), we target these stages with a *software-managed* approach that reshapes access patterns and preferentially serves the hottest data from on-chip storage.

Design Principle. We introduce *Hot-Set–Aware Dynamic Locality*. The core idea is to treat the limited shared memory as a *polymorphic cache*. Instead of costly runtime profiling, we leverage algorithm semantics to identify stage-specific "hot sets" (bursty, repeatedly accessed data), and dynamically reconfigure on-chip storage so the current stage's hot set is served from the lowest-latency tier. Concretely, we instantiate this principle in the three memory-intensive stages of our pipeline: S1, S3, and S5.

Stage-Local Hot Sets. We identify a small hot working set per stage and apply lightweight on-chip handling accordingly. In **S1** (Neighbor Expansion), the hot set includes the *visited set* and *frontier adjacency slices*, which are repeatedly touched during irregular traversal and deduplication. In **S3** (Candidate Ordering), the hot set comprises the *per-point candidate arrays* (IDs and distances) that undergo comparison-heavy ordering passes. In **S5** (Symmetrization), the hot set becomes write-dominated; we next detail how we handle skewed reverse-edge insertions.

S5 Write Hotspots (High-Degree Nodes). In **S5**, destinations that receive disproportionately many reverse-edge insertions within a micro-batch are treated as *write hotspots* (commonly correlated with high-degree nodes under skewed graphs). We address them via a lightweight mechanism of identification, staging, and flushing:

- **Identification:** For each micro-batch, we build a shared-memory histogram over reverse-edge destinations to estimate insertion frequency, and select the Top-K destinations as hotspots under the on-chip budget.
- **Caching:** For each hotspot, we allocate a small on-chip *staging buffer*. We cache only the *append tail* (pending updates) rather than the full adjacency list, aggregating updates locally.
- **Flush:** We flush staged edges using *warp-aggregated atomics*, reserving a contiguous segment in the global adjacency list and converting many random, contended atomic appends into a few coalesced HBM writes.

This keeps the hottest S5 updates on-chip as long as possible and turns contention into bulk, bandwidth-friendly traffic.

Dynamic On-Chip Reconfiguration (Running Example). We track a single shared-memory region across stages, repurposed without device-wide synchronization:

- **Step 1: Visited-Set Cache (S1 Expansion).** Shared memory is set as a *Visited Set* to filter duplicates (e.g., a bitset for compact ID ranges; a cuckoo-style table for sparse IDs), with a small *CSR-like adjacency tile* buffer to stage frontier neighbor slices.
- **Step 2: Candidate Scratchpad (S3 Ordering).** The region is reconfigured into a *Candidate Scratchpad* in Structure-of-Arrays layout (`cand_id`, `cand_distance`), serving comparison-heavy ordering so only compacted top-k results are written back.
- **Step 3: Staging Buffer (S5 Symmetrization).** The region morphs into a *Reverse-Edge Staging Buffer*: hotspot destinations accumulate pending reverse edges on-chip and flush via warp-aggregated atomics as coalesced HBM writes.

Reconfiguration cost is a local, block-scoped switch; it does not introduce device-wide barriers and thus preserves the pipeline's barrier-free execution.

Correctness (Semantics-Preserving). Hot-set caching is an implementation strategy and does not change algorithmic semantics:

- **S1.** Deduplication is idempotent; block-local filtering only changes where the checks happen (on-chip vs. global), not the accepted neighbor set.
- **S3.** Ordering or selection operates on the same candidate multiset; any ties can be deterministically broken by ID.
- **S5.** Staging changes the *write path* (aggregate then flush) but commits the same reverse edges; warp-aggregated atomics only coalesce reservations and stores.

Schematic. Below is a compact schematic of how the same shared-memory region is repurposed across S1/S3/S5:

```
S1: configure_shared(VisitedSet, AdjTile)
-> expand + dedup on-chip -> emit compact candidates
S3: configure_shared(CandIDs, CandKeys)
-> order/select on-chip -> write back top-k
S5: configure_shared(Staging, Hist)
-> identify hotspots -> stage tails -> coalesced flush
```

Remark. Hot-Set–Aware Dynamic Locality reduces random global reads/writes by converting them into predictable on-chip operations, yielding (i) more coalesced global traffic with fewer contended atomics, (ii) smaller, denser working sets for unavoidable global accesses, and (iii) fewer stalls in memory-bound loops via more uniform control flow. The technique is orthogonal to compute-side accelerations (Section §5.1) and composes with out-of-core streaming (Section §5.3).

5.3 Out-of-GPU-Memory Streaming Pipelining

Limited GPU device memory (HBM) prevents loading large-scale dataset onto a single GPU. When vectors reside in host DRAM, naïve staging (copy–compute–copy) serializes I/O and compute, wasting bandwidth and device cycles. Therefore, we design an *out-of-core* pipeline that *streams* data shards from the CPU to the GPU, overlaps transfers with kernel execution, and returns compact results to the host—turning the end-to-end latency per shard into the max of compute and I/O rather than their sum. We first describe how we shard the corpus, stitch sub-indices into a unified index, and finally detail the pipeline that realizes this overlap.

Clustering-Based Partitioning and Host Layout. We first partition the corpus D into balanced shards $D_1, \ldots, D_n$ using a GPU-accelerated coarse quantizer (e.g., Faiss-IVF [26]) or k-means. Shards bound the working set to fit device memory and increase spatial locality for local graph construction. On the host (CPU), vectors are stored in page-locked (pinned) buffers organized in a structure-of-arrays format for coalesced device reads.

Global Connectivity and Hierarchical Merging. Shards are constructed in a streaming manner (described next), and we stitch them via a compact meta-graph G_{meta}. After all sub-indices

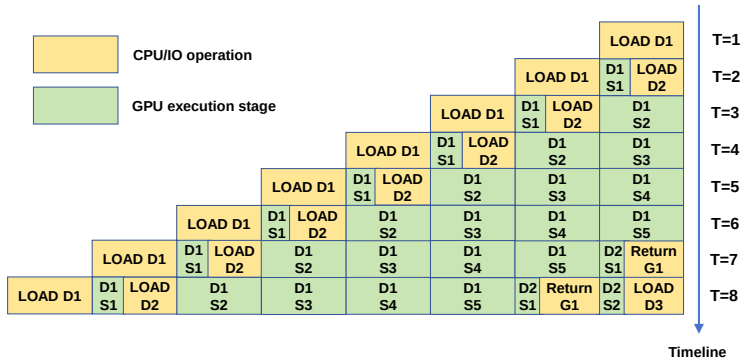


Fig. 4. Overlapped pipeline between GPU computation and CPU–GPU I/O.

$\{G_1, \ldots, G_n\}$ are built, we sample boundary (gateway) points per shard and build a compact meta-graph G_{meta} over $\mathcal{B} = \bigcup_i \mathcal{B}_i$ using an NSG-style construction with tight degree budgets, materializing a small set of rule-consistent cross-shard edges. At query time, G_{meta} provides a few good entry points (and thus a small shard set) quickly; the subsequent traversal remains mostly shard-local for efficiency, but may follow cross-shard edges when beneficial.

Out-of-Core Pipeline Orchestration. To process datasets exceeding GPU memory limits, we implement a standard double-buffered producer–consumer scheme. We allocate two device-side staging buffers and utilize separate CUDA streams (s_{copy} and s_{compute}) to overlap data transfer with kernel execution. The host (producer) stages shards into pinned memory and manages a bounded ring queue, while the device (consumer) executes stages **S1**–**S5** using persistent kernels. This design allows s_{copy} to prefetch shard D_{i+1} asynchronously while s_{compute} builds G_i, effectively hiding PCIe latency (i.e., turning per-shard latency into $\max\{\text{I/O}, \text{compute}\}$).

Topological Integrity of Partitioned Construction. While partitioning alters the construction order, CMANNS maintains the final graph's *topology* and *navigability* via *Global Connectivity*. Since HNSW-like construction is stochastic (seed and order sensitive), we aim for *algorithmic fidelity* rather than bit-wise identity. Partitioning is solely for memory management; partitions do not remain disjoint. *Global Connectivity* integrates shards using *identical pruning criteria* (e.g., NSG's monotonic pruning) to create valid cross-partition edges, yielding a unified graph index consistent with the original graph definition.

Differences from Prior Work. Partitioning for out-of-core GPU construction in CMANNS is purely an *execution strategy*: we preserve *algorithmic fidelity* by using the same neighbor-candidate generation, pruning and selection rule as the vanilla algorithm, so cross-shard edges are rule-consistent rather than heuristic add-ons.

- **Versus CAGRA [38].** GPU-oriented designs such as CAGRA target a different graph family with hardware-friendly structural constraints (e.g., fixed degree and predefined layouts) to simplify SIMT execution. In contrast, CMANNS introduces no new topology constraints: it preserves the rule-based construction (hence the variable-degree behavior) and only changes the *construction schedule* to support out-of-core GPU execution.
- **Versus HNSW [34].** HNSW wide links are optional *additional* heuristic shortcuts beyond the baseline neighbor-selection rule. Our cross-shard edges differ by definition: they are accepted only if they satisfy the *same* baseline pruning or selection criterion, i.e., they are rule-consistent edges obtained from an expanded candidate pool, not additional links outside the vanilla rule.
- **Versus Vamana [22].** While both use partitioning, objectives differ. Vamana (DiskANN) targets *hierarchy-aware* large-scale search under constrained memory and is engineered for out-of-core access, often with boundary-aware handling *during partitioning*. In contrast, CMANNS uses

disjoint, zero-replication partitioning only for *construction-time* out-of-core GPU builds, and maintains navigability via rule-consistent cross-shard edges.

6 Implementation and Cost Model

Codebase and Tooling. CMANNS comprises ∼5K lines of C++17 and CUDA. We use CMake for builds, compile with `-O3` and device-side LTO, and rely on standard GPU libraries where beneficial (e.g., cuBLAS/cuBLASLt or CUTLASS-style kernels for GEMM, CUB/Thrust for scans/selection). Host I/O uses page-locked (pinned) buffers. The implementation is Linux-oriented and includes scripts for reproducible runs and profiling (`nsys`, `ncu`).

Module Layout. (i) *Partitioner & host layout*: GPU-accelerated Faiss-IVF (or k-means fallback) produces balanced shards and SoA-formatted, pinned host buffers. (ii) *Out-of-core engine*: double-buffered H2D/D2H streams and a bounded ring queue for descriptors (§5.3). (iii) *Scheduler*: a producer–consumer orchestrator with backpressure (q) and event fencing. (iv) *Device builder*: persistent, stage-structured kernels for S1–S5 with on-chip reconfiguration (§4, §5.1, §5.2). (v) *Meta-graph*: boundary sampling to build G_{meta}.

Parallelism and Occupancy. We use persistent kernels with per-SM work queues. Tiles are sized by an occupancy model that considers registers per thread, shared-memory per block, and MMA fragment footprints. Warp-uniform control is enforced in hot loops; warp shuffles primitives (`__shfl_*`) and cooperative groups handle reductions and scans.

6.1 Persistent Kernel Execution

To maximize instruction throughput, we implement the construction pipeline as a single persistent kernel. A critical challenge is enabling effective overlap between memory-bound stages and compute-bound stages so that latency can be hidden.

Macro-Level Overlap (System). As detailed in Section 5.3, we overlap the PCIe transfer of the next shard (D_{i+1}) with the kernel execution on the current shard (D_i) using independent CUDA streams and double buffering.

Micro-Level Overlap (Intra-Kernel). Our persistent kernel processes tasks via a global work queue: each block repeatedly fetches the next task using a non-blocking atomic counter. Within each block, stages execute sequentially ($S_1 \to \ldots \to S_5$) with only minimal intra-block synchronization. Across blocks, there is no device-wide barrier: each block independently fetches the next task. Overlap is enabled by:

- **Stage Heterogeneity:** Stages stress different bottlenecks (memory vs. compute), so stalls in one stage do not preclude progress elsewhere under SIMT scheduling.
- **Occupancy Provisioning:** We utilize the occupancy model to maintain adequate resident warps, enabling latency hiding when certain warps stall on long-latency operations.

As formalized in Theorem 1, global phase-locking is unstable, yielding a steady-state mixture of blocks across stages (i.e., different blocks reside in different stages simultaneously). With stage heterogeneity and sufficient residency, SIMT scheduling exploits this mixed-stage execution for effective time-averaged overlap.

Theorem 1 (Phase-Lock Instability)**.** Assume that (i) blocks execute a persistent loop, repeatedly pulling tasks from a global queue, (ii) there is no device-wide resynchronization across blocks between stages (or tasks), and (iii) at least one stage has non-degenerate, data-dependent execution time across tasks. Then, global phase-locking is unstable: blocks desynchronize, and in steady execution, different blocks occupy different stages.

Proof. Pick two blocks b_1, b_2 and let $t_b(n)$ be the wall time when block b completes its n-th iteration. Define $\Delta_n \triangleq t_{b_1}(n) - t_{b_2}(n)$. Let i_n and j_n denote the task indices processed by

b_1 and b_2 in their n-th iteration, respectively; since blocks fetch tasks independently from the global queue, generically $i_n \neq j_n$. Under (i), both sequences $\{t_b(n)\}$ exist for all n, and $\Delta_{n+1} = \Delta_n + \sum_{k=1}^{5} \left(T^{(k)}(i_n) - T^{(k)}(j_n)\right)$, where $T^{(k)}(i)$ is the stage-k execution time for task i. By (iii), stage durations vary across tasks, so the increment is generically nonzero; hence, Δ_n drifts over iterations. By (ii), no global barrier resets Δ_n to zero. Therefore, b_1 and b_2 occupy different pipeline stages at the same wall time, i.e., blocks desynchronize. □

6.2 Cost Model and Shard Size Selection

Under out-of-core execution, we determine the shard size B under the overlapped pipeline by jointly considering (i) steady-state throughput and (ii) an explicit HBM-capacity bound for out-of-memory (OOM) free double buffering.

Performance-Oriented Cost Model. With asynchronous copy/compute overlap, the per-shard time is approximated by the model:

$$T_{\text{shard}}(B) \approx \max\Big(T_{\text{I/O}}(B),\ T_{\text{compute}}(B)\Big),$$
$$T_{\text{I/O}}(B) \approx \max\left\{\frac{B_{\text{in}}(B)}{\beta_{\text{H2D}}},\ \frac{B_{\text{out}}(B)}{\beta_{\text{D2H}}}\right\}.$$

Here, β_{H2D} and β_{D2H} are sustained host→device and device→host bandwidths, $B_{\text{in}}(B)$ is the input vector payload (dominant), $B_{\text{out}}(B)$ is the compact CSR/metadata emitted after **S5** (typically $B_{\text{out}} \ll B_{\text{in}}$), and $T_{\text{compute}}(B)$ is the device compute time for **S1–S5**. We choose B to roughly balance $T_{\text{I/O}}(B)$ and $T_{\text{compute}}(B)$ for high throughput.

HBM Footprint Prediction Model (Memory Bound). For shard size B, the peak-resident HBM footprint is conservatively upper-bounded as:

$$S_{\text{shard}}(B) \approx B\,(DS_{\text{data}} + RS_{\text{id}} + LS_{\text{cand}} + S_{\text{aux}}) + S_{\text{const}}.$$

Here, B is the number of vectors in a shard, D is the vector dimension, R is the hard maximum degree cap, and L is the fixed intermediate candidate-pool size. The terms S_{data}, S_{id}, and S_{cand} denote the byte size of a vector element (e.g., `float`), a node index (e.g., `uint32`), and a candidate record (e.g., distance-ID pair), respectively. S_{aux} accounts for per-point bookkeeping (e.g., visited flags), and S_{const} captures constant overheads such as metadata. $S_{\text{shard}}(B)$ separates into a shard-resident term scaling linearly with B plus constant overhead S_{const}. Stage-local scratch is pre-allocated and reused across **S1–S5** (contributing only its peak), while shard-resident payload remains allocated throughout processing. Since the coefficient is determined by fixed algorithmic bounds (D, R, L), the footprint is predictable and usable to choose an OOM-free shard size under double buffering.

Shard Size Selection. With double buffering, two shards may be in-flight (compute + transfer). We choose B to satisfy:

$$2 \cdot S_{\text{shard}}(B) + S_{\text{pipe}} \leq \gamma \cdot \text{HBM}_{\text{cap}},$$

where S_{pipe} captures fixed pipeline buffers and $\gamma < 1$ reserves headroom. This bound enables precomputing the largest feasible B for stable pipelining without runtime OOM; we then optionally refine B via warm-up to better match $T_{\text{compute}}(B)$ with $T_{\text{I/O}}(B)$.

7 Evaluation

In this section, we evaluate CMANNS across multiple datasets and application scenarios against strong graph-based baselines. We report *full-in-GPU-memory build time*, *out-of-GPU-memory build time*, and *online search quality including latency and recall* at matched accuracy. We also conduct ablation studies to isolate the contributions of each optimization: (i) *Tensor Core–accelerated* distance computation, (ii) *hot-set–aware dynamic locality* for memory-intensive stages, and (iii) the *out-of-GPU-memory streaming pipeline*. In addition, we profile *device memory usage* and *L1/L2 cache hit*

Table 2. Characteristics of datasets.

Dataset	Vector	Dimension	Size	Data Type
SIFT1M	1M	128	470MB	Float
GIST	1M	960	3.6 GB	Float
DEEP	10M	96	3.6 GB	Float
SIFT100M	100M	128	47GB	Float
SIFT200M	200M	128	95GB	Float
BIGCODE	∼10M	768	30GB	Float
DATACOMP	12.8M	768	37GB	Float

Table 3. Index construction time (seconds) for NSG, HNSW, and NN-Descent on three datasets: CMANNS (Ours) vs. FAISS.

Index	NSG			HNSW			NN-Descent		
Dataset	CMANNS	FAISS	Speedup	CMANNS	FAISS	Speedup	CMANNS	FAISS	Speedup
SIFT1M	12.15	39.63	**3.26x**	23.25	44.71	**1.92x**	9.04	26.01	**2.88x**
GIST1M	32.77	427.51	**13.05x**	172.59	469.02	**2.72x**	15.77	134.27	**8.51x**
DEEP10M	105.78	815.27	**7.71x**	470.85	840.80	**1.79x**	102.09	248.24	**2.43x**

rates to quantify resource efficiency and verify that CMANNS sustains high arithmetic throughput while maximizing effective bandwidth.

7.1 Experimental Setup

Settings. Unless otherwise stated, all experiments are conducted on a single server equipped with two Intel Xeon Gold 6426Y CPUs (16 cores / 32 threads each; 32 physical cores / 64 hardware threads total), 503 GB DRAM, a local NVMe SSD, and one NVIDIA A100 GPU (40 GB HBM; CUDA 12.4). The operating system is Ubuntu 20.04.6 LTS and the host compiler is g++ 9.4.0. For the CPU scaling study with 64 physical cores (§7.5), we additionally use a separate server based on AMD EPYC 9554.
Datasets. We evaluate CMANNS on standard public benchmarks widely adopted in ANNS studies [6, 23]: *SIFT* (128D) at scales of **1M** (470,MB), **100M** (47,GB), and **200M** (95,GB); *GIST* (960D) at **1M** (3.6,GB); and *DEEP* (96D) at **10M** (3.6,GB). Furthermore, to assess performance on high-dimensional workloads, we incorporate two large-scale datasets used in the FLASH paper [46]: *BIGCODE* (768D) at ∼**10M** (30,GB) and *DATACOMP* (768D) at **12.8M** (37,GB). Detailed statistics are summarized in Table 2.
Baselines. We benchmark CMANNS against a comprehensive suite of SOTA libraries: (1) **FAISS** [26], the most widely used open-source ANNS library; (2) **FLASH** [46], a novel CPU-based optimization strategy leveraging LUTs; (3) **CAGRA** [38] (via NVIDIA CuVS [36]), the current high-performance GPU-based graph index; and (4) **DiskANN** [22], specifically for its in-memory build.

7.2 Overall Performance

We evaluate CMANNS against the strong CPU baseline FAISS, with the primary goal of accelerating *graph construction* while preserving recall. Following our partitioned workflow, we focus on shard scales typical in practice (1–10M vectors) and report results on **SIFT1M**, **GIST1M**, and **DEEP10M** as shown in Table 3.

The observed variation (GIST1M 13.05× > DEEP10M 7.71× > SIFT1M 3.26×) stems from a combination of arithmetic intensity, memory irregularity, and shard scale. First, *arithmetic intensity* grows with dimension d: distance evaluation dominates high-d dataset, and mapping ℓ_2/IP distances to GEMM increases compute reuse (tiles in shared/L2). Thus, the **GIST1M** approaches the accelerator peak and yields the largest gains. By contrast, the **SIFT1M** offers less compute per byte and correspondingly less benefit from Tensor Cores. Second, larger shards exacerbate *irregular memory*

Table 4. Mapping Vamana and NSW to CMANNS pipeline stages.

Stage	Vamana (DiskANN) Mapping	NSW Mapping
S1	Greedy Graph Traversal	Long-Path Random Search
S2	Dense Dist. (Query vs. Candidates)	Dense Dist. (Query vs. Candidates)
S3	Priority Queue Maintenance	Dynamic Sorted Candidate List
S4	Robust Pruning (α-inequality check)	Heuristic Neighbor Selection (Top-M)
S5	Bidirectional Edge Insertion	Reverse Edge Maintenance

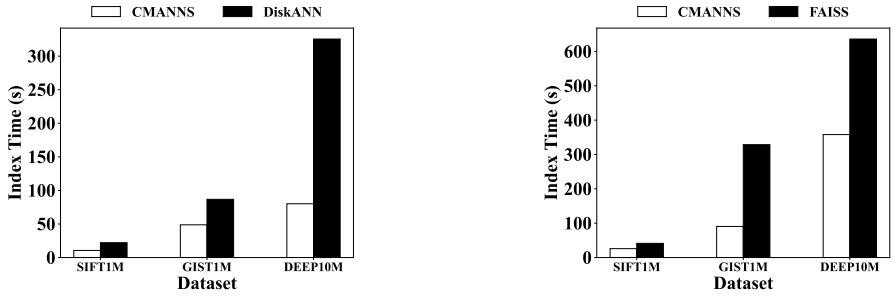
pressure: in the **DEEP10M**, memory-bound phases (expansion/ordering/pruning) benefit from hot-set–aware on-chip caches (visited sets, candidate scratchpads, reverse-edge staging) that coalesce traffic and reduce long-scoreboard stalls, explaining strong speedups even at the smallest dimension (96D). Third, *amortization* matters: small shards (SIFT1M) provide less opportunity to amortize kernel launches and pipeline warm-up, capping speedup despite ample parallelism.

NSG Performance. Gains are concentrated where the redesign regularizes work and raises intensity. In *S2 (distance computation)*, reformulating to GEMM with fused epilogs (bias/add, in-kernel top-k) allows Tensor Cores to supply the bulk of improvements on high-d datasets. In *S1/S3 (expansion/ordering)*, stage-local shared-memory structures (visited-set cache, candidate scratchpad) convert scattered global accesses into warp-cooperative on-chip operations, improving L1/L2 residency and effective bandwidth. In *S4 (pruning)*, micro-GEMMs speed candidate checks, while triangle-inequality screening prunes random probes, thereby containing what would otherwise be a memory-dominated cost. Finally, moving *S5 (symmetrization)* to the GPU and using warp-aggregated atomics collapses many fine-grained appends into a few coalesced writes, trimming the write-bound tail. Notably, CMANNS achieves larger speedups on higher-dimensional datasets (e.g., GIST), where Tensor Core–accelerated distance GEMMs offer greater arithmetic intensity and our locality optimizations more effectively reduce memory stalls.

HNSW Performance. HNSW exhibits smaller but consistent gains than NSG due to its more complex control flow (multi-layer updates, per-node degree budgeting). Our GPU builder executes candidate expansion, distance calculation, neighbor selection, and layer commits on the device. Even so, the graph updates remain synchronization-sensitive; we mitigate this with warp-aggregated atomics and bulk-synchronous layer commits. Empirically, we observe speedups of 2.72× on GIST1M, 1.92× on SIFT1M, and 1.79× on DEEP10M at matched recall, reflecting that accelerating the dominant compute kernels and confining irregular updates yields meaningful end-to-end benefits despite residual synchronization overheads.

NN-Descent Performance. NN-Descent has simple control flow and is dominated by distance evaluations, making it especially amenable to GPU acceleration. Our implementation preserves the downstream NSG structure and recall while substantially reducing build time. At matched recall, we observe speedups of 8.51× on **GIST1M** (its high dimensionality maximizes Tensor Core gains), 2.88× on **SIFT1M**, and 2.43× on **DEEP10M**.

Analysis of Inter-Stage Data Movement Overhead. A potential concern in pipelined GPU execution is extra HBM traffic due to materializing stage intermediates. In our persistent multi-stage kernel, stage handoff is performed within the same kernel invocation, and stage-local intermediates are kept on-chip when possible via shared memory reconfiguration and register reuse (Section 5.2). As a result, HBM writes are dominated by necessary persistent updates (e.g., committed edges) rather than transient per-stage state. We quantify this using profiling on SIFT1M for NSG construction. The measured HBM write traffic is ≈8.9 GB, costing ≈56 ms—only 0.46% of the total 12.15 s execution time. This confirms that inter-stage data movement is not a bottleneck in our pipelined execution.



(a) Vamana testing. (b) NSW testing.

Fig. 5. Evaluation on Vamana and NSW.

7.3 Adaptation to Other Indices

We validate the generality of CMANNS's CM disaggregation framework on *Vamana* (DiskANN [22]) and *vanilla NSW* [33] by mapping their construction onto our Stage 1–Stage 5 pipeline and evaluating construction performance.

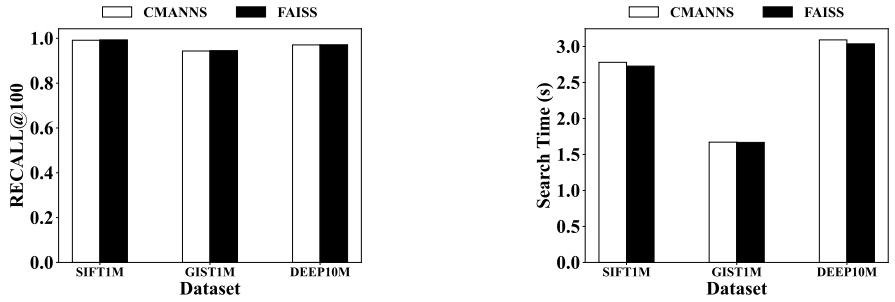
Vamana and NSW Adaptation. Firstly, Table 4 shows that the construction flows of Vamana and NSW can be systematically mapped onto our five-stage pipeline. Although their pruning rules differ (e.g., Vamana's α-based pruning), they rely on the same fundamental operators: candidate expansion/traversal (Stage 1), distance evaluation (Stage 2), candidate ordering (Stage 3), pruning or neighbor selection (Stage 4), and symmetrization/connectivity enforcement (Stage 5). CMANNS accelerates these operators by addressing their dominant hardware bottlenecks: Tensor Cores speed up arithmetic-intensive distance evaluation and pruning (Stage 2/4), while our hot-set-aware dynamic locality mitigates irregular memory accesses in traversal and symmetrization (Stage 1/5). Consequently, CMANNS generalizes to these indices without modifying their algorithmic invariants. For Vamana, we focus on accelerating the *in-memory* topology construction component, decoupled from DiskANN's SSD-based search pipeline.

Vamana Performance Comparison. For the testing of Vamana, we compare against DiskANN's optimized *in-memory* builder (excluding SSD I/O). In Figure 5a, CMANNS achieves **1.78×–4.06×** speedup, peaking on DEEP10M (**4.06×**). This is expected because Vamana's build is dominated by irregular traversal, queue maintenance, and bidirectional insertions (Stages 1/3/5), whose *fractional cost increases with graph scale*. On DEEP10M, the larger working set maximizes these memory-bound phases, so our hot-set–aware locality contributes more to the construction time. Meanwhile, Stages 2/4 still benefit from Tensor Core computation, yielding consistent gains. Overall, these results validate CMANNS's generality across different graph topologies and pruning rules.

NSW Performance Comparison. Figure 5b shows CMANNS speeds up NSW by **1.59×–3.63×** over FAISS CPU, with the largest gain on GIST1M (**3.63×**). Unlike Vamana, NSW construction is more *distance-evaluation heavy* under high dimensionality: computing candidate distance and neighbor selection (Stages 2/4) becomes compute-bound as d grows. Therefore, on GIST1M (960D), Tensor Core–accelerated distance and pruning yield the largest benefit, while our locality optimizations still reduce stalls in traversal/symmetrization (Stages 1/3/5). These results show CMANNS remains effective for flat graph such as NSW, and the peak speedup aligns with where compute dominates.

7.4 Vector Search Quality

In this section, we verify that CMANNS 's faster *construction*, using mixed-precision Tensor Core kernels with FP32 accumulation, preserves search accuracy and latency. Under matched query budgets (identical entry points and stopping criteria), we report *Recall@100* and per-query latency



(a) RECALL@100 testing.

(b) Search time.

Fig. 6. Search quality across datasets. Evaluation uses randomly sampled queries: 5,000 for SIFT and DEEP, and 500 for GIST, drawn from the provided query sets.

using the *unmodified* NSG search procedure on indexes built by CMANNS across datasets as shown in Figure 6. In all cases, CMANNS matches FAISS within $\leq 2\%$ for both Recall@100 and latency (P95), indicating that the GPU-first build preserves the graph structure, navigability, and thus query behavior. Because our contribution targets building rather than querying, we intentionally reuse the canonical graph-based search to isolate index quality without introducing new query heuristics.

7.5 Comparison with SOTA CPU Baseline

To rigorously evaluate CMANNS against state-of-the-art CPU-based construction, we compare with **FLASH** [46] on HNSW. FLASH represents the current peak of CPU optimization, accelerating construction via AVX-512-based Look-Up Tables (LUTs) and cache-efficient memory management. We also introduce two large-scale, high-dimensional datasets, **BIGCODE** and **DATACOMP**, adopted from the FLASH paper. To assess scalability, we benchmark CPU baselines under three hardware regimes: (a) **32-core/64-thread**, (b) **64-core/64-thread** (Physical Cores), and (c) **64-core/128-thread** (SMT enabled). The 64-core experiments use an AMD EPYC 9554 server. For fairness, we tune parameters to strictly meet a high-quality constraint (Recall@100 ≥ 0.98) and report wall-clock construction time. Figure 7 reports the construction time across five datasets under these three hardware regimes.

Standard 32-core/64-thread Performance. CMANNS establishes a commanding lead, outperforming FLASH by **1.59×–2.20×** and FAISS by **1.78×–12.96×**. On **SIFT1M**, CMANNS completes construction in **23.25s**, approximately 1.7× faster than FLASH (39.86s) and 1.9× faster than FAISS (44.71s). The gap widens on workloads with high dimension: on **BIGCODE** and **DATACOMP**, CMANNS (601s and 691s) achieves speedups of **2.2×** and **2.15×** over FLASH, and **12.69×** and **13.0×** over FAISS. These results demonstrate that while FLASH enhances CPU SIMD usage, it remains bound by CPU throughput limitations. Furthermore, FLASH's CPU-centric design scales poorly to GPU architectures, where irregular access patterns induce warp divergence and uncoalesced memory access. In contrast, CMANNS's compute-memory disaggregated design fully leverages Tensor Cores and HBM bandwidth.

Scalability on 64-Core/64-Thread (Physical Cores). Scaling to 64 physical cores reveals *diminishing returns* for baselines due to memory-bandwidth saturation. FLASH shows marginal gains: SIFT1M improves from 39.86s to 35.76s (~1.11×), BIGCODE from 1323s to 1123s (~1.18×), and DATACOMP from 1491s to 1356s (~1.10×). FAISS exhibits similar trends: SIFT1M from 44.71s to 38.8s (~1.15×), BIGCODE from 7635s to 6684s (~1.14×). Despite these upgrades, CMANNS maintains its lead. On **SIFT1M**, CMANNS (23.25s) remains ~1.53× faster than 64-core FLASH. On **BIGCODE** and **DATACOMP**, CMANNS outperforms FLASH by **1.87×** and **1.96×**, and FAISS by **11.1×** and **11.5×**.

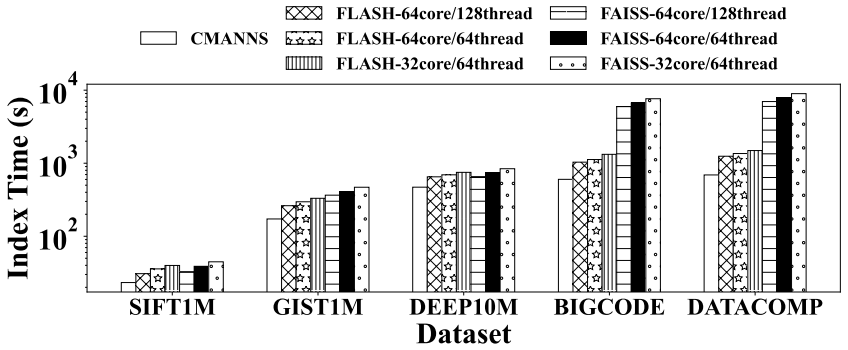


Fig. 7. Comparison with CPU baselines (FAISS and FLASH) under different CPU configurations.

Table 5. Index construction time comparison with CAGRA.

Index	CAGRA Index		
Dataset	CMANNS	NVIDIA CuVS CAGRA	Speedup
SIFT1M	8.57s	11.83s	**1.38x**
GIST1M	12.52s	18.04s	**1.44x**
DEEP10M	73.52s	111.75s	**1.52x**

Overall, CMANNS outperforms FLASH by **1.47×–1.96×** and FAISS by **1.56×–11.47×**, confirming the efficacy of our CM-disaggregated architecture.

Peak Performance at 64-core/128-thread (SMT). Enabling SMT pushes CPU baselines to their throughput limits. FLASH achieves optimal times (30.77s on SIFT1M, 1034s on BIGCODE, 1246s on DATACOMP), as does FAISS (32.72s on SIFT1M, 5950s on BIGCODE, 6987s on DATACOMP). Despite this, CMANNS remains substantially faster, delivering speedups of **1.32×–1.80×** over FLASH and **1.39×–10.11×** over FAISS. Notably, CMANNS extends its lead on high-dimensional workloads where FLASH's quantization is theoretically most effective: **1.72×** on BIGCODE and **1.80×** on DATACOMP against 128-thread FLASH. These results confirm that even with maximal parallelism and AVX-512 acceleration, CPU methods cannot match CMANNS's effective utilization of HBM bandwidth and Tensor Core parallelism.

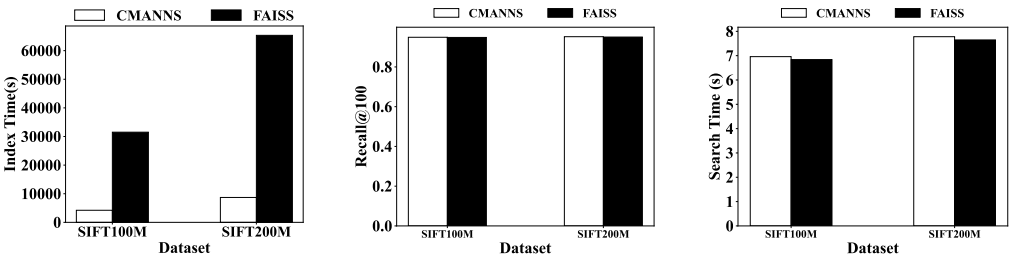
7.6 Comparison with SOTA GPU Baseline

We compare against the GPU SOTA, **CAGRA** [38]. CAGRA alters graph topology (fixed degrees) and uses rank-based (2-hop) pruning. We port CAGRA's construction logic to CMANNS and benchmark against the official NVIDIA CuVS implementation on an A100 GPU. For fairness, we tune it to meet Recall@100 ≥ 0.98.

As shown in Table 5, CMANNS outperforms CuVS CAGRA with **1.38×–1.52×** speedups. On SIFT1M, CMANNS completes in **8.57s** (**1.38×** over CAGRA). On GIST1M and DEEP10M, CMANNS (12.53s and 73.52s) achieves **1.44×** and **1.52×** speedups over CAGRA (18.04s and 111.75s). These results indicate our optimizations (e.g., hot-set-aware locality and Tensor Core utilization) provide substantial acceleration independent of the specific graph.

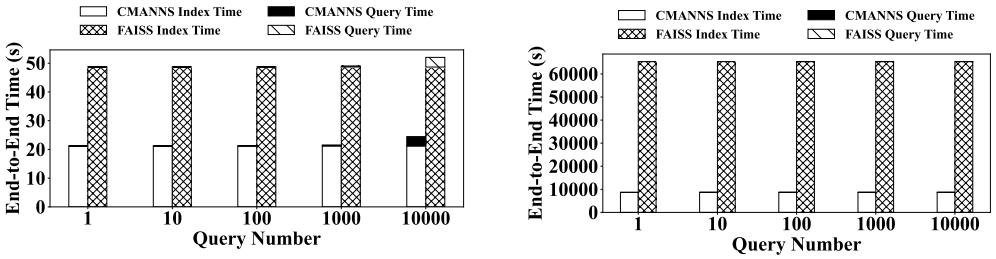
7.7 Out-of-GPU-Memory Evaluation

To assess scalability beyond device memory, we evaluate CMANNS on large dataset with **100M** and **200M** vectors (up to **95 GB**), as shown in Figure 8. The goal is to validate that our out-of-core, double-buffered pipeline accelerates construction while preserving index quality including recall and latency.



(a) Index time. (b) RECALL@100. (c) Search time.

Fig. 8. Out-of-GPU-memory evaluation on SIFT100M and SIFT200M including index time, recall, and search time.



(a) on SIFT1M. (b) on SIFT200M.

Fig. 9. End-to-end evaluation including index constructing time and vector searching time.

Build Time Analysis. On **DEEP10M**, CMANNS reduces NSG build time from **815 s** (CPU FAISS baseline) to **106 s**, a **7.71**× speedup. This trend persists at larger scales: on **100M** and **200M**–vector datasets, CMANNS completes in *1.17 h* and *2.42 h*, versus *8.75 h* and *18.15 h* for FAISS on the same hardware at matched recall. The gains arise from two effects: (i) reformulating distance evaluation as high–intensity GEMMs that exploit Tensor Cores; and (ii) overlapping data movement with computation so that per-shard time approaches max{transfer, compute} rather than their sum. In practice, shard sizes chosen by our cost model keep the device in a steady state while reserving headroom for staging buffers and compact adjacency emission.

Search Quality. Crucially, accelerated construction must not degrade search quality. As shown in Figure 8b and Figure 8c, we randomly sample 5,000 queries from the provided query sets on large-scale datasets and evaluate *Recall@100* and per-query latency for NSG indexes built by CMANNS and FAISS. In all cases, CMANNS matches FAISS within $\leq 2\%$ on both metrics, indicating that GPU-based construction preserves index quality. This validates our design: if partitioning degraded structure, search quality would drop. Matching the in-memory CPU baseline confirms that *partition-then-stitch* preserves graph faithfulness and navigability, keeping topology intact despite the changed construction order.

As a result, CMANNS scales out-of-core by streaming host-resident vectors through a double-buffered GPU pipeline and returning only compact CSR slices. This design sustains high utilization at 100M–200M scale, cutting wall-clock build time from hours to near real-time windows (minutes to low hours) while maintaining recall–latency characteristics comparable to CPU-built graphs.

7.8 End-to-End Performance: Build & Search

In this section, we report the end-to-end time, which includes both *index construction* and *query serving* on two distinct regimes: the in-memory SIFT1M and the large-scale, out-of-core SIFT200M (95 GB raw vectors). For a workload of Q queries,

$$T_{\text{e2e}}(Q) = \underbrace{T_{\text{build}}}_{\text{index time (IT)}} + \underbrace{Q \cdot t_{\text{query}}}_{\text{search time (ST)}} .$$

We compare CMANNS to FAISS at matched recall and identical search budgets (entry, beam/ef, stopping). We vary the query volume $Q \in \{1, 10, 10^2, 1K, 10K\}$ to span interactive to batch regimes and report end-to-end *speedup* $S(Q) = \frac{T_{\text{e2e}}^{\text{FAISS}}(Q)}{T_{\text{e2e}}^{CMANNS}(Q)}$.

The results are shown in Figure 9. They indicate that for NSG, index construction dominates the end-to-end time; consequently, CMANNS's faster builds translate directly into higher $S(Q)$ for small and moderate values of Q.

Performance on SIFT1M. As shown in Figure 9a, on the in-memory SIFT1M dataset, the results indicate that CMANNS can achieve speedups of **2.30**×, **2.30**×, **2.29**×, **2.27**×, and **2.13**× for Q=1, 10, 100, 10^3, and 10^4, respectively. Query costs are very similar between systems (we keep the standard NSG search); e.g., the *single-query* time is ≈ 0.0004 s for both, and for *10,000* queries, the aggregate search time is **3.3554 s** (FAISS) vs. **3.2552 s** (CMANNS), a modest ∼ 3% advantage. Consequently, as Q increases, the fixed IT advantage is amortized by growing ST and $S(Q)$ gradually declines (from 2.30× to 2.13× in our runs), as predicted by the model above.

Performance on SIFT200M. As shown in Figure 9b, on out-of-core SIFT200M, index construction dominates end-to-end time (over 99%), so build-time optimizations translate directly to end-to-end gains. Construction time is 18.15 h for FAISS versus 2.42 h for CMANNS, yielding a **7.5**× speedup. The query term ($Q \cdot t_{\text{query}}$) remains negligible even at high Q, so end-to-end speedup stays close to build-time speedup, confirms that our out-of-core pipeline remains effective under large-scale transfers and **CM-disaggregated** construction sustains its advantage at scale.

Because CMANNS preserves the index structure and uses the canonical query algorithm, search efficiency remains on par with FAISS, while build time is substantially lower. The net effect is a pronounced end-to-end gain when refresh cycles are frequent or the query horizon per build is modest—precisely the regime faced by dynamic vector databases. As workloads are heavily query-bound (very large Q), end-to-end speedup approaches the ratio of search throughputs (near parity, with a slight edge to CMANNS), while still retaining the operational benefit of shorter rebuild windows.

7.9 Efficiency of Different Optimizations

We quantify the contribution of each optimization in CMANNS by selectively enabling (i) Tensor Core–accelerated distance computation (**"Comp"** in Figure 10), (ii) hot–set–aware dynamic locality (**"Mem"**), and (iii) the out–of–GPU–memory streaming pipeline (**"I/O"**). The target index is *NSG*; datasets are SIFT1M, GIST1M, and DEEP10M. Graph parameters (degree/pruning) and search settings are held fixed. Unless noted, the baseline is our CM-disaggregated framework, with *all three optimizations disabled*. We report *a speedup* as Speedup = $\frac{T_{\text{baseline}}}{T_{\text{optimized}}}$ in Figure 10.

Efficiency of "Comp" Optimization. Enabling **Comp** provides the foundational acceleration by reformulating distance evaluation as high–intensity GEMMs on Tensor Cores. Cumulative speedups with `CM+Comp` reach **1.99**× (SIFT1M), **5.93**× (GIST1M), and **2.16**× (DEEP10M), reflecting the dominant role of computation at higher dimensionality (GIST, 960D).

Efficiency of "Mem" Optimization. Adding **Mem** (stage-local shared-memory caches for visited sets, candidate scratchpads, etc.) converts scattered global traffic into warp-cooperative on-chip

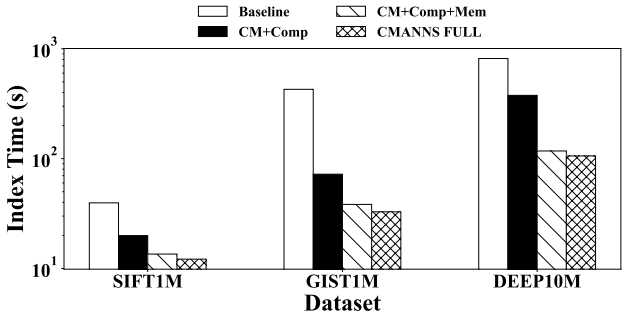


Fig. 10. **Ablation of optimization components.** "Comp" refers Tensor Core–accelerated distance computation; "Mem" refers hot-set–aware on-chip locality; "I/O" refers out-of-GPU-memory streaming pipeline.

Table 6. Microbenchmarks during NSG construction: peak memory usage and L1/L2 cache hit rates (CMANNS vs. FAISS).

Metric	Memory Usage(GB)		Cache Hit Rate(%)			
Dataset	**CMANNS**	**FAISS**	**CMANNS**		**Naïve GPU**	
			L1	**L2**	**L1**	**L2**
SIFT1M	1.88GB	1.79GB	30.6	67.04	20.7	8.33
GIST1M	5.98GB	5.91GB	31.89	48.28	17.87	6.56
DEEP10M	17.92GB	16.56GB	39.13	58.08	23.52	10.47

operations, increasing effective bandwidth and reducing long-latency stalls. Cumulative speedups rise to **2.93**× (SIFT1M), **11.16**× (GIST1M), and **6.94**× (DEEP10M), underscoring that memory-regularization is critical for complex (GIST) and larger (DEEP) shards.

Efficiency of "I/O" Optimization. Finally, enabling **I/O** overlap copies with kernels via double buffering and concurrent streams, pushing cumulative speedups to **3.26**× (SIFT1M), **13.05**× (GIST1M), and **7.71**× (DEEP10M). Even when a shard fits entirely in device memory, we micro-batch edge-selection tasks and stage compact adjacencies, allowing I/O and computation overlap to trim tail latency and keep the device near steady-state saturation (§5.3).

Performance gains are cumulative and complementary: **Comp** raises arithmetic intensity; **Mem** regularizes irregular phases to realize bandwidth; **I/O** hides transfer costs. Together, they yield up to **13.05**× faster builds without altering search behavior.

7.10 Microbenchmark

In this section, we use microbenchmarks to compare *system-level* characteristics of CMANNS against FAISS during NSG construction, focusing on (i) peak construction memory and (ii) cache efficiency (L1/L2 hit rates) to quantify how our design reduces cache misses and improves performance. Peak memory is sampled at steady state; cache metrics are collected using a GPU profiler.

Peak Construction Memory. Table 6 reports peak memory usage. CMANNS incurs only a *marginal* overhead relative to the CPU-based FAISS builder, e.g., **SIFT1M:** 1.88 GB vs. 1.79 GB (+ ∼5.0%); **GIST1M:** 5.98 GB vs. 5.91 GB (+ ∼1.2%); **DEEP10M:** 17.92 GB vs. 16.56 GB (+ ∼8.2%). This small increase stems from GPU-specific preallocation (staging buffers, scratch space, and persistent queues) required to orchestrate massively parallel kernels and double-buffered streaming.

Cache Efficiency and Locality. The most pronounced gains appear in cache behavior. A naïve GPU port shows poor locality (e.g., on **SIFT1M**: L1 hit 20.7%, L2 hit 8.33%), leading to frequent long-scoreboard stalls. With CMANNS 's hot-set–aware shared-memory staging and warp-cooperative gathers/scatters, the same workload attains an L1 hit rate of 30.6% (+9.9 percentage points over the naïve GPU baseline) and an L2 hit rate of 67.04% (+58.7 percentage points over the naïve GPU

baseline), a trend consistent across datasets. By concentrating the recurrent working set (visited sets, candidate scratchpads, reverse-edge staging) in on-chip storage while enforcing coalesced global accesses, CMANNS substantially reduces memory latency and raises effective bandwidth.

In conclusion, CMANNS 's modest increase in peak memory is an intentional, low-cost trade-off for markedly better cache residency and fewer stalls. By separating compute- and memory-dominated phases and keeping hot data on-chip, CMANNS transforms irregular graph construction into bandwidth- and cache-efficient kernels, yielding the end-to-end speedups reported in §7.2.

8 Related Work

Non-graph ANNS Index. Beyond proximity graphs which is discussed in Section §2, a large body of work accelerates ANNS through *quantization*, *space partitioning*, and *learned* encoders. IVF [43, 44, 59] and Product Quantization (PQ) [14, 17, 47] compress vectors and restrict search to a few inverted lists, often with OPQ rotations to reduce distortion, yielding strong memory efficiency and high throughput on CPUs/GPUs (e.g., FAISS) but typically trailing top graph indexes at high recall on challenging high-dimensional data [15, 23, 26]. Tree-based methods [19, 41, 62] and classic LSH variants work [16, 21, 61] well at low to moderate dimensions or under aggressive approximation, but they degrade in the high-dimensional regime [3, 7, 50]. Learning-based indexing (e.g., spectral/supervised hashing [32], learned quantizers [56]) adapts codes to data and improves upon data-independent hashing; yet, it generally underperforms compared to strong graph indexes on large-scale, high-recall targets [26, 52].

GPU-based ANNS Systems and Construction. GPU implementations of IVF/IVF-PQ (FAISS) [9] leverage optimized kernels to achieve high throughput but inherit the recall limits of quantization [26]. GPUs-based graph efforts initially focused on *search*: SONG ports NSW/NSG/DPG to GPUs with warp-synchronous expansions [60]; GGNN restructures traversal for SIMT efficiency [18]; GANNS adds GPU-parallel proximity-graph construction [55]. CAGRA takes a different tack: it *changes the index structure* to a flat, fixed-degree, GPU-friendly graph, seeded by NN-descent and refined to maximize build/search throughput on accelerators [38]. Hybrid CPU–GPU serving, like BANG, keeps the graph in host memory while streaming compressed vectors to the device for distance evaluation [28].

9 Conclusion

We tackled the long-standing bottleneck of graph index *construction* for high-quality ANN graphs by redesigning the builder for GPUs. CMANNS applies compute–memory disaggregation, tensor-core–accelerated distance evaluation, hot-set on-chip locality, and an out-of-core double-buffered pipeline to turn irregular, CPU-bound workflows into high-throughput GPU kernels. Across standard benchmarks (up to 95 GB), CMANNS cuts build time by as much as *13.05*× at matched recall, preserves query latency/accuracy, and keeps peak construction memory close to CPU baselines.

Acknowledgments

We sincerely thank the reviewers for their insightful comments. This work is funded in part by the National Natural Science Foundation of China (NO. 62502193), the Key Program of Natural Science Foundation of Jiangsu under grant (NO. BK20251190, BK20243053), the fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China (No. JYB2025XDXM901), Nanjing "U35" Talent Cultivation Program (No. U (2024) 001), National Key Research and Development Plan of China (2023YFB4502305), Collaborative Innovation Center of Novel Software Technology and Industrialization, Nanjing University. Rong Gu is the corresponding author of this paper.

References

[1] Ilias Azizi, Karima Echihabi, and Themis Palpanas. 2023. Elpis: Graph-based similarity search for scalable data science. *Proceedings of the VLDB Endowment* 16, 6 (2023), 1548–1559.

[2] Ilias Azizi, Karima Echihabi, and Themis Palpanas. 2025. Graph-based vector search: An experimental evaluation of the state-of-the-art. *Proceedings of the ACM on Management of Data* 3, 1 (2025), 1–31.

[3] Jon Louis Bentley. 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM* 18, 9 (1975), 509–517.

[4] Yuzheng Cai, Jiayang Shi, Yizhuo Chen, and Weiguo Zheng. 2024. Navigating labels and vectors: A unified approach to filtered approximate nearest neighbor search. *Proceedings of the ACM on Management of Data* 2, 6 (2024), 1–27.

[5] Jack Choquette, Wishwesh Gandhi, Olivier Giroux, Nick Stam, and Ronny Krashinsky. 2021. Nvidia a100 tensor core gpu: Performance and innovation. *IEEE Micro* 41, 2 (2021), 29–35.

[6] Paul Covington, Jay Adams, and Emre Sargin. 2016. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM conference on recommender systems*. 191–198.

[7] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. 2004. Locality-Sensitive Hashing Scheme Based on p-Stable Distributions. In *SCG*. 253–262.

[8] Wei Dong, Charikar Moses, and Kai Li. 2011. Efficient k-nearest neighbor graph construction for generic similarity measures. In *Proceedings of the 20th international conference on World wide web*. 577–586.

[9] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. 2024. The Faiss library. (2024). arXiv:2401.08281 [cs.LG]

[10] Boyuan Feng, Yuke Wang, Guoyang Chen, Weifeng Zhang, Yuan Xie, and Yufei Ding. 2021. EGEMM-TC: accelerating scientific computing on tensor cores with extended precision. In *Proceedings of the 26th ACM SIGPLAN symposium on principles and practice of parallel programming*. 278–291.

[11] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2019. Fast Approximate Nearest Neighbor Search With The Navigating Spreading-out Graphs. *PVLDB* 12, 5 (2019), 461 – 474. doi:10.14778/3303753.3303754

[12] Wilson WL Fung and Tor M Aamodt. 2011. Thread block compaction for efficient SIMT control flow. In *2011 IEEE 17th international symposium on high performance computer architecture*. IEEE, 25–36.

[13] Yukang Gan, Yixiao Ge, Chang Zhou, Shupeng Su, Zhouchuan Xu, Xuyuan Xu, Quanchao Hui, Xiang Chen, Yexin Wang, and Ying Shan. 2023. Binary Embedding-based Retrieval at Tencent. arXiv:2302.08714 [cs.IR] https://arxiv.org/abs/2302.08714

[14] Jianyang Gao and Cheng Long. 2024. RaBitQ: quantizing high-dimensional vectors with a theoretical error bound for approximate nearest neighbor search. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–27.

[15] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. 2013. Optimized Product Quantization for Approximate Nearest Neighbor Search. In *CVPR*. 2946–2953.

[16] Aristides Gionis, Piotr Indyk, Rajeev Motwani, et al. 1999. Similarity search in high dimensions via hashing. In *Vldb*, Vol. 99. 518–529.

[17] Yutong Gou, Jianyang Gao, Yuexuan Xu, and Cheng Long. 2025. SymphonyQG: Towards Symphonious Integration of Quantization and Graph for Approximate Nearest Neighbor Search. *Proceedings of the ACM on Management of Data* 3, 1 (2025), 1–26.

[18] Fabian Groh, Lukas Ruppert, Patrick Wieschollek, and Hendrik PA Lensch. 2022. Ggnn: Graph-based gpu nearest neighbor search. *IEEE Transactions on Big Data* 9, 1 (2022), 267–279.

[19] Antonin Guttman. 1984. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*. 47–57.

[20] 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.

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

[22] 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. *Advances in neural information processing Systems* 32 (2019).

[23] Herve Jegou, Matthijs Douze, and Cordelia Schmid. 2010. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence* 33, 1 (2010), 117–128.

[24] Wenqi Jiang, Hang Hu, Torsten Hoefler, and Gustavo Alonso. 2025. Fast graph vector search via hardware acceleration and delayed-synchronization traversal. *Proceedings of the VLDB Endowment* 18, 11 (2025), 3797–3811.

[25] Wenqi Jiang, Marco Zeller, Roger Waleffe, Torsten Hoefler, and Gustavo Alonso. 2024. Chameleon: A Heterogeneous and Disaggregated Accelerator System for Retrieval-Augmented Language Models. *Proc. VLDB Endow.* 18, 1 (Sept. 2024), 42–52. doi:10.14778/3696435.3696439

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

[27] Hongshin Jun, Jinhee Cho, Kangseol Lee, Ho-Young Son, Kwiwook Kim, Hanho Jin, and Keith Kim. 2017. Hbm (high bandwidth memory) dram technology and architecture. In *2017 IEEE International Memory Workshop (IMW)*. IEEE, 1–4.

[28] V. Karthik, Saim Khan, Somesh Singh, Harsha Vardhan Simhadri, and Jyothi Vedurada. 2024. BANG: Billion-Scale Approximate Nearest Neighbour Search using a Single GPU. *arXiv:2401.11324* (2024).

[29] Sukjin Kim, Seongyeon Park, Si Ung Noh, Junguk Hong, Taehee Kwon, Hunseong Lim, and Jinho Lee. 2025. {PathWeaver}: A {High-Throughput}{Multi-GPU} System for {Graph-Based} Approximate Nearest Neighbor Search. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*. 1501–1517.

[30] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems* 33 (2020), 9459–9474.

[31] Jimmy Lin. 2024. Operational advice for dense and sparse retrievers: HNSW, flat, or inverted indexes? *arXiv preprint arXiv:2409.06464* (2024).

[32] Yingfan Liu, Xiaotian Qiao, Zhaoqing Liu, Xiaofang Xia, Yinlong Zhang, and Jiangtao Cui. 2025. Deep multi-negative supervised hashing for large-scale image retrieval. *Expert Systems with Applications* 264 (2025), 125795.

[33] 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.

[34] Yu A. Malkov and D. A. Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 42, 4 (2020), 824–836. doi:10.1109/TPAMI.2018.2889473

[35] Magdalen Dobson Manohar, Zheqi Shen, Guy Blelloch, Laxman Dhulipala, Yan Gu, Harsha Vardhan Simhadri, and Yihan Sun. 2024. Parlayann: Scalable and deterministic parallel graph-based approximate nearest neighbor search algorithms. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 270–285.

[36] NVIDIA cuVS. 2024. cuVS: A library for vector search and clustering on the GPU. https://github.com/rapidsai/cuvs.

[37] Shumpei Okura, Yukihiro Tagami, Shingo Ono, and Akira Tajima. 2017. Embedding-based news recommendation for millions of users. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*. 1933–1942.

[38] 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)*. IEEE, 4236–4247.

[39] Zhen Peng, Minjia Zhang, Kai Li, Ruoming Jin, and Bin Ren. 2023. iqan: Fast and accurate vector search with efficient intra-query parallelism on multi-core architectures. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 313–328.

[40] Joobo Shim, Jaewon Oh, Hongchan Roh, Jaeyoung Do, and Sang-Won Lee. 2025. Turbocharging Vector Databases using Modern SSDs. *Proceedings of the VLDB Endowment* 18, 11 (2025), 4710–4722.

[41] Chanop Silpa-Anan and Richard Hartley. 2008. Optimised KD-trees for fast image descriptor matching. In *2008 IEEE conference on computer vision and pattern recognition*. IEEE, 1–8.

[42] Yukihiro Tagami. 2017. Annexml: Approximate nearest neighbor search for extreme multi-label classification. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*. 455–464.

[43] 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)*. 1135–1150.

[44] 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)*. 171–185.

[45] Sairaj Voruganti and M Tamer Özsu. 2025. MIRAGE-ANNS: Mixed Approach Graph-based Indexing for Approximate Nearest Neighbor Search. *Proceedings of the ACM on Management of Data* 3, 3 (2025), 1–27.

[46] Mengzhao Wang, Haotian Wu, Xiangyu Ke, Yunjun Gao, Yifan Zhu, and Wenchao Zhou. 2025. Accelerating Graph Indexing for ANNS on Modern CPUs. *Proceedings of the ACM on Management of Data* 3, 3 (2025), 1–29.

[47] Runhui Wang and Dong Deng. 2020. DeltaPQ: lossless product quantization code compression for high dimensional similarity search. *Proceedings of the VLDB Endowment* 13, 13 (2020), 3603–3616.

[48] Yuke Wang, Boyuan Feng, and Yufei Ding. 2022. QGTC: accelerating quantized graph neural networks via GPU tensor core. In *Proceedings of the 27th ACM SIGPLAN symposium on principles and practice of parallel programming*. 107–119.

[49] Yuke Wang, Boyuan Feng, Zheng Wang, Guyue Huang, and Yufei Ding. 2023. {TC-GNN}: Bridging sparse {GNN} computation and dense tensor cores on {GPUs}. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*.

149–164.

[50] Roger Weber, Hans-Jörg Schek, and Stephen Blott. 1998. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB*, Vol. 98. 194–205.

[51] Chuangxian Wei, Bin Wu, Sheng Wang, Renjie Lou, Chaoqun Zhan, Feifei Li, and Yuanzhe Cai. 2020. Analyticdb-v: A hybrid analytical engine towards query fusion for structured and unstructured data. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3152–3165.

[52] Yair Weiss, Antonio Torralba, and Rob Fergus. 2008. Spectral Hashing. In *NeurIPS*. 1753–1760.

[53] Yuxiang Yang, Shiwen Chen, Yangshen Deng, and Bo Tang. 2025. ParaGraph: Accelerating Graph Indexing through GPU-CPU Parallel Processing for Efficient Cross-modal ANNS. In *Proceedings of the 21st International Workshop on Data Management on New Hardware*. 1–10.

[54] Ziqi Yin, Jianyang Gao, Pasquale Balsebre, Gao Cong, and Cheng Long. 2025. DEG: Efficient Hybrid Vector Search Using the Dynamic Edge Navigation Graph. *Proceedings of the ACM on Management of Data* 3, 1 (2025), 1–28.

[55] Yuanhang Yu, Dong Wen, Ying Zhang, Lu Qin, Wenjie Zhang, and Xuemin Lin. 2022. GPU-accelerated proximity graph approximate nearest Neighbor search and construction. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 552–564.

[56] Dongqing Zhang, Jiaolong Yang, Dongqiangzi Ye, and Gang Hua. 2018. Lq-nets: Learned quantization for highly accurate and compact deep neural networks. In *Proceedings of the European conference on computer vision (ECCV)*. 365–382.

[57] Jianjin Zhang, Zheng Liu, Weihao Han, Shitao Xiao, Ruicheng Zheng, Yingxia Shao, Hao Sun, Hanqing Zhu, Premkumar Srinivasan, Weiwei Deng, et al. 2022. Uni-retriever: Towards learning the unified embedding based retriever in bing sponsored search. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 4493–4501.

[58] 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.

[59] 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)*. 23–40.

[60] 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)*. IEEE, 1033–1044.

[61] Xi Zhao, Yao Tian, Kai Huang, Bolong Zheng, and Xiaofang Zhou. 2023. Towards efficient index construction and approximate nearest neighbor search in high-dimensional spaces. *Proceedings of the VLDB Endowment* 16, 8 (2023), 1979–1991.

[62] Yifan Zhu, Ruiyao Ma, Baihua Zheng, Xiangyu Ke, Lu Chen, and Yunjun Gao. 2024. GTS: GPU-based tree index for fast similarity search. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–27.

Received October 2025; revised January 2026; accepted February 2026