# Parallel Acceleration of the Inverted File Index for Approximate Nearest Neighbor Search

CHE-WEI,CHOU
313551118
Dept. of Computer Science
National Yang Ming Chiao Tung University

TE-YEN, CHIU
313551038
Dept. of Computer Science
National Yang Ming Chiao Tung University

GUAN-LIN, CHEN
313553054
Dept. of Computer Science
National Yang Ming Chiao Tung University

## Abstract

Approximate Nearest Neighbor Search (ANNS) is a critical component in modern vector retrieval systems, enabling efficient search over high-dimensional embeddings. The Inverted File Index (IVF) represents a widely adopted clustering-based ANNS approach that partitions the search space into coarse Voronoi cells, dramatically reducing query complexity from $O(N)$ to $O(n_{\text{probe}} \cdot n_{\text{list,avg}})$. However, despite this algorithmic efficiency, IVF's computational bottlenecks during index construction and query evaluation remain significant at scale. This project presents a comprehensive parallelization of the IVF-Flat algorithm targeting both multi-core CPUs and GPUs. We implement task-level parallelism via OpenMP for concurrent query processing and vector assignment, data-level parallelism through AVX2 SIMD intrinsics for vectorized distance computations, and massively parallel CUDA kernels for GPU acceleration. Our implementation is evaluated on standard benchmarks (SIFT1M and GIST1M) using the Recall–QPS trade-off metric. We systematically analyze performance through profiling tools including Intel VTune and NVIDIA Nsight to identify memory bandwidth limitations and load imbalance challenges inherent to irregular inverted list structures. Preliminary results target 2.5–4× CPU and 5–8× GPU speedups over the single-threaded baseline, while maintaining Recall@10 $\geq 0.95$. This work offers both practical and pedagogical insights into parallelizing classical ANNS algorithms and quantifies the architectural trade-offs between CPU and GPU implementations for large-scale vector search.

## 1 Introduction

In modern artificial intelligence and information retrieval systems, *vector search* has become a fundamental component for semantic matching. Instead of comparing discrete identifiers or keywords, each data item—such as a sentence, an image, or an audio clip—is represented as a high-dimensional *embedding vector* that captures its semantic meaning. When a user submits a query, the system retrieves the $k$ most similar vectors in this space, known as the $k$-nearest neighbors ($k$-NN). This mechanism underpins a wide range of applications, including *Retrieval-Augmented Generation (RAG)* for large language models, web search engines, and recommendation systems. However, a brute-force $k$-NN search, which compares a query vector against every vector in the dataset, quickly becomes computationally prohibitive as data volume grows. The complexity scales linearly with the dataset size, making such exhaustive search infeasible for large-scale applications.

To address this scalability challenge, researchers have developed *Approximate Nearest Neighbor Search (ANNS)* algorithms, which aim to return results close to those of exact $k$-NN search but with drastically reduced computation time. ANNS methods exploit spatial locality and data partitioning to narrow down the search space, reducing the number of distance evaluations while maintaining acceptable recall. This trade-off between speed and accuracy has made ANNS a cornerstone of modern large-scale retrieval systems and vector databases. Among the various ANNS methods, the *Inverted File Index (IVF)* is one of the most widely adopted designs. Its core idea is straightforward: first, partition the entire vector space into a predefined number of coarse clusters using $k$-means; then, for each query, search only within the few clusters whose centroids are closest to the query vector. Each cluster maintains an *inverted list* of assigned data points. This structure significantly reduces the number of comparisons while maintaining low memory overhead and straightforward implementation. IVF can also be combined with techniques such as *Product Quantization (PQ)* to further optimize storage and search speed. Due to these advantages, IVF has been extensively adopted in large-scale similarity search frameworks such as *FAISS* (Facebook AI Similarity Search) and *ScaNN* (Scalable Nearest Neighbor Search by Google).

Despite its efficiency, the traditional IVF pipeline—particularly during index construction and query search—still involves intensive computation on large datasets. The $k$-means clustering stage and the distance calculations during query evaluation are both highly parallelizable tasks. This motivates our project: to design and implement a *parallel version of the IVF algorithm* that leverages modern parallel programming techniques. By parallelizing index construction, vector assignments, and query-level distance evaluations, we aim to accelerate both index building and search operations without compromising accuracy. The project demonstrates how classical ANNS techniques can be re-engineered through parallel programming to achieve scalable, high-performance retrieval suitable for today's data-intensive environments.

## 2 Statement of the Problem

### 2.1 Inverted File Index (IVF)

The application chosen for this project is the *Inverted File Index (IVF)*, a clustering-based Approximate Nearest Neighbor Search (ANNS) algorithm widely used in large-scale vector retrieval systems. The IVF method is built upon a two-stage search process: a coarse quantization stage and a fine search stage. In the first stage, the entire dataset is partitioned into $n_{list}$ coarse clusters using the $k$-means algorithm. Each centroid acts as a representative for its assigned subset of vectors, known as an *inverted list*. In the second stage, a query vector is compared only against a small number of these clusters (typically $n_{probe} \ll n_{list}$), and the search is then refined by computing distances within the corresponding lists.

This hierarchical search structure allows IVF to dramatically reduce the number of distance computations compared to exhaustive $k$-NN search. The total search cost is approximately $O(n_{probe} \cdot n_{list\_avg})$ instead of $O(N)$, where $N$ is the total number of vectors. The accuracy can be controlled by tuning $n_{list}$ and $n_{probe}$, providing a flexible trade-off between speed and recall. In practice, IVF is the foundation of systems like FAISS and ScaNN, often combined with Product Quantization (PQ) or Residual Vector Quantization (RVQ) for further compression and latency reduction.

## 2.2 Motivation

While the Inverted File Index (IVF) substantially reduces the search space compared to exhaustive $k$-NN, its two main phases—*index construction* and *query search*—remain computationally demanding at scale. Crucially, IVF exhibits structural properties that make it particularly well-suited for parallel execution at both the *task level* and the *data level*.

*Task-level parallelism.* IVF naturally decomposes into independent units of work. During index construction, the assignment of each dataset vector to its nearest centroid is independent of other vectors; during query time, different queries do not depend on one another and can be processed concurrently. This independence enables parallel scheduling across large batches of vectors (for building) or queries (for search), improving throughput without altering algorithmic semantics.

*Data-level parallelism.* Within each task, IVF exposes fine-grained, uniform computations. In index construction, computing distances from a vector to the set of coarse centroids follows the same arithmetic pattern across all centroids; in query search, distance evaluations for candidate vectors inside the selected inverted lists are likewise uniform and mutually independent. These regular, element-wise operations admit data-parallel execution with minimal control divergence and predictable memory access patterns.

*Low synchronization and clear locality.* IVF relies on read-mostly shared state (centroids and inverted lists) during both phases. Assignments and per-query scoring can proceed with limited coordination, typically requiring only lightweight writes to per-list or per-query buffers. Moreover, the partitioning of the dataset into inverted lists provides locality: computations are confined to a small number of probed lists, which reduces cross-task interference and improves cache friendliness.

## 2.3 Problem Definition

The Approximate Nearest Neighbor Search (ANNS) problem can be formally defined as follows. Let $\mathcal{X} = \{x_1, x_2, \ldots, x_N\}$ denote a dataset of $N$ vectors in $\mathbb{R}^d$, and let $q \in \mathbb{R}^d$ represent a query vector. The goal of ANNS is to efficiently find a set of $k$ vectors $\mathcal{S}_k(q) \subseteq \mathcal{X}$ such that the elements of $\mathcal{S}_k(q)$ are the nearest neighbors of $q$ under a given distance metric, usually the Euclidean distance:

$$\mathcal{S}_k(q) = \underset{\substack{S \subseteq \mathcal{X} \\ |S|=k}}{\arg\min} \sum_{x_i \in S} \|q - x_i\|_2.$$

In exact $k$-NN search, the entire dataset $\mathcal{X}$ is scanned to compute $\|q - x_i\|_2$ for all $i \in [1, N]$. In contrast, ANNS algorithms aim to find an approximate result $\hat{\mathcal{S}}_k(q)$ such that $\hat{\mathcal{S}}_k(q)$ is close to $\mathcal{S}_k(q)$ but computed with significantly less time and memory cost:

$$\hat{\mathcal{S}}_k(q) \approx \mathcal{S}_k(q), \quad \text{with} \quad |\hat{\mathcal{S}}_k(q)| = k.$$

To evaluate the quality and efficiency of an ANNS system, two key performance metrics are commonly used: *Recall Rate* and *Queries Per Second (QPS)*.

*Recall Rate.* Recall measures the fraction of true nearest neighbors that are successfully retrieved by the approximate search. For a query $q$, the recall rate is defined as:

$$\text{Recall}(q) = \frac{|\hat{\mathcal{S}}_k(q) \cap \mathcal{S}_k(q)|}{k}.$$

The overall recall across a set of queries $Q$ is the average recall:

$$\text{Recall}_{avg} = \frac{1}{|Q|} \sum_{q \in Q} \text{Recall}(q).$$

A higher recall indicates better approximation quality, with $\text{Recall}_{avg} = 1.0$ corresponding to exact search.

*Queries Per Second (QPS).* QPS quantifies the throughput efficiency of the system, representing how many queries can be processed per second. Given a total number of queries $|Q|$ and total search time $T_{search}$ (in seconds), QPS is defined as:

$$\text{QPS} = \frac{|Q|}{T_{search}}.$$

QPS reflects the system's computational efficiency and responsiveness under parallel or large-scale workloads.

In this project, we aim to maximize both *Recall* and *QPS* while minimizing total computation time for index building and querying. These two metrics will jointly evaluate the trade-off between retrieval accuracy and performance scalability in the proposed parallel IVF implementation.

# 3 Proposed Approaches

This project aims to analyze, parallelize, and accelerate the IVF-Flat (Inverted File Flat) approximate nearest neighbor (ANN) search algorithm. The core of the IVF-Flat method involves partitioning the high-dimensional vector space into Voronoi cells, each represented by a centroid. During a search, only the cells closest to the query vector are visited, drastically reducing the computational workload.

The structure of the IVF-Flat algorithm is exceptionally well-suited for parallelization. Its two main phases—probe selection and list search—consist of numerous independent distance calculations. This presents clear opportunities for both **task-level parallelism** (distributing searches across threads) and **data-level parallelism** (using SIMD instructions for distance computations), which our proposed approaches will exploit.

Our plan is twofold: first, we will implement and evaluate CPU-based parallelization; second, we will explore an implementation on the GPU.

## 3.1 Baseline Configuration

To rigorously quantify the performance improvements of our parallel implementations, a baseline version will be established. This version will represent a straightforward, sequential implementation of the algorithm. It will be compiled as a single-threaded application and will not utilize any platform-specific optimizations, such as multi-threading libraries or SIMD intrinsics. All performance gains will be measured against this baseline.

## 3.2 CPU Parallelization Approaches

Our primary CPU optimization strategy is **data parallelism**, leveraging both thread-level parallelism with **OpenMP** and instruction-level parallelism with **SIMD (Single Instruction, Multiple Data)** intrinsics. The acceleration will target the two main computational bottlenecks of the search process.

*3.2.1 System Architecture (CPU).* The CPU architecture adopts a master–worker architecture orchestrated by the OpenMP runtime. The master thread dispatches tasks to a pool of worker threads that process data concurrently, as illustrated in Figure 1.

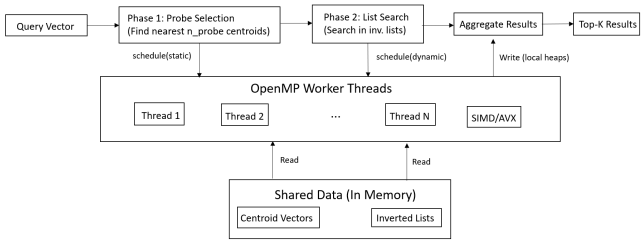


**Figure 1: CPU Parallelization Architecture.**

*3.2.2 Component Functions and Interaction.*

- **Master Thread:** Responsible for coordinating the search workflow. It initiates the OpenMP parallel region and, after all worker threads complete, aggregates their local candidate results into a final, globally sorted top-k list.
- **OpenMP Worker Threads:** These threads perform the actual computations.
  - *Phase 1 (Probe Selection):* The task of computing distances to the `nlist` centroids is partitioned among threads using a `#pragma omp for` with a *schedule(static)* clause, as this workload is uniform.
  - *Phase 2 (List Search):* After the `nprobe` nearest centroids are identified, the task of searching their corresponding inverted lists is dispatched using a *schedule(dynamic)* clause. This handles the load imbalance caused by lists of varying lengths. Each thread maintains a private local min-priority heap for its local top-k candidates.
- **SIMD (AVX2) Acceleration:** This is a key component for instruction-level parallelism. The L2 distance calculation will be implemented using AVX2 intrinsics from the `<immintrin.h>` header. By using `__m256` registers, we can process eight 32-bit floating-point numbers in a single instruction, significantly accelerating all distance computations in both phases.

## 3.3 GPU Parallelization with CUDA

We will investigate the feasibility of offloading the search process to a GPU using CUDA.

*3.3.1 System Architecture (GPU).* The proposed GPU architecture follows a standard host–device execution model, as illustrated in Figure 2.

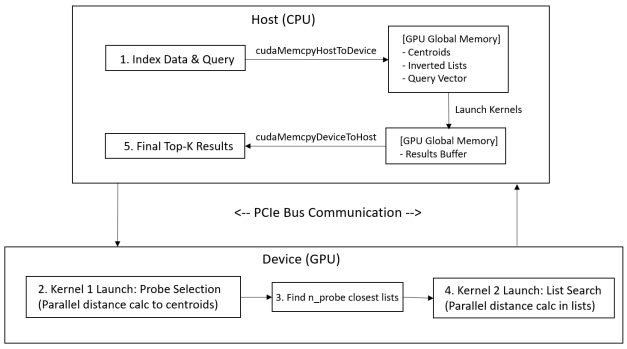


**Figure 2: GPU Parallelization Architecture.**

- **Host (CPU):** Manages data preparation, memory transfers between host and device, and CUDA kernel launches. During index initialization, all index structures (base vectors, centroids, inverted lists) are copied from host RAM to GPU global memory.
- **Device (GPU):** Acts as a massively parallel co-processor.
  - *Kernel 1 (Probe Selection):* A kernel will be launched where each thread computes the distance from the query vector to one centroid.
  - *Kernel 2 (List Search):* A second kernel will be designed to perform distance calculations for vectors within the selected `nprobe` lists. This phase is more challenging because the irregular memory access patterns may cause non-coalesced reads, thereby limiting overall throughput.

*3.3.2 GPU Performance Analysis.* A key part of this investigation will be to analyze the performance bottlenecks of the GPU implementation. If the performance is suboptimal, we will explore the underlying causes, which may include PCIe data transfer overhead, kernel launch latency, or poor memory access efficiency due to the irregular structure of the inverted lists.

# 4 Performance Analysis and Profiling

To identify performance bottlenecks and validate our optimizations, we will use a suite of standard profiling tools.

## 4.1 Profiling Tools

- **Perf (Linux):** For low-overhead analysis of performance counters to identify function-level hotspots.

- **Valgrind (Cachegrind):** For detailed simulation of the cache hierarchy (L1, L2, LLC) to precisely measure cache miss rates and analyze data locality.
- **Intel VTune Profiler:** For in-depth microarchitectural analysis of CPU pipeline utilization, memory bandwidth, and the efficiency of SIMD vectorization.
- **NVIDIA Nsight:** For profiling the CUDA implementation, analyzing kernel performance, memory access patterns, and identifying GPU-specific bottlenecks.

## 4.2 Key Performance Metrics

The evaluation of our parallelized ANN algorithm will be based on a comprehensive set of metrics that cover both application-level effectiveness and system-level performance.

- **Recall-QPS Trade-off Curve:** This is the primary evaluation method for ANN algorithms. We will generate a 2D plot showing Recall@K on one axis and Queries Per Second (QPS) on the other. By varying algorithm parameters (e.g., `nprobe`), we can trace a curve that visualizes the fundamental trade-off between search accuracy and speed.
- **Queries Per Second (QPS):** As the main indicator of search throughput, QPS measures how many queries the system can process per second. It is the reciprocal of the average query latency.
- **Recall@K:** This metric quantifies the accuracy of the approximate search. It is defined as the proportion of the true top-K nearest neighbors that are successfully found by the algorithm.
- **Index Build Time & Memory Usage:** We will measure the time required to construct the index from the base dataset and the total memory footprint of the final index structure. These metrics are crucial for assessing the practical viability of the algorithm.
- **System-Level Diagnostics (Cache Miss Rate, IPC):** To understand the reasons behind the performance outcomes, we will use profiling tools to measure low-level hardware metrics. Cache Miss Rate will reveal the effectiveness of our data layout and memory access patterns, while Instructions Per Cycle (IPC) will indicate how efficiently the CPU's execution units are being utilized, especially after applying SIMD optimizations.

# 5 Language Selection

**Programming languages.** We adopt **C++** for core data structures and system integration, **OpenMP** for CPU-side data/task parallelism, and **CUDA** for GPU acceleration of distance-heavy kernels. We use standard profiling tools (Perf, Intel VTune, NVIDIA Nsight) to guide optimization.

## 5.1 C++ and SIMD for Core Performance

**Roles.** Implement IVF-Flat index structures (centroids, inverted lists), I/O and memory management, and host-side orchestration.
**Performance.** Use **SIMD (AVX2)** intrinsics for L2/inner-product distance on CPU (fallback and re-rank paths); emphasize cache-friendly layouts and aligned/batched accesses.
**Maintainability.** CMake-based builds; hot-spot identification with Perf/VTune; unit tests and microbenchmarks for distance, list scan, and top-$k$.

## 5.2 CUDA for Massively Parallel Acceleration

**Core kernels**

- **Probe selection**: parallel distances from a query to all centroids.
- **List search**: parallel distances within the probed inverted lists and partial top-$k$ selection.

**Optimization strategy.** Overlap H2D/D2H transfers with compute via `cudaMemcpyAsync` and multiple streams; coalesce global loads; stage centroids and list tiles in shared/L2 when beneficial; use warp-level reductions to cut synchronization.

## 5.3 OpenMP for Host-Side Parallelism

**Division of labor.**

- Batch multiple queries with `#pragma omp parallel for` on host pre/post-processing paths.
- Parallelize index construction (vector-to-centroid assignment) with static scheduling; use dynamic scheduling for list scans when list lengths are skewed.

# 6 Related Work

We first sketch three ANNS paradigms—LSH (hashing), HNSW (graph), and IVF (clustering)—and their practical trade-offs in recall, latency, memory, and build cost. We then trace IVF's evolution to IVF-PQ (IVFADC) and related quantization/variant designs (e.g., OPQ, SQ/INT8, RQ, IMI, re-ranking), motivating IVF-based choices for large-scale, cost-efficient search.

## 6.1 Approximate Nearest Neighbor Search

Modern ANNS techniques largely fall into three families: **hashing-based (LSH), graph-based (HNSW), and clustering-based (IVF)**. LSH projects high-dimensional vectors into hash buckets (e.g., random hyperplanes) so queries only compare within a few buckets; it has clean theory, fast indexing, and scales well in distributed settings, but high recall often requires many tables, increasing memory and probe cost. HNSW builds a navigable small-world graph with hierarchical layers and greedy neighbor expansion; in practice, for a given latency budget it often achieves the highest recall, supports online insertions, and has mature CPU/GPU implementations, at the expense of larger index memory and longer build time, with performance sensitive to parameters (e.g., M, efConstruction, efSearch). IVF (inverted file) first performs coarse clustering (e.g., k-means) into nlist cells and, at query time, probes only nprobe cells before re-ranking, which aggressively prunes comparisons. **IVF's key strengths are memory efficiency, a simple engineering path, and excellent compatibility with quantization and GPU parallelism**, yielding a stable latency–throughput–cost trade-off.

## 6.2 Evolution of IVF

IVF follows a coarse-to-fine paradigm. The canonical extension is IVF-PQ : IVF narrows the candidate lists, then Product Quantization (PQ) encodes vectors into compact subspace codes so queries use

lookup-table distance estimation—achieving order-of-magnitude memory compression with high throughput. Subsequent refinements reduce quantization error or improve scan efficiency: OPQ learns an orthogonal rotation before PQ; scalar/8-bit quantization (e.g., SQ/IVF-SQ8) simplifies codebooks and speeds arithmetic; residual/multi-stage quantization (RQ/MQ) incrementally encodes residuals; additive/compound quantization (AQ/CQ) increases expressiveness via codeword combinations; and re-ranking with exact vectors or higher-precision codes improves final recall. Structural variants raise coarse resolution (e.g., IMI, inverted multi-index) or combine IVF with graph navigators (e.g., HNSW as the coarse quantizer). On modern GPUs, LUT reuse, half-precision paths, tensor-core kernels, and SIMD "fast scan" further push IVF-PQ toward high compression, low latency, and large-scale throughput, with nlist/nprobe, codebook size, bits-per-subvector, and re-ranking depth jointly governing the recall–latency–memory frontier.

## 7 Statement of Expected Results

Our objective is to *increase query throughput (QPS) without sacrificing accuracy* on IVF-Flat. We seek Pareto improvements on the Recall–QPS frontier under fixed memory budgets.

### 7.1 Primary Objectives

- **Accuracy preservation.** Maintain recall parity with the baseline at matched operating points (e.g., Recall@10), so users perceive the same result quality while benefiting from parallel execution.
- **Throughput-oriented parallelism.** Exploit task- and data-level parallelism on CPU (OpenMP+SIMD) and GPU (CUDA), while overlapping data movement with computation to sustain higher throughput across both single-query and batched regimes—without changing algorithmic semantics. Target speedups are approximately **2.5–4× on CPU** and **5–8× on GPU** compared to the single-threaded baseline.
- **Latency stability and tail control.** Keep median and tail latencies stable under load via dynamic scheduling, cache-friendly layouts, balanced list scanning, and partial top-$k$ early pruning; batched queries gain smoother end-to-end latency through transfer/compute overlap.

### 7.2 Secondary Targets

- **Memory efficiency**. Report IVF-Flat index footprint (total size, bytes/vector).
- **Build-time improvement**. Parallelize centroid assignment and list construction for $\sim 2\times$–$4\times$ faster builds on multi-core CPUs.
- **Scalability**. Near-linear QPS scaling with CPU threads until LLC/NUMA limits; on GPU, sustained scaling with resident queries/streams until bandwidth-bound.

### 7.3 Datasets

We evaluate on two standard public benchmarks to ensure comparability and robustness:

- **SIFT1M**: 1M vectors, 128-D; classic small/medium-scale ANN benchmark.
- **GIST1M**: 1M vectors, 960-D; higher dimensionality to stress distance kernels and memory bandwidth.

We will report Recall@$k$ ($k \in \{1, 10, 100\}$), QPS, p50/p95 latency, bytes/vector, and index build time on both datasets.

### 7.4 Evaluation Protocol

- **Baselines**. (i) Single-thread IVF-Flat (float, no SIMD); (ii) Multi-thread CPU without SIMD (ablations); (iii) Optional FAISS IVF-Flat sanity check with comparable settings.
- **Operating points**. Sweep `nlist` and `nprobe` to trace Recall–QPS curves; evaluate batches $\{1, 16, 64, 256\}$ to expose latency/throughput regimes and PCIe overlap on GPU.
- **Hardware.** Document CPU model/cores/SIMD width/NUMA; GPU model/SMs/memory bandwidth; driver/toolchain versions.

## 8 Timetable

The project will span seven weeks, culminating in a final presentation and report in Week 7. The development schedule is summarized in Table 1.

**Table 1: Project timeline and milestones.**

| Week | Milestone / Task |
|---|---|
| 1 | Implement single-threaded IVF-Flat baseline in C++. |
| 2 | Add OpenMP task parallelism for centroid assignment and list search. |
| 3 | Integrate AVX2 SIMD optimization and profile with `perf`/VTune. |
| 4 | Develop CUDA kernels for probe selection and list search. |
| 5 | Implement CPU–GPU integration and multi-stream pipelining. |
| 6 | Tune parameters and collect Recall–QPS performance curves. |
| 7 | Final presentation and report submission. |

This schedule ensures that the baseline, CPU, and GPU implementations are completed within seven weeks, allowing sufficient time for evaluation and presentation preparation.

## References


[1] Malkov, Y. A., and Yashunin, D. A. (2020). Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 42(4), 824–836.

[2] Jégou, H., Douze, M., and Schmid, C. (2011). Product quantization for nearest neighbor search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(1), 117–128.

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

[4] Guo, R., et al. (2020). Accelerating large-scale inference with anisotropic vector quantization. In *Proceedings of the International Conference on Machine Learning (ICML)*.