南開大學
Nankai University

计 算 机 学 院

并行程序设计实验报告

---

pthread/OpenMP 实验

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 5 月 29 日

摘要

本实验报告围绕高维向量的近似最近邻搜索 (ANNS) 任务展开，重点实现并优化了 IVF、IVF-PQ、以及 HNSW 索引算法，详细讨论使用 pthread 和 OpenMP 进行并行加速的策略，包括静态与动态线程模型、缓存友好性优化以及与 SIMD 指令结合的应用。实验结果表明，OpenMP 在多数场景下展现出优于 pthread 的性能，尤其在结合动态调度和 SIMD 指令后，IVF 查询延迟在 Recall 为 0.90 时可降至 70 微秒左右。算法层面，IVF-PQ 结合了 IVF 的快速筛选和 PQ 的高效压缩，串行状态下优于单一方法。然而，HNSW 本身因其图结构的串行依赖特性，直接并行化导致负优化，但与 IVF 结合后可实现并行加速。最终，报告总结了不同算法在极致优化 (OpenMP+SIMD) 下的 latency-recall 权衡，并强调了并行化并非万能，还需结合任务结构与硬件特性，找到瓶颈，综合运用多种优化手段。

关键字：ANNS；IVF；IVF-PQ；HNSW；pthread；OpenMP；并行优化

目录

一、 问题背景

在近似最近邻搜索（ANNS）中，算法通常面对的是海量的高维向量数据。如果直接对每一个查询向量逐一遍历所有数据点进行距离计算，代价极其高昂，特别是在大规模向量数据库中几乎不可行。在 SIMD 实验中，我已经引入了 PQ 的方法，通过牺牲一些精度换取效率的提升。在本次实验中，IVF 和 HNSW 也将登场。值得注意的是这些方法是可以同时使用的，因此除了使用 pthread 和 openMP 优化之外，我也将探索分析这些算法的优劣以及组合的效果。

二、 Inverted File(IVF)

(一) IVF 原理

IVF 是一种通过**划分搜索空间减少计算量**的加速方案，目标是通过对数据进行空间划分和预聚类，在查询时只访问小部分候选向量，从而大幅减少计算量。因此在查询前首先需要构建索引。

1. 聚类（KMeans）：首先对 base data 中的所有向量进行 KMeans 聚类，将数据划分为 nlist 个簇。每个簇代表一个子空间，其质心用于粗筛阶段的候选过滤。
2. 建立倒排表（Inverted Index）：对于 base data 中的每个向量，计算其与所有质心的距离，选出最近的质心。将该向量的编号加入对应簇的倒排链表中，即构建“质心 $\rightarrow$ 点集”的映射。
3. 索引结构存储：保存每个簇的质心向量和其对应的向量索引列表；构建完成后，搜索时就不再需要原始向量。

Algorithm 1 IVF 索引构建过程——对应离线预处理阶段

Input: base 向量集合 $X = \{x_1, x_2, \ldots, x_N\}$，聚类数量 $nlist$
Output: 簇中心列表 C，倒排表 I

function BUILD_IVF_INDEX($X, nlist$)
  使用 KMeans 对 X 聚类为 $nlist$ 个簇
  $C \leftarrow$ 聚类得到的 $nlist$ 个质心
  初始化倒排表 $I[1 \ldots nlist] \leftarrow \emptyset$
  for 每个向量 x_i in X **do**
    $cid \leftarrow x_i$ 所属的最近质心编号（即最近簇）
    将 x_i 的索引 i 加入倒排表 $I[cid]$
  end for
  return (C, I)
end function

在这一步中有一个非常关键的超参数：聚类数量 nlist，这个参数的设置和数据规模息息相关，如果设置不得当会显著影响搜索的性能，实验也会对这个参数进行讨论。

(二) 串行查询

1. 原理

1. 粗筛阶段：给定查询向量 q，计算其与所有 nlist 个簇的质心的距离；选择距离最近的 nprobe 个簇作为候选簇。

2. 精筛阶段：遍历这 nprobe 个候选簇内的所有点；对每个候选点计算其与 q 的实际距离；保留最近的前 k 个，作为最终的查询结果。

2. 关键代码

Algorithm 2 IVF 查询过程——对应粗筛和精筛两阶段

Input: 查询向量 q，簇质心列表 C，倒排表 I，base 数据 X，簇数 $nlist$，搜索簇数 $nprobe$，返回数目 k

Output: top-k 的向量列表

```
function IVF_SEARCH(q, C, I, X, nlist, nprobe, k)
    centroidDistList ← []
    for i = 1 → nlist do
        d ← 欧几里得距离 (q, C[i])
        centroidDistList.append(d, i)
    end for
    按照距离对 centroidDistList 排序，选出前 nprobe 个簇
    candidates ← []
    for 每个选中的簇编号 cid do
        for 每个点 xid in I[cid] do
            d ← 欧几里得距离 (q, X[xid])
            candidates.append(d, xid)
        end for
    end for
    对 candidates 按照距离升序排序
    return candidates[1 ... k]
end function
```

3. IVF 串行查询的算法复杂度分析

1. 粗筛阶段在该阶段，程序需要计算查询向量 $\mathbf{q} \in \mathbb{R}^d$ 与所有 n_{list} 个簇中心的距离。设向量维度为 d，则该阶段的计算复杂度为：

$$T_{\text{coarse}} = \mathcal{O}(n_{\text{list}} \cdot d)$$

2. 精筛阶段设数据库中共有 N 个向量，且每个簇平均包含 $\frac{N}{n_{\text{list}}}$ 个向量（在实际聚类后每个簇的数量会有差异，此处只是近似）。对于查询时选取的 n_{probe} 个候选簇，总候选点数约为：

$$M = n_{\text{probe}} \cdot \frac{N}{n_{\text{list}}}$$

对每个候选点，需要计算一次欧几里得距离（$\mathcal{O}(d)$），并在一个大小为 k 的堆中维护 top-k 最近邻（插入代价为 $\mathcal{O}(\log k)$）。因此精筛阶段的时间复杂度为：

$$T_{\text{fine}} = \mathcal{O}\left(M \cdot (d + \log k)\right) = \mathcal{O}\left(\frac{n_{\text{probe}} \cdot N}{n_{\text{list}}} \cdot (d + \log k)\right)$$

3. 总复杂度

$$T_{\text{total}} = \mathcal{O}\left(n_{\text{list}} \cdot d + \frac{n_{\text{probe}} \cdot N}{n_{\text{list}}} \cdot (d + \log k)\right)$$

从复杂度表达式可以看出：

- n_{list} 越大，粗筛阶段计算量增加，但每个簇内的数据量减小，从而降低精筛代价，这个值应该要根据数据规模的大小进行调整；
- n_{probe} 越大，查询精度越高（recall 提升），但候选点增多，查询时间也随之增长；
- 需要找到合适的参数组合 $(n_{\text{list}}, n_{\text{probe}})$ 使整体查询效率较优。

因此，在并行化之前，我在实验中先对这些超参数进行讨论，实现准确率与查询效率的权衡。

(三) pthread 并行优化（静态）

pthread 的实现大体可以分为静态线程和动态线程两个类型 [4]。首先介绍**静态线程池 + 阶段同步**（barrier）的并行优化版本，使用 `pthread` 接口实现多线程查询加速。该方案重点优化了查询阶段中最耗时的两部分：簇中心距离计算和倒排表扫描。

1. 线程模型设计

我采用固定数量的线程（以 8 个为例），其中主线程编号为 0，其余子线程为 1 至 7。所有线程在程序启动时创建，仅初始化一次，进入一个循环等待中心调度的执行阶段。通过使用 `pthread_barrier_t` 作为同步原语，实现不同阶段的统一起始与结束控制，避免复杂的互斥锁或条件变量设计。

2. 阶段划分

整个查询过程被划分为三个阶段：

- **Stage-A: 质心距离计算**。每个线程并行计算查询向量与部分簇中心的欧几里得距离，结果写入共享的距离数组；
- **Stage-B: 列表扫描**。每个线程分配到若干个候选簇，扫描其对应的倒排表，并计算候选点与查询向量的距离，同时使用局部堆维护 top-k 最近邻；
- **Stage-C: 结果归并**。主线程统一收集所有线程的局部堆结果，归并得到最终 top-k 查询结果。

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
```

3. 关键代码

- **线程初始化**：通过 pthread_create 创建 NUM_CHILD 条子线程，并在主线程中统一管理生命周期；

```
/* ---------- 初始化：创建 7 条子线程 & barrier ---------- */
static const int NUM_THREADS = 8;          // 总并行线程（含主线程），以8为例
static const int NUM_CHILD   = NUM_THREADS - 1;
inline void init_static_threads()
{
    static bool inited = false;
    if(inited) return;
    inited = true;
    pthread_barrier_init(&g_barrier, nullptr, NUM_THREADS);   // 包含主线程
    for(long t = 1; t <= NUM_CHILD; ++t){
        pthread_t th;
        pthread_create(&th, nullptr, worker_loop, (void*)t);
        pthread_detach(th);             // 常驻线程，无需 join
    }
}
```

- **任务划分**：将计算任务按照簇数 n_{list} 或 n_{probe} 按照线程数等分（切片），实现负载均衡，实际上由于聚类后的每一类的数量不都一样，所以负载会出现不均衡的情况，后面也会有一个线程池；

```
/* ------------------ Stage-A: centroid 距离 ------------------ */
if(g_stage == STAGE_CENTER){
    int nlist = g_ivf_index.nlist;
    int chunk = (nlist + NUM_THREADS - 1) / NUM_THREADS;
    int st = tid * chunk;
    int ed = std::min(st + chunk, nlist);
    // ...计算st到ed范围的中心点距离
}
/* ------------------ Stage-B: list 扫描 ------------------ */
else if(g_stage == STAGE_SCAN){
    int chunk = (g_real_probe + NUM_THREADS - 1) / NUM_THREADS;
    int rs = tid * chunk;
    int re = std::min(rs + chunk, g_real_probe);
    // ...处理rs到re范围的簇
}
```

- **共享状态管理**：主线程设置**全局**状态变量（如查询向量地址、任务阶段标识等），子线程轮询进入对应执行逻辑；
- **结果合并**：每个线程维护一个 priority_queue 局部堆，最终合并为全局 top-k 结果，保证线程安全和精度一致性。

4. 并行加速理论分析

串行算法的耗时主要由以下三个部分组成：

$$T_s = T_{\text{centroid}} + T_{\text{scan}} + T_{\text{merge}},$$

$$T_{\text{centroid}} = N_{\text{list}}\, d\, t_{\text{fma}},$$
$$T_{\text{scan}} \approx N_{\text{probe}}\, \bar{L}\, d\, t_{\text{fma}},$$
$$T_{\text{merge}} = \varepsilon T_s,\ 0 < \varepsilon \ll 1$$

设相对占比

$$f_1 = \frac{T_{\text{centroid}}}{T_s},\ \ f_2 = \frac{T_{\text{scan}}}{T_s},\ \ f_3 = \frac{T_{\text{merge}}}{T_s},\ f_1 + f_2 + f_3 = 1.$$

- 两计算阶段各平分到 8 线程，负载均衡效率记为 $\eta \in (0, 1]$。在第一阶段，是可以做到平均划分的，1024 是可以被 8 整除的，每个线程计算 128 个簇的质心和查询向量的距离；但是在第二个阶段，**容易出现负载不均衡的情况，无法确保每个簇的倒排记录数量完全相同**，因此这里会有一定的损失，在公式中使用 η 来衡量。
- 两次 `pthread_barrier_wait`，常数开销 $2B$，每个阶段结束后，所有线程需要在继续下一阶段前同步，确保所有线程都完成了当前阶段的工作。因此需要两次屏障同步：第一次在质心计算阶段结束后；第二次在扫描阶段结束后，进入合并阶段前。

$$T_p = \frac{f_1 T_s}{8} + \frac{f_2 T_s}{8\eta} + f_3 T_s + 2B$$

计算得到加速比为

$$S = \frac{T_s}{T_p} = \left(\frac{f_1}{8} + \frac{f_2}{8\eta} + f_3 + \frac{2B}{T_s}\right)^{-1} < 8$$

5. 进一步优化：Cache-Friendly Reordering

在之前的 IVF 实现中，倒排表 `invlists[c]` 记录的仍是“原始向量序号”，扫描 list 时需随机访问

$$\mathbf{v}_i = \texttt{base}[\,i\,] \in \mathbb{R}^d, \qquad i \in \texttt{invlists}[c],$$

导致大量 L1/L2 Cache 失效。为此，我在离线建索引阶段一次性重排底库向量，把同簇成员顺序搬到连续地址，从而把随机读优化为顺序读，提高负载效率 η，可以再降低一点 latency。

(四) pthread 并行优化（动态）

相比于前面介绍的静态线程池模型，动态线程模型采用了“按需创建、用后即毁”的线程管理策略。两者最关键的区别在于资源管理方式与线程生命周期:

1. 与静态线程池的区别

表 1: 动态线程与静态线程池对比

特性	**静态线程池**	**动态线程**
线程生命周期	程序启动时创建，一直存在直到程序结束	仅在并行计算阶段存在，计算完成后立即销毁
资源占用	持续占用系统资源，即使在空闲期	仅在计算期间占用资源，空闲时释放
任务分配	通常使用任务队列进行集中调度	直接分配任务给新创建的线程
适用场景	查询频繁且持续的场景	查询间隔较大或系统负载波动显著的场景

2. 动态线程实现细节

在本实现中，动态线程的核心代码体现在线程的创建和销毁过程：

```
// 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);
}
```

与静态线程池中预先创建线程并循环等待任务不同，动态线程模型中线程仅在 `ivf_search` 函数调用期间存在。每次查询时，系统会根据当前可用处理器核心数和查询参数 `nprobe` 动态确定所需线程数量：

```
int num_cores = sysconf(_SC_NPROCESSORS_ONLN);
int num_threads = std::min(num_cores, nprobe);
```

这种方式避免了潜在的线程过多导致的调度开销。

3. 并行加速分析

动态线程模型相比静态线程池通常具有以下性能特点：

- **优势**：更低的资源占用、更少的上下文切换、更适合突发性计算负载
- **劣势**：频繁的线程创建/销毁带来额外开销（线程创建/销毁开销很大），不适合高频短查询场景

在这个 IVF 搜索实现中，每个查询对 nprobe 个聚类中心内向量的距离计算构成了主要的计算任务。虽然动态线程模型在应对负载波动时展现出良好的灵活性，但在持续高频次的查询场景下，线程创建与销毁的累积开销相当大。在这种情况下，静态线程池模型可能因避免了重复的线程管理开销而表现出更高的效率，在实验中会详细分析这一情况。

（五） OpenMP 并行优化

1. 优化思路

OpenMP（Open Multi-Processing）是一个用于共享内存并行编程的 API，广泛支持于主流 C/C++ 和 Fortran 编译器。它通过编译指令（如 `#pragma omp parallel for`）实现线程级并行，极大地简化了并行程序的开发过程 [3]。

2. 关键代码

- **Step-1（质心距离）**采用 `#pragma omp parallel for schedule(static)`，每个线程独立计算一批质心的距离，无共享写入，零同步开销。

```
#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
```

- **Step-2（倒排表扫描）**选出的 n_{probe} 个倒排表长度有差异，使用`schedule(dynamic)`动态分配任务以缓解负载不均。为消除对全局 Top-k 堆的锁竞争，每个线程维护本地大顶堆，搜索阶段无锁；结束后再将各线程结果归并成全局堆。

```
#pragma omp parallel for schedule(dynamic)//倒排表扫描
    for (int p = 0; p < real_probe; ++p) {
        const int  tid      = omp_get_thread_num();
        auto&      heap     = local_heaps[tid];
        int list_id         = centroid_dists[p].second;
        const auto& invlist = g_ivf_index.invlists[list_id];
        for (int idx : invlist) {
            const float* vec = base + static_cast<size_t>(idx) * dim;
            float dist       = l2_distance(query, vec, static_cast<int>(dim));
            if (heap.size() < k) {
                heap.emplace(dist, idx);
            } else if (dist < heap.top().first) {
                heap.pop();
                heap.emplace(dist, idx);
            }
        }
    }
```

3. 进一步优化：OpenMP+SIMD

在上述的基础上，为了进一步压缩查询延迟，我在线程级并行（OpenMP）的同时引入 SIMD（neon）实现数据级并行，进一步压缩查询延迟。

向量化距离核：将标量版 `l2_distance` 替换为 SIMD 版本，以一次同时处理 16 个浮点数为例。

```
inline float l2_distance_neon(const float* a, const float* b, int dim)
{
    int i = 0;
    float32x4_t vsum = vdupq_n_f32(0.f);        // 4 lanes 累加
    // 每次处理 16 个 float
    for (; i + 15 < dim; i += 16)
    {
        float32x4_t va0 = vld1q_f32(a + i     );
        float32x4_t vb0 = vld1q_f32(b + i     );
        float32x4_t d0  = vsubq_f32(va0, vb0);
        vsum = vmlaq_f32(vsum, d0, d0);         // diff^2 累加
        //...省略后面三组
    }
    // 每次处理 4 个 float
    for (; i + 3 < dim; i += 4)
    {
        float32x4_t va = vld1q_f32(a + i);
        float32x4_t vb = vld1q_f32(b + i);
        float32x4_t d  = vsubq_f32(va, vb);
        vsum = vmlaq_f32(vsum, d, d);
    }
    // 水平求和 vsum
```

```
    float32x2_t vlow  = vget_low_f32(vsum);
    float32x2_t vhigh = vget_high_f32(vsum);
    float32x2_t vp    = vpadd_f32(vlow, vhigh);      // [0]+[1] , [2]+[3]
    float sum = vget_lane_f32(vp,0) + vget_lane_f32(vp,1);
    // 处理剩余元素
    for (; i < dim; ++i) {
        float diff = a[i] - b[i];
        sum += diff * diff;
    }
    return sum;
}
```

(六) 实验

1. 超参数探索：IVF 聚类数量 & 探索簇数

根据对 IVF 复杂度的分析可知，存在一组合适的参数组合 $(n_{\text{list}}, n_{\text{probe}})$ 使整体查询效率较优。首先我探究了在不同的聚类数 nlist 之下，nprobe 和 recall 的关系，使用的数据集的规模是 100K 的，设置 nlist 从 128 至 4096；nprobe 从 1 至 60。实验结果如图1所示。

- 对于固定的 nlist，随着 nprobe 的增加，召回率也随之提高: 搜索更多的聚类会增加找到相关结果的概率。
- 然而，这种提升并非线性，当 recall 达到一定值（0.9 左右）后，再线性提高 nprobe，召回率的增长会逐渐放缓，出现**边际收益递减的情况**，“膝点”大概就处于 $recall = 0.9$。
- 随着 nlist 的增加，需要搜索更多的簇数才能使 recall 保持理想水平：数据被划分得越细致，相关的结果可能分散在更多的聚类中，需要搜索更多的聚类才能找回。

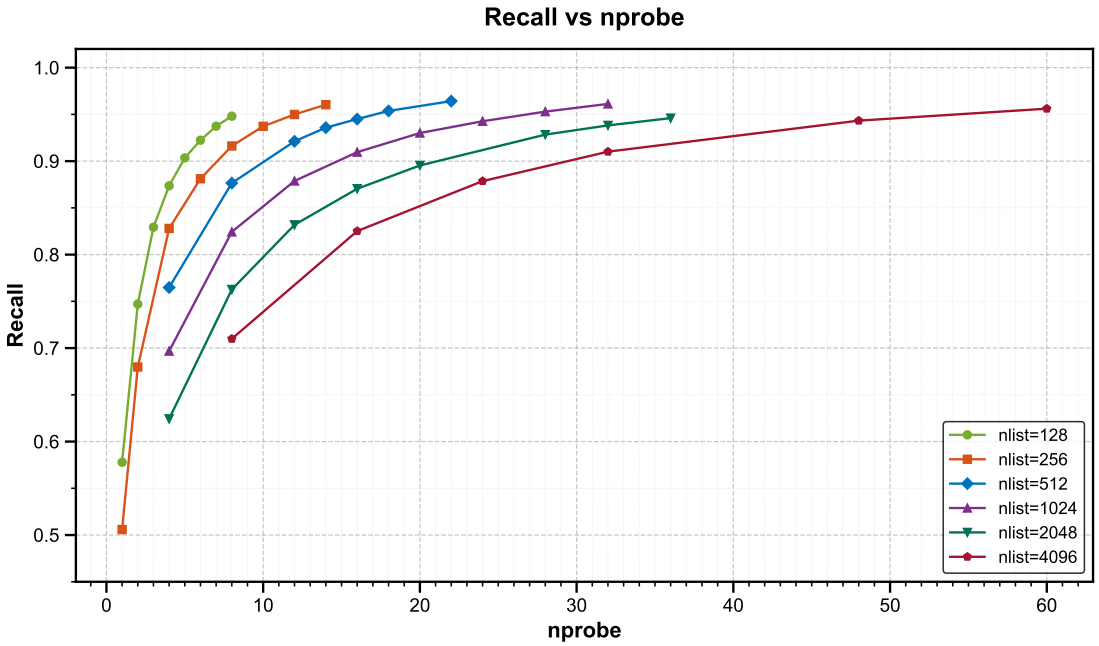

图 1: nlist、nprobe 与 recall 的关系

根据实验结果我发现 nprobe 和 latency 并非完全线性，latency 和 recall 才是衡量本算法超参数选择的根本标准，因此绘制了 latency-recall 帕累托前沿图，实验结果如图2所示。

图2清晰展示了在 ANNS 中，latency 和 recall 之间固有的权衡。通常，为了搜索得更全面、更准确（提高召回率），算法需要检查更多的数据或索引部分（对应于增加 nprobe），这自然会花费更多时间（增加 latency）。

- 我们希望尽量保持高召回率的同时降低查询延迟，这对应图中右下角的部分。这张图可以清晰的看出，nlist=1024 始终最靠近右下角，这是这一系列参数中的最优选择。
- nlist 并非越大越好，应该根据数据规模的大小来定。图中表明本实验使用的 100K 数据集最适合的聚类数量 nlist 为 1024 个，也就是数据规模的 1% 左右。
- 当 **recall 达到 0.90 之后，latency 的增长逐渐不可控，说明需要付出更多的时间代价才能换取少量 recall 的提升**，这与图1的结论一致。

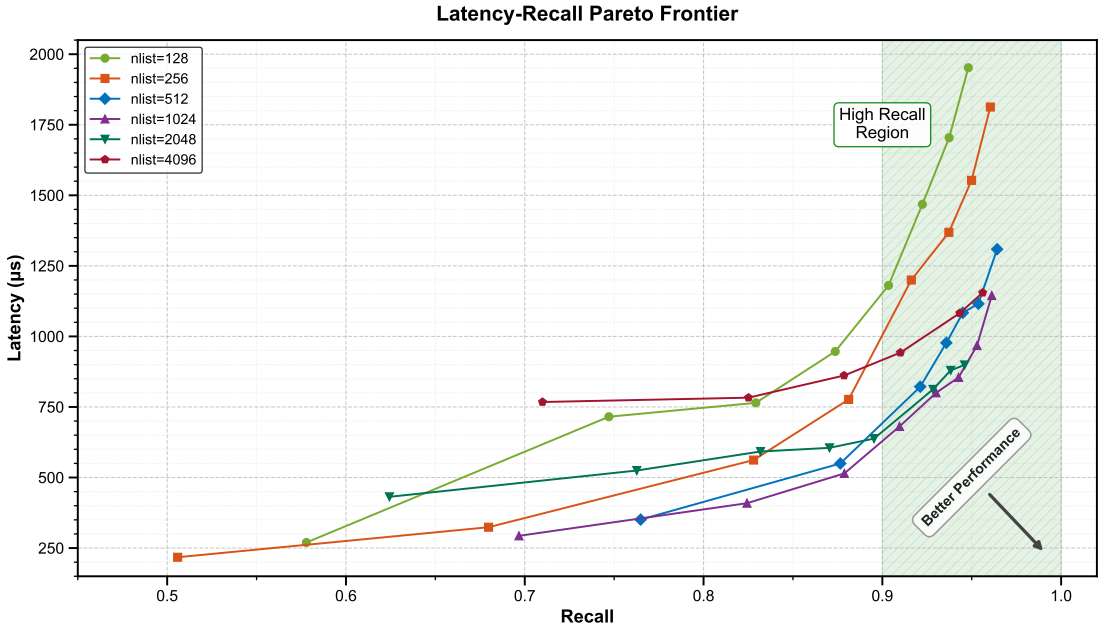

图 2: tradeoff：nlist, latency& recall

延续前几次实验的标准，将 recall 调整至 0.9，结合该实验得出的**聚类数 nlist=1024, nprobe=16，达到的 recall=0.9096**，之后的实验中，都以这个参数设置为准。

2. 查询延迟 & 加速比

为了系统地评估不同并行化方法对 IVF 查询延迟的影响，我设计并实现了以下几种优化方案, 实验结果如图3所示。

- **基准实现 (`flat_latency`)**: 采用串行方式执行查询过程，作为后续并行优化方案的性能对比基准。
- **基于 Pthreads 的并行化**:
  - `pthread_latency_dynamic`：动态线程版本 pthread。
  - `pthread_latency_static`：静态线程版本 pthread，根据实验指导书的建议，暂时将线程数设置为 8。

  - `pthread_latency_static_cache`：在验证上述两种实现之后，我已经知道静态线程在本实验中更有优势，因此在静态线程的基础上，通过对内存进行重排，将相同簇的向量存储在一起，降低查询时的 cache miss 率。

- **基于 OpenMP 的并行化**
  - `openMP`
  - `openMP_simd`: 在 OpenMP 并行化的基础上，结合使用 SIMD 指令。

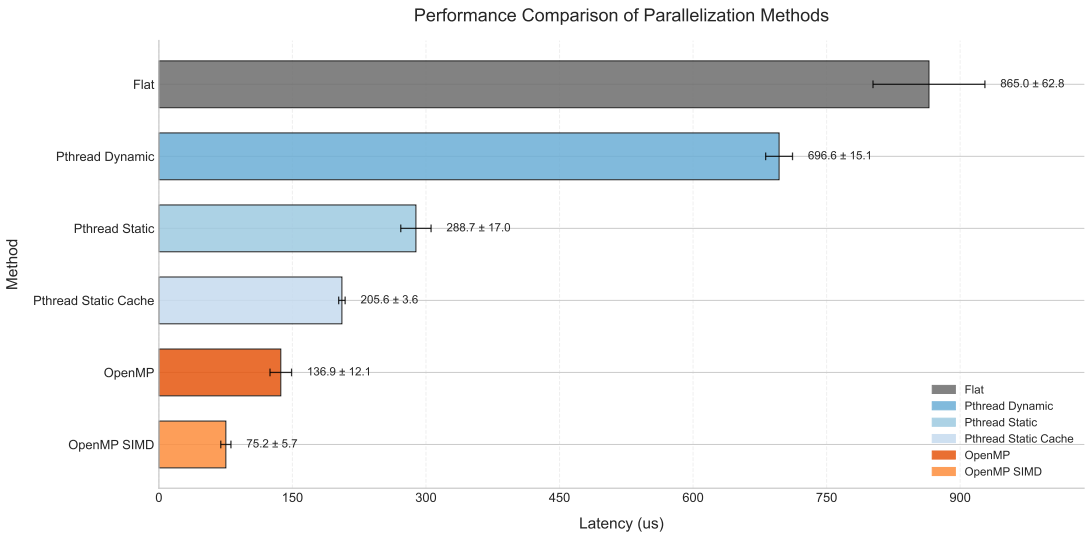

图 3: IVF 各策略查询延迟

- **线程调度方式：静态优于动态**，在本实验环境中，动态线程加速比为 1.24；换成静态划分后可直接提升至 3。
- **内存局部性是下一瓶颈：**在理论分析部分已经可知，静态 8 线程的加速比最高就是 8，但是由于一系列并行开销，实际无法达到这个加速比，在静态划分基础上进行向量重排，使其成为内存友好的索引，相比于内存不友好的版本，加速比提升至 **4**。
- **OpenMP 优势明显:** 利用线程池复用、编译器自动任务分配等技术，单纯 OpenMP 便达到 6.3× 加速，比手写 pthread + cache 优化仍快约 1.5×。
- **OpenMP 配合 SIMD 效果显著：**在 OpenMP 基础上显式向量化后（OpenMP SIMD），延迟几乎再降一半，表明核心算子仍具备可观的运算密度，可以通过数据并行进一步提高效率。*OpenMP + SIMD* 将查询延迟从 865 us 降至 75 us，**查询延迟降低至十微秒级**，达到约 **11.5 的加速比**。

3. VTune 分析——各策略关键指标

我将项目移植到 x86 平台, 利用 VTune 进行更细致的分析。首先我关注 Instructions Retired 和 CPI，分析结果如图4所示。

- 无论是 pthread 还是 OpenMP，**共享内存并行化后，Instructions Retired 都有增加**。这主要是由并行化本身引入的额外开销造成的，包括线程管理开销、同步开销、通信与数据

共享的间接开销等。注意到 OpenMP 的 Instructions Retired 比 pthread 少，这可能是因为 OpenMP 将底层的线程创建、同步和任务调度等复杂管理交给了其高度优化的运行时库来高效执行；相比之下，Pthreads 手动显式编写这些管理逻辑，这导致为实现并行管理功能而执行了更多的指令。

- pthread 和 OpenMP **并行化后 CPI 有所增加**。并且 SIMD 的加入会使 CPI 上升，这符合之前实验的结论。
- 虽然并行化后 Instructions Retired 和 CPI 都有升高，但是，由于**计算任务被有效分解并分配到多个线程**上同时执行，因此最终的查询延迟仍大幅降低。
- 在本例中，动态线程的版本 CPI 和指令总数都比静态版本高，最终耗时也显著高于静态版本，这表明 IVF 这种高频短查询场景对动态线程不利，频繁的线程创建和销毁占用了大量的资源。
- SIMD 是通过数据级并行减少了核心计算所需的指令数，可以和线程级并行有机搭配，降低查询延迟。

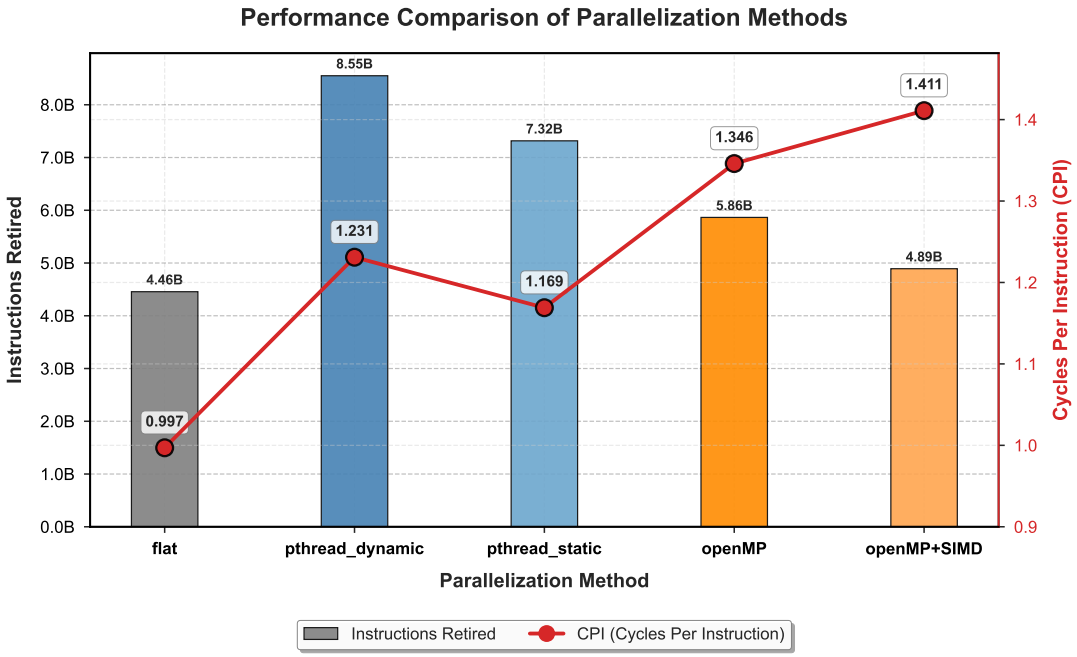

图 4: IVF 各策略 Instructions Retired & CPI

4. VTune 分析——负载均衡

在并行程序优化中，负载均衡是确保所有并行执行单元得到合理分配的工作量，以避免部分单元空闲而其他单元过载，从而最大化并行效率的关键因素。利用 VTune 工具分析不同并行策略在 IVF 查询任务中的负载均衡情况，结果如图5、6、7和8所示。

- 如图5所示，串行算法在执行期间只有一个活动线程，这是串行执行的典型特征，展示了单线程完成全部工作所需的时间模式。所有计算任务均由该线程独立完成。

- 图6展示了 pthread 版本的线程负载情况，可以看到 8 个线程参与了并行计算。然而，这些线程的活跃时长表现出显著的差异。一些线程的绿色条带明显较短，意味着它们较早地完成了分配给它们的任务并进入空闲或等待状态。而另一些线程则持续工作了更长时间。

  这揭示了 pthread 实现在当前 IVF 任务中存在**负载不均衡问题**。整体并行区域的执行时间取决于最后完成工作的线程。那些提前完成任务的线程所代表的 CPU 资源在后续时间内**未能得到充分利用**，导致了并行效率的损失。要想让 pthread 能有较均衡负载，可以考虑**任务池**的方法，如图7，负载相比静态线程版本更均衡些。

- 图8展示了 OpenMP 版本的线程负载。与图6形成鲜明对比，图中所有参与的 OpenMP 线程的绿色活跃条带长度非常接近。主线程**先**进行任务计算，然后创建一组线程计算，并且在相近的时间点结束工作，并行区域的结束边缘显得非常“整齐”。

  这表明 **OpenMP 在此 IVF 任务中实现了非常理想的的负载均衡**。各个线程都得到了相对平均的工作量，并且能够持续有效地利用 CPU 资源直到任务结束。

图 5: 串行算法线程负载表现

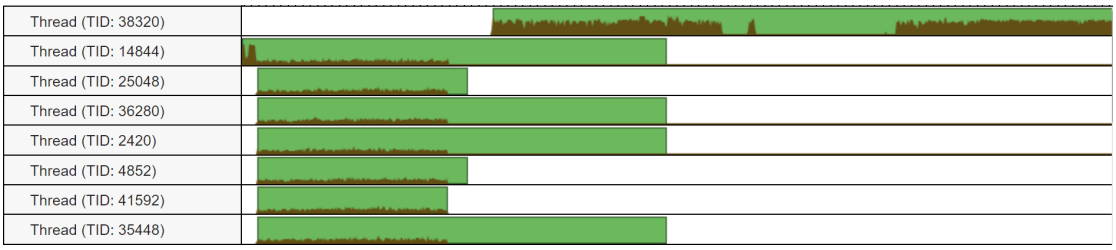

图 6: pthread 线程（静态）负载表现

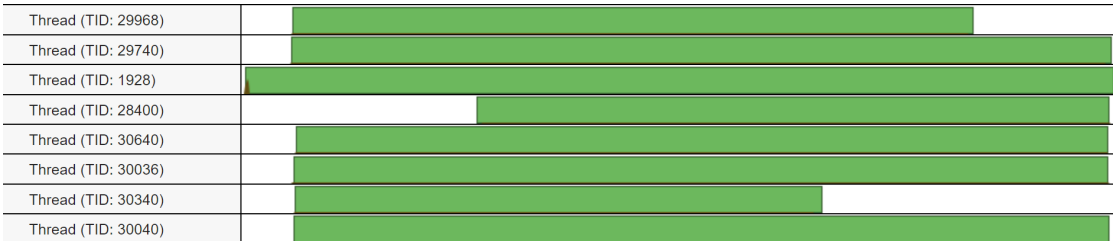

图 7: pthread 任务池版本（优化负载均衡）

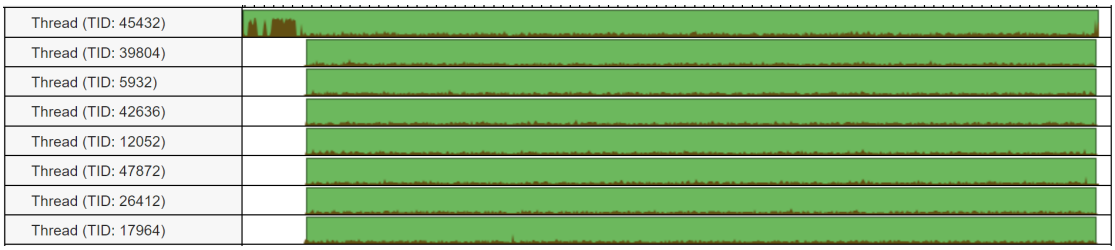

图 8: OpenMP 线程负载表现

5. Cache 优化效果——索引重排

在 pthread 静态线程的基础上，IVF 经过索引重新排布之后，查询延迟又降低 30%，图9展示了优化前后缓存命中的情况。

- 索引重排后，**各级缓存的命中率都有上升，总的访存次数有所下降**，这个优化对 cache 十分友好。
- L2/L3 改善最显著，**命中率翻倍**。索引重新排布后，相同查询批次的工作集被压缩至更小、连续的地址区间，使得中低层缓存可以长期命中。
- L1 本身已接近 100 %，提升空间有限，但 Miss 数仍减少约 5 倍，佐证了乱序索引带来的临界行逃逸问题得到有效缓解。

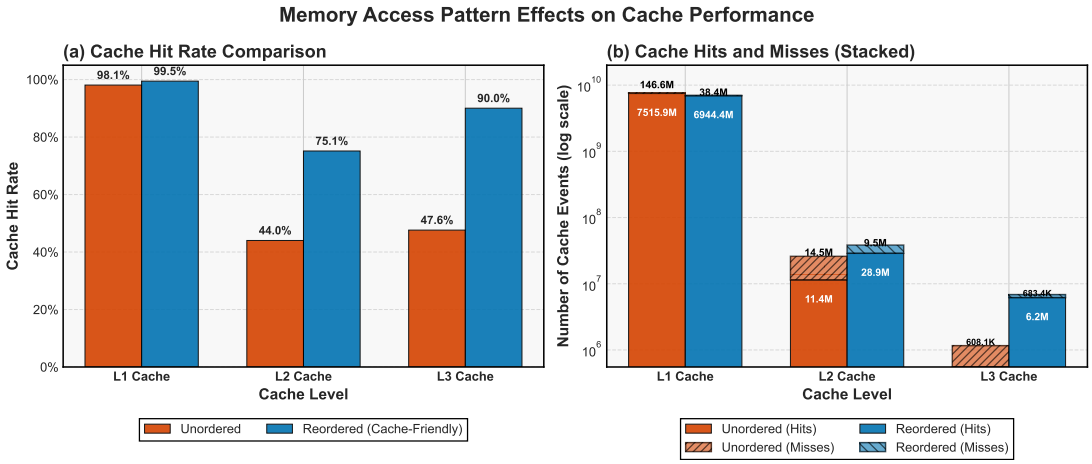

图 9: 索引重排优化 cache 性能

三、 IVF 与 PQ 结合

在之前的实验中我分别单独实现了 IVF 和 PQ [1]，并且实现了并行加速，这两个方法可以结合起来使用，也有两种实现方式：“先 IVF 再 PQ”和“先 PQ 再 IVF”。它们的原理在之前的实验中已经有介绍，下面主要介绍结合的方式。

(一) 原理

1. IVF-PQ

IVF+PQ 的典型流程如下：

1. **粗量化（Coarse Quantization）**：使用 KMeans 将所有向量 $\{\mathbf{x}_j\}_{j=1}^{N} \subset \mathbb{R}^d$ 聚类为 n_{list} 个簇，簇中心为 $\{\mathbf{c}_i\}_{i=1}^{n_{\text{list}}}$。

$$q(\mathbf{x}) = \arg\min_i \|\mathbf{x} - \mathbf{c}_i\|_2.$$

2. **残差编码（Residual Encoding）**：对每个数据库向量 $\mathbf{x}_j$，计算残差

$$\mathbf{r}_j = \mathbf{x}_j - \mathbf{c}_{q(\mathbf{x}_j)}.$$

3. **乘积量化（Product Quantization）**：将 d 维残差向量划分为 m 个子向量，每个子向量做 k_{sub} 个中心的聚类，得到码本 $\{\mathbf{u}_{i,\ell}\}$。每个残差被编码为

$$\text{PQ}(\mathbf{r}_j) = \big[c_{j,1}, c_{j,2}, \ldots, c_{j,m}\big],$$

其中 $c_{j,\ell} = \arg\min_u \|\mathbf{r}_j^{(\ell)} - \mathbf{u}_{\ell,u}\|_2$。

4. **查询（Search）**：对查询向量 $\mathbf{y}$，首先同样用粗量化器找到 n_{probe} 个最近簇，然后对每个簇中的编码残差通过查表计算近似距离：

$$\|\mathbf{y} - \mathbf{x}_j\|_2^2 \approx \|\mathbf{y} - \mathbf{c}_{q(\mathbf{y})}\|_2^2 + \sum_{\ell=1}^{m} d_\ell\big(\mathbf{y}^{(\ell)}, c_{j,\ell}\big),$$

其中

$$d_\ell\big(\mathbf{y}^{(\ell)}, c_{j,\ell}\big) = \|\mathbf{y}^{(\ell)} - \mathbf{u}_{\ell,c_{j,\ell}}\|_2^2.$$

PQ 方法需要使用精排来提高精度，可以调整 rerank 参数，使精度达到一定要求。

Algorithm 3 IVF+PQ 插入与查询伪代码

Input: 数据库 $\{\mathbf{x}_j\}_{j=1}^N$，参数 $n_{\text{list}}, n_{\text{probe}}, m, k_{\text{sub}}$

训练 CoarseQuantizer 和 PQ 码本
for 每个 $\mathbf{x}_j$ **do**
  $i \leftarrow q(\mathbf{x}_j)$
  $\mathbf{r}_j \leftarrow \mathbf{x}_j - \mathbf{c}_i$
  $c_{j,1}, \ldots, c_{j,m} \leftarrow \text{PQ}(\mathbf{r}_j)$
  将编码 $(i; c_{j,1}, \ldots, c_{j,m})$ 加入倒排表第 i 桶
end for
function SEARCH($\mathbf{y}$, K)
  找到 $\mathbf{y}$ 在 CoarseQuantizer 上的 n_{probe} 个最近簇索引集合 S
  候选列表 $\mathcal{C} \leftarrow \emptyset$
  for 每个簇 $i \in S$ **do**
    for 桶中每个编码 $(i; c_{j,1}, \ldots, c_{j,m})$ **do**
      通过查表估算 $\|\mathbf{y} - \mathbf{x}_j\|_2^2$
      将 (j, dist) 加入 $\mathcal{C}$
    end for
  end for
  按 dist 排序取前 K 返回
end function

2. PQ-IVF

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 $\text{PQ}(\mathbf{x}_j)$，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 $\text{PQ}(\mathbf{y})$，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，其有效性和实用性可能不如“先 IVF 再 PQ”。我在实验中也实现了这种方式，可以探究其性能。

(二) IVF 与 PQ 结合的优势

IVF 和 PQ 结合形成的 IVF-PQ 方法比单独使用任一方法都更加高效。

1. 单独方法的局限性

IVF 的局限：虽然 IVF 通过聚类快速缩小搜索空间，但仍需存储完整向量且距离计算开销与维度成正比，内存占用大且计算复杂度高。

PQ 的局限：PQ 通过子空间量化显著降低内存需求和距离计算复杂度，但需要遍历整个数据集，对大规模数据效率低下。

2. 结合后的优势

1. **多层次搜索策略**：IVF-PQ 采用" 粗搜索 + 精细搜索" 的两级策略，将搜索空间从 $O(N)$ 降低到 $O(N/n_{\text{list}} \cdot n_{\text{probe}})$，其中 $n_{\text{probe}} \ll n_{\text{list}}$。
2. **互补性能**：IVF 解决了 PQ 的全局搜索问题，PQ 解决了 IVF 的内存占用和距离计算问题，形成协同效应。
3. **残差编码提高精度**：对残差向量（原始向量减去聚类中心）进行 PQ 编码，使量化更精确，因为残差分布更均匀且范围更小。
4. **计算效率提升**：
   - 单独 IVF：$O((N/n_{\text{list}}) \cdot n_{\text{probe}} \cdot d)$ 次浮点运算
   - IVF-PQ：$O((N/n_{\text{list}}) \cdot n_{\text{probe}} \cdot m)$ 次表查询，且 $m \ll d$
5. **并行友好**：不同聚类的搜索可完全并行，距离表计算可利用 SIMD 指令加速，缓存命中率高。

IVF-PQ 通过结合两种方法的优势，实现了搜索空间缩减和高效距离计算的双重优化。但是二者结合的顺序会影响性能，这会在之后的实现详细分析。

(三) 并行化

1. pthread

通过对 IVF 的并行化实验，我已经知道，在本数据集的环境下，静态线程比动态线程更有优势，因此这里的 pthread 聚焦于静态线程的实现。pthread 并行化的思路与 IVF 部分类似，。

1. 对每个查询，选出 n_{probe} 个最近的 IVF 簇。
2. 创建线程池，将 n_{probe} 个簇的扫描任务**静态划分**为 $T = 8$ 份（7 条子线程 + 1 条主线程），每份负责 $\lceil n_{\text{probe}}/T \rceil$ 个簇，像这样：

$$\big[\, t\times C,\ (t+1)\times C\big), \quad C = \left\lceil \frac{n_{\text{probe}}}{T} \right\rceil$$

线程的创建如下，其余细节和 IVF 中的 pthread 静态线程类似：

```
class ThreadPool {
public:
    explicit ThreadPool(size_t n) : stop(false) {
        for (size_t i = 0; i < n; ++i)
```

```
            workers.emplace_back([this] {
                while (true) {
                    std::function<void()> task;
                    {   std::unique_lock<std::mutex> lk(queueMutex);
                        condition.wait(lk, [this]{return stop||!tasks.empty();});
                        if (stop && tasks.empty()) return;
                        task = std::move(tasks.front()); tasks.pop();
                    }
                    task();                                  // 执行任务
                }
            });
    }
    template<class F> void enqueue(F&& f) {
        { std::lock_guard<std::mutex> lk(queueMutex);
          tasks.emplace(std::forward<F>(f)); }
        condition.notify_one();
    }
    ~ThreadPool() {
        { std::lock_guard<std::mutex> lk(queueMutex); stop = true; }
        condition.notify_all();
        for (auto& t : workers) if (t.joinable()) t.join();
    }
private:
    std::vector<std::thread>              workers;
    std::queue<std::function<void()>>     tasks;
    std::mutex                            queueMutex;
    std::condition_variable               condition;
    bool                                  stop;
};
static ThreadPool g_thread_pool(7);       // 7 个工作线程
```

3. 线程各自完成:

   a) 计算查询残差 $\mathbf{q} - \mathbf{c}_{\text{list}}$;

   b) 构建长为 $M \times K$ 的 LUT（码本距离表）;

   c) 扫描倒排表，累积 Top--k 到线程私有的小根堆 `local_heap`;

   d) 不与其他线程共享任何可写数据 $\Rightarrow$ 无锁。

4. 主线程用 `condition_variable` 阻塞等待 completed_threads $= T - 1$，标志位通过原子加法更新。。

- **任务获取**: 工作者线程在 `condition.wait()` 中阻塞，只有当 `tasks` 非空或 `stop=true` 时被唤醒;
- **生命周期管理**: 析构函数将 `stop` 设为 `true`，广播唤醒所有线程并 `join`，确保无悬挂线程;

2. OpenMP

OpenMP 相比于 pthread 更方便，在 pthread 中我只对最耗时的部分做了并行化，而在 OpenMP 中我在更多的部分都做了并行化，此部分和 IVF 有较大不同，下面详细介绍:

1. 准备工作:获取最大线程数与线程局部存储,在并行区域开始之前,通过 `omp_get_max_threads()` 获取 OpenMP 运行时环境可用的最大线程数 `T`。该值用于预先初始化线程局部数据结构数组的大小。

2. 阶段一:并行计算查询向量与聚类中心的距离,使用 `#pragma omp parallel for schedule(static)` 指令并行化了对所有 `g_ivfpq_index.nlist` 个聚类中心距离的计算。`schedule(static)` 子句将迭代空间静态地、均匀地分配给线程组中的各个线程。每个线程独立计算一部分聚类中心的距离，并将结果（距离和聚类中心 ID）存入共享数组 `cent_dists` 的对应位置。

```
#pragma omp parallel for schedule(static)
for (int cid=0; cid<g_ivfpq_index.nlist; ++cid) {
    float d = l2_distance(query,
                          g_ivfpq_index.centroids[cid].data(),
                          static_cast<int>(dim));
    cent_dists[cid] = {d, cid};
}
```

3. 阶段二：并行扫描 `nprobe` 个倒排表，这是查询过程中最为耗时的部分。我创建了一个大的并行区域 (`#pragma omp parallel`)，在此区域内，每个线程首先分配其私有的工作缓冲区 `q_residual` (用于存储残差) 和 `lut` (距离查找表)。随后，使用 `#pragma omp for schedule(dynamic)` 指令将 `real_probe` 个倒排列表的处理任务动态分配给线程团队中的线程。

```
#pragma omp parallel
{
    std::vector<float> q_residual(dim);
    std::vector<float> lut(M * K); // M, K 为PQ参数

    #pragma omp for schedule(dynamic)
    for (int pi = 0; pi < real_probe; ++pi) {
        int list_id = probe_ids[pi];
        // ... (a) 计算残差 q_residual ...
        // ... (b) 构建 LUT lut ...
        // ... (c) 扫描倒排表，计算 adist ...
        // ... 更新 local_heaps[omp_get_thread_num()] ...
    }
}
```

4. 阶段三：合并局部堆

5. 阶段四：并行精排 (`rerank_k`，对 `approx_heap` 中选出的 `R` 个候选向量进行精确的 L2 距离计算。这一过程同样通过 `#pragma omp parallel for schedule(static)` 进行并行化。每个线程处理一部分候选向量，计算精确距离，并将结果存入其对应的线程局部精排堆 `local_rerank_heaps[omp_get_thread_num()]`。

```
// 假设 cand 向量已从 approx_heap 准备好，R 为实际精排数量
std::vector<std::priority_queue<DistIdx>> local_rerank_heaps(T);
#pragma omp parallel for schedule(static)
for (int i = 0; i < R; ++i) {
    int id = cand[i].second;
    // ... vec = base + (size_t)id * dim; (获取原始向量)
    // ... dist = l2_distance(query, vec, ...); (计算精确距离)
    auto& current_thread_heap = local_rerank_heaps[omp_get_thread_num()];
    // ... 更新 current_thread_heap ...
}
```

精排完成后，这些 `local_rerank_heaps` 同样需要合并到最终的结果堆 `final_heap` 中。

3. OpenMP+SIMD

在 OpenMP 的基础上，结合 neon 的 SIMD 指令可以继续优化。

- **显式 SIMD:** 我实现了一个专门的 L2 距离计算函数 `l2_neon`:

```
static inline float l2_neon(const float* a, const float* b, int dim)
{
    float32x4_t vacc = vdupq_n_f32(0.f); // 初始化累加器向量
    int i = 0;
    // 主循环：一次处理16个float（4个float32x4_t向量）
    for (; i + 15 < dim; i += 16) {
        float32x4_t a0 = vld1q_f32(a+i   ); /* Vector load */
        float32x4_t b0 = vld1q_f32(b+i   );
        // ...（类似加载 a1,b1, a2,b2, a3,b3）...
        float32x4_t d0 = vsubq_f32(a0,b0); /* Vector subtract */
        // ...（类似计算 d1, d2, d3）...
        vacc = vmlaq_f32(vacc,d0,d0); /* Vector multiply-accumulate: vacc += d0*
    d0 */
        // ...（类似累加 d1*d1, d2*d2, d3*d3）...
    }
    // 处理剩余不足16但大于等于4的元素
    for (; i + 3 < dim; i += 4) {
        float32x4_t da = vsubq_f32(vld1q_f32(a+i), vld1q_f32(b+i));
        vacc = vmlaq_f32(vacc, da, da);
    }
    // 水平加法（Horizontal sum）将向量累加器中的4个float值相加
    float32x2_t low = vadd_f32(vget_low_f32(vacc), vget_high_f32(vacc));
    float sum = vget_lane_f32(low,0) + vget_lane_f32(low,1);

    // 处理最后不足4个的剩余元素（标量处理）
    for (; i < dim; ++i) {
        float d = a[i] - b[i]; sum += d*d;
    }
    return sum;
}
```

此 `l2_neon` 函数被应用于以下几个关键的距离计算环节，均在 OpenMP 的并行线程内部被调用:

  - **质心距离计算（阶段一）:** 每个线程在计算查询向量与分配给它的部分粗聚类中心之间的距离时，调用 `l2_neon`。
  - **查找表 (LUT) 构建（阶段二）:** 在计算查询向量残差的子向量与子码本中各码字间的距离时，调用 `l2_neon`。
  - **精排（阶段四）:** 若启用重排序，计算候选向量与查询向量间的精确 L2 距离时，调用 `l2_neon`。

- **融合方式总结:** OpenMP 与 SIMD 的融合体现在一个层次化的并行模型中:

  - **线程级并行 (TLP):** OpenMP 通过 `#pragma omp parallel` 和 `#pragma omp for` 等指令，将宏观任务（例如，处理不同的倒排列表、不同的粗聚类中心或不同的重排序候选）分配给多个 CPU 核心上的不同线程执行。
  - **数据级并行 (DLP):** 在每一个 OpenMP 线程的执行路径内部，SIMD 技术被用来加速计算密集型的小循环。

(四) 实验

根据 IVF 实验的经验，IVF 部分的超参数设置为 nlist=1024，nprobe=16。默认查询精度仍然控制在 recall=0.9，PQ 部分的超参数将精度保持在 0.9 即可，前置实验探索发现聚类数量 16，子空间划分 16，rerank=110 就是一个性能较好的选择，之后的实验以此为准。

1. 查询延迟 & 加速比

如图10，可以直观看出，将 IVF 和 PQ 结合，可以有效降低查询延迟，IVF-PQ 相比于只用 IVF 的版本**快 100%**，证明结合的有效性。接下来的并行化都以 IVF-PQ 为例。

- **IVF 和 PQ 的结合顺序对性能有影响：** 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。我也对此有思考：
  - 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。这也解释了为何 PQ-IVF 在文献中出现的并不多。现在来看，IVF-PQ 是业界常用的高性能大规模向量检索方案。
- pthread 静态线程的加速比为 1.16，OpenMP 的加速比为 2.6。OpenMP 在本实验中相较于 Pthread 静态调度展现出更优性能，得益于更优的负载均衡机制，采用动态调度策略。这使得任务能够根据线程的实时完成情况动态分配，有效避免了 Pthread 静态划分任务时的负载不均衡问题，从而提高了 CPU 核心的整体利用率。
- SIMD 可以帮助 OpenMP 再获得 12% 的提升。

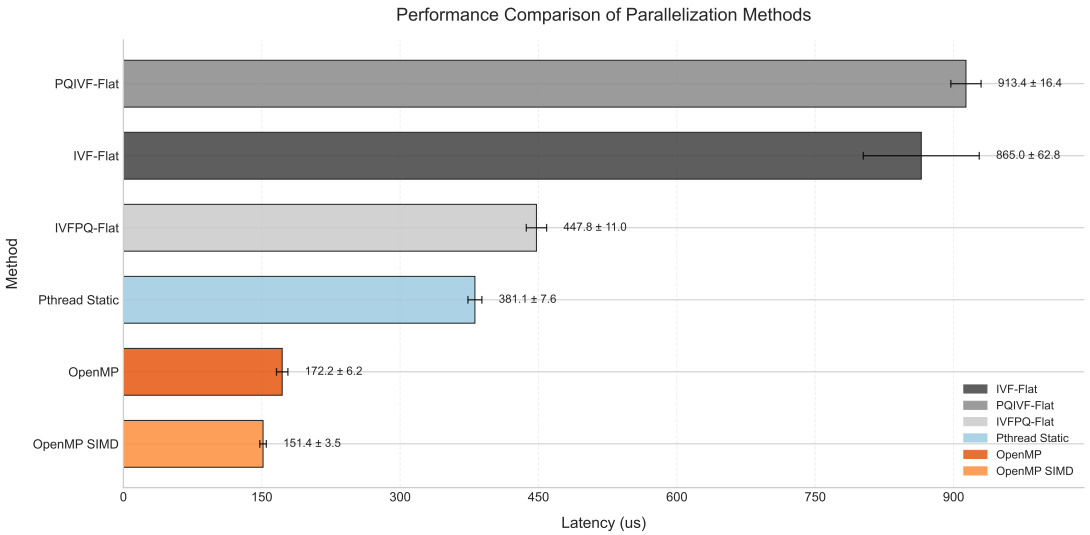

图 10: IVF-PQ 各查询策略延迟

2. 线程数对 pthread 的影响

线程数不是越多越好, 这实验手册中的原话, 因此我在这一部分研究了不同线程数量对 IVF-PQ 查询延迟的影响。**在服务器中，线程数从 2 增加到 8，查询延迟不断下降，似乎线程的增加是有利的**，由于服务器貌似不支持 8 线程以上的线程申请，这一部分在本地 x86 测试，结果如图11所示。

- **并非线程越多越好**：随着线程数从 2 增加到 8，查询延迟显著降低，在 8 个线程时，延迟达到最低点。这表明在此范围内，增加线程可以有效利用多核处理能力，提高并行度，从而缩短查询时间。
- 线程超过一定数量后，增加线程带来的并行化收益不足以抵消线程创建、调度、同步以及资源竞争（如 CPU 缓存、内存带宽等）所带来的开销，导致整体性能下降，且稳定性下降。

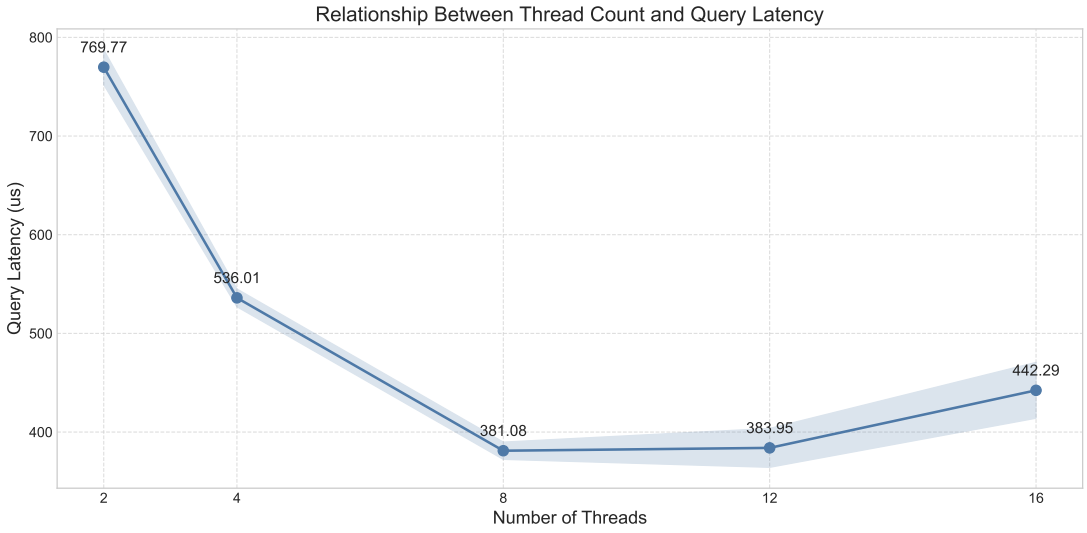

图 11: 线程数与性能的关系

四、 HNSW

(一) 原理

HNSW (Hierarchical Navigable Small World graphs) 主要是通过构建一个层次化的图结构来加速近似最近邻搜索，每一层都是一个“可导航的小世界网络”，这种结构使得从任意节点出发，通过贪心算法（总是走向离目标最近的邻居）可以快速地找到目标 [2]。在本次实验中，项目已经提供好了 hnswlib 项目包，也实现了索引构建和串行查询，因此接下来重点考虑并行化思路。

(二) 邻居距离并行化

在 HNSW 的搜索算法中，当程序迭代到一个新的当前节点 v_c 时，需要遍历其所有邻居 v，对于每个未曾访问过的邻居，都需要计算它与查询点 q 之间的距离 $D(v, q)$，以便决定是否将其加入候选队列 pq 和结果堆 H 。

利用 pthread/OpenMP 对这一邻居距离计算过程进行并行化处理。为了平衡并行化的开销（如线程创建、同步等）和收益，此处引入了一个并行化阈值（`PARALLEL_THRESHOLD`）。仅当待计

算距离的邻居数量超过此阈值时，才启用并行计算模式；若邻居数量较少，则继续采用传统的串行计算方式，以避免不必要的并行开销。

1. pthread

hnsw 也是“高频短查询”，根据之前实验的经验，使用静态线程版本的 pthread 更为合适，尽量避免频繁创建和销毁线程，关键代码和之前的静态 pthread 类似，此处不再展示，仅列出关键步骤。

- **线程池初始化与销毁**：在程序启动时，通过 initialize_static_thread_pool 函数创建并初始化一组固定数量的工作线程。
- **工作线程函数**：每个工作线程执行一个循环函数（worker_thread_function）。在循环中，线程使用条件变量（cond_task_available）和互斥锁（task_mutex）等待新任务的分配。一旦被唤醒，线程会检查是否有分配给自己的任务以及是否收到关闭信号。
- **任务参数传递与分配**：当需要并行计算邻居距离时，主线程分配函数，负责将邻居列表的计算任务划分为若干子任务。每个子任务的参数（如处理的索引范围、数据指针等）被存储在一个共享的结构体数组中，每个工作线程对应数组中的一个元素。主线程通过互斥锁保护对这些共享数据的写入，并通过条件变量广播通知所有工作线程任务已就绪。
- **同步机制**：使用 pthread_barrier_t 类型的屏障（task_completion_barrier）来实现主线程与工作线程之间以及工作线程相互之间的同步。只有当所有参与当前计算的工作线程和主线程都到达屏障时，它们才能继续执行。

2. OpenMP

OpenMP 的并行比较好实现，使用 #pragma omp parallel for schedule(dynamic) 对这部分进行优化，schedule(dynamic) 可以在不同邻居距离计算耗时略有差异时提供一定的负载均衡。

```
const size_t PARALLEL_THRESHOLD = 8;
//unvisited_neighbors 是一个包含未访问邻居ID的容器，data_point 是查询点的数据指针
//dist_func_param_ 是距离函数的参数
std::vector<dist_t> distances(unvisited_neighbors.size());
if (unvisited_neighbors.size() > PARALLEL_THRESHOLD) {
    // 并行计算未访问过的邻居与查询点的距离
    #pragma omp parallel for schedule(dynamic)
    for(size_t i = 0; i < unvisited_neighbors.size(); i++) {
        tableint candidate_id = unvisited_neighbors[i];
        // 假设 getDataByInternalId 是线程安全的或无副作用的
        char *currObj1 = getDataByInternalId(candidate_id);
        // 假设 fstdistfunc_ 是线程安全的或可重入的
        distances[i] = fstdistfunc_(data_point, currObj1, dist_func_param_);
    }
} else {
    // 串行计算距离
    for(size_t i = 0; i < unvisited_neighbors.size(); i++) {
        tableint candidate_id = unvisited_neighbors[i];
        char *currObj1 = getDataByInternalId(candidate_id);
        distances[i] = fstdistfunc_(data_point, currObj1, dist_func_param_);
    }
}
```

3. 预期效果与讨论

这种并行化**对 HNSW 大概率是负优化**，因为 HNSW 的图结构导致邻居节点访问高度随机，多线程并行会造成严重的 cache miss 和内存带宽竞争；同时，向量距离计算本身很快，而每层访问的邻居数量有限（通常 16 个以下），**线程创建和同步的开销往往超过计算收益**；更关键的是，HNSW 搜索的贪心特性使得图遍历具有**串行依赖性**，真正的瓶颈在于遍历过程而非单次距离计算。

(三) IVF+HNSW

IVF+HNSW 是一种混合索引策略，它首先利用 IVF 的特性将数据集划分为若干簇，通过查询点与簇质心的比较，快速筛选出 nprobe 个最相关的候选簇。随后，在这些选定的簇内部，利用为每个簇单独构建的 HNSW 图索引进行高效的局部搜索。

既然 HNSW 本身难以进行并行化，那么就保留其串行执行，引入 IVF，在这一步进行并行化，之前的实验已经证明对 IVF 的并行化很高效。对于 IVF 的并行化已经介绍过，此处省略。

(四) 实验

1. 查询延迟 & 加速比

为了和之前的实验保持一致，依然通过参数的调整使 recall=0.90。实验结果如图12所示。

- **高效的 HNSW**：此方法的串行版本查询延迟就在 150 微秒左右，这已经是本实验中 IVF、IVF-PQ 通过并行优化得比较好的水平了。图索引相当高效，但是对其并行加速并不容易。
- **直接并行化邻居距离计算是负优化**：无论是使用 pthread 还是 OpenMP 实现的邻居距离并行化，其性能均劣于 flat 基线，表明这种并行化策略**引入的并行开销超过了并行带来的收益**，导致了性能下降。OpenMP 出色的负载均衡和任务分配使其好于 pthread。但是整体上仍然是负优化，这种并行方式行不通。
- **IVF+HNSW 可以有效提高查询效率（正优化）**：在使用 OpenMP 进行并行化的情况下，终于使查询延迟降低，加速比达到 1.37。
- **SIMD 和 OpenMP 配合良好（优化 IVF+HNSW）**：加速比达到 1.63，查询延迟稳定在 94 微秒，这让查询延迟进入**十微秒级**。

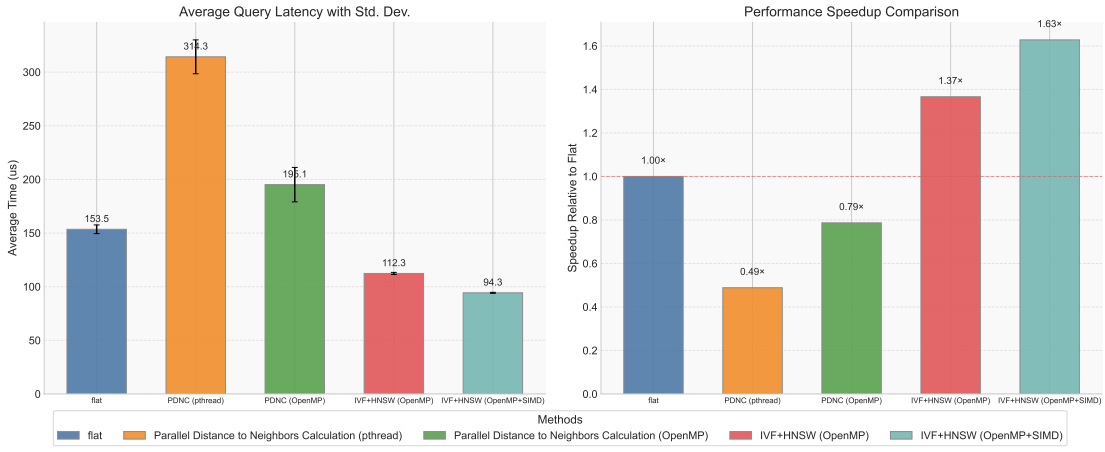

图 12: 各策略优化 HNSW 查询延迟 & 加速比

2. VTune 分析——邻居距离并行负优化的原因

如表2所示，与串行版本相比，Pthread 和 OpenMP 并行版本都引入了**巨大的额外开销**，具体表现为总执行指令数分别增加了约 3.7 倍和 2.2 倍，同时每指令周期数（CPI）也从 1.051 分别上升到了 1.421 和 1.502。即使有多个线程参与运算，这些额外的指令开销（源于线程管理、数据处理和同步）以及 CPI 的升高共同抵消了并行计算的潜在收益，导致负优化。

表 2: 邻居距离并行 Instructions Retired & CPI

Method	Instructions Retired	CPI
flat	1429600000	1.051
pthread	5296300000	1.421
OpenMP	3170746300	1.502

五、 总结

本次实验围绕高维向量近似最近邻搜索（ANNS）任务，重点实现并优化了 IVF、IVF-PQ、HNSW 等索引算法，并采用 pthread 和 OpenMP 进行并行加速。通过系统设计和实验对比，显著提升了查询效率，验证了并行编程在大规模搜索场景下的实际价值。

在并行化方面，OpenMP 在绝大多数场景下比 pthread 更有优势，通过线程池复用和动态调度提升负载均衡效率的同时，简单的并行过程使得 OpenMP 即高效又便捷；结合 SIMD 指令后，**IVF 查询延迟降至 70 us（Recall=0.90）**。

在算法层面，IVF 与 PQ 的结合兼顾速度与精度，串行状态下明显优于单独使用任一方法；但是，**IVF-PQ 的参数需要进一步讨论，精排的数量不宜太多，否则并行加速的效果不如 IVF**，而 HNSW 结构不适合深度并行，容易出现并行负优化，和 IVF 结合之后可以实现并行加速。图13总结了本次实验三个方法极致优化下（OpenMP+SIMD）的 latency-recall trade-off。

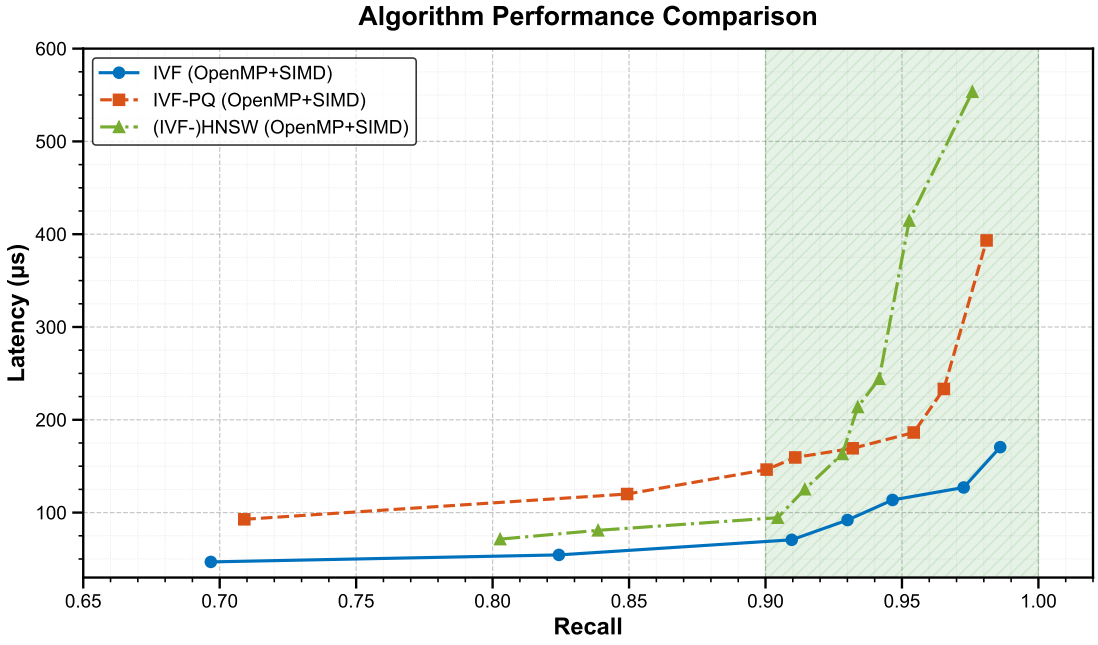

图 13: IVF，IVF-PQ，HNSW 极致优化（OpenMP+SIMD）latency-recall trade-off

本次实验已经将查询延迟降低至 100us 以内（Recall=0.90），我有一下启发和思考：

- **并行不是万能的：并行化效果依赖任务结构与硬件特征**，像 IVF 查询这种可划分为多个独立子任务的结构非常适合并行化，OpenMP 和 pthread 都能显著加速；但 HNSW 的贪心搜索过程具有较强的**串行依赖**，盲目并行反而造成性能下降。
- **综合运用多种优化方法：**就目前而言，线程级并行带来的提升比之前都大，但是 cache 优化，数据并行 SIMD 等方法还可以在此基础上进一步提升。关键是**理解任务本质，找到瓶颈**，做对应的优化。

跳转至：源代码地址，主要结构如下：

类别	**文件**

IVF 相关代码	`ivf_flat.h`
	`ivf_pthread_static.h`
	`ivf_pthread_static_cache.h`
	`ivf_pthread_dynamic.h`
	`ivf_openMP.h`
	`ivf_openMP_simd.h`
	`main_ivf_ncache.cc`
	`main_ivf_cache.cc`

IVF-PQ 相关代码	`ivfpq.h`
	`ivfpq_pthread.h`
	`ivfpq_openmp.h`
	`ivfpq_openmp_simd.h`
	`main_pqivf.cc`
	`main_ivfpq.cc`

IVF-PQ-x86	包含 IVF 和 IVF-PQ 的 x86 版本

HNSW 相关代码	`ivf+hnsw.h`
	`hnswlib/`（第三方库）
	`main.cc`

HNSW-x86	包含 HNSW 的 x86 版本

参考文献

[1] 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, 2011.

[2] Yu 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.

[3] 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, 5 2025. 并行程序设计 Lab3_2，使用 OpenMP 多线程编程.

[4] 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, 5 2025. 并行程序设计 Lab3_1，使用 Pthread 多线程编程.