GPU-Native Approximate Nearest Neighbor Search with IVF-RaBitQ: Fast Index Build and Search

Jifan Shi
Nanyang Technological University
Singapore
jifan002@e.ntu.edu.sg

Jianyang Gao
Nanyang Technological University
Singapore
gaoj0017@e.ntu.edu.sg

James Xia
NVIDIA
USA
jamxia@nvidia.com

Tamás Béla Fehér
NVIDIA
Germany
tfeher@nvidia.com

Cheng Long
Nanyang Technological University
Singapore
c.long@ntu.edu.sg

ABSTRACT

Approximate nearest neighbor search (ANNS) on GPUs is gaining increasing popularity for modern retrieval and recommendation workloads that operate over massive high-dimensional vectors. Graph-based indexes deliver high recall and throughput but incur heavy build-time and storage costs. In contrast, cluster-based methods build and scale efficiently yet often need many probes for high recall, straining memory bandwidth and compute. Aiming to simultaneously achieve fast index build, high-throughput search, high recall, and low storage requirement for GPUs, we present IVF-RaBitQ (GPU), a GPU-native ANNS solution that integrates the cluster-based method IVF with RaBitQ quantization into an efficient GPU index build/search pipeline. Specifically, for index build, we develop a scalable GPU-native RaBitQ quantization method that enables fast and accurate low-bit encoding at scale. For search, we develop GPU-native distance computation schemes for RaBitQ codes and a fused search kernel to achieve high throughput with high recall. With IVF-RaBitQ implemented and integrated into the NVIDIA cuVS Library, experiments on cuVS Bench across multiple datasets show that IVF-RaBitQ offers a strong performance frontier in recall, throughput, index build time, and storage footprint. For Recall≈0.95, IVF-RaBitQ achieves 3.0× higher QPS than the state-of-the-art graph-based method CAGRA, while also constructing indices 14.7× faster on average. Compared to the cluster-based method IVF-PQ, IVF-RaBitQ delivers on average over 4.5× higher throughput while avoiding accessing the raw vectors for reranking.

PVLDB Reference Format:
Jifan Shi, Jianyang Gao, James Xia, Tamás Béla Fehér, and Cheng Long. GPU-Native Approximate Nearest Neighbor Search with IVF-RaBitQ: Fast Index Build and Search. PVLDB, 19(11): 3454 - 3467, 2026.
doi:10.14778/3836663.3836701

Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/Stardust-SJF/cuvs_rabitq.

Proceedings of the VLDB Endowment, Vol. 19, No. 11 ISSN 2150-8097.
doi:10.14778/3836663.3836701

1 INTRODUCTION

Approximate Nearest Neighbor Search (ANNS) on high-dimensional vector data has become a fundamental component in modern data management and machine learning. It supports key applications such as semantic search [6], Retrieval-Augmented Generation (RAG) [14], and recommender systems [10, 15, 33]. With the widespread adoption of embedding-based models [13, 21, 43, 44], production workloads now often involve massive high-dimensional vectors [49, 54]. As a result, designing scalable and efficient ANNS solutions has become a long-standing and active research topic.

A variety of ANNS solutions have been proposed, among which graph-based methods and cluster-based methods are most commonly used in industrial systems. Graph-based methods usually construct proximity graphs over data points and perform greedy traversal to locate nearest neighbors. Representative algorithms, such as HNSW [38], NSG [17], and Vamana graph [25], are known for their competitive recall-performance trade-offs. Cluster-based methods, most notably inverted-file (IVF) indices, partition the vector space into coarse clusters and restrict search to a subset of clusters. These methods offer simple data organization and efficient index construction, which enable easy integration with existing database systems. Furthermore, both types of methods are often enhanced by leveraging vector quantization for reducing the costs of storing and accessing raw vectors and/or the costs of computing distances among raw vectors [1, 18, 22, 25, 30]. Most of these solutions were originally developed and optimized for CPU architectures.

As data scale and query volume continue to grow, many real-world ANNS deployments choose to use GPUs as their accelerators due to GPUs' massive parallelism and high memory bandwidth. For instance, cuVS, which is a GPU-based ANNS library from NVIDIA, has been widely used to handle batched queries on large-scale data [41]. Existing GPU-based ANNS methods largely follow the same two paradigms as their CPU counterparts, namely graph-based approaches [23, 28, 31, 42, 51, 52, 56, 57, 60] and cluster-based approaches [8, 26, 48, 53, 55, 59]. Graph-based GPU methods redesign proximity graph building and traversal to better match GPU execution, leveraging warp-level parallelism, shared-memory caching, and fine-grained scheduling to mitigate control-flow divergence and irregular memory access. Representative systems such as SONG [60], cuHNSW [28], GGNN [23], and CAGRA [42] demonstrate that graph-based ANNS can achieve strong recall and competitive query performance on GPUs.

Compared with graph-based indices, cluster-based GPU methods offer a different set of trade-offs. A key merit is that cluster-based methods expose *regular computation patterns* that align well with GPU primitives: coarse cluster selection can be implemented using dense linear algebra (e.g., GEMM for computing query–centroid distances), and the subsequent probing within selected clusters can be parallelized as large, contiguous streams of codes or vectors. This makes it natural to exploit GPU batching and achieve high throughput. Moreover, cluster-based indices typically have *simpler structures* than proximity graphs, with more predictable memory access, which eases GPU memory management and improves memory coalescing.

Motivated by the above considerations, this paper focuses on *cluster-based* ANNS methods on GPUs. Within this family, IVF-Flat and IVF-PQ are widely used baselines [26, 35, 46, 48, 59]. IVF-Flat stores full-precision vectors and performs brute-force scanning within selected clusters. Its accuracy is strong, but the scan stage is often dominated by memory bandwidth due to high-dimensional floating-point loads. IVF-PQ reduces memory traffic by compressing vectors using product quantization (PQ) [30]. However, it requires learning codebooks during index construction, introducing additional overhead on building complexity. Also, the compressed representation only provides approximate distance estimates without strict accuracy guarantees. As a result, achieving high recall often requires reranking with the raw vectors (e.g., raw vectors are stored in main memory and are accessed during search), which increases the memory footprint and complicates the overall pipeline.

Recently, a new vector quantization method called RaBitQ was proposed [19, 20], which improves upon PQ in several important aspects. First, RaBitQ does not require codebook training. Second, RaBitQ enables more accurate distance estimation with tight, theoretically grounded error bounds. RaBitQ has been combined with IVF for CPU-based ANN search and has shown consistent improvements over PQ [19, 20]. Nevertheless, bringing RaBitQ to GPUs and integrating it with IVF is non-trivial and introduces several algorithmic and system challenges. Specifically, the quantization procedure of RaBitQ involves searching over rescaling factors and evaluating candidates, which can be expensive and control-flow heavy. When implemented on CPU, RaBitQ's distance evaluation step benefits from wide SIMD vectorization and specialized bit-level routines (e.g., AVX-512-style packed/shuffle operations and fast bit-scan / popcount primitives) to accelerate asymmetric inner products and reduce per-dimension overhead [18]. On GPUs with SIMT (Single Instruction, Multiple Thread) execution models, the control-flow heavy quantization becomes warp-divergent, and the CPU SIMD-optimized distance kernels have no direct equivalent. A naive port either underutilizes the GPU or incurs excessive memory traffic and kernel overhead. Realizing RaBitQ effectively on GPUs therefore requires GPU-native design, including scalable quantization strategies, GPU-oriented distance-evaluation primitives, and careful integration with the IVF probe-and-scan pipeline.

In this paper, we present **IVF-RaBitQ (GPU)**, hereafter **IVF-RaBitQ**, the first GPU-native ANNS solution that tightly integrates IVF coarse partitioning with RaBitQ quantization, for achieving fast index build, high-throughput search, high recall, and compact storage simultaneously. We summarize our technical contributions to the development of IVF-RaBitQ as follows. First, during index build, apart from building the IVF index, we reformulate RaBitQ quantization for GPU execution and quantize data vectors into compact RaBitQ codes. Specifically, we design a cluster-at-a-time, block-per-vector quantization pipeline. Within the pipeline, we develop a two-phase parallel grid search algorithm for exploring the rescaling factors in RaBitQ quantization. This design is motivated by the observation that the objective function over rescaling factors is approximately unimodal. Specifically, it applies a coarse-to-fine narrowing strategy which is inspired by ternary search. These techniques enable scalable and high-throughput RaBitQ quantization for massive data vectors on GPUs. Second, during search, we develop GPU-native distance estimation methods based on RaBitQ quantization codes for scanning IVF lists. Specifically, we adopt a two-stage distance estimation strategy for finding nearest neighbors of a query, namely a filtering stage based on the 1-bit RaBitQ code (which consists of the most significant bit of each dimension of the code), followed by a refinement stage using the ex-code (which consists of the remaining bits of each dimension of the code) only for candidates that pass the filter. Since the filtering stage (which checks many data vectors) dominates the overall computation, we develop two GPU-native inner-product computation schemes, namely one based on *lookup tables* and the other based on *bitwise decomposition*, to accelerate this stage. Third, we design a fused cluster-local search kernel for finding nearest neighbors locally within a cluster, which integrates distance filtering, refinement, and top-K selection into a single GPU kernel, substantially reducing kernel-launch overhead and global-memory traffic. Fourth, to support the index build and search pipelines, we design a GPU-oriented index layout that aligns data organization with access patterns of the fused kernel, enabling high memory coalescing and cache efficiency.

We also integrate IVF-RaBitQ build and search pipelines into NVIDIA **cuVS** library [46] and **cuVS Bench** [40], enabling reproducible benchmarking and direct comparisons with existing GPU baselines under a unified evaluation framework. We conduct extensive experiments using cuVS Bench with diverse datasets, demonstrating IVF-RaBitQ's advantages in the trade-off among recall, throughput, index build time, and storage footprint. For query performance, we evaluate methods mainly under large-batch query settings to fully utilize GPU resources. Compared with the state-of-the-art graph-based GPU method, CAGRA, IVF-RaBitQ achieves comparable or higher throughput (0.8×~8.2× speedup for Recall≈0.95) while reducing index construction time by up to order of magnitude (3.4×~55.3× speedup) with far less storage requirement. Compared with IVF-PQ implemented in NVIDIA cuVS Library, IVF-RaBitQ achieves significantly higher query throughput (1.4×~58.3× speedup for Recall≈0.95) and accuracy under similar index build cost and storage budget. These results show that IVF-RaBitQ is a practical and scalable solution for large-scale GPU-based ANNS problems.

2 PRELIMINARIES

2.1 Approximate NN Search and the IVF Index

In this paper, we target the *batched* setting of approximate nearest neighbor search (ANNS): given a batch of queries $\mathbf{Q} \in \mathbb{R}^{n_q \times D}$, we want to find the K approximate nearest neighbors from a set of N data vectors $\mathbf{O} \subseteq \mathbb{R}^D$ for every query in the batch.

In the IVF index, vectors are first partitioned into clusters using a clustering algorithm (e.g., K-means [36] or its variations [3, 37]). The search process is usually carried out in two stages [30]. First, given a query vector, a few nearest clusters are identified based on the distances between their centroids to the query. Then, these clusters are probed by computing the distances between the vectors in the clusters and the query vector and finding the nearest neighbors. With different settings of the number of clusters to probe, IVF provides different accuracy-efficiency trade-offs.

2.2 The RaBitQ Quantization Method

We first review the process of quantizing a raw vector $\mathbf{o}_\mathrm{r}$ in RaBitQ [19, 20]. First, it normalizes $\mathbf{o}_\mathrm{r}$ to a unit vector $\mathbf{o}$ wrt a center vector $\mathbf{c}$, i.e., we have $\mathbf{o} = \frac{\mathbf{o}_\mathrm{r}-\mathbf{c}}{\|\mathbf{o}_\mathrm{r}-\mathbf{c}\|}$. Then, it focuses on quantizing the unit vector $\mathbf{o}$. Note that the direction of $\mathbf{o}_\mathrm{r}$ is captured by $\mathbf{o}$ and the norm of $\mathbf{o}_\mathrm{r}$ can be pre-computed and stored. Specifically, it constructs a codebook by shifting, normalizing, and randomly rotating vectors whose coordinates are B-bit unsigned integers, where B could be 1, 2, 3, ... Formally, given the number of dimensions of a vector D and the number of bits used to quantize each dimension B, the codebook $\mathcal{G}_\mathrm{r}$ is defined as follows:

$$\mathcal{G} := \left\{ -\frac{2^B-1}{2} + u \;\middle|\; u = 0, 1, 2, 3, \ldots, 2^B-1 \right\}^D \tag{1}$$

$$\mathcal{G}_\mathrm{r} := \left\{ \mathbf{P}\frac{\mathbf{x}}{\|\mathbf{x}\|} \;\middle|\; \mathbf{x} \in \mathcal{G} \right\} \tag{2}$$

where $\mathbf{P}$ is a random orthogonal matrix [27].

To quantize the vector $\mathbf{o}$, it takes $\mathbf{o}$'s nearest vector in the codebook $\mathcal{G}_\mathrm{r}$, which we denote by $\hat{\mathbf{o}}$, as its quantized vector. Let $\hat{\mathbf{x}}$ be the vector in $\mathcal{G}$, which corresponds to $\hat{\mathbf{o}}$ (note that each vector in $\mathcal{G}_\mathrm{r}$ corresponds to a vector in $\mathcal{G}$ (before normalization and rotation)). This process can be formally described as follows.

$$\hat{\mathbf{x}} = \arg\min_{\mathbf{x}\in\mathcal{G}} \left\| \mathbf{P}\frac{\mathbf{x}}{\|\mathbf{x}\|} - \mathbf{o} \right\|^2 = \arg\max_{\mathbf{x}\in\mathcal{G}} \left\langle \frac{\mathbf{x}}{\|\mathbf{x}\|}, \mathbf{o}' \right\rangle \tag{3}$$

where $\mathbf{o}' = \mathbf{P}^{-1}\mathbf{o}$ is the data vector after rotation.

According to Equation 3, the quantization process is to find $\hat{\mathbf{x}}$, which is the vector $\mathbf{x} \in \mathcal{G}$ that maximizes the cosine similairty with the data vector after rotation, i.e., $\mathbf{o}'$. According to the analysis in [19], the vector $\hat{\mathbf{x}}$ must be closest to a *re-scaled* vector $\mathbf{o}'$ for some *rescaling factor* t, i.e., $t \cdot \mathbf{o}'$. Therefore, while there are $2^{B \cdot D}$ vectors in $\mathcal{G}$, the quantization process does not need to enumerate all of them. Instead, it only enumerates those which are closest to $t \cdot \mathbf{o}'$ for some t. Furthermore, while there are infinite rescaling factors, there are only $D \cdot 2^{B-1}$ *critical* ones, at which the vectors that are closest to $t \cdot \mathbf{o}'$ in $\mathcal{G}$ would change [19]. As a result, the quantization process can explore these *critical* rescaling factors only. In summary, the quantization process is to identify and enumerate $D \cdot 2^{B-1}$ critical rescaling factors. For each factor t, it finds the vector $\mathbf{x}$ in $\mathcal{G}$ that is closest to $t \cdot \mathbf{o}'$ and computes its cosine similarity with $\mathbf{o}'$. Finally, it returns the vector $\hat{\mathbf{x}}$, which has the greatest cosine similarity.

Given a raw query vector $\mathbf{q}_\mathrm{r}$, we review the process of estimating the Euclidean distance between $\mathbf{q}_\mathrm{r}$ and a raw data vector $\mathbf{o}_\mathrm{r}$ in RaBitQ as follows. It first rotates $\mathbf{q}_\mathrm{r}$ with the same matrix $\mathbf{P}$ used for rotating the data vectors. We denote the query vector after rotation by $\mathbf{q}'$. It then estimates the distance between $\mathbf{q}_\mathrm{r}$ and $\mathbf{o}_\mathrm{r}$ based on several variables: namely (1) the dot product between the RaBitQ code of $\mathbf{o}$, i.e., $\hat{\mathbf{x}}$, and the rotated query vector, $\mathbf{q}'$ and (2) some factors [18]. For detailed expressions of the computation, we refer readers to the documentation of the RaBitQ Library [50].

3 THE GPU-BASED IVF-RABITQ METHOD

3.1 Overview

IVF-RaBitQ (GPU) is a GPU-native ANNS solution that combines an IVF coarse partition with RaBitQ quantization and implements both *index build* and *search* as GPU-friendly pipelines. The overview of its workflow is described as follows.

Index build. Given a set of N raw data vectors $\mathbf{O} \in \mathbb{R}^{N\times D}$ (D is the vector dimensionality), we first learn n_k IVF centroids and assign each data vector to its nearest centroid, thereby forming n_k inverted lists. Within each cluster, we normalize the vectors with respect to the centroid, rotate them with a random orthogonal matrix $\mathbf{P}$, and quantize them using RaBitQ and obtain their codes. Each code is stored in two parts: (i) the most significant bit of each dimension (which we call the *1-bit RaBitQ code* or simply the *1-bit code*) and (ii) the remaining bits (which we call the *ex-code*). To enable efficient large-scale RaBitQ quantization on GPU, we write GPU kernels to process vectors in a cluster-wise manner, and develop GPU-friendly algorithms for exploring the rescaling factors used in RaBitQ quantization (Section 3.2).

Search. In IVF-RaBitQ, queries are processed in batches (Section 3.3). Given a query batch $\mathbf{Q} \in \mathbb{R}^{n_q\times D}$ and a probe count n_probe, we first apply the same random orthogonal transformation $\mathbf{P}$ used in index build to the queries and select the n_probe closest IVF clusters for each query. The search workload is then organized at the granularity of (*query*, *cluster*) *pairs*, enabling independent and parallel processing within each probed cluster. For each (query, cluster) pair, we perform a cluster-local search that evaluates candidates stored in the corresponding inverted list and finds the top-K candidates (i.e., K nearest neighbors) of the query within the cluster. Specifically, it adopts a two-stage distance estimation strategy: a filtering stage based on the 1-bit RaBitQ code, followed by a refinement stage using the ex-code only for candidates that pass the filter. Since the filtering stage checks many data vectors and dominates the overall computation, we design GPU-native inner-product computation methods to accelerate this stage (Section 3.4.1), and further integrate it with filtering, refinement, and top-K selection into a fused GPU kernel to reduce kernel-launch and memory-access overheads (Section 3.4.2). Finally, the top-K candidates obtained from all probed clusters are merged to produce the global top-K result for each query.

GPU index layout. To efficiently support ANNS on GPU, we design a flattened Compressed Sparse Row (CSR)-like layout [7] for storing the clusters of the IVF index. We use it to store other elements of the index as well, including separate contiguous arrays for 1-bit codes, ex-codes, and auxiliary factors. We further adopt an interleaved layout for 1-bit codes that is tailored to the access pattern of the dominant filter stage, leading to improved memory coalescing on the GPU (Section 3.4.3).

3.2 Index Build on GPU

3.2.1 Clustering, Normalization and Rotation. We run GPU-based Balanced K-means [45] to learn n_k centroids and assign each database vector to its nearest centroid. This produces the centroid matrix $\mathbf{C} \in \mathbb{R}^{n_k \times D}$ and a cluster id for each vector. Within each cluster, we normalize each vector to unit ℓ_2 norm wrt its corresponding centroid. For simplicity, when the context is clear, we refer to the normalized vectors as (data) vectors and denote a data vector by $\mathbf{o}$. We then apply a shared random orthogonal matrix $\mathbf{P}$ to rotate data vectors and centroids, which is implemented using General Matrix Multiplication (GEMM). Alternatively, we implement the roatation as a fast Hadamard transform combined with Kac's random walk on GPU, which we borrow from the RaBitQ Library [50]. Since $\mathbf{P}$ is orthogonal, the rotation preserves inner products and distances between data vectors, centroids, and therefore does not alter exact nearest-neighbor relationships. We denote a data vector after rotation by $\mathbf{o}'$.

3.2.2 GPU-based RaBitQ Quantization for Data Vectors. Given a set of N data vectors $\mathbf{o}$ and a budget of B bits per dimension, we aim to develop a GPU-based solution for quantizing the data vectors using RaBitQ, i.e., computing the RaBitQ codes of the data vectors.

Kernel granularity and thread mapping. We first decide the kernel granularity (i.e., how many vectors are processed per launch), which directly affects the parallelism effectiveness by determining how computation and memory resources are scheduled. Quantizing *one vector at a time* is prohibitively expensive when the dataset contains millions to billions of vectors, because kernel-launch overhead and per-vector scheduling costs accumulate quickly, and the GPU's massive parallelism (hundreds of thousands of concurrent threads) is difficult to fully utilize. At the other extreme, quantizing *all vectors at once* can saturate the GPU, but it imposes a high GPU memory footprint (especially for intermediate buffers) and is inflexible across datasets of different sizes. We therefore adopt a *cluster-at-a-time* design: each kernel launch processes the vectors in one cluster, which gives a natural and bounded unit of work aligned with the IVF structure, and makes the implementation largely data-size independent.

Next, we decide the mapping between GPU execution units and vectors. A straightforward approach is to assign a single thread to each vector, analogous to the parallelization strategy typically employed on CPUs. However, this prevents intra-vector parallelization: the computation required by RaBitQ for a single vector (which involves exploring multiple critical scaling factors) becomes serialized within one thread, limiting throughput. Moreover, adjacent threads in a warp would typically access different vectors located at different regions in global memory, which can easily lead to *uncoalesced* memory accesses and lower effective bandwidth. Additionally, under a cluster-at-a-time execution schedule, the number of concurrently processed vectors may be insufficient to fully utilize the GPU. Thus, we adopt a *block-per-vector* mapping as a robust and flexible design point, which has several advantages: (1) it enables fine-grained parallelism for a vector, e.g., it can explore multiple critical scaling factors in parallel; and (2) the threads in the same warp access contiguous (or near-contiguous) locations of the same vector, yielding *coalesced* global-memory accesses. This improves both compute efficiency and memory throughput, and makes it easier to maintain high occupancy when operating cluster by cluster.

Efficient exploration of rescaling factors. The standard RaBitQ quantization algorithm enumerates $D \cdot 2^{B-1}$ critical rescaling factors in ascending order and evaluates them using incremental updates [19]. To support this efficient enumeration, previous CPU implementation maintains per-vector state, typically using a small priority queue to generate the next rescaling factor without explicitly materializing or sorting all candidates [19]. However, this sequential enumeration relies on fine-grained state updates and involves irregular control flow, which is not friendly to GPU execution model.

To achieve high accuracy across dimensionalities while maintaining high performance, we propose a GPU-friendly search for the rescaling factor. Recall that the quantization process of RaBitQ explores a set of critical rescaling factors. For each factor t, it identifies a vector $\mathbf{x} \in \mathcal{G}$ (which is the closest to $t \cdot \mathbf{o}'$) and computes the dot product $\left\langle \frac{\mathbf{x}}{\|\mathbf{x}\|}, \mathbf{o}' \right\rangle$ (or $\langle \mathbf{x}, \mathbf{o}' \rangle / \|\mathbf{x}\|$). Finally, it returns the vector $\hat{\mathbf{x}}$, which has the greatest dot product. This quantization process can be regarded as one of optimizing an objective function $\langle \mathbf{x}, \mathbf{o}' \rangle / \|\mathbf{x}\|$ of t. We observe this objective function is often *approximately unimodal* in practice (i.e., it tends to increase then decrease with a single peak (possibly with small local fluctuations)). A classical *ternary search* locates the maximum of a unimodal function by repeatedly shrinking the interval; however, it is inherently iterative and not ideal for GPU control flow.

Inspired by ternary search, we adopt a *two-phase grid search* that achieves a similar coarse-to-fine narrowing effect using a fixed number of GPU-parallel rounds. Given a D-dimensional normalized vector $\mathbf{o}'$ (after rotation) and bit-width B, we first determine the search range of factors that could affect quantization result. The algorithm then proceeds in two phases. In the coarse search phase, we uniformly sample N_{coarse} candidate rescaling factors across the search range and evaluate them in parallel using multiple threads. For each candidate t, we find the code $\mathbf{x}_{\text{cur}}$ that is closest to $t \cdot \mathbf{o}'$ and compute the objective $\langle \mathbf{x}_{\text{cur}}, \mathbf{o}' \rangle / \|\mathbf{x}_{\text{cur}}\|$. A parallel reduction then identifies the best candidate t_{center}. In the fine search phase, we narrow the search range to a small neighborhood around t_{center} and repeat the parallel evaluation with N_{fine} samples to refine the solution. The final rescaling factor t_{max} is used to produce the quantization code $\hat{\mathbf{x}}$. On the GPU, for each vector, we assign a thread block to do the sampling and the reduction in parallel. In this way, the algorithm evaluates only $N_{\text{coarse}} + N_{\text{fine}}$ sampled factors per vector, rather than enumerating all $O(D \cdot 2^B)$ critical factors as in the standard RaBitQ quantization algorithm.

3.3 Search Pipeline on GPU

3.3.1 Batch Rotation. Let $\mathbf{Q} \in \mathbb{R}^{n_q \times D}$ be a batch of queries. We rotate the batch using the same shared orthogonal matrix $\mathbf{P}$ as in the index build. This produces a contiguous rotated query batch $\mathbf{Q}'$ that is reused by all subsequent stages in the search pipeline.

3.3.2 Cluster Selection. We select the n_{probe} closest clusters for each query. Let $\mathbf{C} \in \mathbb{R}^{n_k \times D}$ be the centroid matrix and $\mathbf{Q}' \in \mathbb{R}^{n_q \times D}$ be the rotated query batch. The squared Euclidean distance matrix

between queries and centroids can be written in matrix form as

$$\mathbf{D} = \mathbf{s}_\mathrm{q}\mathbf{1}_{n_k}^\top + \mathbf{1}_{n_q}\mathbf{s}_\mathrm{c}^\top - 2\,\mathbf{Q}'\mathbf{C}^\top, \tag{4}$$

where $\mathbf{s}_\mathrm{q} \in \mathbb{R}^{n_q}$ and $\mathbf{s}_\mathrm{c} \in \mathbb{R}^{n_k}$ store the squared ℓ_2 norms of queries and centroids, respectively. The dominant term $\mathbf{Q}'\mathbf{C}^\top$ is computed via a single GEMM on the GPU, and a parallel top-K kernel [58] is then used to select the n_probe nearest clusters per query.

3.3.3 Probe Scheduling via Workload Reorganization. After cluster selection, we reorganize the workload explicitly as a set of (query, cluster) pairs, one pair per probed cluster of each query. We sort these pairs by cluster identifier before launching the search kernels for large batches. This reorganization is adopted from cuVS's IVF-PQ [46], which converts the irregular per-query probe lists into an execution order that is friendly to GPU: first, blocks that process the same (or nearby) clusters execute closer in time, improving cache locality for the RaBitQ codes; second, the clusters of a fixed query across the batch are less likely to be probed concurrently, which gives a tighter running top-K threshold (i.e., the current K-th best distance for each query) to enable more aggressive pruning in later probes. Before these workloads are mapped to SMs on the GPU, we compute query-dependent factors once per query and prepare the query representation needed by the chosen inner-product method (Section 3.4.1); these results are reused across that query's n_probe cluster scans.

3.3.4 Cluster-local Search and Two-stage Distance Estimation. For each (query, cluster) pair, we compute *cluster-local* top-K candidates (which are the query's K nearest neighbors locally within the cluster) using a two-stage estimator:

- **Stage 1 (filter):** compute a fast 1-bit distance estimate (and a lower bound) based on the 1-bit RaBitQ code and prune vectors whose values exceed the current threshold (i.e., the current K-th best distance for each query).
- **Stage 2 (refinement):** for the surviving candidates, read additional bits of the RaBitQ code (ex-code) to compute a more accurate estimated distance and then select the cluster-local top-K candidates.

For simplicity, we refer to the estimated distances in Stage 1 and Stage 2 as the *1-bit estimated distance* and the *refined estimated distance*, respectively. Since the inner-product computation between the query and the 1-bit RaBitQ codes in Stage 1 dominates the computation (note that the computation is conducted for all vectors in the probed clusters) [18], we develop two GPU-native inner-product computation schemes tailored to the GPU execution model (Section 3.4.1). Moreover, this two-stage estimator is realized on the GPU as a four-step cluster-local search procedure, including (1) 1-bit estimated distance computation, (2) candidate selection (i.e., retaining vectors whose estimated distances pass the current threshold), (3) refined estimated distance computation, and (4) in-block top-K selection (i.e., selecting the best K candidates within a thread block). These steps are fused into a single GPU kernel to reduce kernel launch overhead and global-memory traffic (Section 3.4.2).

3.3.5 Batch-level Merge. Each block outputs cluster-local candidates. For each query, we merge candidates from its n_probe clusters and run a GPU top-K to produce the final approximate nearest neighbours.

3.4 Kernel and Data-Layout Co-design

3.4.1 GPU Inner-Product Computation for 1-bit Estimated Distance. Recall in RaBitQ, we need to compute the inner product between a query vector $\mathbf{q}'$ (after rotation) and the 1-bit RaBitQ code of a data vector, which we denote by $\hat{\mathbf{x}}_\mathrm{b}$, for computing the 1-bit estimated distance between the query and the vector (Section 2.2). That is, the inner product is computed in an asymmetric form, where query vectors are represented in higher precision (e.g., floating-point), while data vectors are encoded as compact 1-bit RaBitQ codes. Existing CPU-oriented acceleration techniques [18] are based on SIMD instructions and cannot be directly ported to GPUs due to fundamental differences in execution models and available instruction sets. Thus, a straightforward approach would be to expand the binary codes into byte values and reuse conventional inner-product operators. However, such a design is inefficient on GPUs, as it introduces unnecessary type conversions and leads to suboptimal utilization of GPU compute and memory resources.

To address the problem, we develop two GPU-native inner-product computation schemes for this type of inner product. The first scheme adopts a *lookup table based strategy* that trades shared-memory capacity for reduced arithmetic operations, while the second scheme reformulates the inner product into a *bitwise decomposition* that exploits GPU-friendly bitwise operations.

(1) Lookup Table based Strategy. We precompute query-dependent lookup tables (LUTs) and reuse them across all data vectors when evaluating the 1-bit estimated distance. Different from GPU implementations of IVF-PQ (e.g., cuVS [46]) that build LUTs over PQ codewords (i.e., per PQ dimension/subspace), we construct LUTs that aggregate inner-product contributions over multiple dimensions of the 1-bit RaBitQ code. Formally, we partition the dimensions into groups of U bits and construct multiple lookup tables accordingly. Let $m = D/U$ be the number of partitioned blocks, and

$$\hat{\mathbf{x}}_\mathrm{b}^{(j)},\ \mathbf{q}'^{(j)} \in \mathbb{R}^U, \qquad j = 1, \ldots, m$$

be the subvectors corresponding to the j-th block of the 1-bit code $\hat{\mathbf{x}}_\mathrm{b}$ and the query vector (after rotation) $\mathbf{q}'$, respectively. For each block, we precompute a lookup table (LUT) that stores the inner products between $\mathbf{q}'^{(j)}$ and all possible 2^U binary patterns and store the values. Specifically, given the jth block, we define the corresponding LUT as follows:

$$L_j : \{0,1\}^U \to \mathbb{R} \tag{5}$$

$$L_j(\mathbf{b}) = \langle \mathbf{q}'^{(j)}, \mathbf{b} \rangle = \sum_{i=1}^{U} \mathbf{q}'[(j-1)U + i] \cdot \mathbf{b}[i]\ ,\ \mathbf{b} \in \{0,1\}^U \tag{6}$$

Given a binary code vector $\hat{\mathbf{x}}_\mathrm{b}$, the inner product $\langle \hat{\mathbf{x}}_\mathrm{b}, \mathbf{q}' \rangle$ can then be computed by summing the LUT entries corresponding to each block:

$$\langle \hat{\mathbf{x}}_\mathrm{b}, \mathbf{q}' \rangle = \sum_{j=1}^{m} \langle \hat{\mathbf{x}}_\mathrm{b}^{(j)}, \mathbf{q}'^{(j)} \rangle = \sum_{j=1}^{m} L_j\big(\hat{\mathbf{x}}_\mathrm{b}^{(j)}\big). \tag{7}$$

In practice, each binary block $\hat{\mathbf{x}}_\mathrm{b}^{(j)}$ is interpreted as an integer index $k \in \{0, \ldots, 2^U - 1\}$, allowing the inner product computation to be performed via m table lookups followed by a summation.

According to Section 2.2, in IVF-RaBitQ, the inner product between $\hat{\mathbf{x}}_\mathrm{b}$ and $\mathbf{q}'$ has no relationship with which cluster $\hat{\mathbf{x}}_\mathrm{b}$ belongs

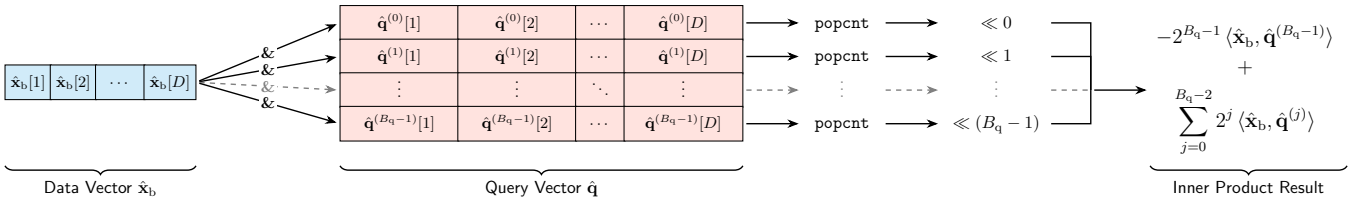

Figure 1: Bitwise Inner Product Computation. $\hat{\mathbf{x}}_\mathrm{b}[i]$ denotes the ith dimension (bit) of the data vector $\hat{\mathbf{x}}_\mathrm{b}$, and $\hat{\mathbf{q}}^{(j)}[i]$ denotes the jth bit of the query vector's ith dimension $\hat{\mathbf{q}}[i]$.

to. That is, to compute the 1-bit estimated distance between a query vector and different data vectors, the LUTs only need to be computed once and can be reused across all data vectors. For each query, we have D/U LUTs, where each LUT uses the U bits as entry and has a size of `sizeof(lut_type)` $\times 2^U$ bytes. To enable high-throughput execution, an ideal case should be that LUTs from different queries can reside in an SM's shared memory (e.g., up to 164 KB on Ampere and 228 KB on Hopper-class GPUs) at the same time, so that multiple queries can be processed concurrently on a single SM. Thus, we set U to 4 to achieve the best performance. For example, if we use FP16 to store inner product results in LUT, for a 1,024-dimensional query, we only need $1024/4 \times 2^4 \times 2\text{B} = 8\text{KB}$ to store all LUTs. Once the LUTs are loaded into the GPU's shared memory, they can be shared with hundreds of threads within a thread block with low access latency, enabling parallel inner-product evaluation for large batches of 1-bit RaBitQ codes.

(2) Bitwise Decomposition based Strategy. While the LUT-based approach is effective, it relies on query-dependent tables residing in shared memory, which can limit on-chip residency and parallelism for high-dimensional vectors or GPUs with relatively small shared memory. To complement this memory-centric design, we revisit the inner-product computation itself to develop a compute-centric scheme that minimizes shared-memory usage and better exploits GPU bitwise instructions.

The classical inner-product computation processes vectors dimension by dimension. On GPUs, however, there is no native instruction that multiplies a packed bit string directly with floating-point values. As a result, a straightforward implementation must first extract each 1-bit entry from the bit string $\hat{\mathbf{x}}_\mathrm{b}$ (at least at byte granularity) before performing the multiplications. Inspired by the bitwise decomposition in RaBitQ [20], we instead adopt a *bitwise* inner-product computation that contrasts with the classical dimension-wise approach. The key idea is to reorganize the computation so that each instruction processes multiple dimensions at once using bit operations. To enable this, we first quantize each rotated query into B_q-bit signed integers $\hat{\mathbf{q}}$. Based on Section 2.2, this quantization is performed once per query and reused across all probed clusters, contributing negligible overhead to the overall search pipeline. With the quantized query, the target inner product $\langle \hat{\mathbf{x}}_\mathrm{b}, \mathbf{q}' \rangle$ can be rewritten as:

$$\langle \hat{\mathbf{x}}_\mathrm{b}, \mathbf{q}' \rangle \approx \Delta_\mathrm{q} \cdot \langle \hat{\mathbf{x}}_\mathrm{b}, \hat{\mathbf{q}} \rangle. \tag{8}$$

where Δ_q is the query quantization scale (step size) that maps the integer-domain inner product back to its original scale (e.g., floating-point) .

Let $\hat{\mathbf{q}}[i]$ be the ith dimension of $\hat{\mathbf{q}}$ where $1 \le i \le D$ and $\hat{\mathbf{q}}^{(j)}[i] \in \{0, 1\}$ be the jth bit of $\hat{\mathbf{q}}[i]$ where $0 \le j < B_\mathrm{q}$ (please refer to presentation of $\hat{\mathbf{q}}$ in Figure 1 for illustration). The bitwise decomposition can be represented using the following equation:

$$\langle \hat{\mathbf{x}}_\mathrm{b}, \hat{\mathbf{q}} \rangle = \sum_{i=1}^{D} \hat{\mathbf{x}}_\mathrm{b}[i] \cdot \hat{\mathbf{q}}[i] \tag{9}$$

$$= \sum_{i=1}^{D} \hat{\mathbf{x}}_\mathrm{b}[i] \cdot \left(-2^{B_\mathrm{q}-1}\, \hat{\mathbf{q}}^{(B_\mathrm{q}-1)}[i] + \sum_{j=0}^{B_\mathrm{q}-2} 2^j\, \hat{\mathbf{q}}^{(j)}[i] \right) \tag{10}$$

$$= -2^{B_\mathrm{q}-1} \langle \hat{\mathbf{x}}_\mathrm{b}, \hat{\mathbf{q}}^{(B_\mathrm{q}-1)} \rangle + \sum_{j=0}^{B_\mathrm{q}-2} 2^j \langle \hat{\mathbf{x}}_\mathrm{b}, \hat{\mathbf{q}}^{(j)} \rangle \tag{11}$$

This equation depicts that the inner product can be viewed as a weighted sum of inner products between binary vectors $\hat{\mathbf{x}}_\mathrm{b}$ and $\hat{\mathbf{q}}^{(j)}$. As shown in Figure 1, the inner product between two binary vectors can be efficiently computed via bitwise AND (&) and population count (POPCNT) operations. Since GPU registers are 32 bits wide, we can process 32 dimensions within a single operation, which provides an advantage over classical dimension-wise inner-product computation. Once $\langle \hat{\mathbf{x}}_\mathrm{b}, \hat{\mathbf{q}}^{(j)} \rangle$ is computed, the final weighted sum can be obtained through several shift and add operations. In this way, the bitwise method converts compression into computation: the compact 1-bit representation is not merely a storage format, but also the format in which the dominant inner product is evaluated. Compared to the LUT-based approach, the bitwise inner product trades additional integer arithmetic for a substantially smaller shared-memory footprint. On GPU, only the quantized query representation needs to be stored in shared memory (e.g., 0.5 KB for a 1,024-dimensional query with 4-bit quantization), making this approach particularly suitable for high-dimensional settings or scenarios where shared memory is the limiting resource.

3.4.2 Fused Cluster-local Search Kernel. Building on the optimized inner-product computation, we next design a fused cluster-local search kernel to efficiently execute the two-stage distance estimation on the GPU. Recall in Section 3.3.4 that the cluster-local search procedure for a (query, cluster) pair (which finds the K nearest neighbors of the query within the cluster) consists of four logical steps, namely (1) 1-bit estimated distance computation, (2) candidate selection, (3) refined estimated distance computation, and (4) in-block top-K selection. A straightforward approach is to implement each of these steps as a separate kernel. However, due to data dependencies between consecutive steps, this design requires launching multiple kernels sequentially, where each kernel loads

its inputs from global memory and writes intermediate results back to global memory. We note that shared memory is scoped to a single thread block and cannot be shared across blocks in different kernels, which means that it is necessary to store intermediate results in global memory between stages. Moreover, when workload imbalance arises across thread blocks, later kernels must wait for the completion of earlier ones, leading to long-tail effects and underutilization of streaming multiprocessors (SMs).

To address these inefficiencies, we design a fused cluster-local search kernel that integrates all four logical steps into a single kernel execution. Rather than launching multiple kernels for different stages, the fused design propagates intermediate results across stages using on-chip memory, enabling block-level synchronization instead of kernel-level synchronization. This design significantly reduces kernel launch overhead, global-memory traffic, and synchronization costs, especially under workload imbalance.

Kernel organization. The fused kernel adopts a block-centric execution model, where each block takes a (query, cluster) pair as an input and returns the cluster-local K-nearest neighbours for the query. Upon launch, a block first loads the query-dependent representation into shared memory. Depending on the selected inner-product computation method (Section 3.4.1), this representation is either the precomputed lookup tables (LUTs) or the quantized query vector used for bitwise inner-product evaluation. After that, threads in the block collaboratively scan the data vectors in the cluster and compute lower bounds of the 1-bit estimated distances. Each thread processes one or more vectors and evaluates the inner product between the query representation and the 1-bit RaBitQ codes. Based on the inner product results, the 1-bit estimated distance lower bounds are computed and compared against a query-specific threshold (the current K-th largest distance among the candidates found so far across clusters). Vectors whose lower bounds exceed the threshold are pruned early, while the remaining vectors are selected as candidates for further refinement. Then, the fused kernel performs distance refinement for the surviving candidates. Since the number of candidates typically constitutes a small fraction of the cluster size, we assign groups of threads within a warp to cooperatively process each candidate. Specifically, threads within a warp jointly compute the inner product between the query and the ex-codes of a candidate, which helps amortize the cost of accessing additional code bits and achieves better workload balance for cases where the number of candidates is small. Using the refined inner-product results together with the associated auxiliary factors, the block computes the refined estimated distance for these candidates. Finally, each block conducts a cluster-local top-K selection using shared memory. After all vectors in the cluster have been processed, the block writes the cluster-local top-K results back to global memory for subsequent merging across clusters, and updates the query-specific threshold atomically.

Shared-memory management via stage-aware reuse. A key challenge of kernel fusion is managing shared-memory usage across stages while preserving sufficient SM residency. Unlike CPU programs that can allocate and free memory dynamically, the shared-memory footprint of a GPU kernel must be determined at launch time, as it directly constrains the number of thread blocks that can concurrently reside on an SM. Excessive shared-memory allocation can therefore reduce occupancy and limit overall throughput. We observe that the shared-memory requirements of different stages in the fused kernel do not fully overlap in time. Specifically, the memory used to store the query-dependent representation (i.e., lookup tables or quantized query vectors) is only needed during the 1-bit distance estimation stage, while later stages require shared memory primarily for storing intermediate inner-product results, candidate metadata, and top-K buffers. Exploiting this property, we reuse shared-memory regions across stages whenever their lifetimes do not overlap: (1) the shared-memory region initially used to store the LUTs or quantized queries is reused to store intermediate inner-product results during distance refinement; (2) the shared memory allocated across the first three stages is repurposed as a sorting buffer for maintaining the cluster-local top-K results in the final stage. By carefully coordinating these reuse patterns, the fused kernel achieves a substantially smaller peak shared-memory footprint than a naive allocation that reserves separate buffers for all stages.

3.4.3 GPU-Oriented Index Layout. To support efficient execution of the fused kernel, we co-design a GPU-oriented index layout that aligns data organization with cluster-local access patterns. IVF-RaBitQ follows a flattened IVF organization inspired by Compressed Sparse Row (CSR) [7] techniques, where vectors belonging to the same cluster are stored contiguously and indexed via an offset array. Building on this baseline organization, we further reorganize RaBitQ codes and auxiliary data to improve data locality, memory coalescing, and cache efficiency during GPU execution. This layout is co-designed with the fused kernel to enable high-throughput search on modern GPUs.

Stage-aware data decomposition. To match the multi-stage execution of the fused kernel, the data associated with each cluster is decomposed into multiple contiguous arrays according to their usage patterns. Specifically, we separate the RaBitQ representation into short codes (the 1-bit RaBitQ codes used in the filtering stage) and long codes (the remaining ex-codes used in the refinement stage). Correspondingly, auxiliary factors required for distance estimation are split into short-factor and long-factor arrays, used for computing 1-bit distances and refined distances, respectively. In addition, a `Pid` array stores the unique identifier of each vector within the cluster. As illustrated in the left part of Figure 2, IVF-RaBitQ stores each component of the index in a separate flattened array. The offset array contains $n_k + 1$ entries, where two consecutive offsets delimit the segment of one cluster. These offsets are shared by all arrays, so the short codes, short factors, long codes, long factors, and vector identifiers (`Pid`) belonging to the same cluster can be located consistently. This stage-aware decomposition improves data locality and avoids loading unused data in early stages of the fused kernel.

Interleaved layout for 1-bit codes. Efficient inner-product computation in the filtering stage requires high memory bandwidth when accessing the 1-bit RaBitQ codes. On GPUs, global memory accesses are serviced via aligned memory transactions (e.g., 32, 64, or 128 bytes), and accesses from consecutive threads in a warp can be coalesced when they target adjacent addresses. To exploit this behavior, we design an interleaved layout for the 1-bit codes. The right part of Figure 2 further shows how the short-code segment of

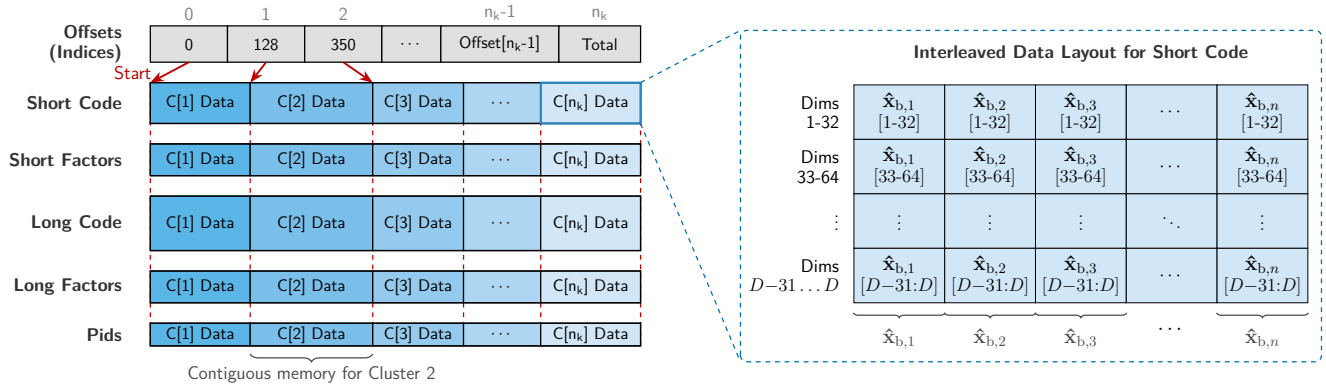

Figure 2: GPU-oriented IVF-RaBitQ index layout.

a cluster is organized. For a cluster containing n vectors of dimension D, the 1-bit code of each vector is packed every 32 dimensions (e.g., dimensions 1–32, 33–64) into a 32-bit unsigned integer. Instead of storing all packed words of one vector consecutively, we store the packed words of all n vectors for the same 32-dimensional group contiguously before moving to the next group. With this layout, threads within a warp access adjacent 32-bit words when processing the same dimension group across different vectors. As a result, memory accesses are coalesced into 128-byte transactions and fit within a single L1 cache line, enabling efficient utilization of GPU memory bandwidth during inner-product computation.

4 EXPERIMENTS

4.1 Experimental Setup

Experimental Platform. The GPU-based experiments are conducted on a machine provided by Amazon Web Services [2]. The machine is equipped with a single NVIDIA L40S GPU with 48 GB of device memory, and an AMD EPYC 7R13 processor providing 8 virtual CPU cores with 64GB main memory. The NVIDIA driver version is 590.48.01 with CUDA 13.1. The CPU-based experiments are performed on a machine equipped with 2 Intel Xeon Gold 6418H processors and 1 TB of DDR5-4800 registered ECC memory, organized as 16 × 64 GB dual-rank DIMMs across two NUMA nodes. Both machines run Ubuntu 22.04 LTS (x86_64).

Datasets. We use nine modern real-world high-dimensional vector datasets to evaluate the performance, as detailed in Table 1. These datasets are taken from the popular ANNS benchmarks and reflect real-world ANNS search scenarios [11, 29, 43, 47]. They cover image (*SIFT* [30], *DEEP* [4, 5], including DEEP-1M with 256 dimensions and DEEP-150M, a 150M-vector cut from DEEP-1B with 96 dimensions, *ImageNet* [12], and *GIST* [30]), code (*CodeSearchNet* [24]), and text (*OpenAI-3072-1M*, *OpenAI-1536-5M*[43] and *Wiki-all* [47]). For each dataset (except for Wiki-all and SIFT), we exclude 10,000 vectors from the dataset as query vectors, and use the rest of them as base data vectors for index construction. For Wiki-all and SIFT, the dataset includes 10,000 query vectors already.

Baselines. We compare our method against several modern ANN methods. For GPU-based methods, we first choose the widely used cluster-based methods, IVF-Flat and IVF-PQ, implemented in the cuVS Library [46]. These methods are built on the inverted file (IVF) index. *IVF-Flat* [16] stores all database vectors in uncompressed form. *IVF-PQ* [9] applies product quantization [30] to compress database vectors into compact codes. It provides two options, namely one with refinement (named IVF-PQ (w/ref)) and the other without refinement (named IVF-PQ (wo/ref)). IVF-PQ (w/ref) re-evaluates shortlisted candidates produced based on the PQ codes using raw data vectors to improve recall. Apart from GPU-based methods that are based on clustering, we also include *CAGRA* [42], a state-of-the-art graph-based GPU ANNS algorithm that constructs and searches fixed-degree proximity graphs fully in parallel. For CPU-based methods, we include *RaBitQLib* [18] (which implements IVF-RaBitQ for ANNS with efficient SIMD instructions). Among these competitors, IVF-PQ and RaBitQLib provide the option for vector quantization. For IVF-PQ (w/ref) and IVF-PQ (wo/ref), we choose the best index settings (e.g., quantization dimensions, quantization bits, and number of clusters) for searching at Recall=0.95 to evaluate the time-accuracy tradeoff. For IVF-RaBitQ and RaBitQLib, we quantize each dimension of vectors to 8 bits unless mentioned otherwise. For RaBitQ quantization on GPU, we empirically set N_{coarse} and N_{fine} to 64 and 32, respectively.

Table 1: Datasets used in evaluation. N is the total number of database vectors and D is the number of dimensions.

Name	Type	N	D	Size (GB)
SIFT-1M	Image	1,000,000	128	0.52
DEEP-1M	Image	1,000,000	256	1.02
ImageNet	Image	1,281,167	512	2.62
CodeSearchNet	Code	1,374,067	768	4.22
GIST	Image	1,000,000	960	3.84
OpenAI-3072-1M	Text	1,000,000	3072	12.29
OpenAI-1536-5M	Text	5,000,000	1536	30.78
Wiki-all	Text	10,000,000	768	30.75
DEEP-150M	Image	150,000,000	96	57.60

Benchmark. All experiments with GPU are conducted using *cuVS Bench* [40], an open-source benchmarking suite developed by NVIDIA for evaluating approximate nearest neighbor (ANN)

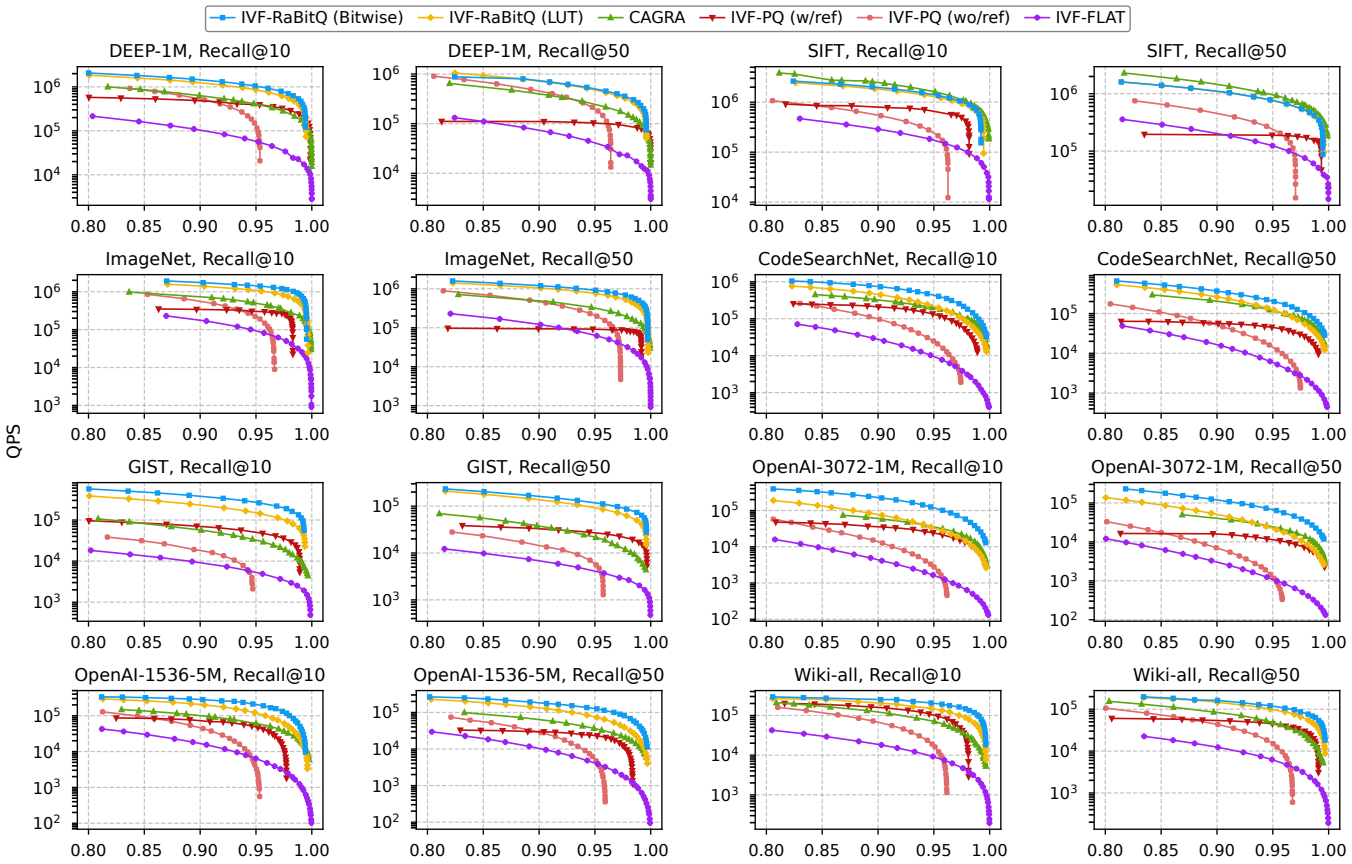

Figure 3: Time–accuracy trade-off of ANN search on representative datasets (log-scale). *Bitwise* and *LUT* denote the two GPU inner-product methods used by IVF-RaBitQ, while *w/ refine* and *w/o refine* indicate whether distance refinement is enabled for IVF-PQ.

algorithms on GPUs. It enables fair comparison across methods by enforcing consistent data loading, query batching, and performance measurement under GPU-resident settings. Unless mentioned otherwise, we use $batchsize = 10^4$ for the benchmark.

4.2 Experimental Results

4.2.1 GPU ANNS Search Performance. In this subsection, we evaluate the GPU-based ANN search performance of IVF-RaBitQ against other representative baselines. For IVF-RaBitQ, we report two different GPU search modes: *IVF-RaBitQ (LUT)* and *IVF-RaBitQ (Bitwise)*, which correspond to two different inner product computation methods (recall Section 3.4.1) used in our fused kernel. Our competitors include IVF-PQ (with and without refinement), IVF-Flat, and CAGRA. All indexes are pre-loaded into GPU memory before the search starts. Figure 3 reports the time-accuracy trade-off for these methods on representative datasets. The performance is evaluated in QPS (query-per-second), and the accuracy is measured by Recall@K (the ratio of true nearest neighbors found in the top-K results returned by an ANNS algorithm). Based on the results, we have the following observations. (1) Comparison among IVF-based methods. Across all datasets, IVF-RaBitQ significantly outperforms IVF-PQ in terms of throughput, especially at higher recall targets. Although IVF-PQ with refinement (denoted as IVF-PQ (w/ref)) improves retrieval accuracy by introducing an additional refinement stage, it still incurs considerable computational and memory overhead when using raw vectors for reranking. In contrast, IVF-RaBitQ attains a more favorable efficiency–accuracy balance without relying on reranking of raw vectors. To be specific, for Recall=0.95, IVF-RaBitQ has 2.4×~58.3× speedup compared to IVF-PQ (wo/ref), and 1.4×~10.0× speedup compared to IVF-PQ (w/ref). (2) Comparison with CAGRA. IVF-RaBitQ demonstrates comparable or higher throughput than CAGRA across most datasets and recall ranges. IVF-RaBitQ benefits from compression while CAGRA works with uncompressed data. On average, IVF-RaBitQ (Bitwise) achieves a speed up of 2.9×, 3.3×, 4.0× compared to CAGRA for recall=0.9, 0.95, 0.99@10, respectively. The performance advantage becomes more evident on higher recall, which highlights the advantage of cache reuse while probing many clusters for a large batch of queries. We note that when the dimensionality is relatively low (e.g., SIFT with $D = 128$), CAGRA has a slightly better performance than IVF-RaBitQ, which is attributed to a lower computational overhead of CAGRA's graph traversal at low dimensionality. (3) Bitwise vs. LUT-based GPU kernels. We further investigate two inner product

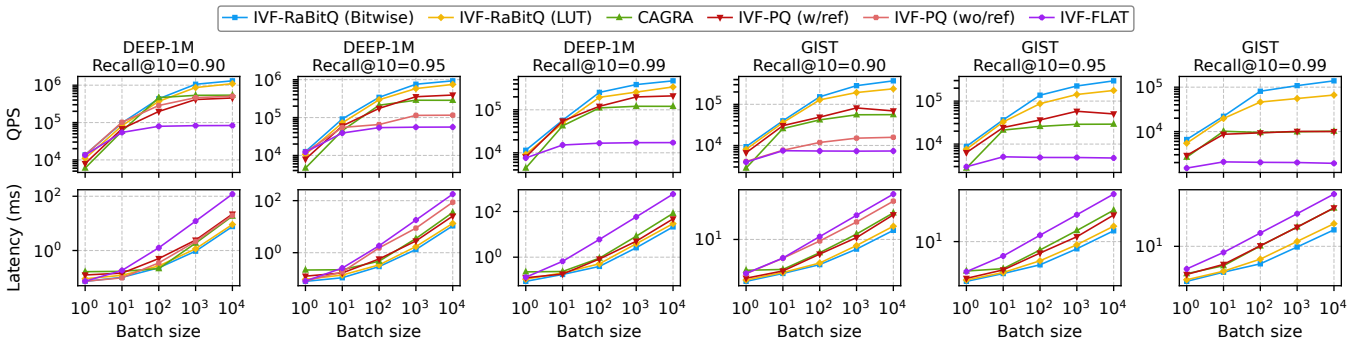

Figure 4: Throughput and Latency scaling with batch size on DEEP-1M and GIST dataset.

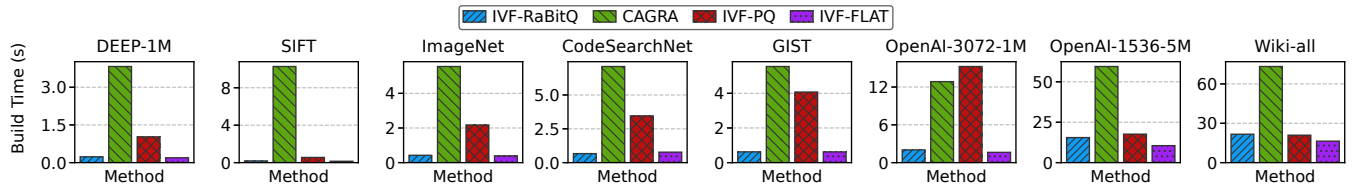

Figure 5: Index build time across different datasets. Each subplot compares build time across methods.

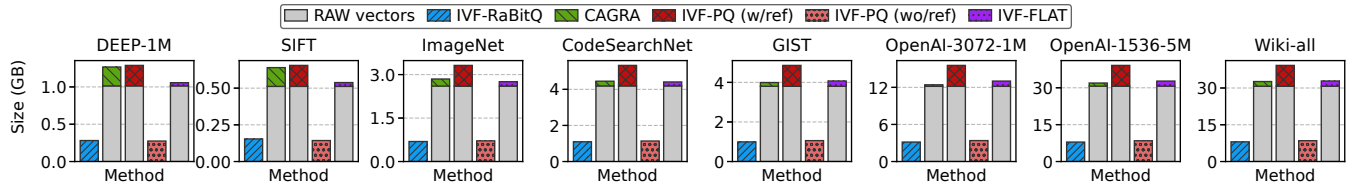

Figure 6: Storage requirement of different indexing methods. CAGRA and IVF-PQ (w/ref) require RAW vectors for refinement, thus the storage of RAW vectors is included.

computation methods for IVF-RaBitQ: a bitwise inner-product kernel and a lookup-table (LUT) based kernel. Both implementations achieve high performance across all datasets. As dimensionality increases, the LUT-based kernel is bounded by shared-memory capacity, whereas the bitwise formulation remains lightweight and scalable which is reflected by gap between the two methods on the OpenAI-3072-1M dataset. We note that the relative performance of these two methods may vary based on the memory bandwidth and shared memory size of different GPUs. (4) Scalability across dataset sizes and dimensionalities. Across datasets ranging from 1M to 10M vectors and dimensions from 256 to 3072, IVF-RaBitQ maintains robust performance and consistent throughput advantages.

4.2.2 Scalability with Query Batch Size. We compare the methods by varying the query batch size. Figure 4 reports the throughput and latency as a function of batch size for target Recall@10 of 0.90, 0.95, and 0.99. In terms of throughput, IVF-RaBitQ is comparable to IVF-PQ (w/ref) and CAGRA at small batch sizes (1 and 10), while significantly outperforming both at larger batch sizes (100, 1,000 and 10,000), where GPU parallelism is more fully utilized. In terms of latency, all methods achieve sub-millisecond response times at small batch sizes. As the batch size increases, IVF-RaBitQ maintains substantially lower latency than the competing methods, demonstrating favorable scalability across workload intensities.

4.2.3 Index Build Time. Figure 5 shows the index build time for the indexes used in Section 4.2.1. For all IVF-based methods, we use one-tenth of the dataset vectors to train the clusters and align the number of clusters to be trained. Also, we align the IVF-PQ and IVF-RaBitQ to use the same compression ratio for quantization (i.e., 8 bits per dimension). We observe that IVF-RaBitQ has a very short index build time across all evaluated datasets. When compared with CAGRA, the proposed method attains an average speedup of 14.7×. This is because CAGRA needs to spend more time on its KNN graph building and graph optimization procedures. This overhead becomes increasingly dominant as the dataset scale and dimensionality grow. Compared with other IVF-based methods, IVF-RaBitQ is overall faster than IVF-PQ (with a 4.2× speedup on average) and is only suboptimal to IVF-Flat.

4.2.4 Storage Footprint. We analyze the storage footprint of different ANN indices, which is the total amount of data that is stored to support search, as shown in Figure 6. We note that IVF-PQ (wo/ref) uses the same compression ratio (8 bits per dimension for a

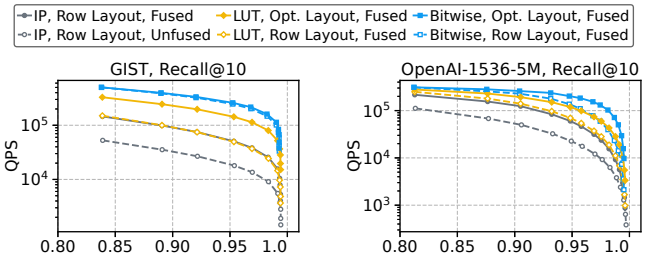

Figure 7: Ablation study for kernel and data layout co-design.

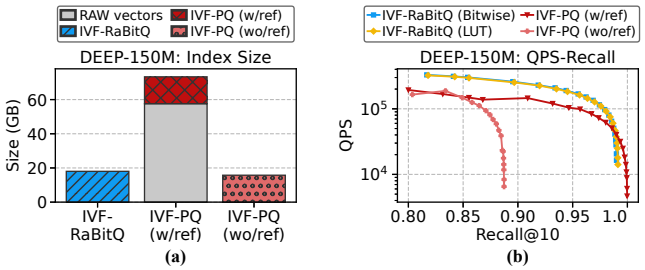

Figure 8: ANNS search on DEEP-150M, where the raw vectors exceed the memory capacity of a single L40S GPU: (a) storage footprint of IVF-RaBitQ and IVF-PQ variants, and (b) time-accuracy tradeoff.

floating-point vector) as IVF-RaBitQ. CAGRA, IVF-Flat, and IVF-PQ with refinement need to store the raw data vectors to support exact or refined distance computation, which significantly increases their storage demand. Compared to these methods, IVF-RaBitQ only needs less than 25% of their storage requirement to achieve a comparable recall. Compared with IVF-PQ (wo/ref), which does not require raw vectors, IVF-RaBitQ achieves a similar storage footprint while delivering substantially higher recall and query throughput. This advantage could enable deploying large-scale ANN systems within the limited memory budget of modern GPUs.

4.2.5 Ablation Study for Kernel and Data Layout Co-design. We conduct an ablation study to quantify the individual contributions of the three key design dimensions in IVF-RaBitQ: inner-product computation method, data layout, and kernel organization. As shown in Figure 7, we evaluate the following variants: (1) three inner-product computation methods—naïve floating-point inner product (IP), lookup-table based (LUT), and bitwise decomposition based (Bitwise); (2) two data layouts—a row-major naïve layout (Row Layout) and our optimized bit-major interleaved layout (Opt. Layout); and (3) two kernel organizations—separate unfused kernels and our fused cluster-local search kernel. The results confirm that each design dimension contributes meaningfully to overall performance: the fully optimized LUT and Bitwise kernels achieve over 5× higher throughput than the naïve unfused baseline at recall levels above 0.95. Notably, the benefit of the optimized data layout is more pronounced on high-dimensional datasets, where global-memory traffic is more intensive and coalesced access patterns yield a larger reduction in memory stalls.

4.2.6 ANNS Search for Datasets Exceeding GPU Memory. We evaluate IVF-RaBitQ on DEEP-150M, whose raw base vectors occupy about 57.6 GB, which is larger than the 48 GB device memory of a single NVIDIA L40S GPU, yet within the 64 GB host memory capacity. Figure 8 reports both the storage footprint and the time–accuracy trade-off for all applicable methods. CAGRA and IVF-Flat cannot handle this dataset due to their large runtime memory requirements. Note that both of them use raw vectors on GPU for searching. IVF-RaBitQ requires only about 18.0 GB for its index and does not require raw vectors during search. In contrast, IVF-PQ with refinement must retain both the compressed index and the raw vectors, resulting in a total footprint of 73.3 GB. For search performance, IVF-RaBitQ achieves 1.58× and 1.70× higher QPS than IVF-PQ (w/ref) at Recall ≥ 0.90 and ≥ 0.95, respectively, while IVF-PQ (wo/ref) cannot reach these recall levels at all.

4.2.7 Quantization Accuracy and Runtime. We evaluate the trade-off between quantization accuracy and runtime by varying N_{coarse} and N_{fine} in our two-phase parallel grid search method. In this experiment, we quantize the normalized vectors without building an index structure, and report the mean ℓ_2 quantization error alongside the wall-clock quantization time. Figure 9 shows that, on the GIST dataset with 8-bit quantization, setting both N_{coarse} and N_{fine} to as few as 16 already yields a mean quantization error below 0.008. We adopt slightly conservative defaults of $N_{\text{coarse}} = 64$ and $N_{\text{fine}} = 32$ to ensure high quantization accuracy with negligible additional overhead.

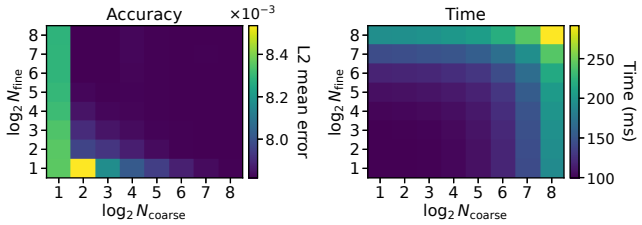

Figure 9: Quantization accuracy and runtime of the GPU-based RaBitQ quantization at 8 bits on the GIST dataset.

4.2.8 Comparison with CPU-based Counterpart. We further compare IVF-RaBitQ (GPU) with the bitwise inner product computation method and its counterpart on CPU, for which we adopt the implementation from the RaBitQLib [18]. The CPU counterpart is bound to one physical CPU socket (utilizing 48 threads) on a fixed NUMA node, with memory allocated from the same NUMA node to avoid cross-socket migration and remote-NUMA effects. Figure 10 (a) shows the build time comparison between RaBitQLib and IVF-RaBitQ (GPU) on GIST dataset, where IVF-RaBitQ (GPU) is 42.3× faster. Figure 10 (b) reports the time-accuracy trade-off curves on the GIST dataset. At Recall@10 = 0.9, 0.95, and 0.99, IVF-RaBitQ (GPU) achieves QPS speedups of 19.8×, 16.8×, and 15.1×, respectively over its CPU counterpart.

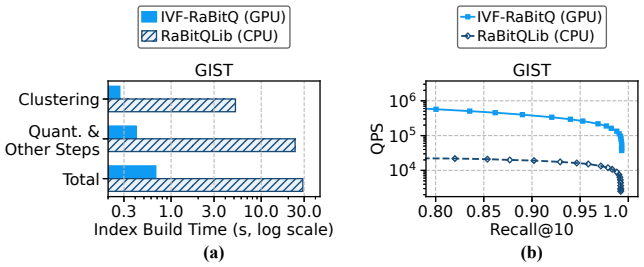

Figure 10: (a) Index build time breakdown, (b) Time-accuracy trade-off of IVF-RaBitQ and its CPU counterpart (RaBitQLib) on GIST dataset. The GPU implementation runs on an NVIDIA L40S, and the CPU counterpart runs on an Intel Xeon Gold 6418H with 48 threads.

5 RELATED WORK

Graph-Based ANNS on GPU. Graph-based methods typically build a proximity graph and answer queries by iteratively expanding a candidate frontier from one or a few entry points, while pruning candidates based on distance comparisons. The graph-based ANNS search algorithms work cooperatively with the graph index, and recent GPU designs mainly focus on accelerating graph traversal and candidate management (e.g., priority queues, visited sets, and hash-table-based filtering) under massive parallelism. Graph-based GPU Nearest Neighbor (GGNN) [23] proposes a parallel, bottom-up hierarchical construction algorithm that recursively merges smaller, independent sub-graphs into a final global index. GGraphCon [57] is a GPU-based graph construction framework which uses a divide-and-conquer framework that partitions the graph building into different small tasks, and a sort-based strategy to build high-quality graphs in parallel. CAGRA [42] uses a fixed-degree graph and rank-based edge reordering to improve GPU load balance and graph reachability, while its search kernel employs software warp splitting and forgettable hash-table management to reduce visited-set overhead. BANG [51] targets billion-scale graph search beyond GPU memory by keeping a compressed representation of vectors on the GPU while accessing the graph index from host memory, thereby optimizing the CPU–GPU memory layout rather than changing the graph traversal paradigm. A recent work, concurrent with ours, develops a graph-based method called Jasper for ANNS on GPU, which uses RaBitQ for vector quantization and supports data updates [39]. We note that this work computes the rescaling factor on the CPU, while performing the rescaling and rounding on the GPU - an execution split that distinguishes it from our approach. Tagore [34] focuses on graph index construction and formulates graph-pruning procedures as Collect-Filter-Store kernels, enabling parallel edge pruning for refinement-based graph indexes. These methods accelerate complex graph construction and irregular, query-dependent graph traversal, where performance is dominated by frontier expansion, duplicate detection, neighbor-list access, and candidate pruning. In contrast, IVF-RaBitQ follows a cluster-based scan over selected inverted lists and avoids maintaining graph frontiers or visited sets during search.

Cluster-based ANNS on GPU. While most graph-based methods focus on maximizing the search throughput, the cluster-based methods usually deliver faster index builds, lower memory usage, and a strong balance between speed and accuracy [32]. In 2016, Wieschollek et al. proposed a two-level product and vector quantization tree that reduces the number of vector comparisons for ANNS tree traversal, which firstly demonstrates GPU performance is superior to CPU performance in large-scale ANN problems [53]. Later, FAISS [26] develops a high-performance top-K selection algorithm on a GPU and integrates it with IVF-PQ for billion-scale ANNS search on multiple GPUs. Chen et al. proposed a hierarchical inverted index structure generated by line quantization to enable more fine-grained search in IVF [8]. Recently, Nvidia cuVS [9] further optimizes LUT residency in shared memory (via low-precision LUT types and kernel-path switching), and employs fused/early-stop kernels for IVF-PQ, making it a state-of-the-art implementation for IVF-PQ on GPU. Rummy [59] and FusionANNS [48] both focus on the scenario when vector data is larger than the GPU global memory, where Rummy is based on pure IVF and FusionANNS further extends it with Product Quantization and a navigation graph. JUNO [35] observes that PQ-based ANN search may perform unnecessary pairwise distance calculations and accumulations, and therefore introduces a sparsity- and locality-aware search algorithm mapped to ray-tracing cores and tensor cores. In contrast to these GPU cluster-based solutions that primarily build on IVF-Flat/IVF-PQ (and their codebook-based distance estimators), and optimize either PQ-LUT evaluation, cross-device data placement, or re-ranking I/O, our IVF-RaBitQ integrates RaBitQ and introduces GPU-native quantization, inner-product computation, and fused-kernel designs tailored to this method.

6 CONCLUSION

We present IVF-RaBitQ (GPU), an end-to-end GPU-native ANNS framework that integrates IVF partitioning with RaBitQ quantization. IVF-RaBitQ enables fast index build, high-throughput search, high recall, and compact storage by combining a scalable GPU quantization pipeline, GPU-native inner-product computation, a fused cluster-local search kernel, and a GPU-oriented index layout. We further integrate IVF-RaBitQ into NVIDIA **cuVS** and the **cuVS bench** pipeline to support reproducible evaluation and fair comparisons with existing GPU baselines. Experiments on various datasets show that IVF-RaBitQ offers a strong trade-off frontier across recall, throughput, build time, and memory footprint, especially in high-recall regimes. In the future, we plan to explore extensions such as support for dynamic index updates and transactional operations.

ACKNOWLEDGMENTS

We sincerely appreciate the insightful discussions and constructive suggestions provided by Artem Chirkin, Akira Naruse, and Hiroyuki Ootomo at NVIDIA, as well as Yutong Gou and Yuexuan Xu at Nanyang Technological University. This research is supported by the Ministry of Education, Singapore, under its Academic Research Fund (Tier 2 Award MOE-T2EP20224-0011 and Tier 1 Award (RG20/24)). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of the Ministry of Education, Singapore.

REFERENCES

[1] Cecilia Aguerrebere, Ishwar Singh Bhati, Mark Hildebrand, Mariano Tepper, and Theodore Willke. 2023. Similarity Search in the Blink of an Eye with Compressed Indices. *Proc. VLDB Endow.* 16, 11 (July 2023), 3433–3446. https://doi.org/10.14778/3611479.3611537

[2] Amazon Web Services, Inc. 2026. Amazon EC2 G6e Instances. https://aws.amazon.com/ec2/instance-types/g6e/. Accessed: 2026-02-23.

[3] David Arthur and Sergei Vassilvitskii. 2007. k-means++: the advantages of careful seeding. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms* (New Orleans, Louisiana) *(SODA '07)*. Society for Industrial and Applied Mathematics, USA, 1027–1035.

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

[5] Artem Babenko and Victor Lempitsky. 2017. Product Split Trees. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 6316–6324. https://doi.org/10.1109/CVPR.2017.669

[6] Fedor Borisyuk, Siddarth Malreddy, Jun Mei, Yiqun Liu, Xiaoyi Liu, Piyush Maheshwari, Anthony Bell, and Kaushik Rangadurai. 2021. VisRel: Media Search at Scale. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining* (Virtual Event, Singapore) *(KDD '21)*. Association for Computing Machinery, New York, NY, USA, 2584–2592. https://doi.org/10.1145/3447548.3467081

[7] Aydin Buluç, Jeremy T. Fineman, Matteo Frigo, John R. Gilbert, and Charles E. Leiserson. 2009. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures* (Calgary, AB, Canada) *(SPAA '09)*. Association for Computing Machinery, New York, NY, USA, 233–244. https://doi.org/10.1145/1583991.1584053

[8] Wei Chen, Jincai Chen, Fuhao Zou, Yuan-Fang Li, Ping Lu, Qiang Wang, and Wei Zhao. 2019. Vector and line quantization for billion-scale similarity search on GPUs. *Future Generation Computer Systems* 99 (2019), 295–307. https://doi.org/10.1016/j.future.2019.04.033

[9] Artem Chirkin, Akira Naruse, Tamás Fehér, Yong Wang, and Corey Nolet. 2024. Accelerating Vector Search: NVIDIA cuVS IVF-PQ (Part 1, Deep Dive). NVIDIA Technical Blog. https://developer.nvidia.com/blog/accelerating-vector-search-nvidia-cuvs-ivf-pq-deep-dive-part-1/ Accessed: 2026-07-18.

[10] Yannis Chronis, Helena Caminal, Yannis Papakonstantinou, Fatma Özcan, and Anastasia Ailamaki. 2025. Filtered Vector Search: State-of-the-Art and Research Opportunities. *Proc. VLDB Endow.* 18, 12 (Aug. 2025), 5488–5492. https://doi.org/10.14778/3750601.3750700

[11] CSE, CUHK. 2018. GQR Datasets. https://www.cse.cuhk.edu.hk/systems/hash/gqr/datasets.html. Accessed: 2026-01-29.

[12] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 248–255. https://doi.org/10.1109/CVPR.2009.5206848

[13] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv:1810.04805 [cs.CL] https://arxiv.org/abs/1810.04805

[14] Xin Luna Dong. 2024. The Journey to A Knowledgeable Assistant with Retrieval-Augmented Generation (RAG). In *Proceedings of the 17th ACM International Conference on Web Search and Data Mining* (Merida, Mexico) *(WSDM '24)*. Association for Computing Machinery, New York, NY, USA, 4. https://doi.org/10.1145/3616855.3638207

[15] Ming Du, Arnau Ramisa, Amit Kumar K C, Sampath Chanda, Mengjiao Wang, Neelakandan Rajesh, Shasha Li, Yingchuan Hu, Tao Zhou, Nagashri Lakshminarayana, Son Tran, and Doug Gray. 2022. Amazon Shop the Look: A Visual Search System for Fashion and Home. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining* (Washington DC, USA) *(KDD '22)*. Association for Computing Machinery, New York, NY, USA, 2822–2830. https://doi.org/10.1145/3534678.3539071

[16] Tamás Fehér, Artem Chirkin, Christina Zhang, and Mahesh Doijade. 2023. Accelerated Vector Search: Approximating with NVIDIA cuVS Inverted Index. NVIDIA Technical Blog. https://developer.nvidia.com/blog/accelerated-vector-search-approximating-with-nvidia-cuvs-ivf-flat/ Accessed: 2026-07-18.

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

[18] Jianyang Gao, Yutong Gou, Yuexuan Xu, Jifan Shi, Zhonghao Yang, and Cheng Long. 2025. The RaBitQ Library. In *The 1st Workshop on Vector Databases*. https://openreview.net/forum?id=OeZHhOsFir

[19] Jianyang Gao, Yutong Gou, Yuexuan Xu, Yongyi Yang, Cheng Long, and Raymond Chi-Wing Wong. 2025. Practical and Asymptotically Optimal Quantization of High-Dimensional Vectors in Euclidean Space for Approximate Nearest Neighbor Search. *Proc. ACM Manag. Data* 3, 3, Article 202 (June 2025), 26 pages. https://doi.org/10.1145/3725413

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

[21] Google AI. 2024. Embeddings | Gemini API | Google AI for Developers. https://ai.google.dev/gemini-api/docs/embeddings. Accessed: 2026-02-01.

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

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

[24] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2020. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. arXiv:1909.09436 [cs.LG] https://arxiv.org/abs/1909.09436 Accessed: 2026-07-18.

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

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

[27] William B. Johnson and Joram Lindenstrauss. 1984. Extensions of Lipschitz Mappings into a Hilbert Space. In *Conference in Modern Analysis and Probability*. Contemporary Mathematics, Vol. 26. American Mathematical Society, Providence, RI, 189–206. https://doi.org/10.1090/conm/026/737400

[28] js1010. 2021. cuhnsw: CUDA implementation of Hierarchical Navigable Small World Graph algorithm. https://github.com/js1010/cuhnsw. GitHub repository, Accessed: 2026-02-01.

[29] Elias Jääsaari, Ville Hyvönen, Matteo Ceccarello, Teemu Roos, and Martin Aumüller. 2025. VIBE: Vector Index Benchmark for Embeddings. arXiv:2505.17810 [cs.LG] https://arxiv.org/abs/2505.17810

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

[31] 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 *Proceedings of the 2025 USENIX Conference on Usenix Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '25)*. USENIX Association, USA, Article 89, 17 pages.

[32] Jack Li. 2025. Understanding IVF Vector Index: How It Works and When to Choose It Over HNSW. Milvus Blog. https://milvus.io/blog/understanding-ivf-vector-index-how-It-works-and-when-to-choose-it-over-hnsw.md Accessed: 2026-01-03.

[33] Wen Li, Ying Zhang, Yifang Sun, Wei Wang, Mingjie Li, Wenjie Zhang, and Xuemin Lin. 2020. Approximate Nearest Neighbor Search on High Dimensional Data — Experiments, Analyses, and Improvement. *IEEE Transactions on Knowledge and Data Engineering* 32, 8 (2020), 1475–1488. https://doi.org/10.1109/TKDE.2019.2909204

[34] Zhonggen Li, Xiangyu Ke, Yifan Zhu, Bocheng Yu, Baihua Zheng, and Yunjun Gao. 2025. Scalable Graph Indexing using GPUs for Approximate Nearest Neighbor Search. *Proc. ACM Manag. Data* 3, 6, Article 360 (Dec. 2025), 27 pages. https://doi.org/10.1145/3769825

[35] Zihan Liu, Wentao Ni, Jingwen Leng, Yu Feng, Cong Guo, Quan Chen, Chao Li, Minyi Guo, and Yuhao Zhu. 2024. JUNO: Optimizing High-Dimensional Approximate Nearest Neighbour Search with Sparsity-Aware Algorithm and Ray-Tracing Core Mapping. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 549–565. https://doi.org/10.1145/3620665.3640360

[36] J. MacQueen. 1967. Some Methods for Classification and Analysis of Multivariate Observations. In *Proc. Fifth Berkeley Symp. Math. Statist. Probab.*, Vol. 1. University of California Press, Berkeley, CA, 281–297.

[37] Mikko I. Malinen and Pasi Fränti. 2014. Balanced K-Means for Clustering. In *Proceedings of the Joint IAPR International Workshop on Structural, Syntactic, and Statistical Pattern Recognition - Volume 8621* (Joensuu, Finland) *(S+SSPR 2014)*. Springer-Verlag, Berlin, Heidelberg, 32–41. https://doi.org/10.1007/978-3-662-44415-3_4

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

[39] Hunter McCoy, Zikun Wang, and Prashant Pandey. 2026. GPU-Accelerated ANNS: Quantized for Speed, Built for Change. arXiv:2601.07048 [cs.DB] https://arxiv.org/abs/2601.07048

[40] NVIDIA Corporation. 2026. cuVS Benchmark — cuVS documentation. https://docs.rapids.ai/api/cuvs/stable/cuvs_bench/. Accessed: 2026-02-26.

[41] NVIDIA Corporation. 2026. NVIDIA cuVS: GPU-Accelerated Vector Search and Data Clustering. NVIDIA Developer. https://developer.nvidia.com/topics/ai/generative-ai/cuvs Accessed: 2026-07-18.

[42] 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 Computer Society, Los Alamitos, CA, USA, 4236–4247. https://doi.org/10.1109/ICDE60146.2024.00323

[43] OpenAI. 2024. New embedding models and API updates. https://openai.com/index/new-embedding-models-and-api-updates/. Accessed: 2026-01-29.

[44] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. 2021. Learning Transferable Visual Models From Natural Language Supervision. arXiv:2103.00020 [cs.CV] https://arxiv.org/abs/2103.00020

[45] RAPIDS AI. 2026. K-Means — cuVS C++ API Documentation. RAPIDS cuVS documentation. https://docs.rapids.ai/api/cuvs/stable/cpp_api/cluster_kmeans/ Accessed: 2026-01-26.

[46] RAPIDS AI. 2026. rapidsai/cuvs: cuVS – a library for vector search and clustering on the GPU. GitHub repository. https://github.com/rapidsai/cuvs Accessed: 2026-01-03.

[47] RAPIDS Team. 2024. Wiki-all Dataset — cuVS Documentation. https://docs.rapids.ai/api/cuvs/stable/cuvs_bench/wiki_all_dataset/. Accessed: 2026-01-29.

[48] 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 *Proceedings of the 23rd USENIX Conference on File and Storage Technologies* (Santa Clara, CA, USA) *(FAST '25)*. USENIX Association, USA, Article 11, 15 pages.

[49] Dan Vanderkam, Rob Schonberger, Henry Rowley, and Sanjiv Kumar. 2013. *Nearest Neighbor Search in Google Correlate*. Technical Report. Google. https://research.google/pubs/nearest-neighbor-search-in-google-correlate/ Accessed: 2026-07-18.

[50] VectorDB-NTU. 2026. RaBitQ Library Documentation. Online documentation. https://vectordb-ntu.github.io/RaBitQ-Library/ Accessed: 2026-07-18.

[51] Karthik Venkatasubba, Saim Khan, Somesh Singh, Harsha Vardhan Simhadri, and Jyothi Vedurada. 2025. BANG: Billion-Scale Approximate Nearest Neighbour Search Using a Single GPU . *IEEE Transactions on Big Data* 11, 06 (Dec. 2025), 3142–3157. https://doi.org/10.1109/TBDATA.2025.3581085

[52] Hui Wang, Wan-Lei Zhao, Xiangxiang Zeng, and Jianye Yang. 2021. Fast k-NN Graph Construction by GPU based NN-Descent. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management* (Virtual Event, Queensland, Australia) *(CIKM '21)*. Association for Computing Machinery, New York, NY, USA, 1929–1938. https://doi.org/10.1145/3459637.3482344

[53] Patrick Wieschollek, Oliver Wang, Alexander Sorkine-Hornung, and Hendrik P. A. Lensch. 2016. Efficient Large-Scale Approximate Nearest Neighbor Search on the GPU. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2027–2035. https://doi.org/10.1109/CVPR.2016.223

[54] Wikipedia contributors. 2024. Wikipedia:Pageview statistics — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Wikipedia:Pageview_statistics. Accessed: 2026-02-01.

[55] Jingyi Xi, Chenghao Mo, Ben Karsin, Artem Chirkin, Mingqin Li, and Minjia Zhang. 2025. VecFlow: A High-Performance Vector Data Management System for Filtered-Search on GPUs. *Proc. ACM Manag. Data* 3, 4, Article 271 (Sept. 2025), 27 pages. https://doi.org/10.1145/3749189

[56] 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 (DaMoN '25)*. Association for Computing Machinery, New York, NY, USA, Article 7, 10 pages. https://doi.org/10.1145/3736227.3736237

[57] 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)*. 552–564. https://doi.org/10.1109/ICDE53745.2022.00046

[58] Jingrong Zhang, Akira Naruse, Xipeng Li, and Yong Wang. 2023. Parallel Top-K Algorithms on GPU: A Comprehensive Study and New Methods. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, CO, USA) *(SC '23)*. Association for Computing Machinery, New York, NY, USA, Article 76, 13 pages. https://doi.org/10.1145/3581784.3607062

[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 *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation* (Santa Clara, CA, USA) *(NSDI'24)*. USENIX Association, USA, Article 2, 18 pages.

[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)*. 1033–1044. https://doi.org/10.1109/ICDE48307.2020.00094