南開大學
Nankai University

计 算 机 学 院

并行程序设计实验报告

ANN 期末研究报告

张奥喆

年级：2023 级

专业：计算机科学与技术

指导教师：王刚

2025 年 7 月 6 日

摘要

近年来，大规模向量检索在推荐系统、搜索引擎及多模态理解等场景中日益重要，但高维空间的“维度灾难”使精确最近邻搜索在时延与吞吐上难以满足实用需求。为此，本文系统研究了近似最近邻搜索（ANNS）中具有代表性的几类算法——乘积量化（PQ）、倒排文件索引（IVF）与分层图索引（HNSW）以及他们的结合策略——在多层次并行体系结构上的性能优化策略。

本文首先在 ARM 与 x86 CPU 上挖掘指令级并行（SIMD）、线程级并行（OpenMP/pthread)和进程级并行(MPI)的潜力;并进一步结合算法-硬件协同的 FastScan、OpenMP+SIMD、MPI+SIMD 以及 IVF-HNSW 等混合策略；随后在 NVIDIA GPU 上，借助矩阵化改写与批处理技术，对核心距离计算进行大规模并行。所有实验统一采用 DEEP100K 数据集，并借助 perf 与 VTune 进行瓶颈剖析。

实验结果表明：在 CPU 端，最优 OpenMP+SIMD 方案可为 IVF 带来最高 11.5× 的加速；GPU 端对暴力搜索的批量矩阵化改写实现了最高 1648× 的吞吐提升；FastScan 通过寄存器内查表将 PQ 查询延迟降低 4 倍；IVF-HNSW、分区 HNSW 等策略在保证 0.90 以上召回的同时，提供最高 1.75x 的加速比。本文也对比了专业工具库 Faiss-GPU，分析其性能出众的原因。综合分析指出，算法-硬件协同设计、并行开销权衡与负载均衡是设计 ANNS 并行化的关键因素。

关键字：ANN；并行计算；SIMD；OpenMP；MPI；GPU；IVF；PQ；HNSW

目录

一、 概述

(一) 问题背景与挑战

最近邻搜索 (Nearest Neighbor Search, NNS) 旨在高维空间中，为给定的查询向量 (query) 找到距离其最近的一个或多个数据点。随着深度学习技术的发展，图像、文本、音频等非结构化数据可以被有效地转换为高维向量嵌入 (embedding)，并且在嵌入空间中保持语义相似性，即内容相似的原始数据其对应的向量在空间中的距离也相近。这使得 NNS 技术成为推荐系统、搜索引擎、人脸识别、大规模语言模型等众多应用的核心基石。

然而，NNS 面临着严峻的“维度灾难”问题。当数据维度增高时，数据点之间的距离差异变得不再显著，空间变得稀疏，这使得性能急剧下降。最直接的暴力搜索方法，即计算查询向量与数据库中所有向量的距离，其计算复杂度为 $O(N \cdot d)$（其中 N 为数据库大小，d 为向量维度）。在当今动辄千万甚至数十亿规模的向量数据库和数百维的向量面前，暴力搜索的计算开销是无法接受的。

为了解决这一挑战，近似最近邻搜索 (Approximate Nearest Neighbor Search, ANNS) 应运而生。ANNS 的核心思想是放弃寻找绝对精确的最近邻，转而寻求**在可接受的精度损失范围内，以极高的效率找到足够接近的邻居**。其性能通常由两个核心指标来衡量：

- **查询延迟 (Latency)**: 处理单个查询所需的平均时间，单位通常是微秒 (µs)。
- **QPS:** 每秒查询数，鉴于 GPU 的批处理工作机制，性能评估的重点应从查询延迟转向更能真实反映系统吞吐量优势的 QPS，为了统一性能度量，这个指标也可以通过查询延迟转换过来。
- **召回率 (Recall@k)**: 算法返回的 k 个结果中，与真实最近的 k 个结果的交集大小与 k 的比值。公式为: $Recall@k = \frac{|KNN(q) \cap ANNS(q)|}{k}$。

ANNS 算法的设计目标便是在保证高召回率（如 90% 以上）的前提下，尽可能地降低查询延迟。

(二) 研究思路与结构

面对 ANNS 的性能挑战，利用现代计算硬件的并行能力是提升其效率的关键路径。本报告旨在系统性地探索和比较主流 ANNS 算法在不同并行计算架构下的优化策略、性能表现与瓶颈所在。

本文将围绕具有代表性的最近邻算法展开，包括了 NNS 和 ANNS:

1. **暴力搜索**: 作为精确搜索的基准，用于展示并行化的巨大潜力。
2. **乘积量化 (PQ)**: 一种通过向量量化加速距离计算的代表性方法。
3. **倒排文件索引 (IVF)**: 一种通过划分搜索空间来减少候选集大小的常用方法。
4. **图索引 (HNSW)**: 当前业界性能领先的、基于图结构的搜索算法。

此外，我也尝试对上述的方法进行结合使用，探索更好的优化思路。

(三) 并行技术与实验环境概述

1. 并行计算模型简介

实验涉及的并行技术涵盖了从指令级到节点级的多个层次:

- **SIMD (Single Instruction, Multiple Data)**: 数据级并行，允许单条指令同时对多个数据元素执行相同的操作，非常适合加速向量、矩阵等规整数据的计算。本实验主要使用 ARM 平台的 NEON 和 x86 平台的 AVX 指令集。
- **共享内存并行**: 线程级并行, 在单个计算节点内, 多个线程共享同一地址空间, 通过 pthread 或 OpenMP 等 API 协调执行。其优势在于线程间通信开销低，但受限于单节点的内存带宽和核心数。
- **分布式内存并行**: 进程级并行，在多个计算节点组成的集群上，每个节点（进程）拥有独立的内存空间，通过消息传递接口（Message Passing Interface, MPI）进行通信和同步。其优势在于可扩展性强，能利用大规模集群资源。
- **大规模异构并行**: 利用 GPU 等协处理器进行计算。GPU 拥有数千个计算核心，尤其擅长处理高度并行的计算密集型任务。本实验使用 NVIDIA GPU 和 CUDA 编程模型。

2. 实验平台与数据集

实验中的所有实验均在统一的硬件平台和数据集上进行，以保证结果的可比性。

- **硬件环境**: 实验所用服务器 CPU 为 ARM 架构，鲲鹏服务器。部分剖析性能的对比实验在 x86 平台的 CPU 上进行。GPU 实验使用 NVIDIA RTX3090。
- **软件环境**: 操作系统为 OpenEuler，编译器为 g++，统一使用 O2 优化。
- **数据集**: 实验统一采用公开的 DEEP100K 数据集。该数据集包含 10 万个 96 维的浮点向量，是从 DEEP1B 数据集中抽样而来，常用于 ANNS 算法的基准测试。
- **分析工具**: 使用 Linux 下的 perf 工具和 Intel VTune Profiler 对程序的性能瓶颈（如 CPU 周期、指令数、缓存命中率、内存带宽等）进行深度剖析。

二、 暴力搜索的并行优化

(一) 算法原理与串行瓶颈

暴力搜索，或称为穷举搜索、Flat Search，是最基础也是最精确的最近邻搜索方法。其原理极其简单：对于给定的一个查询向量 q，遍历数据库中全部 N 个基准向量 $\{x_1, x_2, ..., x_N\}$，逐一计算 q 与每个 x_i 之间的距离 $\delta(q, x_i)$，并始终维护一个大小为 k 的候选列表（通常是最大堆），用以保存当前找到的距离最近的 k 个向量。遍历结束后，候选列表中的向量即为最终的 Top-K 最近邻。

该算法的瓶颈显而易见：其时间复杂度为 $O(N \cdot d)$，其中 N 是数据库大小，d 是向量维度。每一次查询都需要进行 N 次距离计算，而每次距离计算又包含 d 次乘加操作。当 N 和 d 很大时，总计算量极为庞大，导致查询延迟极高，无法满足实时应用的需求。

(二) CPU 并行加速

1. SIMD

在 CPU 端，最直接的优化方式是利用 SIMD 指令集加速向量间的距离计算。以本实验使用的内积距离（$1 - \sum q_i \cdot x_i$）为例，核心的计算瓶颈在于 d 维向量的点积运算。通过使用 ARM NEON 指令集，我们可以将原本需要 d 次循环的标量乘加操作，替换为少量的向量指令。例如，

使用 128 位的 NEON 寄存器可以一次性装载 4 个 32 位浮点数，通过 vmlaq_f32（向量乘加）等指令，单条指令即可完成 4 个维度的乘加运算，理论上可带来 4 倍的加速。为了进一步挖掘并行潜力，可以同时使用多个寄存器，一次处理 8 个、16 个甚至更多的浮点数。

Listing 1: SIMD 向量乘法

```
float32x4_t sum1 = vdupq_n_f32(0); // 第一个累加寄存器
float32x4_t sum2 = vdupq_n_f32(0); // 第二个累加寄存器
// 每次处理8个浮点数(使用2个NEON寄存器)
for (size_t i = 0; i + 7 < vecdim; i += 8)
{
    // 加载第一个向量的4个float
    float32x4_t v1_1 = vld1q_f32(b1 + i);
    float32x4_t v1_2 = vld1q_f32(b1 + i + 4);
    // 加载第二个向量的4个float
    float32x4_t v2_1 = vld1q_f32(b2 + i);
    float32x4_t v2_2 = vld1q_f32(b2 + i + 4);
    // 执行乘法并累加到sum寄存器
    sum1 = vmlaq_f32(sum1, v1_1, v2_1);
    sum2 = vmlaq_f32(sum2, v1_2, v2_2);
}
```

2. OpenMP+SIMD

由于之前 OpenMP 并没有优化暴力搜索，因此本次实验我使用 OpenMP+SIMD 进一步优化暴力搜索。SIMD 指令集能够显著加速单个向量的距离计算，但它本质上仍是在单核内进行的指令级并行。对于现代多核 CPU 而言，只使用 SIMD 而忽略了多核的并行潜力是一种巨大的资源浪费。

OpenMP 是一个用于共享内存并行编程的 API，它能够简单地通过编译器指令（pragma）将串行代码中计算密集型的循环（如暴力搜索中的外层循环）并行化，自动地将循环任务分配到多个 CPU 核心上执行 [10]。

1. **线程级并行**: 使用 OpenMP 的 `#pragma omp parallel for` 指令，将遍历 N 个基准向量的外层循环分解为多个独立的任务块。例如，如果有 N 个向量和 T 个线程，OpenMP 会将 N 个向量大致均分，每个线程负责计算其中约 N/T 个向量与查询向量 q 的距离。这些线程在不同的 CPU 核心上同时运行，实现了任务的并行处理。

2. **指令级并行**: 在每个独立的线程内部，当计算单个基准向量 x_i 与查询向量 q 的距离时，继续沿用前述的 SIMD 优化方法。也就是说，每个线程在执行自己的那一小部分距离计算任务时，其内部的点积运算依然由 ARM NEON 等 SIMD 指令集来加速。

并行搜索与局部 Top-K 计算阶段将搜索任务分发给多个线程。每个线程独立计算一部分向量的距离，并维护一个自己的“Top-K”结果队列。

Listing 2: 并行搜索与局部 Top-K 计算

```
// 最终返回的全局优先队列
std::priority_queue<std::pair<float, uint32_t>> final_queue;
#pragma omp parallel
{
```

```
    // 每个线程维护一个本地的优先队列，用于存储局部的Top-K结果
    std::priority_queue<std::pair<float, uint32_t>> local_queue;

    // 使用OpenMP将循环并行化，静态调度有助于均匀分配任务
    #pragma omp for schedule(static)
    for (int i = 0; i < base_number; ++i) {
        // 使用SIMD指令集加速单个向量的距离计算
        float dis = InnerProductSIMDNeon(base + i * vecdim, query, vecdim);
        ...
    }
}
```

临界区与全局结果合并阶段是在所有线程完成各自的搜索任务后，需要将它们找到的局部 Top-K 结果汇总，以得到最终的全局 Top-K。

Listing 3: 临界区与全局结果合并

```
    // 使用临界区，确保同一时间只有一个线程能执行合并操作
    #pragma omp critical
    {
        while (!local_queue.empty()) {
            // 将局部队列的元素逐个尝试合并到全局队列中
            auto& item = local_queue.top();
            if (final_queue.size() < k) {
                final_queue.push(item);
            } else if (item.first < final_queue.top().first) {
                final_queue.pop();
                final_queue.push(item);
            }
            local_queue.pop();
        }
    }
} // #pragma omp parallel 区域结束
```

3. 实验验证

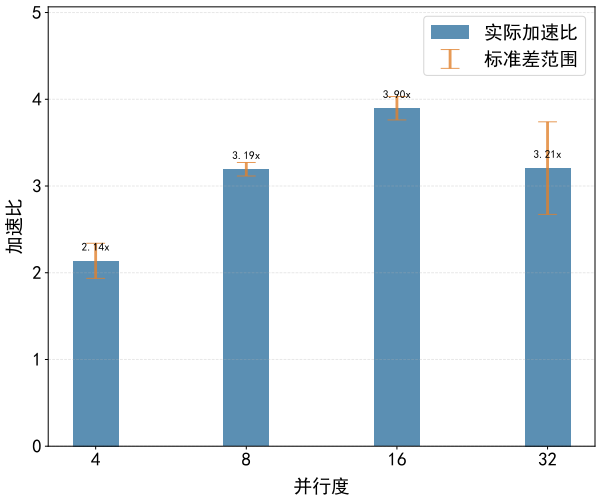
实验结果如图1a所示。

- 从一次算 4 个浮点数到一次算 16 个浮点数，尽管查询延迟在降低，加速比在上升，但是实测加速比远达不到理论的加速比，使用更多的寄存器并不会带来线性的效率提升。
- **性能拐点出现在并行度 16，可以得到接近 4 倍的加速比**：并行度 8 和 16 的标准差较小，表明这些设置下系统性能更稳定；但是此时已经出现收益递减了，查询效率的提升不如之前并行度 4 至 8 的提升；当并行度到 32 时，发现性能竟然下降了，并且性能波动较大，这反映了**一味地提高寄存器的数量并不会一直带来性能的提升**，甚至适得其反。

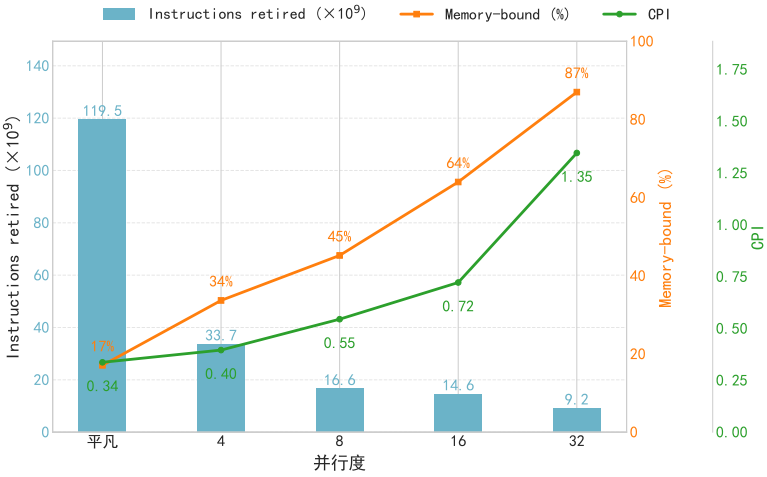
为了分析 SIMD 优化相比于平凡算法的优势，同时为了找到并行度增长的瓶颈，在服务器上使用 Perf 工具进行剖析，重点关注 Instructions Retired、CPI 和 Memory Bound。得到的性能数据如图1b所示。

- **SIMD 大幅减少了 Instructions Retired，这是并行加速的主要来源**。从平凡算法到一次算 4 个浮点数，指令完成数的确减少了 $\frac{3}{4}$，相比于并行度 4，并行度 8 在此基础上减少 50%，目前的收益的确是线性的，但是从并行度 8 到 16 再到 32，这样的收益就边际递减了，这一部分指令数减少的比例并不与线程数成正比，表明存在并行开销（比如高并发下的额外管理）。
- CPI 随着并行度的上升而增高。CPI 值越低表示处理器效率越高，当 CPI 超过 1 时，意味着平均每条指令需要多于一个时钟周期执行，处理器效率显著下降。注意到，从 16 线程到 32 线程，CPI 陡然增加了 87.5%，也超过了 1（1.35），这是并行度 32 状态下，性能下降的直接证明之一。
- **Memory-bound 逐渐升高**。从平凡算法的 17% 一直逐步上升（基本是线性的）到 34 %、45 %、64 %、87 % , 瓶颈逐渐转变为内存带宽的限制。在并行度 32 时, 87 % 的 Memory-bound 基本宣告带宽被榨干，这意味着处理器大部分时间都在等待内存操作完成。

综合以上的分析，可以得出结论，总体耗时随并行度呈“先降后升”的倒 U 形：SIMD 在 4 至 8 并行度阶段带来近乎线性的收益——指令完成数锐减，而 CPI 和 Memory-bound 仍处于低位，因此总时间快速下降；当并行度扩展到 16 线程，**指令数的递减进入边际报酬递减区，而同步、调度等并行开销开始显现**，CPI 抬升到 0.72、Memory-bound 跃至 64 %，耗时降幅明显变缓；继续扩展到 32 线程时，虽然指令数再降 37 %，但 **87 % 的 Memory-bound 使内存带宽基本饱和**，CPI 飙升到 1.35——CPU 大部分周期都在等待数据——并行开销反客为主，致使总耗时反而回弹。也就是说，性能拐点出现在 16 线程：此前耗时主要受计算量驱动，此后则主要受内存吞吐与线程管理开销主导。



(a) SIMD 加速比



(b) vtune 分析 SIMD 并行

图 1: SIMD 加速比与性能分析

OpenMP+SIMD 实测**加速比在 15x 至 20x**，使用多线程的计算任务获得了显著的性能提升。OpenMP 的更详细分析会在 IVF 部分体现。

(三) GPU 端大规模并行 (CUDA)

1. 实现原理

为了利用 GPU 进行加速，必须摒弃“逐个查询、逐个比较”的串行思维，转向“**批量处理、并行计算**”的模式，也就是将距离计算重构为矩阵乘法 [11] [6]。

假设我们有 m 个查询向量组成的查询批次，每个查询向量维度为 d。我们可以将其组织成一个 $m \times d$ 的矩阵 Q。整个数据库的 N 个向量可以组织成一个 $N \times d$ 的矩阵 B。我们希望计算出每个查询向量与每个数据库向量的内积，这恰好可以由矩阵乘法 $D = Q^T \times B$ 得到。结果将是一个 $m \times N$ 的矩阵 D_{ist}，其中每个元素 $D_{\text{ist}}[i][j]$ 就等于第 i 个查询向量与第 j 个数据库向量的内积。

通过这种方式，原本 $m \times N$ 次独立的内积计算（每次计算包含 d 次乘加）被统一成了一次大规模的矩阵乘法。这正是 GPU 最擅长的计算类型，能够将成千上万的计算核心全部调动起来，实现并行加速。

计算前，需在 GPU 设备为数据库向量 `d_base`、查询批次 `d_queries`、距离矩阵 `d_distances` 及结果索引 `d_indices` 分配显存。其中，数据库向量仅需从主机（CPU）一次性传输 [9]。

Listing 4: GPU 显存分配与数据初始化

```
float *d_base, *d_queries, *d_distances;
int *d_indices;
cublasHandle_t handle;

// 为各矩阵分配GPU显存
cudaMalloc(&d_base, base_number * vecdim * sizeof(float));
cudaMalloc(&d_queries, BATCH_SIZE * vecdim * sizeof(float));
// ... (为d_distances, d_indices分配)

// 将数据库一次性从CPU拷贝到GPU
cudaMemcpy(d_base, base, base_number * vecdim * sizeof(float),
    cudaMemcpyHostToDevice);
cublasCreate(&handle);
```

此阶段在循环内批处理查询。首先调用 cublasSgemm [8]，通过矩阵乘法高效计算内积距离矩阵 `d_distances`。接着，启动自定义 CUDA 核函数 find_topk_heap，利用高速共享内存为每个查询并行地从距离矩阵中筛选出 Top-k 结果。

Listing 5: 核心计算：矩阵乘法与 Top-K 筛选

```
// 1. cuBLAS矩阵乘法: D = Q^T * B
const float alpha = 1.0f, beta = 0.0f;
cublasSgemm(handle, CUBLAS_OP_T, CUBLAS_OP_N,
            current_batch_size, base_number, vecdim,
            &alpha, d_queries, vecdim, d_base, vecdim,
            &beta, d_distances, current_batch_size);

// 2. 自定义核函数并行筛选Top-K
size_t shared_mem_size = threads * k * (sizeof(float) + sizeof(int));
find_topk_heap<<<blocks, threads, shared_mem_size>>>(
    d_distances, d_indices, base_number, current_batch_size, k);
```

```
cudaDeviceSynchronize();
```

最后，将 GPU 上计算完成的 Top-k 结果索引 `d_indices` 拷贝回 CPU 内存，以供后续使用。

Listing 6: 将结果从 GPU 拷贝回 CPU

```
cudaMemcpy(h_batch_indices, d_indices,
           current_batch_size * k * sizeof(int),
           cudaMemcpyDeviceToHost);
```

2. 实验测试

实验对比了 GPU 并行方案与单线程 CPU 串行基准的性能，并通过调整批处理大小 (Batch Size) 来观察 GPU 的吞吐率变化。

表 1: GPU 暴力搜索性能对比 (Recall=1.0)

算法方案	QPS (queries/sec)	加速比
CPU Flat (基准)	105.0 ± 0.2	1.0x
GPU (Batch Size=500)	$13,809.0 \pm 115.8$	130.3x
GPU (Batch Size=1000)	$27,861.1 \pm 461.3$	261.6x
GPU (Batch Size=2000)	$50,568.2 \pm 211.7$	483.9x
GPU (Batch Size=5000)	$107,893.8 \pm 120.8$	1027.5x
GPU (Batch Size=10000)	$173,140.5 \pm 2177.3$	**1648.6x**

如表1所示，GPU 并行方案取得了惊人的性能提升。随着 Batch Size 的增大，GPU 的计算资源被更充分地利用，加速比几乎呈线性增长。当 Batch Size 达到 10000 时，相较于 CPU 基准实现了高达 **1648.6 倍**的加速。注意由于数据规模较小，因此还没有出现性能拐点，如果面对更大规模的数据，需要找到较优的 Batch Size。

三、 乘积量化 (PQ) 的并行优化

(一) 算法原理

乘积量化 (PQ) 是一种旨在通过向量量化技术，以极低的内存占用和计算开销来近似计算向量间距离的方法。其核心思想是将一个高维向量空间分解为多个低维子空间的笛卡尔积 [5]。

1. 索引构建

1. **向量分解:** 将所有训练向量沿维度轴均匀切分为 M 段。
2. **子码本训练:** 对第 $m \in \{1, \ldots, M\}$ 个子空间，将其对应的所有训练数据子向量 $\{\mathbf{x}^{(m)}\}$ 作为输入，运行 K-means 算法，从而得到该子空间的码本 $\mathcal{C}^{(m)}$。

2. 查询流程

在查询阶段，目标是近似计算查询向量 $\mathbf{q}$ 与数据库中某条编码向量之间的距离。

1. **查询向量分解:** 与索引构建阶段保持一致，将查询向量 $\mathbf{q}$ 分解为 M 个子向量 $\{\mathbf{q}^{(m)}\}_{m=1}^{M}$。

2. **构建距离查找表 (Look-Up Table, LUT):** 对于每个子空间 m, 预先计算查询子向量 $\mathbf{q}^{(m)}$ 到该空间所有 K 个码字 $\mathbf{c}_k^{(m)}$ 的平方欧氏距离，构建一个查找表。

$$d_k^{(m)} = \left\|\mathbf{q}^{(m)} - \mathbf{c}_k^{(m)}\right\|_2^2, \quad \forall k \in \{1, \ldots, K\}$$

此步骤为每个查询生成 $M \times K$ 个距离值。

3. **扫描与近似距离累加:** 遍历数据库中的每条编码向量 $\mathbf{q}(\mathbf{x}) = \left[i^{(1)}, \ldots, i^{(M)}\right]$。通过查询预先计算好的 LUT，将各子空间的距离加和，得到近似的全局距离 $\widetilde{d}(\mathbf{q}, \mathbf{x})$。

$$\|\mathbf{q} - \mathbf{x}\|_2^2 = \sum_{m=1}^{M} \left\|\mathbf{q}^{(m)} - \mathbf{x}^{(m)}\right\|_2^2 \approx \sum_{m=1}^{M} \left\|\mathbf{q}^{(m)} - \mathbf{c}_{i^{(m)}}^{(m)}\right\|_2^2 = \sum_{m=1}^{M} d_{i^{(m)}}^{(m)} =: \widetilde{d}(\mathbf{q}, \mathbf{x})$$

4. **候选集排序与精确重排 (Rerank):** 使用一个小顶堆实时维护当前距离最近的 Top-L 个候选结果（L 通常大于最终需要的 k 值)。在获得这 L 个候选编码后，提取它们对应的原始高维向量，并与查询向量 $\mathbf{q}$ 计算精确距离，最后对这 L 个候选项进行重新排序，以提升最终的召回率。

PQ 通过将高维空间中的昂贵距离计算，转化为 M 次简单的查表操作，极大地降低了计算复杂度。但其瓶颈在于，仍然需要线性扫描整个数据库的编码。

(二) 核心挑战与并行优化策略

传统 PQ 的查询流程中，对 LUT 的访问是瓶颈所在。由于数据库向量的编码是随机的，导致访问 LUT 时的内存地址是分散的、不连续的，这造成了严重的缓存不命中，难以利用 CPU 的 SIMD 指令进行有效加速。

1. 跨向量并行化

一次性计算 W 个向量与查询向量 q 的距离 [13]。该策略的核心步骤是在处理第 k 个子空间时:

1. **加载 W 个码字:** 高效加载 W 个向量的第 k 个码字 $o_k^1, \ldots, o_k^W$ 。
2. **SIMD 聚集:** 使用一条 SIMD gather 指令，根据这 W 个码字索引，并行地从 LUT_k 中查找并加载 W 个对应的预计算距离值 $[LUT_{k,o_k^1}, \ldots, LUT_{k,o_k^W}]$。
3. **SIMD 加法:** 使用一条 SIMD 加法指令，将加载的 W 个距离值分别累加到 W 个向量对应的累加器中。

通过迭代所有子空间 $k = 1 \ldots m$，最终得到 W 个向量的 PQ 距离。这种方法将 gather 指令的开销**摊销**到 W 个向量上，并充分利用了 SIMD 的并行算术运算能力，从而实现加速。

局限： 通过查阅资料发现，**ARM 的 NEON 并不支持原生的 gather 指令** [2]，我的关键代码中的 gather 实现实际上是手动执行“查找 + 构建”这个过程，这大大降低了 SIMD 的优化效果。此外，我在实验中也编写了 AVX 的跨向量 SIMD 优化版本（X86 平台)，使用了原生的 AVX2 的 gather 指令，可以预期原生的 gather 指令效果应该好于手动实现的 gather。

2. OpenMP

在这一部分我也实现了 OpenMP 优化的 PQ，将每个步骤中的数据并行任务分配给多个 CPU 核心, 将查询流程中数据独立的、计算密集的 `for` 循环, 通过 `#pragma omp parallel for` 指令分配到多核并行执行。并行化 M 个独立子空间的距离表计算任务。并行化对海量数据库的扫描，以计算 PQ 近似距离。并行化对候选集中向量的精确距离计算。

3. FastScan

我复现了业界先进的 **PQ-FastScan** [1] 技术。它通过算法与数据布局的协同设计，巧妙地将随机访存问题转化为寄存器内部的高效操作。其核心创新点有两个：

1. **数据布局重排 (Block-of-16)**: 在索引构建阶段，不再按向量连续存储编码，而是将数据重新组织。以 16 个向量为一个块 (block)。这样，在查询时，一次内存读取就可以顺序地载入 16 个不同向量的同一个子空间的编码，将随机读优化为了顺序读。

2. **寄存器内查表**: FastScan 使用 4-bit 量化（$K = 16$）[4]。对于一个子空间，查询向量到 16 个码字的距离可以被量化后存入一个 16 字节的 u8 数组中，这个数组恰好可以被完整地加载到一个 128 位的 NEON 寄存器中作为“查找表”。然后，利用 ARM NEON 提供的 `vqtbl1q_u8`（向量查表）指令，以从内存中加载的 16 个 4-bit 编码为索引，直接在这个存有 LUT 的寄存器中并行地完成 16 次查表操作。

Listing 7: Fastscan 关键优化

```
/* ---- 布局转换：把 codes[i][m] 转成 block-of-16 ---- */
for(size_t i=0;i<N;++i){
    size_t blk=i/16,pos=i%16;
    for(int m=0;m<M;++m)
        codes_fs[ blk*M*16 + m*16 + pos ] = codes[i][m];
}
/* ---- LUT 量化 ---- */
float maxv = *std::max_element(dist.begin(), dist.end());
float alpha = maxv/255.f;
for(int t=0;t<M*K;++t) LUT_q8[t] = uint8_t(dist[t]/alpha + 0.5f);
/* ---- FastScan 核心：一次处理 16 向量 ---- */
uint16x8_t acc0=vdupq_n_u16(0), acc1=acc0;
for(int m=0;m<M;++m){
    uint8x16_t lut  = vld1q_u8(LUT_q8 + m*16);          // 16×u8
    uint8x16_t code = vld1q_u8(blk_ptr + m*16);         // 16×u8
    uint8x16_t d    = vqtbl1q_u8(lut, code);            // 查表
    acc0 = vaddq_u16(acc0, vmovl_u8(vget_low_u8 (d)));  // 0..7
    acc1 = vaddq_u16(acc1, vmovl_u8(vget_high_u8(d)));  // 8..15
}
vst1q_u16(acc, acc0);   vst1q_u16(acc+8, acc1);     // 写回 16 距离
```

通过这两个设计，PQ-FastScan 彻底消除了查询时对主存中 LUT 的随机访问，连 cache 都不需要访问了，将核心计算限制在了 CPU 寄存器内部，从而实现了极致的性能。

(三) 实验验证与性能分析

1. 查询延迟 & 加速比

实验对比了平凡 PQ、尝试使用跨向量 SIMD 优化的 PQ，以及 PQ-FastScan 的性能。为了保证公平，通过调整 rerank 参数使得所有方法的召回率均在 0.90 左右。

如图2所示，跨向量并行化有一定效果，加速比为 1.18x，而 PQ-FastScan 的性能优势是压倒性的。相较于平凡 PQ，它实现了超过 **4 倍**的加速，基本符合论文中的结论 [1]。

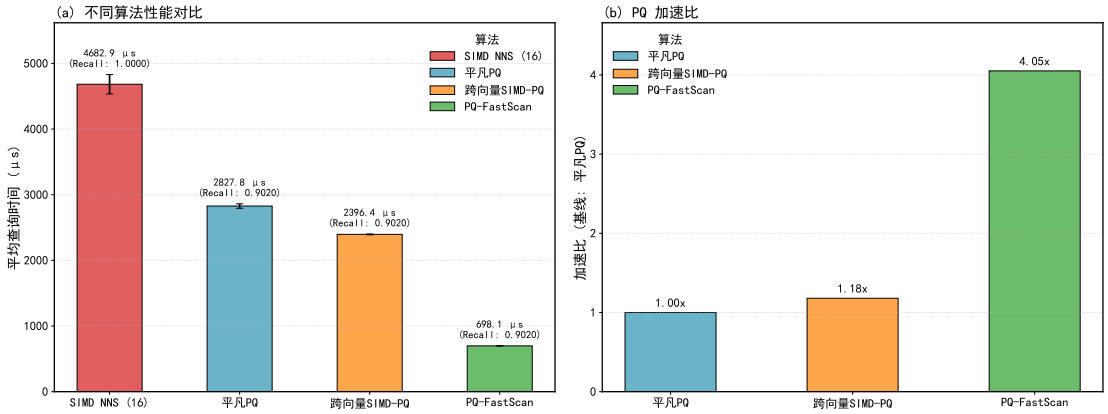


图 2: PQ 各并行策略查询延迟与加速比

2. Perf 剖析：访存优化分析

为了从底层数据验证 PQ-FastScan 的成功原理，以及 gather 指令原生/非原生之间的差距，我使用 perf 工具分析了不同实现的内存加载 (Memory Load) 次数和指令数 (Instruction Retired)。

如图3所示，PQ-FastScan 的内存加载次数相比平凡 PQ 削减了近 **90%**，指令数也减少了约 78%。这雄辩地证明了其性能提升的根源在于成功地将访存瓶颈消除，使得 CPU 可以专注于高效计算。

此外，在 x86 上测试发现，AVX2 的原生 gather 指令带来了显著的性能提升。加速比达到 1.48 倍，远超在 ARM NEON 平台上通过软件模拟 gather 指令所实现的 1.18 倍加速比。这凸显了以下几点:

1. SIMD 技术对 PQ 查询的加速效果，在很大程度上依赖于底层指令集的支持。
2. 缺乏原生硬件支持（如 NEON 需模拟 gather）会限制 SIMD 的性能增益。
3. 原生 gather 指令的核心优势在于**利用硬件并行性**高效处理分散内存读取，减少了指令数量与开销。因此，在进行优化时，**必须将硬件对关键指令的支持度作为核心考量因素**。

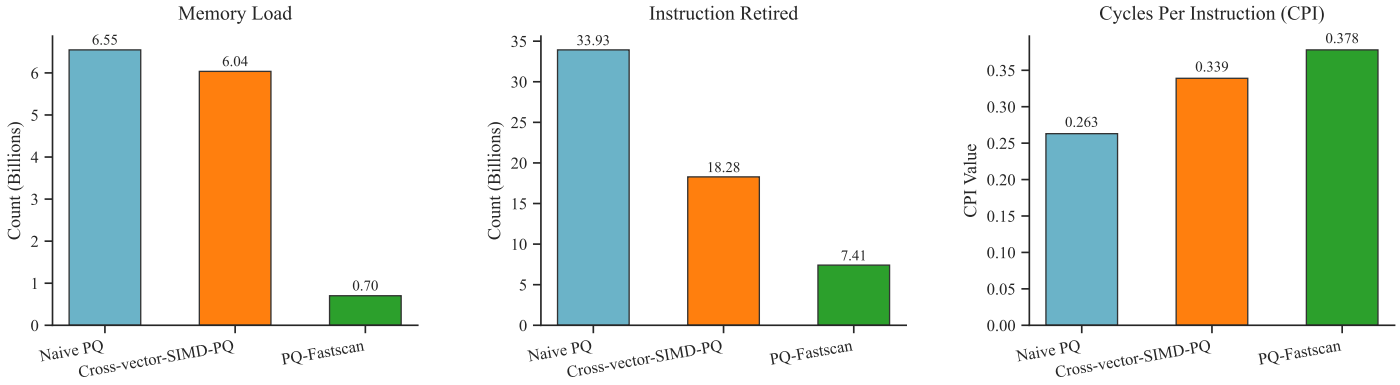


图 3: PQ 优化性能分析

3. recall-latency tradeoff

PQ 作为一种近似算法，其精度依赖于后续的精确重排 (rerank) 阶段。rerank 的候选数量越多，召回率越高，但查询延迟也越长。实验探究了这一权衡关系，如图4。

1. **普遍的权衡关系**: 对于图中所示的四种方法都存在共同的趋势：**随着召回率（X 轴）的提升，QPS（Y 轴）相应下降**。这验证了核心的权衡关系——为了追求更高的搜索精度（高召回率），需要扫描和重排序更多的候选向量，这直接导致了计算量的增加和查询耗时的延长，从而降低了系统的整体吞吐能力。

2. **算法性能对比**:

   - `flat` (红色虚线): 作为基线方法，其性能在四种方法中最低。
   - `cross-vector SIMD` (绿色点划线): 该算法的性能略优于传统的 `flat` 方法。
   - `openmp (8 threads)` (紫色点线): 使用 OpenMP 在大部分情况下都获得了将近 2x 的加速比，效果比 SIMD 更佳。
   - `fastscan` (蓝色实线): 在所有召回率水平上都表现出最优的性能。在相同的召回率要求下，它的 QPS 显著高于所有其他方法，意味着其查询延迟最低，效率最高。

3. **高召回率区间的性能变化**: 在图表右侧的绿色高亮区域（召回率 > 0.90），所有算法的性能下降趋势都变得更为陡峭。这表明，当召回率要求非常高时，**为了找到极少数相关的遗漏项，性能成本会急剧上升**。即便如此，`fastscan` 算法的领先优势依然十分明显，能够在维持高召回率的同时，提供远高于其他方法的查询效率。这对于需要在精度和延迟之间做出最佳选择的实际应用场景具有重要意义。

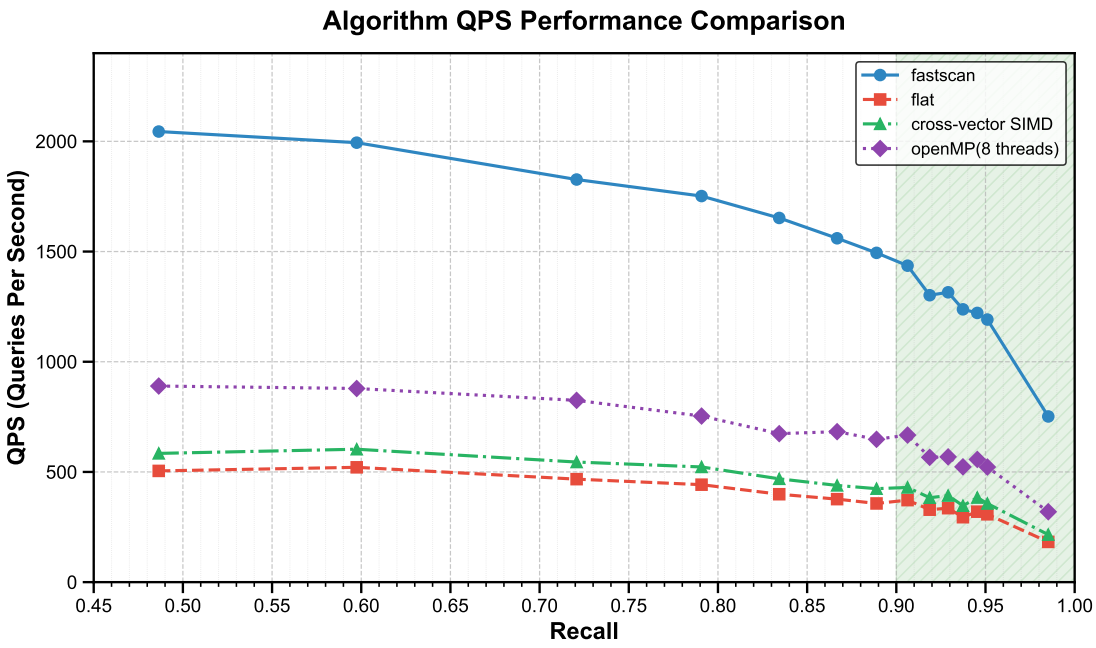


图 4: PQ 各并行策略 recall-latency trade-off

四、 倒排文件索引 (IVF) 的并行优化

(一) IVF 算法原理

倒排文件索引（Inverted File, IVF）是一种应用广泛的近似最近邻搜索（ANNS）加速策略，其核心思想是通过对高维向量空间进行划分，将全量搜索问题转化为一个范围更小的精确搜索问题，从而在保证较高召回率的同时，大幅降低查询的计算开销。其过程主要分为离线索引构建和在线查询两个阶段。

- **索引构建（离线阶段）**首先，使用 K-Means 等聚类算法将数据库中的所有向量预先划分成 n_{list} 个簇。然后，为每个簇建立一个“倒排列表”，该列表记录了所有属于该簇的向量的 ID。最终，索引由两部分构成：所有簇的质心向量和每个簇对应的倒排列表。
- **查询流程（在线阶段）**当给定一个查询向量时，搜索过程分两步进行：
  - **粗筛**: 计算查询向量与所有 n_{list} 个簇质心的距离，并选取距离最近的 n_{probe} 个簇作为候选目标（其中 $n_{\text{probe}} \ll n_{\text{list}}$）。
  - **精筛**: 仅在粗筛阶段选出的 n_{probe} 个候选簇所对应的倒排列表中，获取所有向量 ID，并读取这些向量进行精确的距离计算，最终返回距离最近的 Top-K 个结果。

IVF 算法通过两阶段搜索，将暴力搜索的复杂度从 $\mathcal{O}(N \cdot d)$ 显著降低至约 $\mathcal{O}(n_{\text{list}} \cdot d + \frac{N \cdot n_{\text{probe}}}{n_{\text{list}}} \cdot d)$，其中 N 是数据库大小，d 是向量维度。算法的性能高度依赖于两个关键超参数的设置：

- $n_{\textbf{list}}$ **(聚类中心数)**: 该值在粗筛成本和精筛成本之间形成权衡。n_{list} 越大，粗筛越慢，但每个簇包含的向量越少，精筛越快。
- $n_{\textbf{probe}}$ **(查询的簇数量)**: 该值在搜索精度和搜索速度之间形成权衡。n_{probe} 越大，搜索范围越大，召回率（精度）越高，但查询耗时也相应增加。

因此，在实际应用中，需要根据具体的数据规模和性能要求，选择最优的参数组合。

(二) 共享内存并行 (pthread)

pthread，全称为 POSIX Threads，是一个标准化的 C 语言编程接口 API，用于创建和操作线程。它允许程序在单个进程内创建多个执行流（即线程），从而在多核处理器上实现并行计算。作为一种底层的线程库，pthread 提供了对线程生命周期、同步（如互斥锁、条件变量、barrier 等）和数据共享的精细控制。这种灵活性使得开发者能够实现复杂的并行逻辑，但同时也要求开发者手动处理线程管理和同步的复杂性，以避免竞态条件和死锁等问题。我利用 pthread 实现了两种线程模型对 IVF 查询过程进行并行化加速：静态线程池和动态线程模型 [12]。

1. 静态线程模型

静态模型在程序启动时初始化一个固定数量的线程池(例如 8 个),通过屏障(`pthread_barrier_t`)进行同步，避免了频繁创建和销毁线程的开销。查询过程被划分为三个阶段：

- **Stage-A: 质心距离计算**。将 n_{list} 个质心的距离计算任务静态均匀分配给所有线程。
- **Stage-B: 倒排表扫描**。将 n_{probe} 个候选簇的扫描任务分配给各线程，每个线程使用独立的局部堆维护 top-k 结果，以避免锁竞争。
- **Stage-C: 结果归并**。主线程收集所有子线程的局部堆结果，合并得到最终的全局 top-k。

Listing 8: 静态线程模型：主线程执行两阶段任务和调度

```
// Stage-A: 质心距离计算
g_stage = STAGE_CENTER;
pthread_barrier_wait(&g_barrier);  // 通知所有线程开始
for (int cid = 0; cid < local_end; ++cid)
    cent_dists[cid] = { l2_distance(query, centroids[cid].data(), dim), cid
        };
pthread_barrier_wait(&g_barrier);  // 等所有线程完成
// Stage-B: 倒排列表扫描
g_stage = STAGE_SCAN;
pthread_barrier_wait(&g_barrier);  // 通知所有线程开始扫描
for (int lid : my_lists) {
    for (int idx : invlists[lid]) {
        float d = l2_distance(query, base + idx * dim, dim);
        update_topk(heap, d, idx);
    }
}
pthread_barrier_wait(&g_barrier);  // 等待所有线程结束
```

理论加速比受限于负载均衡效率 η 和同步开销 B（`pthread_barrier_wait`），公式为：

$$= \left(\frac{f_1}{N_{threads}} + \frac{f_2}{N_{threads} \cdot \eta} + f_3 + \frac{2B}{T_s} \right)^{-1} < N_{threads}$$

其中 f_1, f_2, f_3 分别为质心计算、列表扫描和合并阶段的时间占比，T_s 为总时长。为进一步优化，我还实现了**缓存友好重排**，在索引构建时将同簇向量在内存中连续存储，将随机访存优化为顺序访存，显著提高了 L2/L3 缓存命中率，查询延迟进一步降低。

由于 IVF 的算法特点，倒排表的长度是不一样的，并且可能差距较大，单纯地按照线程数等分会造成**负载不均**的问题，解决的办法是**任务池**，创建好的线程会不断地从任务队列中动态地“拉取”任务进行处理，一旦处理完成就拉取下一个任务，这样线程的利用率更高，有效缓解负载不均的问题。

2. 动态线程模型

动态模型为每次查询请求按需创建线程，任务完成后立即销毁。

- **优势：** 空闲时无资源占用，适合突发性计算负载。
- **劣势：** 线程创建和销毁的开销**巨大**。对于 IVF 这类高频、短时查询场景，该开销远超并行计算带来的收益，导致性能远不如静态线程池。

Listing 9: 动态线程模型

```
// 创建线程
std::vector<pthread_t> threads(num_threads);
for (int i = 0; i < num_threads; i++) {
    pthread_create(&threads[i], nullptr, search_thread, &thread_args[i]);
}
// 等待所有线程完成
for (int i = 0; i < num_threads; i++) {
```

```
    pthread_join(threads[i], nullptr);
}
```

(三) 共享内存并行 (OpenMP)

OpenMP 通过编译指导（#pragma）极大地简化了并行代码的编写，并展现出优于手动管理线程的 pthread 的性能。

- **质心距离计算：** 使用 `#pragma omp parallel for schedule(static)`，将计算任务静态均匀分配，此阶段负载均衡。

```
#pragma omp parallel for schedule(static)
    for (int cid = 0; cid < nlist; ++cid) {
        float dist = l2_distance(query, g_ivf_index.centroids[cid].data(),
                                 static_cast<int>(dim));
        centroid_dists[cid] = {dist, cid};
    }
std::partial_sort(...);                          // 选出 nprobe 个质心
```

- **倒排表扫描：**考虑到各倒排表长度不均,采用 `#pragma omp parallel for schedule(dynamic)` 进行任务的动态调度。**这有效地解决了 pthread 静态版本中的负载不均衡问题**，是其性能更优的关键。每个线程同样使用局部堆来避免同步开销。

```
#pragma omp parallel for schedule(dynamic)//倒排表扫描
    for (int p = 0; p < real_probe; ++p) {
        ...
    }
```

1. 结合 SIMD 的极致优化

为进一步压榨性能，在 OpenMP 线程级并行的基础上，引入了 SIMD 数据级并行。通过将核心的 L2 距离计算函数（`l2_distance`）替换为手动编写的 NEON 指令版本，实现了在每个线程内部对向量运算的加速。

- **融合模型：** 形成了线程级并行与数据级并行的层次化并行模型。

(四) 分布式内存并行 (MPI)

1. 实现原理

为了利用多台节点的计算资源，MPI 提供了一种分布式内存并行模型，通过在多个独立的进程间通信来协调任务，将计算负载分发到不同的处理器或计算节点上。对 IVF 算法的优化来说，核心思想与共享内存类似，即将 n_probe 个倒排列表的扫描任务分配给不同的 MPI 进程 [3]。

MPI 优化主要体现在以下几个方面:

1. **倒排列表搜索的负载均衡与并行：** 这是 MPI 优化的核心。在获取到 $nprobe$ 个最近的聚类中心列表后，这些列表会被均匀地分配给不同的 MPI 进程。具体分配策略是，将第 p 个 probe 列表分配给进程 p (mod world_size)。每个进程只负责搜索分配给它的倒排列表，从而将计算任务分散到各个处理器上。

Listing 10: MPI 版本 IVF 搜索核心逻辑 - 局部搜索

```
inline std::priority_queue<DistIdx>
ivf_search_mpi(const float *base,const float *query,std::size_t base_number,
    std::size_t dim,std::size_t k,int nprobe)
{
    // 2.1 先各自找到 query 最近的 nprobe 个 centroids
    std::vector<DistIdx> cent_dists; cent_dists.reserve(g_ivf_index.nlist);
    for (int cid = 0; cid < g_ivf_index.nlist; ++cid) {
        ...
    }
    std::partial_sort(cent_dists.begin(),cent_dists.begin() + std::min(nprobe
        , g_ivf_index.nlist),cent_dists.end());
    // 2.2 按 “probe 序号 % world_size == rank” 把列表平均分给各进程
    std::priority_queue<DistIdx> local_topk; // 大顶堆
    for (int p = 0; p < nprobe && p < g_ivf_index.nlist; ++p) {
        if (p % env.size != env.rank) continue; // 这一 probe 不归我管
        int list_id     = cent_dists[p].second;
        const auto &invlist   = g_ivf_index.invlists[list_id];
        for (int idx : invlist) {
            ...
        }}}
```

2. **结果收集与全局 Top-K：** 每个 MPI 进程在自己的倒排列表上执行搜索，并维护一个局部的 Top-K 结果（大顶堆）。搜索完成后，所有进程通过 MPI 的 `MPI_Gather` 操作将各自的局部 Top-K 结果收集到根进程（rank 0）。根进程负责将所有局部结果汇总，并从中选出最终的全局 Top-K 最近邻。

2. MPI+OpenMP

MPI 与 OpenMP 的结合旨在利用多节点（通过 MPI）和单节点内多核心（通过 OpenMP）的计算资源，以期达到更优的并行加速效果。其核心实现思路如下：

- **簇间并行 (MPI):** 首先，利用 MPI 实现进程级别的并行。在 N 个聚类中心被选出后，这些中心对应的倒排列表的搜索任务被分配给 P 个 MPI 进程
- **簇内并行 (OpenMP):** 在每个 MPI 进程内部，当处理分配给它的某个特定簇的倒排列表时，利用 OpenMP 指令进行线程级别的并行。具体来说，遍历该倒排列表中的所有向量，计算查询向量与这些库向量之间的距离，这一过程通过 OpenMP 的并行循环 (#pragma omp parallel for) 来加速。每个 OpenMP 线程计算一部分向量的距离，并将结果汇总到共享结果容器中。
- **结果汇总:** 每个 MPI 进程首先通过 OpenMP 线程的协作，得到其负责的簇中的局部 Top-K 结果。随后，所有 MPI 进程的局部 Top-K 结果通过 MPI_Gather 操作汇总到主进程（rank 0）。主进程最后对收集到的所有局部结果进行全局排序，得到最终的全局 Top-K 结果。

理论上，MPI+OpenMP 的混合并行模型有望提供比单一并行策略（纯 MPI 或纯 OpenMP）更高的加速比。**两级并行带来的加速:** 假设有 P 个 MPI 进程，每个进程内使用 T 个 OpenMP 线程。

理想情况下，如果任务能够完美划分且无额外开销，总加速比可以期望达到 $P \times T$。但是实际受限于并行开销，这种理想情况无法达到：

- **过细的并行粒度与高昂的开销**：对于簇内搜索，如果倒排列表的平均长度不够大，使用 OpenMP 进行并行化时，每个线程分配到的计算任务可能过少。此时，OpenMP 线程创建、管理和同步的开销可能超过并行计算带来的收益。
- **MPI 与 OpenMP 资源竞争与通信瓶颈加剧：** MPI 进程和 OpenMP 线程可能在 CPU 核心、内存带宽等资源上产生竞争，导致整体效率下降。不当的线程/进程绑定策略也可能加剧此问题。

在实际测试中，MPI+OpenMP 的组合表现为**负优化**，主要原因是实验的数据规模太小，单个节点就已经足以满足计算需求，多节点只会**徒增并行开销**。在面对海量的数据时，MPI 的优势才能体现。

3. MPI+SIMD

这部分同样实现了 **MPI+SIMD** 的混合版本，在每个 MPI 进程内部使用 SIMD 加速距离计算，形成线程级并行与数据级并行的层次化并行模型。

（五） GPU 并行加速

1. 实现原理

IVF 的 GPU 加速和暴力搜索类似，只是 batch 分组有一些优化。

1. **并行化的粗筛**: 将查询批次与所有簇质心的距离计算重构为一次 cuBLAS 矩阵乘法，并用并行核函数选出 Top-n_probe。
2. **基于簇重合度的查询分组**: 这是一个 CPU 端的预处理步骤。通过分析一个批次内不同查询所需访问的簇列表的重合度，将目标簇高度重合的查询分为一组。
3. **分组化的并行精筛**: 对于每个查询组，一次性将它们所需访问的所有簇的并集加载到 GPU 显存中。然后启动一个自定义 CUDA 核函数，并行地在这一小块数据上为组内的每个查询完成精筛。

该方案通过查询分组，极大地提升了数据局部性和访存效率，避免了为每个查询反复加载数据。

此外，作为对比，本研究也调用了业界顶尖的 **Faiss-GPU** 库来实现 IVF。使用 Faiss-GPU 实现 IVF 搜索的流程被极大地简化了，主要分为索引构建和搜索两个阶段。

1. **索引构建**: Faiss 提供了一个便捷的函数 `faiss::gpu::index_cpu_to_gpu`，它可以将一个在 CPU 上训练好并填充了数据的 IVF 索引，“克隆”到 GPU 上。在这个过程中，Faiss 会自动处理所有的数据传输、显存分配，并对数据布局进行优化（例如转置存储）以适应 GPU 的访存模式。
2. **搜索**: 搜索过程只需要调用 GPU 索引对象的 `search` 方法。所有复杂的并行操作，包括查询批处理、粗筛、精筛、多 Stream 异步执行等，都由 Faiss 在内部自动完成。

(六) 实验测试

1. 超参数探索

如图5所示，对于这个 DEEP100K 数据集，聚类数量 nlist=1024 为佳，之后的 IVF 相关实验都会以这个数据为准。

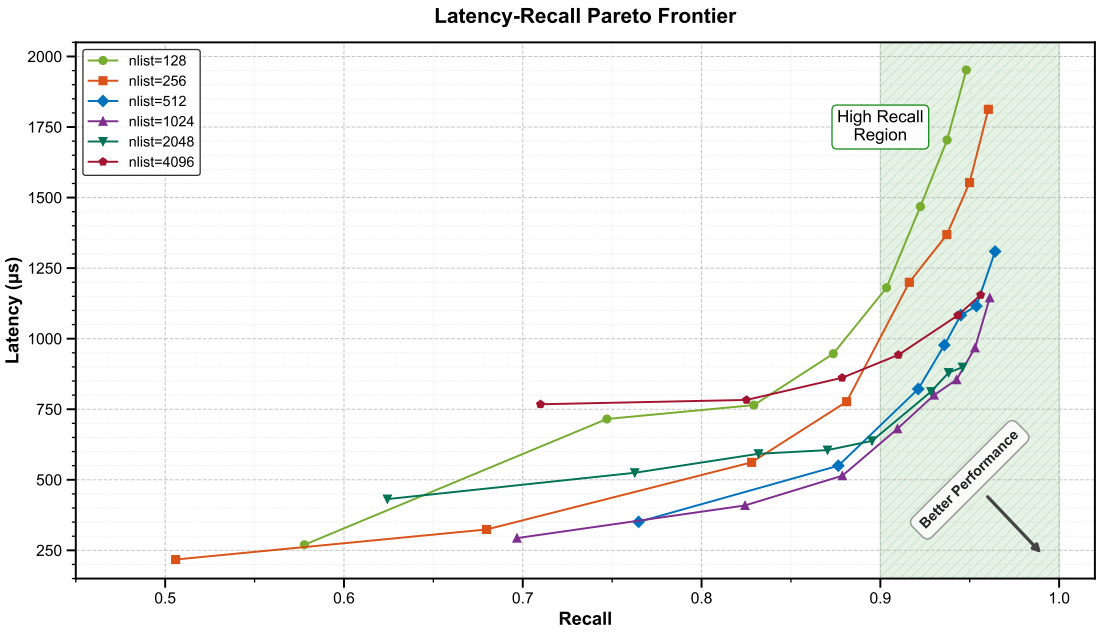


图 5: IVF 超参数探索

2. 线程数/进程数对性能的影响

实验验证，如图6和图7所示，无论是多线程并行还是多进程并行，都有共同的结论：**线程数/进程数并不是越多越好**。根据实验结果，pthread 和 OpenMP 都使用 8 个线程为佳，MPI 使用一个节点 8 个进程为佳。

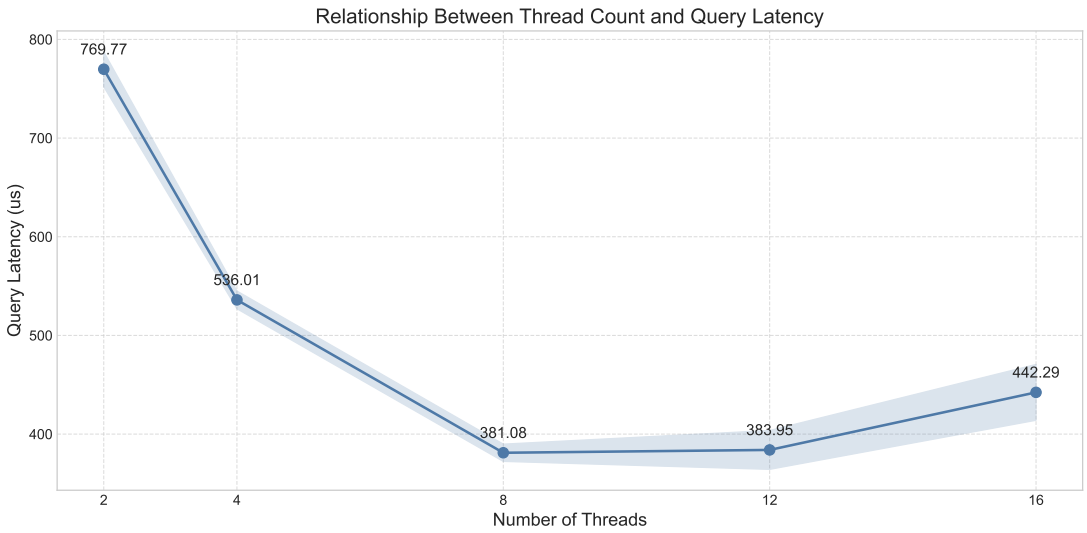


图 6: pthread 线程数对查询延迟的影响

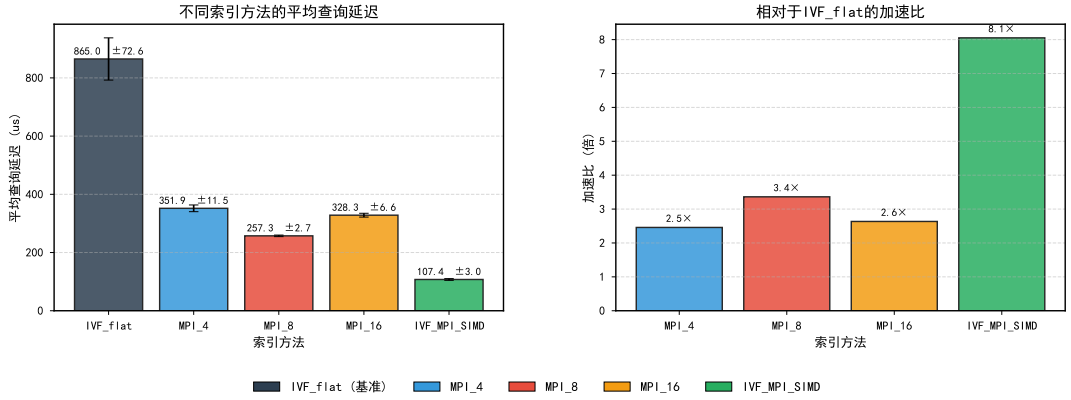


图 7: MPI 进程数对查询延迟的影响

3. recall-latency tradeoff

综合 IVF 的 CPU 相关实验的所有数据，绘制出图9，IVF 串行版本的性能已经超过了并行优化的暴力搜索的性能，这凸显了近似最近邻搜索（ANNS）的核心价值，即通过牺牲一定的召回率来换取查询性能的巨大提升。此外，从图中可以得出以下几点结论：

1. **普遍的 Tradeoff 关系**：为了获得更高的召回率，算法需要搜索更多的倒排列表或访问更多的向量，这导致了查询延迟的升高，是一个典型的性能与精度之间的权衡。
2. **OpenMP+SIMD 性能最优**：在所有的并行策略中，IVF(OpenMP+SIMD)（红色点划线）表现出最优的性能。在任意给定的召回率水平上，它几乎都具有最高的 QPS，这意味着它能在保证同样精度的前提下，提供最低的查询延迟，**加速比高达 11.5x**。这充分说明了线程级并行（OpenMP）与数据级并行（SIMD）结合的巨大优势。
3. **SIMD 优化的显著效果**：OpenMP 和 MPI 结合 SIMD 后都能使性能更上一层楼，展现了线程级并行与数据级并行的层次化并行模型的优势。
4. 基于 **OpenMP** 的两种策略整体优于 **pthread** 和 **MPI** 的策略。
   - OpenMP 相比于 pthread，拥有更灵活的并行调度，**有效解决负载不均衡的问题**，此外，这是一个高度优化的库，调用也更加方便，相比于我对 pthread 的实现，不仅代码更简洁、不易出错，而且最终的运行效率也更高。
   - MPI+SIMD 的加速比为 8.1x，MPI 的优势在于跨节点、跨机器的可伸缩性，但对于**这个数据集规模，单节点足以应对查询需求**，没有体现出 MPI 的优势，反而 MPI 的并行开销拖累了查询性能，**经过 VTune 的分析（如图8），单节点下，MPI 的 Instructions Retired 和 CPI 都高于 OpenMP，性能不如 OpenMP**。
     - OpenMP 则是在同一进程内部启动多个线程，所有线程共享同一块物理内存，对候选向量或索引块的访问可以直接通过共享缓冲区完成，省去了进程边界的拷贝和同步开销，内存带宽与缓存命中率的优势被放大（这对 SIMD 也是有利的）。
     - **负载均衡**：OpenMP 的动态调度策略（schedule(dynamic)）在倒排表长度差异较大时可以更灵活地负载均衡，MPI 暂时并没有实现相关特性。
   - 基于 **pthread** 的两种策略（pthread_static_cache 和 pthread_dynamic）加速比分别为 4x 和 1.24x。正如我分析的那样，IVF 查询是一个高频的短时查询，动态线程的**反复线程创建和销毁**也带来巨大的开销，性能不如静态线程模型。

事实上，IVF 的 OpenMP+SIMD 优化是我在鲲鹏服务器（CPU）上实现的单次查询时间最短的一个策略，查询延迟在 70us 左右（reccall=0.90）。

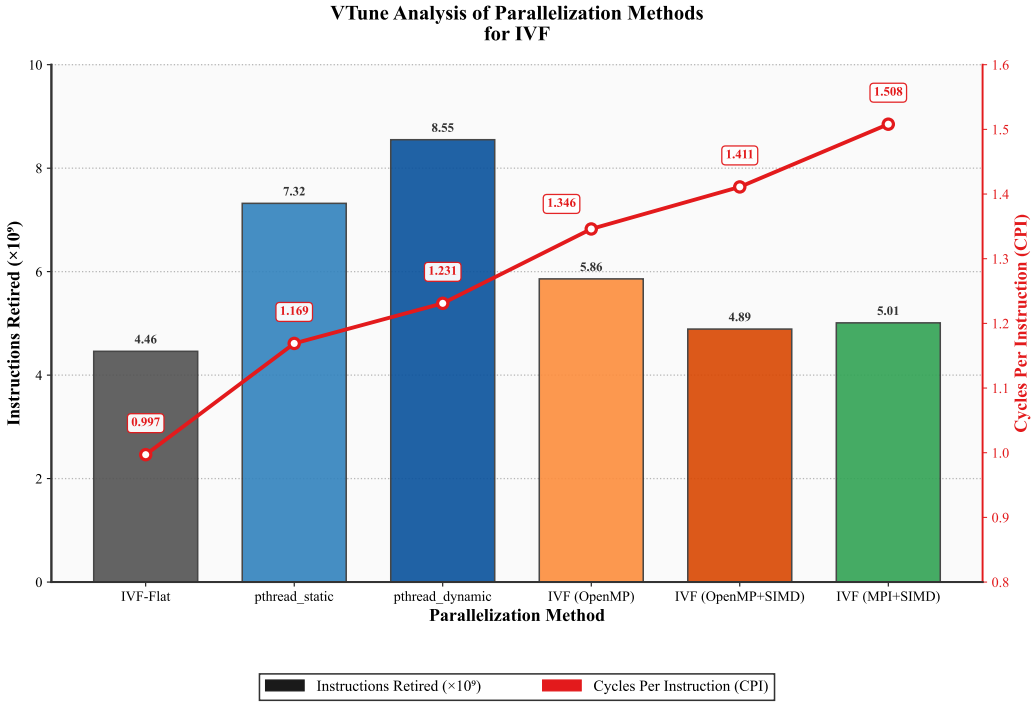


图 8: VTune 分析 IVF 各策略性能

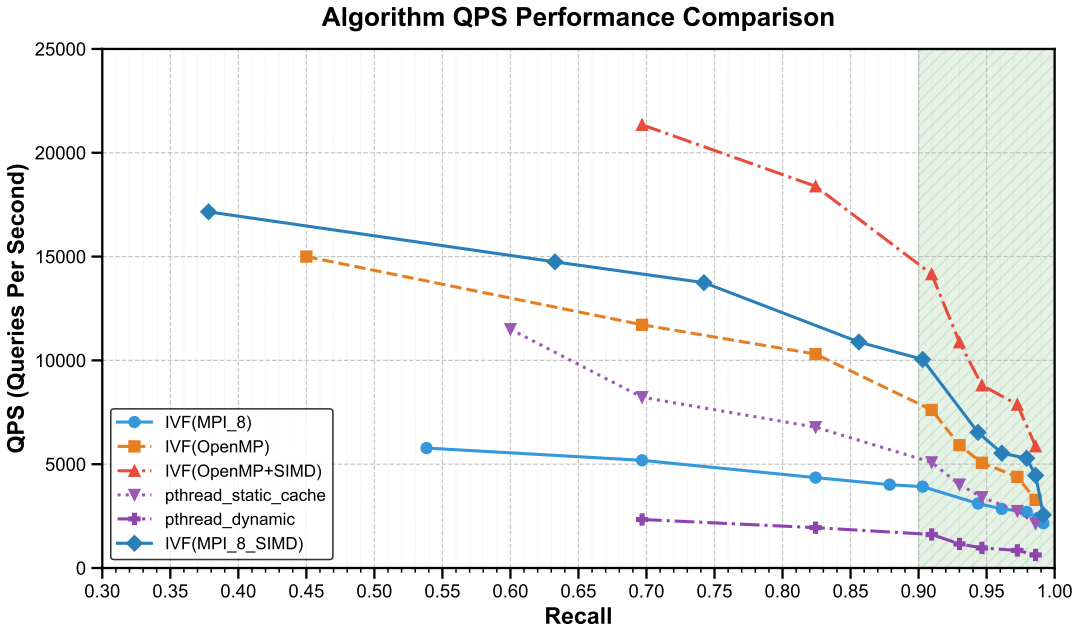


图 9: IVF 各并行策略 recall-latency tradeoff

4. GPU 并行加速效果

- 我自己实现的 `IVF_gpu` 版本相较于 CPU 版本，实现了约 **47**× 到 **73**× 的性能提升。
- 而 `IVF_faiss_gpu` 版本的表现则更为出色，其加速比达到了 **345**× 到 **564**×。这也体现了 faiss 库的成熟和先进，其在底层 CUDA 核函数的设计、内存访问模式以及并行计算调度上都进行了深度优化，从而实现了极致的性能表现。

表 2: 不同召回率 (Recall) 下的加速比 (Speedup) 对比

方法	**召回率 (Recall)**					
	0.68	0.81	0.9	0.95	0.986	0.995
IVF_gpu	47.07x	69.91x	58.52x	66.24x	73.01x	49.18x
IVF_faiss_gpu	363.49x	425.91x	345.59x	416.41x	**564.14x**	516.38x

5. Faiss-GPU 库的高度优化

下载了 Faiss-GPU 的库后, 我也浏览了关于 IVF 优化的代码。总的来说, 由于刚入门 CUDA 的 GPU 编程，其优化的方式远超出我的能力水平。

- **数据布局（Index Build 时）**
  - 倒排表转为 *SoA + block-of-32*，这种布局在 GPU 上更加高效。
  - 手写版通常直接 `cudaMemcpy` 原始向量，float32 存 global，导致 L2 miss 与 global load inefficiency 极高。
- **核函数融合**
  - 距离计算 + Top k 更新写成单一 kernel, 局部小顶堆放寄存器, 省掉一次 global round-trip 与 kernel launch。
  - 手写版多核函数流水，我的方法导致每个数据项需要被写入全局内存两次。
- **Warp 级算子**
  - 1 warp 负责 1 query，借助一些 sync 方法做 SIMD 级 top k 归约；Tensor Core 自动触发 `mma` 汇编。
  - 手写版在整个线程块（block，包含多个 warp）级别维护一个堆结构，用于存储 top-k 结果。这种方法需要频繁在块内同步数据。
- **多 Stream 与 Pipeline**
  - 质心 GEMM、倒排扫描、H2D 复制分配到不同 `cudaStream_t`, 事件同步, 实现 PCIe 与计算完全重叠。
  - 我的实现只用一条流水线来处理所有事情，导致任务只能按顺序排队执行。这通常会造成 GPU 计算单元和数据传输总线在某些时间段处于“无事可做”的空闲状态，造成资源浪费。

faiss-gpu 把 ANNS 全流程“翻译成 GPU 的语言”——从 **访存模式、核函数融合、warp 级算子、流水并发到运行期调度**无一遗漏，因此相较于仅用 cuBLAS 的朴素实现可带来额外加速。

五、 IVF-PQ 的并行优化

(一) IVF-PQ 原理

IVF-PQ 是对标准 IVF 索引的进一步优化，其核心目标是大幅压缩索引体积并加速精筛阶段的距离计算。它将 PQ 应用于残差向量来实现这一目标。

1. 核心思想与构建流程

与 IVF 在倒排列表中存储原始向量不同，IVF-PQ 的构建与存储流程如下：

1. **计算残差**: 与 IVF 相同，首先通过 K-Means 将数据集聚类。对于每个向量 $\mathbf{x}_j$，计算其与所属簇中心 $\mathbf{c}_i$ 的**残差向量** $\mathbf{r}_j = \mathbf{x}_j - \mathbf{c}_i$。这个残差向量通常比原始向量具有更小的范数和更集中的分布，因此更利于压缩。
2. **残差的乘积量化**: 对所有残差向量 $\mathbf{r}_j$ 应用 PQ 编码。具体来说，将 d 维残差向量切分为 m 个子空间，并为每个子空间独立学习一个包含少量中心点的小码本。这样，每个残差向量就被编码为一个由 m 个码字索引组成的紧凑代码，从而极大地降低了索引的存储需求。

2. 查询流程与距离计算

1. **粗筛**: 查询阶段与 IVF 完全相同，通过计算查询向量与所有簇中心的距离，定位到 n_{probe} 个最邻近的候选簇。
2. **精筛**: 无需解压编码来重构向量，而是直接利用查询向量和 PQ 码本进行**非对称距离计算 (ADC)**。该方法通过预计算查询向量的各个子向量与对应码本中所有中心点的距离，并以查表（LUT）后求和的方式，高效地估算出查询向量与数据库中每个被编码向量的近似距离。

综上，IVF-PQ 将精筛阶段的大量浮点向量距离计算，转换为了对极短编码的、基于查表的快速运算。这在内存占用和查询速度上带来了巨大提升，但代价是引入了量化误差。在实际应用中，这种误差可以通过调整 n_{probe} 或在得到初步候选集后，增加一个可选的精确重排（rerank）步骤来弥补精度损失。

3. PQ-IVF 效果不佳

PQ+IVF 是将 PQ 与倒排索引顺序对调的一种变体：

1. **预先量化**：对所有原始向量先做 PQ 压缩，得到压缩码 PQ($\mathbf{x}_j$)，并存储在磁盘上。
2. **编码索引**：使用倒排索引（如 IVF）对压缩码进行聚类，将压缩向量分配到不同桶中。
3. **查询**：对查询向量 $\mathbf{y}$，先 PQ 编码得到 PQ($\mathbf{y}$)，再访问 n_{probe} 个桶，比较码间距离。

查询相关资料发现，关于 PQ-IVF 这种组合方式在文献和实际应用中比较少见，其有效性和实用性不如“先 IVF 再 PQ”。PQ-IVF 的效果和 IVF 的效果类似，甚至不如；而 IVF-PQ 的效果明显好于单独使用 IVF。

- 若采用 PQ→IVF 的顺序，即先将数据做 PQ 后再进行倒排，由于聚类和检索都是基于已有量化误差的编码，易导致桶划分不准确，从而影响召回率，效果往往不如单独使用 IVF。
- 而采用 IVF→PQ 的顺序，即先基于原始向量进行准确的倒排索引分桶，再在每个桶内对残差向量进行 PQ 编码，保留了可靠的剪枝能力，同时在候选范围内实现更高效的近似搜索。

(二) 实验验证

IVF-PQ 的并行优化思路和 IVF、PQ 的类似，此处不再重复，我也新实现了 MPI+SIMD 的版本，让性能对比更加完善。如图10所示。

- recall=0.90 时，IVF-PQ 的串行算法相比于 IVF **快 48%**，显示出 IVF 和 PQ 结合的优势。
- recall=0.90 时，OpenMP+SIMD 的加速比为 2.9x，MPI+SIMD 的加速比为 2.2x。和 IVF 实验结论保持一致，MPI 效果略差于 OpenMP。此外，我发现 IVF-PQ 虽然串行时间比 IVF 快，但是并行化的加速比不如 IVF 高。这主要是因为 PQ 需要用 rerank 提高精度，这部分并行化效果并不好。

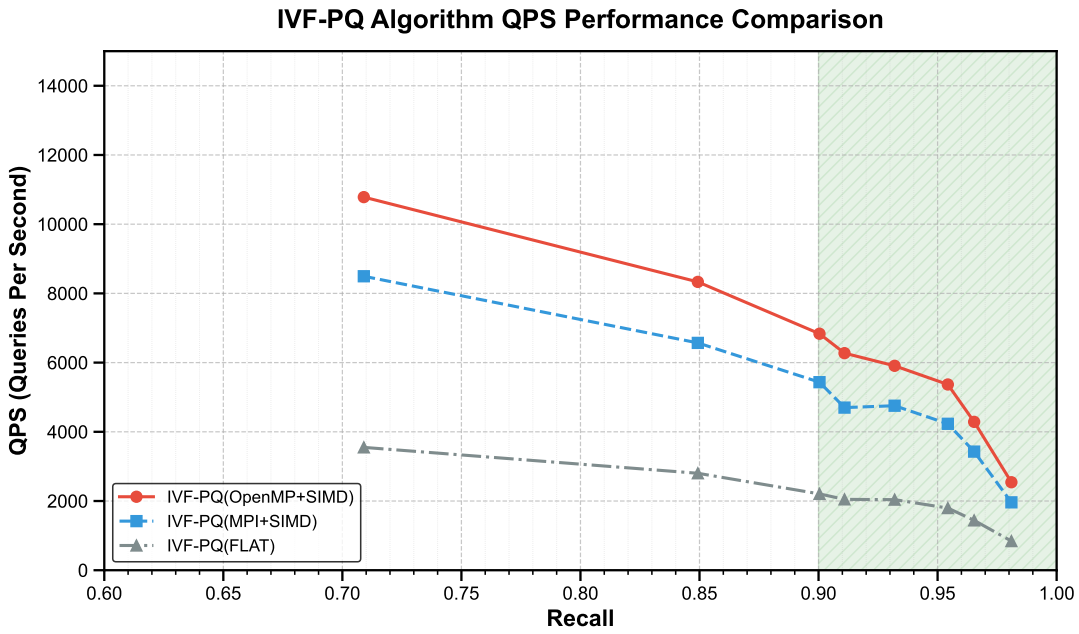


图 10: IVF-PQ 各并行策略 recall-latency tradeoff

六、 图索引 (HNSW) 的并行优化

(一) 算法原理与并行化难点

HNSW (Hierarchical Navigable Small World graphs) 是当前性能最顶尖的 ANNS 算法之一。它通过构建一个层次化的图结构来加速搜索。每一层都是一个“可导航的小世界网络”，上层图更稀疏，作为快速通道；下层图更密集，用于精确查找。搜索时，从顶层图的入口点开始，贪心地走向离查询目标最近的邻居，直到找到局部最优解，然后进入下一层继续搜索，直至最底层 [7]。

HNSW 的主要并行化难点在于其核心的贪心遍历过程具有**强串行依赖性**。下一步访问哪个节点，完全取决于当前节点及其邻居的距离计算结果，这使得将单次查询的图遍历过程分解给多个线程并行执行变得极其困难。实验证明，盲目地对邻居距离计算等局部操作进行并行化，其引入的线程创建和同步开销远大于计算收益，会导致显著的**负优化**。探索的过程中我没能直接对 HNSW 正优化，因而曲线救国，使用混合策略。

(二) 混合策略一：IVF-HNSW

1. 原理与实现

既然 HNSW 的核心遍历难以并行，那么思路就转向在其上层结构进行并行化。IVF-HNSW 是一种有效的混合策略：

1. 首先利用 IVF 将数据集划分为 n_list 个簇。
2. 然后，在每个簇内部的数据点上，独立地构建一个 HNSW 图索引。
3. 查询时，先通过 IVF 的粗筛找到 n_probe 个候选簇，然后并行地在这些簇对应的 n_probe 个独立的 HNSW 图上进行搜索。

这样，并行化的对象就从 HNSW 内部的遍历，变为了对多个独立 HNSW 图的搜索任务，这些任务彼此无关，非常适合并行。我使用 OpenMP 和 MPI 实现了对 IVF-HNSW 外层 IVF 部分的并行化。

(三) 混合策略二：分区 HNSW (Partitioned HNSW)

1. 原理与实现

分区 HNSW 是另一种更高层次的并行策略，它完全摒弃了 IVF。

1. 将整个数据集随机地、均匀地水平分割成 P 个独立的分区（P 通常等于线程/进程数）。
2. 为每个数据分区独立地构建一个完整的 HNSW 图索引。
3. 查询时，将查询向量广播给所有 P 个线程/进程，在、每个线程/进程在自己负责的分区 HNSW 图上执行并行的局部搜索。
4. 最后，收集所有进程的局部 Top-K 结果，合并得到全局 Top-K。

相比 IVF-HNSW，分区的 HNSW 有以下几个优势：

- **索引构建简化**：分区 HNSW 不需要执行 K-means 聚类这一耗时且迭代的预处理步骤，索引构建过程更为直接，且各分区索引的构建可以完全并行，进一步缩短构建时间。
- **潜在的更高召回率**：查询时对所有分区都执行搜索，且每个分区内部的 HNSW 搜索质量足够高，则这种方法有可能达到更高的召回率（**上限更高**），因为它本质上是在**搜索整个数据集**。
- **负载均衡潜力**：通过合理的划分策略（例如均匀的随机划分），可以更好地平衡不同 MPI 进程的计算负载，避免某些进程因处理过大或过于复杂的簇而成为瓶颈，对 MPI 来说，IVF 带来的负载不均衡问题比较难解决。

(四) 实验验证与性能分析

如图11所示，recall=0.90 时：

- HNSW 串行版本的性能已相当出色，远超 IVF 等传统方法。
- 直接并行优化 HNSW 的效果不好，是**负优化**。其延迟 (195.1µs) 远高于 Flat 基线 (153.5 µs)。

- 将 IVF 与 HNSW 结合的混合索引方法能有效降低延迟。其中，结合 OpenMP 进行多线程并行以及 SIMD 指令集并行优化的策略取得了最佳效果，其平均查询延迟最低，仅为 94.3µs，**加速比为 1.62x**。MPI 仍然略逊于 OpenMP。

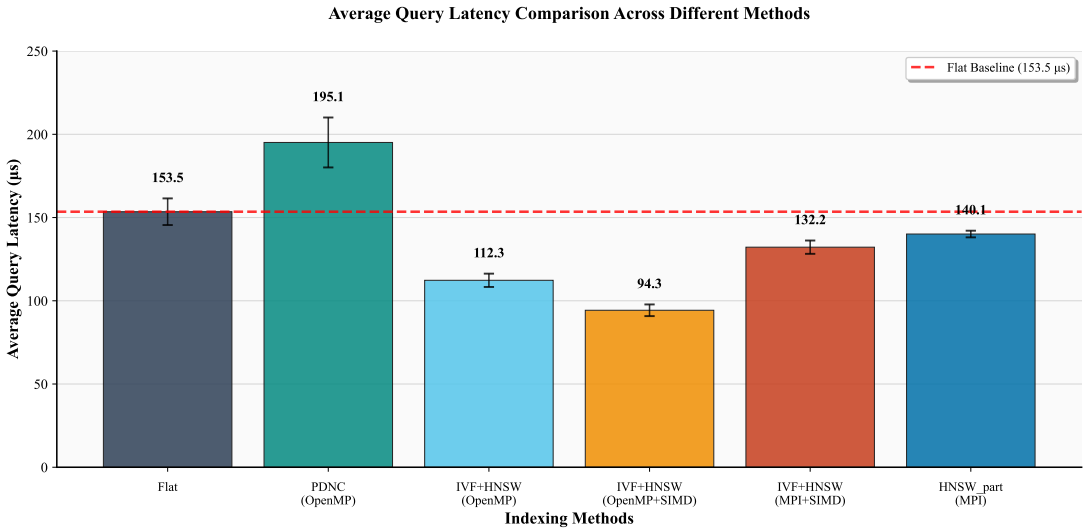


图 11: HNSW 各并行策略查询延迟

recall-latency 实验结果如图12所示，这张图展示了所有算法都呈现出召回率与 QPS 之间的反向关系，**这张曲线图和之前的图都不太一样，不同算法策略的曲线出现了明显的交叉关系，不同 recall 最优的策略也不一样**。

1. **性能对比**:
   - **HNSW_flat**：作为基准，其 QPS 性能在所有算法中最低，但其曲线也相对平滑，没有出现剧烈的性能骤降。
   - **IVF-HNSW(OpenMP+SIMD)**：在 recall=0.93 及之前，openMP 的加速表现出最高的 QPS 性能，当召回率超过 0.92 后，其 QPS 下降速度最快。
   - **IVF-HNSW(MPI+SIMD)**：recall<0.92 的范围内，MPI 优化的 QPS 优于串行 HNSW 和 MPI 分区 HNSW。但是在此之后，性能又不如串行的 HNSW。
   - **分区 HNSW_MPI**：该算法的 QPS 曲线相对平稳，在整个召回率范围内保持了较好的稳定性。尤其是 recall 在 0.95 以上，其他并行策略都出现了**负优化**的情况，而 MPI 优化的分区 HNSW 仍然是**正优化**，并且当 **recall=0.979 时，加速比达到 1.75，这个成绩比 recall=0.90 时更亮眼**，表现出较强的鲁棒性。
2. **最优工作区域**:
   - 图中标注的绿色区域（recall>0.90）表示高性能的区域，其中多种算法的性能开始**波动或交叉**。在这个区域内，不同的优化策略可能面临不同的挑战，或者达到了各自的性能瓶颈。
   - 选择何种算法取决于具体的应用需求：如果对查询速度有极高要求，且能接受略低的召回率，IVF-HNSW(OpenMP+SIMD) 在召回率 0.80-0.92 区间表现最佳；如果需要更高的召回率且对速度也有要求，MPI 优化的分区 HNSW 更合适。

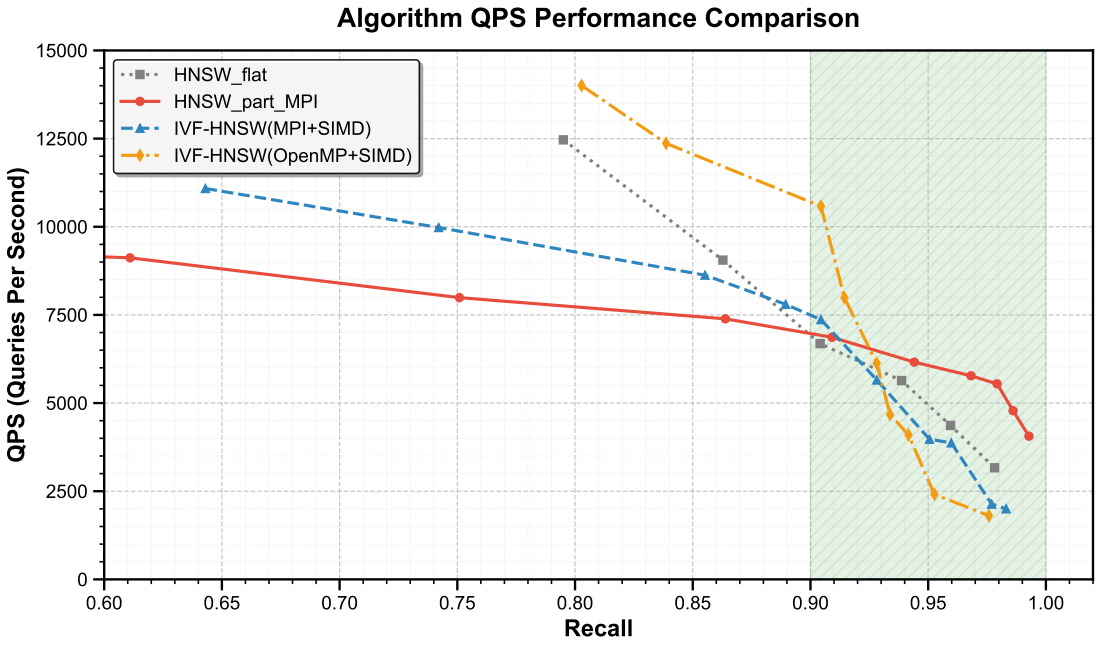


图 12: HNSW 各并行策略 recall-latency tradeoff

七、 总结

(一) 核心结论

这学期的 ANN 实验我探索了主流算法的并行加速策略，经过努力，所有算法都能**正优化**，对于 DEEP100K 数据集上，CPU 并行可以优化到 1x 至 11.5x 的加速比，GPU 并行可以获得数百倍到上千倍的加速。实验普遍反映了以下结论：

1. **不存在普适的“最佳”并行策略**：最优解深度依赖于算法的内在结构与目标硬件的架构特性。计算密集型的暴力搜索在 GPU 上通过算法重构获得最大收益；访存密集型的 PQ 需要通过精巧的数据布局重排（fastscan）来释放 SIMD 潜力；可分解的 IVF 在各类并行平台上均表现良好；而强串行依赖的 HNSW 则需要通过高层级的混合策略来实现并行化。

2. **算法与硬件的协同设计是关键**：高性能的实现往往源于将算法的计算模式“翻译”成硬件最擅长的语言。无论是将暴力搜索映射为 GPU 的矩阵乘法，还是将 PQ-FastScan 的查表操作限制在 CPU 寄存器内，都体现了这一核心思想。

3. **并行开销的权衡是优化的核心**：实现有效的并行加速，关键在于对并行化引入的各类开销进行精确的评估与权衡。例如，SIMD 优化需应对内存带宽瓶颈，OpenMP 的优势在于负载均衡，MPI 的主要开销源于节点间通信与结果合并，而 GPU 则依赖批处理来最大化吞吐。在实践中，召回率-延迟 (Recall-Latency) 权衡图为此类决策提供了关键的量化依据，是指导策略选择的重要工具。

4. **专业工具库的巨大价值**：通过与 Faiss-GPU 的对比，我看到一个高度工程化的专业库在底层设计上的深度优化，能够带来远超手动实现的性能。

(二) 设计启发

本次综合性的并行化实验研究带来了以下几点深刻的设计启发：

- **拥抱混合并行**：将不同层次的并行模型（如线程级 + 数据级，或分布式 + 线程级）相结合，是充分挖掘现代异构硬件潜力的关键。
- **性能分析工具指导优化**：只有通过 perf、VTune 等工具进行定量的性能剖析，才能准确地定位瓶颈（计算、访存、通信、同步），从而进行针对性的优化。
- **并行友好型算法设计**：与其在现有复杂算法上强行并行，不如在设计算法之初就将并行化考虑在内。分区 HNSW 的成功便是一个范例，它通过简化数据划分，天然地实现了更好的负载均衡和并行伸缩性。

(三) 未来工作展望

本报告的工作仍有进一步拓展的空间：

- **更深度的混合并行**：使用更大规模的数据集，探索 MPI 的优势。
- **索引构建的并行化**：本次实验主要关注查询优化，针对海量的数据，索引构建也极其耗时，未来可系统研究如何并行化 K-Means 聚类、HNSW 图构建等耗时的离线过程。
- **更前沿的 ANNS 算法探索**：对更新的量化技术（如 OPQ）或图索引算法进行并行化研究。
- **能效分析**：在追求极致速度的同时，研究不同并行策略的功耗和能效比（Performance per Watt），这在大型数据中心的部署中至关重要。

跳转至：源代码地址，包含这学期所有 ANN 并行化的相关代码和数据资料。

参考文献

[1] Fabien André, Anne-Marie Kermarrec, and Nicolas Le Scouarnec. Cache locality is not enough: High-performance nearest neighbor search with product quantization fast scan. In *Proceedings of the 42nd International Conference on Very Large Data Bases*, volume 9, pages 1033–1044. VLDB Endowment, 2016.

[2] Arm Limited. Arm c language extensions (acle) for neon intrinsics. https://developer.arm.com/architectures/instruction-sets/simd-isas/neon, 2025.

[3] Chenying Du, Chenfei Zhou, Junyun Mai, Junlong Li, Yanfeng Ding, Zhiyuan Hua, Derong Kong, and Yifei Zhang. 并行程序设计实验指导书: MPI 编程. Technical report, Nankai University, May 2025.

[4] FAISS contributors. Fast accumulation of pq and aq codes (fastscan). https://github.com/facebookresearch/faiss/wiki/Fast-accumulation-of-PQ-and-AQ-codes-(FastScan), 2025.

[5] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(1):117–128, 2010.

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

[7] Yury A Malkov and Dmitry A Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence*, 42(4):824–836, 2018.

[8] NVIDIA Corporation. *cuBLAS Library User's Guide*, 2024. CUDA Toolkit 12.4.

[9] NVIDIA Corporation. *CUDA C++ Programming Guide*, 2024. Version 12.4.

[10] 曾泉胜, 陈静怡, 唐明昊, 华志远, 张逸非, and 孔德嵘. 并行程序设计实验指导书: openmp 编程. Technical report, Nankai University, 2025.

[11] 李世阳, 孙辉, 华志远, 孔德嵘, and 张逸非. GPU 编程实验指导书. Technical report, Nankai University, 2025.

[12] 杜忱莹, 周辰霏, 周浩, 陈静怡, 唐明昊, 华志远, 张逸非, and 孔德嵘. 并行程序设计实验指导书: pthread 编程. Technical report, Nankai University, 2025.

[13] 杜忱莹, 安祺, 徐一帆, 曹珉浩, 华志远, 张逸非, and 孔德嵘. 并行程序设计实验指导书: SIMD 编程实验. Technical report, Nankai University, 2025.