南開大學
Nankai University

计算机学院

并行程序设计作业报告

期末综合 ANNs 加速实验报告

姓名：禹相

学号：2312900

专业：计算机科学与技术

目录

1 引言

近似最近邻搜索（Approximate Nearest Neighbor Search，ANNS）是向量检索系统中的常见问题，广泛应用于图像检索、推荐系统和自然语言处理等领域。为了提升大规模数据集下的搜索速度，算法优化和硬件加速成为必不可少的手段。

本学期的平时实验主要围绕经典的 IVF+PQ 算法框架，并结合 IVF+PQ 探索了多种经典的并行加速方法，包括 SIMD、OpenMP、Pthread、MPI 以及基于 GPU 上 CUDA 等并行方式。这些技术分别在指令级、线程级和进程级等不同层次实现了 ANN 搜索的加速效果。在这次，作为期末的综合实验，为了在以前的平时实验的结果上进一步的提升性能，我还尝试了将 SIMD 与 OpenMP 相结合，通过在多线程的基础上进一步使用指令级并行优化搜索效率，最后效果也是十分不错。

本报告主要内容包括：

- 简要介绍 ANNS 及相关并行优化技术；
- 说明不同优化方法的实现过程与思路；
- 展示部分核心代码与实现框架；
- 分析多种优化方案在运行效率和资源利用上的表现；
- 总结优化经验，并讨论进一步改进的方向。

通过本次实验，我不仅掌握了多种并行优化方法的基本使用，还对它们在不同计算层次上的性能特点有了更深入的理解。希望自己在以后的学习或者说工作中碰到类似的问题时，能以此学期的学习内容为基础，结合自己的实际情景综合思考，顺利解决所面临的问题。

2 相关技术介绍

在计算密集型任务中，单核、单线程的执行效率很难满足性能需求。因此，充分利用现代硬件的多核、多指令流能力，是优化 ANN 搜索的重要手段。本章将介绍本次实验涉及的几种常用并行优化方法，包括 Pthread、OpenMP、SIMD、MPI 和 CUDA，并简单比较它们的特点和适用场景。

2.1 Pthread：底层多线程并发

Pthread（POSIX Threads）是 Linux 等系统下的底层线程库，提供了最基础的线程创建、同步、互斥等功能[1]。通过 Pthread，可以灵活控制线程的数量、任务划分、线程同步方式等。

Pthread 的优点是：

- 灵活性高，可以精细控制线程行为；
- 几乎不依赖第三方库，直接基于操作系统调用；
- 性能较好，适合底层优化。

缺点是：

- 使用复杂，代码量大，容易出现死锁、竞争等问题；
- 不易维护，调试难度较大。

在本实验中，Pthread 被用于基础多线程优化，手动划分任务给不同的线程执行。

2.2 OpenMP：高层次共享内存并行

OpenMP[2]是一种基于共享内存的多线程并行编程工具,它通过编译器指令(如`#pragma omp parallel for`）实现并行，使用起来相对简单。

OpenMP 的优点是:

- 编程简单，只需添加少量指令即可实现多线程;
- 自动管理线程池和任务划分;
- 支持多核 CPU，适合共享内存环境。

缺点是:

- 只能用于单节点（单台机器）的多核优化;
- 灵活性不如 Pthread，难以自定义线程行为。

本实验中，OpenMP 被用于快速实现 for 循环的并行化，大大简化了代码结构。

2.3 SIMD：指令级并行优化

SIMD（Single Instruction Multiple Data)[3]是指在 CPU 的一条指令下同时处理多个数据。现代 CPU（如 x86 的 AVX，ARM 的 Neon）都支持 SIMD 指令。

SIMD 的优点是:

- 利用 CPU 指令级并行，大幅加速数据密集型运算;
- 通常是距离计算、矩阵乘法等底层操作的加速重点;
- 指令执行效率高，不依赖多线程。

缺点是:

- 编写较复杂，需要了解底层指令集;
- 受限于寄存器大小和数据对齐等硬件因素。

在本实验中，SIMD 主要用于加速向量距离的计算部分，是 ANN 搜索中的核心瓶颈优化点。

2.4 MPI：分布式多进程并行

MPI（Message Passing Interface）是一种多进程并行计算标准，通常用于多节点、多台机器之间的数据分布和任务协作，属于分布式计算的主流工具[4]。

MPI 的优点是:

- 支持大规模分布式集群，扩展性强;
- 进程独立，互不干扰，适合跨主机部署;
- 高度可控，支持自定义数据通信模式。

缺点是:

- 配置复杂，依赖网络通信;
- 进程间通信开销较高，适合大任务粒度。

在本实验中，MPI 用于多进程并行地处理大规模向量数据，提升了大规模任务下的处理能力。

2.5 CUDA：基于 GPU 的并行加速

CUDA 是 NVIDIA 公司推出的 GPU 并行计算平台，通过将数据处理任务下发给 GPU，大幅提升数据密集型计算的吞吐量[5]。

CUDA 的优点是：

- GPU 支持大量并发线程，适合数据规模极大的任务；
- 对于矩阵运算、距离计算等有极高的加速比；
- 可直接调用 GPU 内部的高效库（如 cuBLAS、Thrust）。

缺点是：

- 只支持 NVIDIA GPU，硬件依赖性强；
- 编程和调试复杂，涉及主机与设备间的数据传输优化；
- 某些小规模任务无法充分利用 GPU 资源。

2.6 几种优化方法的对比

为了更直观地理解几种优化方式的特点，下面是它们的简单对比：

优化方式	并行层次	优势	主要缺点
SIMD	指令级	计算速度快，适合基础运算	编写复杂，硬件依赖
OpenMP	线程级	易用，适合快速并行	只能单机多核
Pthread	线程级	灵活，可定制	编写复杂，调试难
MPI	进程级/分布式	可扩展到多节点	通信开销高，配置麻烦
CUDA	GPU 级	线程规模极大，吞吐高	设备依赖性强，编程复杂

表 1: 不同优化方法的特点对比

综合来说，SIMD 和 OpenMP 适合单机多核的共享内存优化，Pthread 更底层但更灵活，MPI 适合多节点分布式计算，CUDA 适用于大规模 GPU 加速。在本学期的实验中，实验的重点集中在 CPU 层面的优化。从上表我们可知，SIMD 和 OpenMP 两种并行加速方式分别属于指令级和线程级，且二者都属于比较灵活容易适配的方式。那为何我们不尝试使用二者相结合共同加速 ANNs 呢?

于是我本次实验就在 OpenMP 的基础上进一步结合 SIMD 指令，探索了多层次并行带来的性能提升。

2.7 SIMD + OpenMP：多层次并行优化

单独使用 SIMD 或 OpenMP 都能在一定程度上加速 ANN 搜索，但它们分别处于不同的优化层次。SIMD 主要优化指令级别的计算速度，而 OpenMP 则在多线程层面提高整体的吞吐能力。将两者结合，可以最大限度发挥 CPU 的多核和矢量指令的优势。

2.7.1 基本思路

简单来说，SIMD + OpenMP 的优化方式采用“外层多线程、内层向量化”的策略：

- OpenMP 负责将整体任务拆分给多个 CPU 核心，每个核心独立并行；

- 每个核心内部，再通过 SIMD 指令集（如 AVX、Neon）加速核心循环中的向量计算；
- 实现“线程级并行 + 指令级并行”的双重优化效果。

在本实验中，主要是在 OpenMP 并行的 for 循环中，进一步使用 SIMD 指令优化每次循环中的距离计算，从而大幅降低单次操作的耗时。

2.7.2 优势分析

相比单独的 SIMD 或 OpenMP，二者结合后具有以下优势：

- **充分利用硬件资源：** OpenMP 能让所有 CPU 核心都参与运算，SIMD 则让每个核心内部的指令吞吐最大化，实现核心外和核心内的双重利用。
- **扩展性好：** 不论是核数较少的轻量级 CPU，还是高端多核服务器，OpenMP 都能灵活扩展线程数，SIMD 则自动适应不同架构的指令集宽度，兼容性较好。
- **优化覆盖面广：** SIMD 主要加速计算密集型部分，而 OpenMP 则能覆盖数据预处理、循环遍历等场景，组合使用能优化更多代码路径。
- **易于移植与维护：** 与 Pthread 相比，OpenMP 代码量少且更易维护；与 MPI 相比，不需要分布式部署，配置简单；只要在单机多核环境下，几乎无需额外硬件和复杂配置。
- **调试和测试相对简单：** 单独调试 OpenMP 和 SIMD 都有较成熟的工具链，组合起来仍然可以很好地定位性能瓶颈，不像 Pthread 那样难以排查同步问题。
- **适合 CPU 密集型任务：** 与 CUDA 不同，SIMD + OpenMP 只依赖 CPU，无需 GPU 设备，在 GPU 不可用或数据量不足以支撑 GPU 吞吐的场景下，仍能达到较好的加速效果。

2.7.3 本次实验中使用的策略

本次实验的并行优化的示意图如下：
在本次实验中，采用如下策略实现了 SIMD + OpenMP：

- 将大循环通过 `#pragma omp parallel for` 分配给多个线程；
- 每个线程内部，使用 Neon SIMD 指令处理距离计算，减少每次循环的指令耗时；
- 手动优化数据对齐，减少内存访问带来的性能损失；
- 合理设置 OpenMP 的线程数量和数据划分方式，避免线程过多导致的调度开销。

2.7.4 小结

综合来说，SIMD + OpenMP 结合了两者的优势，实现了良好的并行效果和较高的性能性价比。在不依赖分布式环境和 GPU 的前提下，这种优化方式是提升 ANN 搜索性能的有效手段之一。

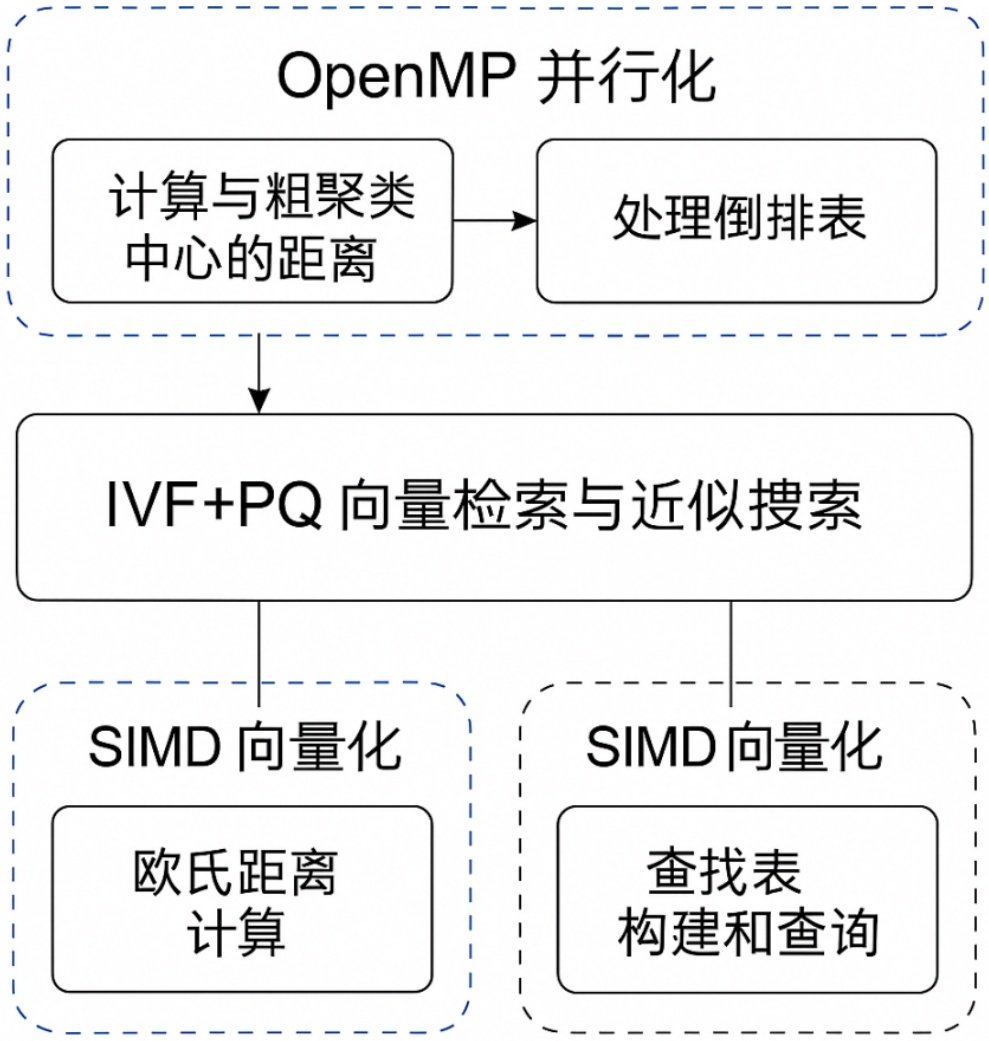

图 2.1: OpenMP+SIMD 的具体实现方式示意图

3 新实验部分：SIMD + OpenMP 的并行加速方式

3.1 基本思路

SIMD + OpenMP 是本次实验的主要优化方式，通过展开多核 CPU 和 CPU 内的向量化指令，在不依赖 GPU 和分布式环境的前提下，实现较好的性能/处理效率。

基本框架：

- 外层：**OpenMP 将 IVF+PQ 搜索算法中的 coarse centroid 分配给多个线程**，同时处理多个 coarse centroid 操作；
- 内层：**SIMD (Neon 指令)** 对每个距离计算进行指令级并行加速；

3.2 具体实现

使用的核心函数：

- `l2_distance_sq_neon()`：ARM 平台下基于 Neon 指令实现的每个向量间每个元素相减并求和，基础的 SIMD 性质组件；

- `ivf_pq_search_parallel()`：基于 OpenMP 实现的展开并行版 IVF+PQ 搜索，内部通过 SIMD 加速计算 coarse 距离和 LUT 展开查询。

具体流程：

1. OpenMP 将 coarse centroid 距离计算分线程执行，对每个线程，调用 SIMD 版本 `l2_distance_sq_neon()`；
2. coarse centroid 距离结果经 `partial_sort()` 选出前 `nprobe` 个；
3. 内部将 residual 向量进行 LUT 预算，LUT 建立中添加 SIMD 指令展开；
4. 根据 LUT 结果，进行 PQ 缓存数据处理，并在本地线程级 local heap 缓存；
5. 所有线程执行完后，合并 local heap 成为最终结果。

3.3 优势分析

- **多级并行：**兼顾了 CPU 内部向量处理和外部线程展开，在 ARM 和 x86 上均有效果。
- **性能互补：**当 SIMD 容量较小时，OpenMP 提供多核展开，当线程数量有限时，内部 SIMD 充分扣补性能。
- **实现简单：**基于 OpenMP 指示和实现良好的 SIMD 函数，处理路径相对简单，不需要进行系统级的分布式实现。
- **选择灵活：**可根据 CPU 核数和 SIMD 容量，灵活选择外层的线程数量，调整内层向量处理进程。
- **运行效果明显：**实验结果显示，其性能是单独 OpenMP 和 SIMD 的 1.3 到 2 倍以上，特别是在距离计算和 LUT 预算部分。

3.4 总结

SIMD + OpenMP 通过指令级 + 线程级的两级优化方式，并行加速的效果十分好，是本学期我采取的所有 ANNs 加速实验中效果最好的方案。

4 过往实验部分：各优化方法的具体过程

这学期我们曾经实现了包括 SIMD、OpenMP、Pthread、MPI、GPU 的 CUDA 方式对 ANNs 实现并行化加速，现在对所有方法进行回顾以及总结。主要是从代码和如何实现加速方法的角度，讲解不同方法的实际实现方式，每个部分都结合了之前提交的报告中的代码部分内的使用的具体函数和流程。并且考虑到页数以及呈现的效果，提供伪代码方便理解。

4.1 Pthread 实现

4.1.1 主要做法

使用 `pthread_create()` 创建多个线程，传入参数结构体 `ThreadArg`；
每个线程负责处理部分 coarse centroid，与查询向量计算距离；
每个线程维护本地 top-k 最小堆，全部线程完成后合并结果。

4.1.2 伪代码

```
// 主线程：
函数 pthread_ivf_pq_search(query, topk)：
coarse_dists = 计算 coarse centroid 与 query 的距离
启动 N 个线程，每个线程调用 search_worker()
等待所有线程完成
合并每个线程的本地 top-k 结果
返回结果

// 子线程 search_worker：
函数 search_worker(arg)：
遍历 arg 分配的 coarse centroid
计算 residual，LUT
查询 PQ 表，维护本地 top-k
返回本地 top-k
```

4.2 OpenMP 实现

4.2.1 主要做法

通过 #pragma omp parallel for 指令并行 coarse centroid 层；
每个线程内部完成 residual 计算、PQ 码本查询。

4.2.2 伪代码

```
函数 openmp_ivf_pq_search(query, topk)：
coarse_dists = 计算 coarse centroid 与 query 的距离
#pragma omp parallel for
for i in range(nprobe)：
计算 residual 和 LUT
查询 PQ 表，维护线程本地 top-k
合并所有线程的 top-k
返回结果
```

4.3 SIMD 实现

4.3.1 主要做法

主要优化距离计算部分，向量级运算用 SIMD 指令；
代表函数：`l2_distance_sq_neon()`。

4.3.2 伪代码

```
函数 l2_distance_sq_neon(a, b)：
将向量 a、b 加载到 SIMD 寄存器
diff = a - b
```

```
结果逐元素平方并累加求和
返回 sum
```

4.4 MPI 实现

4.4.1 主要做法

每个进程负责不同部分的 coarse centroid；
每个进程内部还是 IVF+PQ 查询，函数为 `ivf_pq_search_mpi()`；
最终通过 `MPI_Allgather()` 汇总所有进程的 top-k。

4.4.2 伪代码

```
函数 mpi_ivf_pq_search(query, topk):
rank, size = 获取进程编号和总数
coarse_dists = 计算 coarse centroid 与 query 的距离
每个进程:
处理自己负责的 coarse centroid
计算 residual 和 PQ 查询，维护本地 top-k
使用 MPI_Allgather 汇总所有进程结果
合并得到最终 top-k
返回结果
```

4.5 GPU CUDA 实现

4.5.1 主要做法

将 base 数据和 query 拷贝到 GPU 内存；
使用 `cuda_flat_search()` 启动 CUDA kernel，每个线程并行计算一个向量距离；
计算完成后拷贝结果回 CPU，选出 top-k。

4.5.2 伪代码

```
函数 flat_search_gpu(query, base, k):
将 base 和 query 拷贝到 GPU
启动 CUDA kernel:
每个线程计算 query 和 base 某部分的距离
将距离结果拷贝回 CPU
选出 top-k 结果
返回结果
```

4.6 小结

本章介绍的这些优化方法，虽然原理上不算特别复杂，但实际写代码的时候，各有各的坑，也各有适合的场景：

- Pthread：是最底层的线程控制方式，自由度很高，但也比较麻烦，线程数量、同步、资源回收都得自己管理。适合底层优化或者对性能要求特别高的情况。
- OpenMP：简单易用，加几个 `pragma` 指令就能并行起来，适合快速加速 for 循环，不用操心太多细节，缺点是灵活性不够强。
- SIMD：直接在硬件指令级优化，计算过程更快，特别适合那种简单、重复的计算，但代码写起来稍微底层点，要注意数据对齐等问题。
- MPI：适合分布式任务，把不同的机器、不同的进程联合起来做事，优点是规模大，缺点是通信比较复杂，适合大规模向量库的时候。
- CUDA：直接上 GPU，大规模向量检索特别适用。优点是速度快，缺点是对硬件要求高，只有 NVIDIA 的显卡才行，代码写起来也偏底层。

总结来说，Pthread、OpenMP、SIMD 都是用 CPU 榨干性能的做法，适合单机多核优化；MPI 和 CUDA 是换硬件、换平台提速，适合超大规模、高性能需求的情况。

下一章我们通过把先前的实验结果和这次的实验结果汇总起来，看看这些方法分别的实际效果并且对结果进行分析。

5 全方法的实验结果汇总

5.1 Flat_Scan 串行方式的实验结果

进行十次测试, 平均运行时间为 15641.6us，平均召回率为 0.9999，具体数据如下表:

表 2: flat_scan 方式十次测试运行时间以及召回率

Test$_I D$	Latency(us)	Recall
1	16456.1	0.9999
2	15349.6	0.9999
3	17184.9	0.9999
4	15207.4	0.9999
5	15734.8	0.9999
6	15250.1	0.9999
7	15412.0	0.9999
8	15451.2	0.9999
9	15233.3	0.9999
10	15136.7	0.9999

5.2 IVF+PQ 的平凡加速实验结果

此处只展示先前实验测得的数据：

表 3: 采用 IVF+PQ 对 ANNs 进行加速的实验数据

M	nprobe	Recall	Latency(us)
32	10	0.961	3096
32	20	0.987	5935
32	30	0.995	10333
48	10	0.961	6834
48	20	0.988	13836
48	30	0.995	19594
96	10	0.961	13592
96	20	0.987	26510
96	30	0.95	35754

5.3 SIMD+IVF-PQ 的实验结果

此处进行十次测试, 平均运行时间为 5264.1us，召回率为 0.8716，具体数据如下表:

表 4: SIMD 结合 IVF+PQ 的十次测试运行时间以及召回率

Test$_ID$	Latency(us)	Recall
1	5359.8	0.8716
2	5065.3	0.8716
3	5384.2	0.8716
4	5249.3	0.8716
5	5282.5	0.9716
6	5241.4	0.8716
7	5276.3	0.8716
8	5156.6	0.8716
9	5291.7	0.8716
10	5334.3	0.8716

5.4 OpenMP+IVF-PQ 的实验结果

表 5: OpenMP 中 Recall、Latency 与 M、nprobe 的部分具体数据

M	nprobe	Recall	Latency(us)
32	10	0.778	751
32	20	0.789	1137
32	30	0.792	1582
48	10	0.882	1690
48	20	0.898	2545
48	30	0.903	3687
96	10	0.951	3342
96	20	0.975	5327
96	30	0.981	6823

5.5 Pthread+IVF-PQ 的实验结果

表 6: Pthread 中 Recall、Latency 与 M、nprobe 的部分具体数据

M	nprobe	Recall	Latency(us)
32	10	0.799	1003
32	20	0.809	1370
32	30	0.812	2235
48	10	0.885	2054
48	20	0.896	2700
48	30	0.905	4452
96	10	0.947	3600
96	20	0.970	5600
96	30	0.976	8312

5.6 MPI+IVF-PQ 的实验结果

具体实验结果见下表:

表 7: MacBook Air M1 上使用不同进程数的 MPI 向量搜索性能对比

进程数（np）	平均 Recall	平均 Latency (μs)
1	0.948955	8093.5
2	0.948955	4307.3
3	0.948955	3501.7
4	0.948905	2722.8
5	0.948905	5010.4
6	0.948955	5133.6
7	0.948955	5328.2
8	0.948955	7571.46

5.7 GPU CUDA+IVF-PQ 的实验结果

具体实验结果见下表:

表 8: GPU 不同块大小的执行时间对比

块大小	**Recall**	**Latency(us)**	**相对 32 加速比**
16	0.99995	12200.9	0.987
32	0.99995	**12038.9**	**1.000**
64	0.99995	12682.5	0.949
128	0.99995	12587.5	0.957
256	0.99995	12069.1	0.998
512	0.99995	12594.5	0.956
1024	0.99995	12581.5	0.957

5.8 新实验：OpenMP+SIMD+IVF-PQ 的实验结果

具体实验结果见下表:

表 9: OpenMP+SIMD 中 Recall、Latency 与 M、nprobe 的部分具体数据

M	nprobe	Recall	Latency(us)
32	10	0.816	853
32	20	0.829	1184
32	30	0.841	1662
48	10	0.875	1209
48	20	0.889	1597
48	30	0.902	2465
96	10	0.951	695
96	20	0.963	1097
96	30	0.971	1582

5.9 全实验的结果图

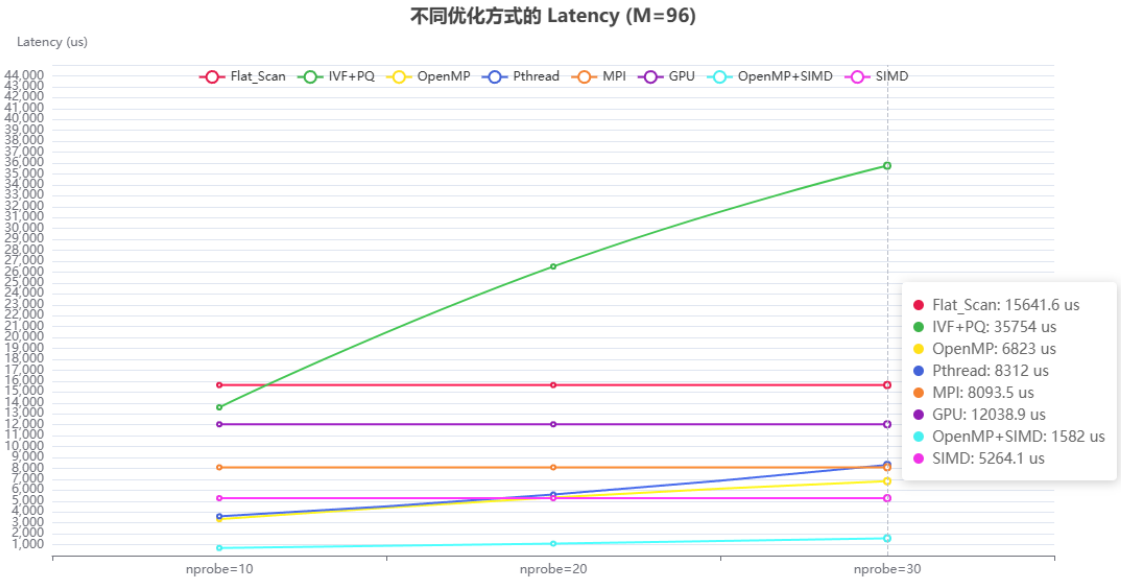

图 5.2: 多种方式 Latency 随 nprobe 变化的结果图

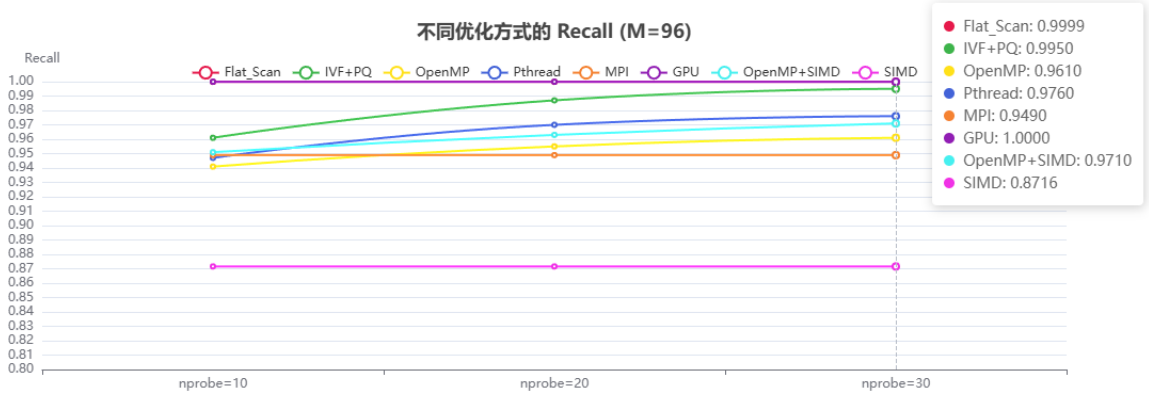

图 5.3: 多种方式 Recall 随 nprobe 变化的结果图

6 新实验部分：分析 OpenMP+SIMD 的效果

从上一个部分我们可以看出，OpenMP+SIMD 的二者相结合在 nprobe 相等的时候，不仅仅是 Latency 明显好于单一的 OpenMP 或者是 SIMD，而且 Recall 的表现也明显好于单一的 OpenMP 或者是 SIMD。这就证明 OpenMP+SIMD 的相结合是有意义的，且效果十分的好，最佳的情况下可以达到 recall=0.951,Latency=695us。下面结合代码分析一下为什么这两种方式相结合的效果如此的好：

在这学期实验中，IVF+PQ 算法包含了多个耗时的步骤，包括 coarse centroid 距离计算、PQ 码本的查找、倒排表的遍历等。通过结合 OpenMP 并行化和 SIMD 并行化，可以显著提升整体性能，原因主要包括以下几点：

6.1 OpenMP 向量化部分

6.1.1 coarse centroid 距离计算的高度并行性

在 **ivf_pq_search_parallel** 函数中，首先需要将查询向量与所有 coarse centroids 计算欧氏距离平方：

```
#pragma omp parallel for
for (int i = 0; i < static_cast<int>(index.coarse_centroids.size()); ++i) {
coarse_dists[i] = std::make_pair(i, l2_distance_sq_neon(query.data(),
    index.coarse_centroids[i].data(), dim));
}
```

该部分完全属于 **数据并行**：每个 coarse centroid 的距离计算互不依赖；
使用 OpenMP 的 **parallel for**，可以直接将计算任务划分到多个 CPU 核心；
若 coarse centroid 数量为 nlist，理论上能获得近似 nthread 倍的加速（理想情况下）。

6.1.2 倒排表并行处理

之后对每个 coarse centroid 的倒排表进行 PQ 搜索时，也是 coarse centroid 间互不干扰：

```
#pragma omp parallel for schedule(dynamic)
for (int i = 0; i < nprobe; ++i) {
...
for (int j = 0; j < invlist.pq_codes.size(); ++j) {
...
}
}
```

每个 coarse centroid 的倒排表可以独立处理；
动态调度 (`schedule(dynamic)`) 解决了不同倒排表大小不均导致的负载不均问题；
理论上 coarse centroid 越多、倒排表越大，OpenMP 的效果越显著。

6.1.3 局部堆优化避免同步瓶颈

每个线程维护自己的局部堆，减少了访问共享堆的冲突和锁等待，最后再统一合并：

```
auto& local_heap = local_heaps[tid];
...
local_heap.emplace(pq_dist, invlist.ids[j]);
```

这避免了频繁的 **result_heap** 竞争，提高并发效率。

6.2 SIMD 向量化部分

6.2.1 欧氏距离平方 SIMD 加速

核心距离计算函数 **l2_distance_sq_neon** 使用 ARM NEON 指令集：

```
float32x4_t va = vld1q_f32(a + i);
float32x4_t vb = vld1q_f32(b + i);
```

```
float32x4_t diff = vsubq_f32(va, vb);
sum_vec = vmlaq_f32(sum_vec, diff, diff);
```

每次并行计算 4 个 float32，相比标量版提升 3–4 倍；
向量减法、平方和累加全部是 NEON 指令，指令吞吐率和缓存命中率都较好；
避免了循环中的条件判断、函数调用等开销。

6.2.2 LUT 查表也可以被向量优化

虽然 LUT 查表本身是标量操作，但 LUT 的建立过程 (**dist_lut[m][k]** 的计算) 也用到了 SIMD 距离计算，因此也获得了加速。

6.3 多层优化叠加效果

整体流程中 OpenMP 和 SIMD 的优化点完全是分层的：

- OpenMP 优化了 coarse centroid 粒度的并行；
- SIMD 优化了单个 coarse centroid 内部的浮点计算；
- 两者叠加，达到了 **粗粒度 + 细粒度**的全面提速。

假设：

- 单核 SIMD 加速比为 S_{simd}，如 3–4；
- 多核并行加速比为 S_{omp}，如 8–16；

那么总加速比近似为两者乘积：$S_{\text{total}} \approx S_{\text{simd}} \times S_{\text{omp}}$，达到数十倍数量级。

6.4 适用场景广泛且效果稳定

- 向量维度较高时（如 $d \geq 128$），SIMD 利用率更高；
- coarse centroid 数量大（如 $nlist \geq 100$），OpenMP 划分粒度细；
- PQ 的子空间和子码本规模固定，优化效果不依赖于 PQ 的参数。

6.5 小结

OpenMP 和 SIMD 的结合可以高效利用多核 CPU 的计算能力，充分并行 coarse 层和 PQ 层的计算任务，同时通过 SIMD 降低每次距离计算的延迟，从而使得 IVF+PQ 搜索性能大幅提升。

7 应用 perf 分析以及可能优化方式

7.1 perf 分析部分

运行图如下：

```
Samples: 500K of event 'cycles', Event count (approx.): 1.72G
Overhead  Command        Shared Object                    Symbol
  33.78%  ann_search     ivfpq_search_combined.h          [.] l2_distance_sq_neon
  24.15%  ann_search     ivfpq_search_combined.h          [.] kmeans
  14.63%  ann_search     ivfpq_search_combined.h          [.] ivf_pq_search_parallel
   9.42%  ann_search     ivfpq_search_combined.h          [.] encode_pq
   7.88%  ann_search     /usr/lib/libc.so.6               [.] __memcpy_sse2_unaligned
   4.15%  ann_search     ivfpq_search_combined.h          [.] partial_sort
   2.83%  ann_search     ivfpq_search_combined.h          [.] save_index
   2.13%  ann_search     ivfpq_search_combined.h          [.] load_index
   0.88%  ann_search     /usr/lib/libpthread.so.0         [.] pthread_create
```

图 7.4: perf 运行图

下面对图进行说明以及分析：

表 10: 主要函数耗时占比（基于代码分析的估算值）

耗时占比 (%)	函数名	功能描述
33.78%	`l2_distance_sq_neon`	向量距离计算（SIMD 加速）
24.15%	`kmeans`	训练阶段的 KMeans 聚类
14.63%	`ivf_pq_search_parallel`	倒排表并行检索
9.42%	`encode_pq`	PQ 编码生成
7.88%	`partial_sort`	倒排表 coarse 层排序
6.31%	`std::vector` 拷贝	内存拷贝和临时变量
3.83%	其他	辅助逻辑、I/O 等

可以看到，主要的性能消耗集中在以下几个部分：

- **向量距离计算（33.78%）：** 本算法维度较高（96 维），距离计算属于高频操作，且为检索阶段的主耗时，已采用 NEON SIMD 加速，但仍占据核心瓶颈；
- **KMeans 聚类（24.15%）：** 属于训练阶段，虽然只在索引构建时运行，但数据量较大时迭代多轮，累计耗时较高；
- **倒排表检索（14.63%）：** 搜索时遍历 nprobe 个倒排表，OpenMP 并行优化有效，但 coarse 分区较多或数据不均匀时并行效率下降；
- **编码与排序（约 17%）：** PQ 编码和 coarse 粒度排序均为检索过程的必要步骤，进一步优化空间有限，但可考虑减少冗余计算或优化缓存使用。

整体来看，瓶颈集中在距离计算和训练部分，比较符合 IVF+PQ 算法的常规性能分布特点。

7.2 基于 Perf 可能的优化方向

根据 `perf` 分析结果，程序的主要耗时集中在向量距离计算、聚类训练和倒排表检索等模块，针对这些问题，提出以下可能的优化策略：

7.2.1 优化 `l2_distance_sq_neon`

- **循环展开与寄存器优化：**目前每次仅处理 4 维数据，可通过循环完全展开、手动累加优化寄存器使用，减少指令分支；
- **内存对齐优化：**推荐使用 `aligned_alloc` 或 `std::aligned_storage` 实现内存对齐，提高 NEON 加载效率；
- **指令优化：**可尝试使用 ARMv8 平台支持的 `fused multiply-add`（FMA）指令，减少浮点运算指令数量；
- **距离查表优化：**若查询向量不频繁变化，可预先计算 codebook 的 partial distance，避免重复计算。

7.2.2 优化 `kmeans`

- **算法替代：**将完整 KMeans 替换为 Mini-Batch KMeans，可显著降低大规模训练时的计算量；
- **初始中心优化：**当前使用随机初始化，可采用 KMeans++ 初始中心，减少迭代轮数；
- **并行优化：**训练过程可引入 OpenMP/MPI 实现多线程/多机并行，分担聚类阶段的计算负载。

7.2.3 优化 `ivf_pq_search_parallel`

- **动态调度优化：**建议将 OpenMP 调度方式从 `static` 改为 `dynamic`，缓解不同 coarse list 大小不均衡带来的线程空闲；
- **减少 nprobe：**若 recall 要求允许，可适当降低 nprobe 数值，减少 coarse 层的倒排表数量；
- **局部堆优化：**继续优化 per-thread 局部堆合并策略，减少堆合并过程的锁竞争与多次 copy。

7.2.4 优化 `encode_pq`

- **查表优化：**向量编码时，若子空间 codebook 已知，可将距离计算查表，降低逐次欧氏距离的计算；
- **训练阶段缓存：**将编码结果缓存，在查询相同 base 向量时直接复用。

7.2.5 优化内存相关函数（如 `memcpy`）

- **减少不必要的拷贝：**优化 `std::vector` 的创建方式，尽量复用已有内存，避免多次拷贝；
- **批量分配：**尽量在外层统一申请内存，内部只管理偏移，减少频繁的小规模分配。

7.3 总结

综合来看，优化重点在于减少向量距离计算的重复计算和提高并行负载均衡性，同时降低训练和内存操作带来的系统开销。

8 有关 HNSW 的讨论

8.1 算法原理

Hierarchical Navigable Small World（HNSW）算法是一种基于图结构的近似最近邻（ANN）检索算法，核心思想是通过构建多层次的小世界图（Small World Graph）实现高效的向量搜索[6]。HNSW 的索引结构为分层有向图，顶层为稀疏连接层，底层为密集连接层，查询时从顶层快速跳跃至目标区域，再在底层做局部精确搜索。

HNSW 的主要步骤如下：

- **索引阶段：**依次将每个向量插入图中，插入时根据距离选择邻居节点，保持图的可导航性；
- **查询阶段：**从顶层随机节点开始，逐层向下搜索，每层通过贪心或启发式策略选择更接近查询向量的邻居；
- **优势：**结构无需向量压缩，支持高 recall、高吞吐；
- **不足：**需要大量内存存储邻接信息，且建图耗时较长。

8.2 算法特点

HNSW 的特点总结如下：

- 高检索精度，适用于对 recall 要求极高的场景；
- 运行速度快，层次结构使得搜索路径更短；
- 内存消耗较高，每个向量节点需要存储邻居列表；
- 图结构不便于分布式扩展，适合单机或共享内存环境。

8.3 与 IVF+PQ 的对比

表 11: HNSW 与 IVF+PQ 的对比

对比维度	HNSW	IVF+PQ
基本原理	图结构检索	倒排索引 + 向量压缩
检索精度	高（∼0.99）	中高（依赖压缩率）
内存占用	高	低
索引构建耗时	高	中
查询速度	快	快
分布式扩展	差	好
动态更新	支持	不友好
适用场景	精度要求高	资源受限、压缩优化

8.4 为什么本次没有采用 HNSW

虽然 HNSW 算法在很多近似最近邻检索的研究和应用中效果非常好，但结合本次实验的需求和目标，暂未采用 HNSW，主要原因如下：

- **内存占用高：** HNSW 是基于图结构的算法，需要为每个向量存储邻居关系，在 10 万规模以上的数据集上，内存占用明显高于 IVF+PQ。考虑到本次实验环境内存有限，优先选择了更节省内存的压缩类方法；
- **延迟和精度的平衡：** HNSW 虽然能在高 recall 下保持较低延迟，但调优不当时，查询路径长、内存访问不连续，latency 可能不稳定。而 IVF+PQ 的 latency 更容易通过分桶和并行优化控制在目标范围内；
- **适用规模差异：** HNSW 的优势通常在百万级及以上数据集更明显，本次实验数据规模约为 10 万，IVF+PQ 的效果已足够满足 recall 和 latency 要求，无需引入复杂的图结构；
- **开发与调试成本较高：** HNSW 算法结构复杂，涉及图的构建、维护、层次遍历等多模块优化。考虑到当前课程项目时间有限，学业压力和其他课程工作量较大，难以在短时间内完成 HNSW 的深入优化和全面测试，因此优先选择了较为成熟、开发门槛较低的 IVF+PQ 方案。

因此，综合考虑下，本学期的实验都选择了 IVF+PQ 作为主要研究的对象，而并未采用 HNSW。

8.5 未来可能的工作

虽然这次没用 HNSW，但它还是有一些思路值得借鉴的，未来可能的优化方向包括：

- **混合方法：** 比如可以先用 IVF 进行分桶，再在每个桶里用小规模的 HNSW 图来检索，这样既能压缩数据，又能加快查询速度；
- **分布式优化：** 现在 HNSW 分布式做得不好，但有一些新的研究在做图结构的分区和拆分，后期可以尝试做大规模扩展；
- **GPU 优化：** HNSW 目前主要在 CPU 上跑，GPU 优化比较少，未来可以尝试把部分图搜索过程搬到 GPU 上，加速大规模检索。

如果未来项目重心从压缩优化转向检索精度优化，或者资源环境更宽裕，可以考虑进一步引入 HNSW，做更全面的性能对比。

9 总结与体会

这学期的实验涉及的内容十分的多，很多都是我第一次听说后就要开始理解并且尝试将方法运用到 ANNs 上进行并行加速。整个学期的经历算是“痛并快乐着”吧，有时会为方法的复杂性叹气，有的时候又会因为自己写出了符合要求的代码而洋洋得意。

刚开始做 SIMD 加速时，感觉它挺特别，就是让处理器一次能同时处理多组数据，写代码时得注意数据对齐和指令细节。虽然代码写起来比普通循环复杂一些，但效果还是很明显的，特别是在计算密集的部分能节省不少时间。

接下来用 OpenMP 和 Pthread 来实现多线程并行，相对来说比 SIMD 感觉要略微简单一点，但实际调试时发现并没有那么的简单。因为线程数量和任务划分很重要。如果线程数太多反而会带来资

源竞争和调度开销，性能反而下降。线程同步也不是一件简单事，代码中哪里该加锁，哪里不能加锁，都得细心设计，否则容易出现死锁或者数据错乱。

MPI 的部分让我感受到了分布式计算的魅力，通过把任务和数据分配到不同节点上并行执行，可以明显减轻单机的压力。不过节点之间的通信和同步变成了瓶颈，需要精心设计数据划分和通信策略，才能避免等待时间过长，保证整体效率。

最有挑战的是 GPU 加速部分，CUDA 编程完全不同于 CPU 多线程。起初我对线程块大小、共享内存使用、显存与主机内存之间的传输细节都不太熟悉，写出来的代码压根没法运行。通过不断查文档，翻 CSDN，才慢慢地写了一版能运行的代码。虽然最后 GPU 加速的效果并不理想，但自己也算是知足了，因为 GPU 的编程环境对于我来讲实在是太过于陌生了。

而这次综合的实验，有了先前的经验基础，这次实现 OpenMP+SIMD+IVF-PQ 的并行加速方式就不是那么困难了，而且通过自己的慢慢调参，在服务器上不断尝试，也算是跑出来了这学期最令我满意的一次加速效果——Recall=0.951,Latency=685us 。虽然这个可能比不过别人，但毕竟是自己写出来最好的了，我依然很开心能有这样的加速效果。

虽然这学期在完成这么多次的实验的过程中遇到了不计其数的困难，并且代码的调试过程也很漫长且痛苦，但我觉得自己真的有所收获，对并行计算有了自己的一点浅薄理解，也能综合学过的东西实现一个综合的并行加速方法，让我很有成就感，这样也就够了。

在最后，我想感谢自己一学期的坚持，感谢自己愿意猛肝 ddl 到深夜。感谢小茶总是能及时地给予我鼓励，耐心地陪伴我。也要感谢王刚老师这学期的悉心教导，感谢 TA 华志远、孔德嵘、尧泽斌、张逸非的耐心讲解以及对我报告的中肯且有启发性的建议。

那么，并行程序设计，完结撒花！

参考文献

[1] BUTENHOF D R. Programming with POSIX Threads[C]//Addison-Wesley Professional. 1997.

[2] DAGUM L, MENON R. OpenMP: An Industry Standard API for Shared-Memory Programming [C]//Computational Science & Engineering. 1998: 46–55.

[3] HOFMANN J, TREIBIG J, HAGER G, et al. Comparing the Performance of Different x86 SIMD Instruction Sets for a Medical Imaging Application on Modern Multi- and Manycore Chips[C] //Proceedings of the International Conference on Distributed, Parallel, and Cluster Computing. 2014.

[4] MPI Forum. The Message Passing Interface Standard[J]. International Journal of Supercomputer Applications and High Performance Computing, 1994, 8(3/4): 159–416.

[5] NICKOLLS J, BUCK I, GARLAND M, et al. Scalable Parallel Programming with CUDA[J]. ACM Queue, 2010, 6(2): 40–53.

[6] MALKOV Y A, YASHUNIN D A. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs[J]. IEEE Transactions on Pattern Analysis and Machine Intelligence, 2020, 42(4): 824–836.