南開大學
Nankai University

南　　开　　大　　学

计　算　机　学　院

并行程序设计实验报告

ANN 选题 SIMD 优化实验

姓名：申健强

学号：2313119

专业：计算机科学与技术

指导教师：王刚

2025 年 4 月 29 日

目录

一、 前言

(一) 实验环境

实验环境：

- 本地 CPU：Intel(R) Core(TM) Ultra 5 125H 1.20 GHz
- 线上测试：OpenEuler 服务器[1]
- IDE：Vscode
- 本地编译器：gcc11.4.0
- simd 指令集：本地：SSE/AVX，线上：Neon

(二) 实验内容概述

1. ANN 算法背景

背景：最近邻搜索（Nearest Neighbor Search, NNS）旨在在高维空间中找到与查询向量最相似的向量。由于深度学习技术可以将图像、文本、音频等非结构化数据转换为高维向量，并保持语义相似性（例如相似图片的向量距离较近），NNS 在推荐系统、图像检索和大语言模型等领域具有广泛应用。然而，精确 NNS 在高维大规模数据集上的计算复杂度极高（例如，数据集规模 $N > 10^6$，维度 $d > 100$），使得其在实际场景中难以应用。因此，近似最近邻搜索（ANNS）成为更实用的替代方案，旨在在保持较高准确度的同时显著降低计算开销。

问题定义：

- 给定数据集 $X = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N\} \subset \mathbb{R}^d$。
- 对于查询向量 $\mathbf{q} \in \mathbb{R}^d$，ANNS 的目标是找到集合 $S \subseteq X$，使得 S 中的向量与 $\mathbf{q}$ 足够接近，并满足一定的准确度要求。

准确度衡量：常用召回率（recall）评估 ANNS 的准确性：

$$\text{recall@}k = \frac{|KNN(\mathbf{q}) \cap KANN(\mathbf{q})|}{k}$$

其中 $KNN(\mathbf{q})$ 是精确的 k 个最近邻集合，$KANN(\mathbf{q})$ 是 ANNS 返回的 k 个近似最近邻集合。本实验中，我们关注在达到特定召回率（0.9）时的算法延迟。

2. ANNS 的优化方向

ANNS 的优化主要集中在以下两个方面：

1. **加速距离计算**：
   - 使用 SIMD 指令并行化距离计算，提升单次计算的效率。
   - 采用量化技术（如 PQ、OPQ、SQ）减少浮点运算量，降低计算复杂度。
2. **减少访问的数据点数量**：

[1]考虑到我们需要进行许多次测试用于比对不同优化对于 recall 和 latency 的影响，而在线上测试相对不便，对算法层面而言我们对加速比的分析收到平台影响较小，故最终我们的加速比分析将大部分基于 x86 平台，而选择较为均衡，表现良好的算法实现到服务器上提交进行测试

- 构建高效索引结构（如 HNSW、IVF）以快速定位候选集，减少需要计算的向量数量。
- 使用混合方法（如 PQ + rerank）结合快速筛选和精确重排序，在保证准确度的同时优化性能。

具体优化技术:

- **SIMD 并行**：利用 CPU 的 SIMD 指令集（如 SSE/AVX 或 Neon）加速向量运算。
- **PQ 和 OPQ**：通过量化压缩向量数据，减少存储和计算需求。
- **SQ**：简单高效的量化方法，结合 SIMD 进一步提升计算速度。
- **混合方法**：通过 PQ 快速筛选候选集并结合 rerank 提高准确性。

本次实验基于 DEEP100K 数据集（维度为 96 的图像向量），重点探索上述优化技术在 ANN 中的实现与性能表现。

(三) 实验代码

与第一次实验相同，本人实验的所有代码，实验结果，包括但不限于可视化和算法脚本都可以在https://github.com/sjq0098/Parallel_Programs.git中查看。

二、 实验原理概述

(一) SIMD 并行

原理: SIMD（单指令多数据）通过在单个指令下同时处理多个数据元素来加速计算。在 ANN 中，SIMD 主要用于优化距离计算（如欧几里得距离或内积距离，这里我们使用内积距离），以提升搜索性能。

实验原理数学推导:

- **欧几里得距离**：对于两个向量 $\mathbf{x}$ 和 $\mathbf{y}$，其欧几里得距离定义为：

$$\delta(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_{i=1}^{d} (x_i - y_i)^2}$$

- **内积距离**：内积距离常用于余弦相似度计算，定义为：

$$\delta(\mathbf{x}, \mathbf{y}) = 1 - \sum_{i=1}^{d} x_i \cdot y_i$$

  SIMD 并行计算多个维度的乘积和累加，从而实现加速内积运算。同时 SIMD 通过将多个维度的差值计算、平方和求和操作并行化，显著减少欧氏距离的计算时间。

实现: 通过 SIMD 指令集（如 x86 的 SSE/AVX 或 ARM 的 Neon），我们可以将向量数据打包到寄存器中，并行执行加法、减法、乘法等操作。

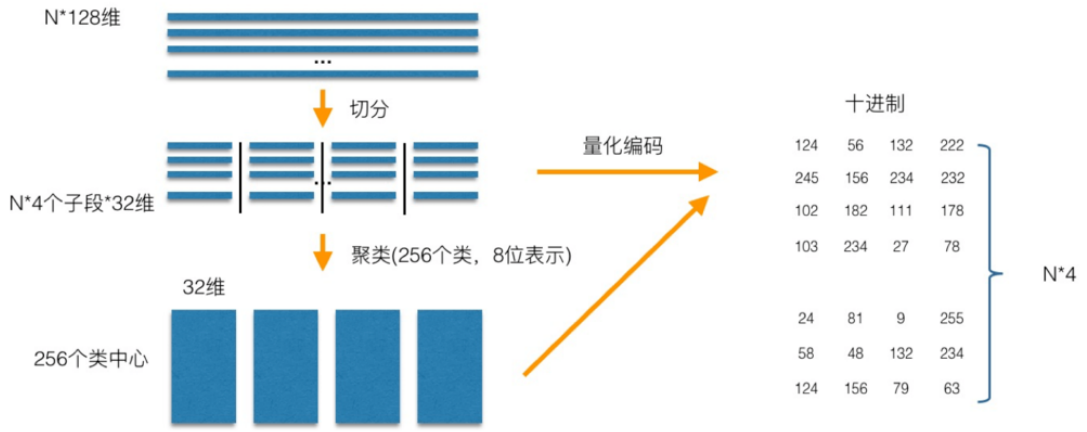

图 1: PQ 原理展示图

(二) SQ（Scalar Quantization）

原理：SQ（标量量化）将每个维度的浮点数值映射为低精度整数（例如 8 位无符号整数），从而减少存储需求并加速计算。

数学推导：

- 对于每个维度 i，将 x_i 映射到 $[0, 255]$ 范围内：

$$x_i' = \text{round}\left(255 \cdot \frac{x_i - \min_i}{\max_i - \min_i}\right)$$

其中 $\min_i$ 和 $\max_i$ 是第 i 维的最小值和最大值。

距离计算：使用量化后的整数值近似计算距离，结合 SIMD 指令（如处理 16 个 8 位整数的并行运算）进一步提升效率。

优势：SQ 实现简单，存储开销低，且易于与 SIMD 结合加速计算。

(三) PQ（Product Quantization）

原理：PQ（乘积量化）通过将高维向量分解为多个低维子空间并分别量化，通过距离查表 LUT 降低存储和计算开销，同时保持近似距离计算的准确性。

距离计算：对于查询向量 $\mathbf{q}$，预先计算其子向量与所有聚类中心的距离，存储在查找表 LUT 中。在搜索时，通过查找表快速计算量化向量与 $\mathbf{q}$ 的近似距离。

优势：PQ 将存储需求从原始浮点向量减少到量化索引，同时通过查找表加速距离计算。

缺点：PQ 采用近似的方式实现距离的计算优化，但近似计算的也有降低精度的代价，这个代价我们将在混合方法出给出一定改善。

(四) OPQ（Optimized Product Quantization）

原理：OPQ 是 PQ 的改进版本，通过引入旋转矩阵优化子空间划分，减少量化误差，提高搜索准确性。

数学推导：

- 引入正交旋转矩阵 $\mathbf{R}$，将原始向量 $\mathbf{x}$ 变换为 $\mathbf{x}' = \mathbf{R}\mathbf{x}$。
- 对变换后的向量 $\mathbf{x}'$ 应用 PQ 量化。

- 优化目标是最小化量化误差：

$$\min_{\mathbf{R},\mathbf{C}} \sum_{\mathbf{x}} \|\mathbf{x} - \mathbf{R}^{-1}\hat{\mathbf{x}}'\|^2$$

其中 $\hat{\mathbf{x}}'$ 是 $\mathbf{x}'$ 的量化表示，$\mathbf{C}$ 是聚类中心集合。

实现：通过迭代优化 $\mathbf{R}$ 和 $\mathbf{C}$，OPQ 在保持 PQ 高效性的同时，进一步提升量化精度。

缺点：OPQ 能够在保留 PQ 的高效性的同时提升量化精度，但缺点在于索引构建成本过高[2]。

（五） 混合方法（PQ + rerank）

原理：混合方法结合 PQ 的快速近似搜索和 rerank 的精确重排序，先通过 PQ 筛选候选集，再对候选集进行精确计算以提高准确性。

数学推导：

- **PQ 搜索**：利用 PQ 量化向量快速计算近似距离，生成候选集 S'。
- **rerank**：对 S' 中的向量使用原始向量计算精确距离，排序后输出最终结果。

优势：该方法在保证高召回率的同时，通过 rerank 提升了搜索精度，适用于对准确性要求较高的场景。

三、 关键实现概述

基于上述原理我们有以下实现，由于本地我们主要利用 x86 平台测试，数据的加载指令等我们也用此举例，在本地测试之后我们会进行迁移工作到服务器端进行测试：

（一） SIMD 优化实现

SIMD 优化是后续优化的基础，主要实现串行数据的向量化便于 SIMD 并行，针对向量间距离计算，提高现代处理器的并行计算能力的利用率。具体设计算法如下

1. **基础内积距离计算优化**
   - 利用 AVX2 指令集实现 8 个浮点数并行计算
   - 通过寄存器重用和指令重排减少内存访问开销
2. **向量规范化优化**
   - 向量预先封装，减少后续代码的复杂性
   - 预计算向量平方和，避免查询时重复计算
   - 使用 SIMD 加速向量规范化过程

Algorithm 1 SIMD 优化的内积距离计算

```
function ComputeInnerProduct(x, y, dim)
    i ← 0, sum256 ← _mm256_setzero_ps()
    while i + 7 < dim do
        vx ← _mm256_loadu_ps(x + i)
```

[2]在本次实验中由于 OPQ 方法的矩阵训练成本过高和本地索引构建的局限性并未能体现出其优越性，我们在接下来的实现中会细说此事

```
        vy ← _mm256_loadu_ps(y + i)
        vmul ← _mm256_mul_ps(vx, vy)
        sum256 ← _mm256_add_ps(sum256, vmul)
        i ← i + 8
    end while
    水平求和 sum256 得到 ip
    处理剩余元素
    return 1.0 − ip                                          ▷ 内积距离
end function
```

(二) 标量量化 (SQ) 实现

SQ 实现通过量化将浮点值映射到 8 位无符号整数，同时结合 SIMD 加速计算。主要包括：

1. **向量量化过程**
   - 计算每个向量的最小值和最大值作为量化参数
   - 按比例将浮点值映射到 [0,255] 范围的 uint8 值
2. **量化后的距离计算**
   - 结合反量化和 SIMD 加速内积计算
   - 批量处理 8/16 个量化值，提高计算吞吐量

Algorithm 2 标量量化 (SQ) 实现

```
function QuantizeVector(vec, idx, dim)
    找到向量中的最小值 min_val 和最大值 max_val
    scale ← (max_val − min_val)/255.0
    offset ← min_val
    for j ← 0 to dim − 1 do
        normalized ← (vec[j] − min_val)/scale
        quantized ← uint8(min(255, max(0, normalized)))
        存储量化值
    end for
end function
function ComputeDistanceSQ(query, code, scale, offset, dim)
    使用 AVX2 批量处理 16 个量化值
    反量化: value = code × scale + offset
    计算与查询向量的内积
    return 1.0 − innerproduct
end function
```

(三) 乘积量化 (PQ) 实现

PQ 实现将高维向量分解为子向量并单独量化，通过查表加速距离计算：

1. **索引构建 (本地 python 调包实现)**
   - 将向量分为 m 个子向量

- 对每个子空间进行 K-means 聚类，生成码本和编码文件保存备用

2. **距离计算加速**

- 预计算查询向量与码本的距离表 (LUT)
- 通过查表方式快速近似计算距离

Algorithm 3 乘积量化 (PQ) 实现

```
function BuildPQIndex(base, n, dim, m, k)
    将 dim 维向量分为 m 个子向量
    for i ← 0 to m − 1 do
        从基础数据中提取第 i 个子向量
        对子向量进行 K-means 聚类得到 k 个中心
        存储码本 codebooks[i]
    end for
    for i ← 0 to n − 1 do
        量化向量 base[i] 得到 m 个量化索引
        存储量化索引
    end for
end function
function SearchPQ(query, k)
    预计算距离表 distTable[m][k]
    初始化最小优先队列 results
    for i ← 0 to n − 1 do
        dist ← 0
        for j ← 0 to m − 1 do
            centerId ← codes[i][j]
            dist ← dist + distTable[j][centerId]
        end for
        更新结果队列 results
    end for
    return results
end function
```

(四) 优化乘积量化 (OPQ) 实现

OPQ 在 PQ 基础上引入正交变换，提高量化精度，其实现与 PQ 大致类似，再次不做赘述：

1. **正交矩阵训练**

- 迭代优化正交矩阵和码本，最小化量化误差
- 使用降维分解实现矩阵优化

2. **索引构建与搜索**

- 在变换空间中构建 PQ 索引
- 搜索时先变换查询向量，再按 PQ 方式计算距离

（五） 混合方法 (PQ+rerank) 实现

混合方法结合 PQ 的高效筛选和精确重排序：

1. **两阶段搜索**
   - 使用 PQ 快速生成候选集
   - 对候选集使用原始向量计算精确距离并重排序
2. **候选集大小自适应**
   - 根据目标召回率动态调整候选集大小
   - 平衡查询延迟和搜索准确性

Algorithm 4 混合方法 (PQ+rerank) 实现

function SEARCHPQRERANK(**query**, k, $rerank\_factor$)
  $k' \leftarrow k \times rerank\_factor$
  $candidates \leftarrow$ SearchPQ(**query**, k')
  初始化结果队列 $results$
  for $i \leftarrow 0$ **to** $|candidates| - 1$ **do**
    $id \leftarrow candidates[i].id$
    $dist \leftarrow$ 使用原始向量计算 **query** 与 **base**$[id]$ 的精确距离
    更新 $results$
  end for
  return 取 $results$ 中前 k 个结果
end function

四、 实验设计与实现

本实验使用 DEEP100K 数据集，实现并测试不同优化方法在 ANN 中的性能表现。以下是详细的实验设计与实现：

（一） 数据集处理与评估指标

1. **数据集分割**
   - 使用 DEEP100K 数据集（10 万个 96 维向量）
   - 选择查询列表文件 query 中的两千条查询需求作为查询集，base 作为基础数据集
   - 使用预计算精确 K 近邻 gt 数据集作为基准（ground truth）
2. **评估指标**
   - 主要指标：查询延迟（latency，毫秒）
   - 约束条件：召回率（recall@10）>= 0.9
   - 系统消耗（较为次要）：内存使用、索引构建时间

(二) 实验流程设计

1. **索引构建阶段**[3]

Algorithm 5 索引构建流程

```
procedure BuildIndex(method, parameters)
    加载 DEEP100K 基础数据集
    switch method do
        case SQ
            实例化 ScalarQuantizer
            调用 build 方法量化向量
        case PQ
            实例化 ProductQuantizer
            训练码本
            编码基础数据集（参数 M，Ksub）
        case OPQ
            训练正交变换矩阵
            实例化 OptimizedPQ
            训练码本并编码数据集（参数 M，Ksub，迭代次数 I）
    保存索引到文件
end procedure
```

2. **查询评估阶段**如前文所述，我们为了方便测试和对比我们在本地进行了加速的比对测试，在本地我们对 main.cc 文件进行了改造，让其能够测试多种方法并返回其 recall 与 latency 进行记录，其中我们本地测试时创建了新的编译和测试脚本。

Algorithm 6 查询评估流程

```
procedure EvaluateQuery(method, parameters)
    加载预构建索引
    预先加载查询集和 ground truth
    for query in queries do
        start_time ← 当前时间
        results ← 执行查询
        end_time ← 当前时间
        latency ← end_time − start_time
        计算 recall@10
        记录结果
    end for
    计算平均延迟和平均召回率
    return avg_latency, avg_recall
end procedure
```

(三) 实现细节

篇幅所限，具体可查阅上文提到的 GitHub 仓库。

[3]其实没有指定 method，一股脑全构建出来了，但按理来说应该保持程序健壮性，所以我们在这里写有分支的版本

(四) 优化参数调整

实验中重点调整以下参数，分析其对性能的影响：

1. **SQ 优化参数**
   - 量化数据类型：8 位、16 位等
   - SIMD 指令集选择：AVX2、SSE4、Neon 等
2. **PQ 及 OPQ 优化参数**
   - 子空间数量 m：4, 8, 16, 32
   - 码本大小 k：256（单字节量化）
3. **混合方法参数**
   - 重排序候选数量
   - 重排序前置过滤阈值调整

通过系统调整这些参数，找到在召回率 0.9 约束下延迟最低的配置。

五、 实验结果可视化分析

(一) 本地主要实验结果

我们在本地测试得到的主要实验结果如表1：

(二) 召回率与延迟的权衡分析

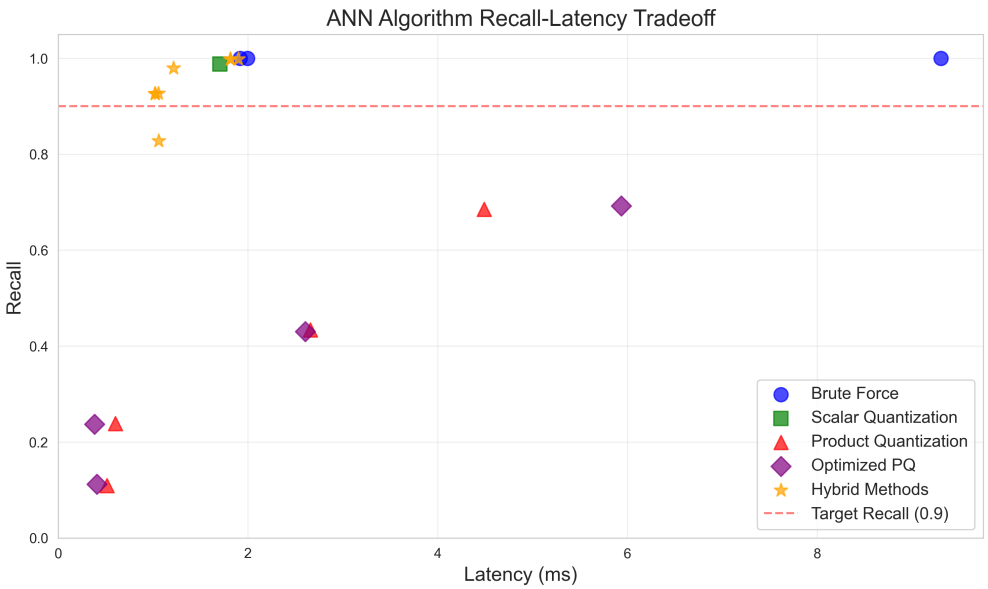

图 2: ANN 算法召回率-延迟权衡分析

图2展示了不同 ANN 算法在召回率和查询延迟之间的权衡关系。从图中可以清晰地看出：

表 1: ANN 算法综合性能对比（最优值标粗）

算法	召回率	延迟 (毫秒)	加速比
标量暴力搜索	**1.0000**	9.30	1.00×
SSE 优化暴力搜索	**1.0000**	2.00	4.66×
AVX 优化暴力搜索	**1.0000**	1.92	4.85×
标量量化 (SQ)	0.9878	1.70	5.46×
乘积量化 (PQ) M=4	0.1095	0.52	18.04×
乘积量化 (PQ) M=8	0.2385	0.60	15.41×
乘积量化 (PQ) M=16	0.4340	2.66	3.50×
乘积量化 (PQ) M=32	0.6851	4.49	2.07×
优化乘积量化 (OPQ) M=4	0.1124	0.41	22.86×
优化乘积量化 (OPQ) M=8	0.2372	**0.39**	**24.16**×
优化乘积量化 (OPQ) M=16	0.4303	2.60	3.57×
优化乘积量化 (OPQ) M=32	0.6920	5.94	1.57×
混合搜索 (PQ16+ 精确重排序)	0.9265	**1.02**	9.09×
混合搜索 (PQ32+ 精确重排序)	0.9982	1.81	5.13×
混合搜索 (OPQ16+ 精确重排序)	0.9269	1.05	8.82×
混合搜索 (OPQ32+ 精确重排序)	0.9982	1.90	4.90×
PQ16+R50	0.8277	1.06	8.77×
PQ16+R100	0.9265	**1.02**	9.09×
PQ16+R200	0.9798	1.21	7.66×
PQ16+R500	0.9982	1.81	5.13×

- 暴力搜索算法（特别是标量实现）虽然提供了精确召回，但其查询延迟也最高。
- 标量量化 (SQ) 技术维持了接近暴力搜索的高召回率，同时显著降低了延迟。
- 乘积量化方法 (PQ 和 OPQ) 随着子空间数量 (M) 的增加，召回率逐渐提高，但查询延迟也相应增加。
- 混合搜索方法位于图的左上区域，表明它们在保持高召回率的同时实现了低延迟，是性能与精度平衡的最佳选择。

对于并行算法而言，算法的 recall 和 latency 的 tradeoff 关系比较明显。混合搜索 (PQ16+ 精确重排序) 和 PQ16+R100 实现了超过 92% 的召回率，同时延迟仅为 1.02 毫秒，表现出最佳的综合性能平衡，体现出先近似查找再进行精确查找方法的优越性。

另一个比较需要注意的是，SQ 算法在本数据集中表现优异可能是因为对于归一化的数据集而言，找到其上下限并进行分为 256 个标度本身颗粒度就很小了，这导致 SQ 的表现优异。

(三) SIMD 指令集加速效果

图3展示了 SIMD 指令集对暴力搜索算法的加速效果。从图中可以清晰看出:

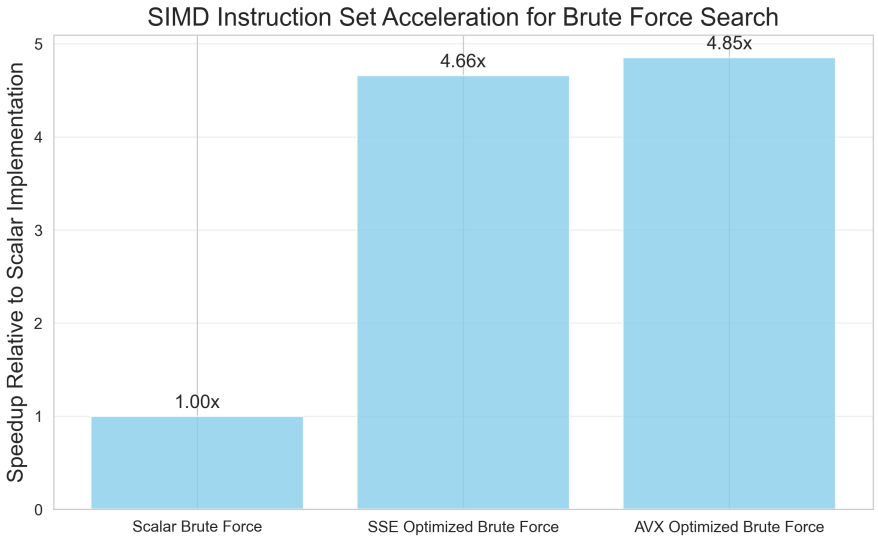

图 3: SIMD 指令集对暴力搜索的加速效果

这些加速在保持近似 100% 召回率的前提下实现，证明了 SIMD 向量化对距离计算的高效性。

(四) 子空间数量对 PQ 和 OPQ 性能的影响

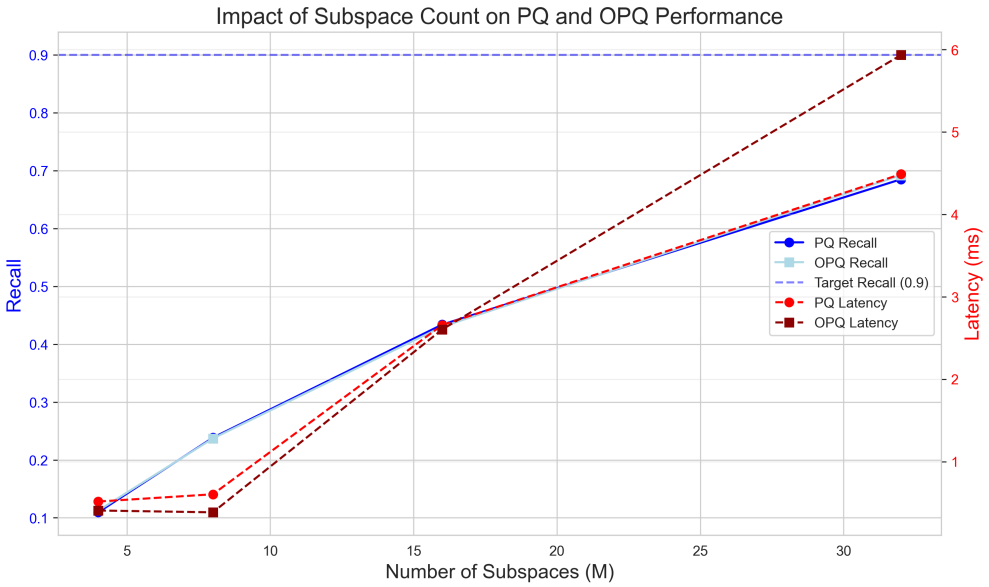

图 4: 子空间数量对 PQ 和 OPQ 性能的影响

图4展示了子空间数量 (M) 对 PQ 和 OPQ 性能的影响。从图中可以观察到:

- 随着子空间数量的增加，PQ 和 OPQ 的召回率逐渐提高，但查询延迟也显著增加。
- 在低子空间数量 (M=4,8) 时，OPQ 的召回率略高于 PQ，且延迟更低，显示了优化变换矩阵带来的益处。
- 当 M 值较大时 (M=16,32)，两种方法的召回率差异减小，但 OPQ 的延迟略高于 PQ。

- 对于追求高召回率的应用场景，M=32 是较好的选择；而对于追求低延迟的场景，M=4 或 M=8 可能更为合适。

对于上述的实验结果我们需要进行一定的解释，**为什么 OPQ 算法对比 PQ 算法优化没有那么显著?**

因为在索引构建中，进行多次迭代的索引构建很耗时，出于时间与 ddl 考虑，OPQ 构建的索引都只进行了一轮迭代，所以优化相对没那么显著。

为什么重排序能够相对单纯的 PQ 提升那么显著?

这个问题存在两方面的解释，其一是我们在进行 kmeans 聚类时没有使用内积距离实现，导致后续使用内积举例作并行优化时的使用的聚类本身具有一定的近似性，而重排序消除了这种近似性，从而使得精度提升很快。其二，在计算过程中 PQ 使用计算时 Float32 运算的误差存在累积，导致 PQ/OPQ 算法的正确率下降，而重排序计算时使用原始向量精度会略微高一些。

（五） 重排序因子对性能的影响

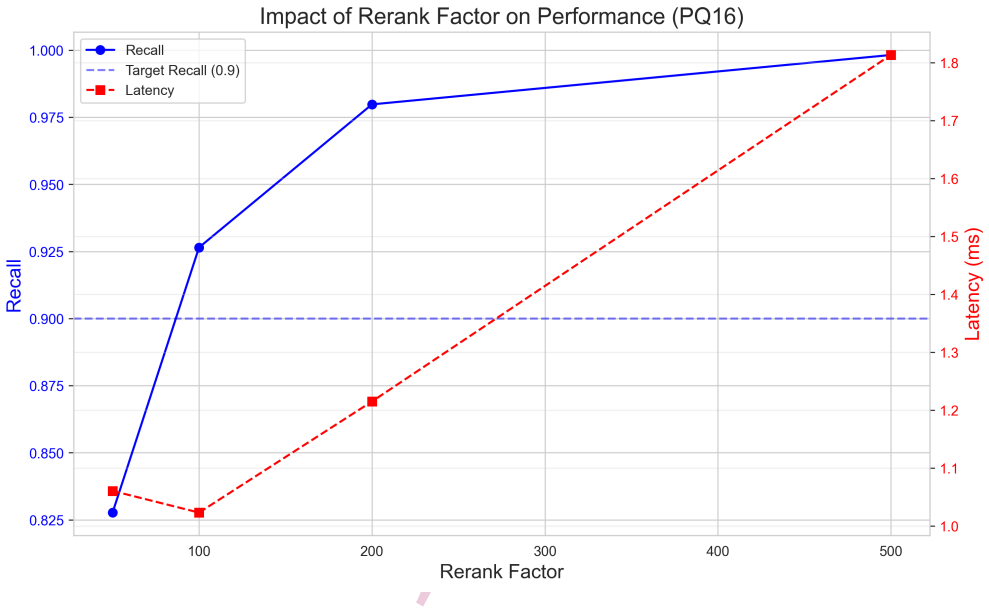

图 5: 重排序因子对混合搜索性能的影响

图5分析了重排序因子对混合搜索方法性能的影响。从图中可以得出以下结论：

- 随着重排序因子的增加，召回率呈现明显的上升趋势，但查询延迟也相应增加。
- 重排序因子为 100 时，实现了 92.65% 的高召回率，同时维持了较低的查询延迟 (1.02 毫秒)，是精度和性能的良好平衡点。
- 当重排序因子增加到 500 时，召回率进一步提高到 99.82%，接近暴力搜索的精度，但延迟增加到 1.81 毫秒，已经和暴力搜索没有区别甚至超越了暴力搜索的时间消耗。

出于平衡考虑，对于大多数实际应用，重排序因子在 100-200 之间可能是最佳选择，既保证了高召回率，又维持了较低的延迟。

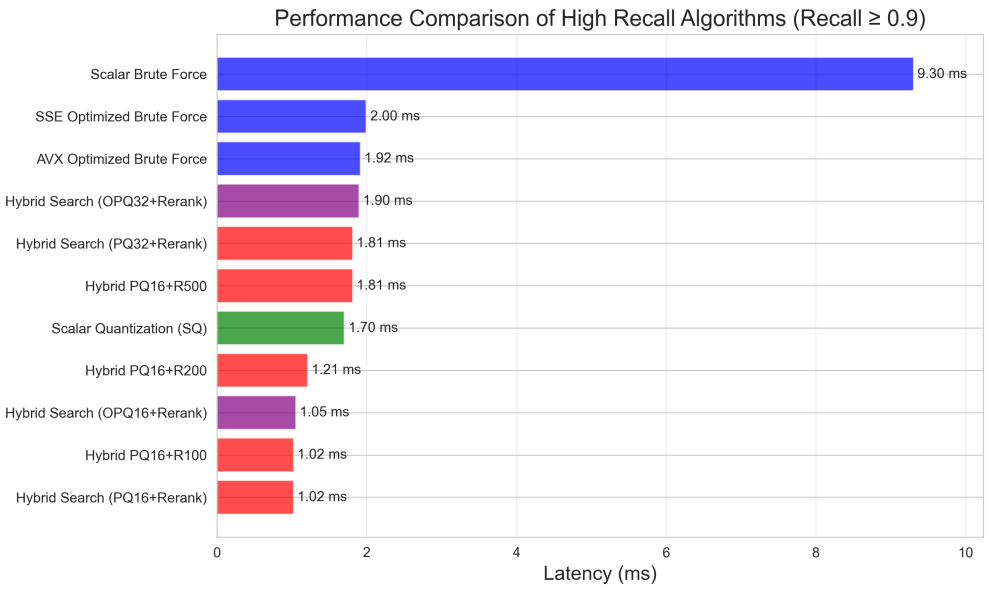

图 6: 高召回率算法 (召回率 0.9) 性能对比

(六) 高召回率算法性能对比

图6对比了所有召回率 0.9 的算法的性能。从图中可以清晰看出:

- 混合搜索 (PQ16+ 精确重排序) 和 PQ16+R100 表现最为出色，在保持 92.65% 高召回率的同时，查询延迟仅为 1.02 毫秒。
- 混合搜索 (OPQ16+ 精确重排序) 也表现良好，召回率为 92.69%，延迟为 1.05 毫秒。
- 标量量化 (SQ) 虽然提供了 98.78% 的高召回率，但其延迟 (1.70 毫秒) 高于混合方法。
- 暴力搜索方法，即使经过 SIMD 优化，其延迟仍明显高于混合方法，表明量化和多阶段搜索策略有效。

(七) 召回率与加速比的关系

图7展示了不同算法在召回率和加速比之间的权衡关系。从图中我们可以观察到:

- 优化乘积量化 (OPQ) 和乘积量化 (PQ) 在低子空间配置 (M=4,8) 下提供了最高的加速比 (超过 15 倍)，但召回率较低 (约 10%-24%)。
- 混合搜索方法实现了良好的平衡，在保持超过 90% 召回率的同时提供 7-9 倍的加速比。
- 标量量化 (SQ) 位于中间位置，提供了 5.46 倍的加速比和 98.78% 的高召回率。

由上面的对比分析我们可以得到：对于要求极高精度的应用，标量量化或 SIMD 优化的暴力搜索是合适的选择，我们最好直接实现精确算法；而对于大多数实际应用如推荐系统或别的对精度要求相对较低的应用，混合搜索方法可以提供更好的性能-精度平衡。

(八) OpenEuluer 服务器运行结果补充

我们将方法迁移到 Neon 平台，进行了实验结果测试和补充如表2:

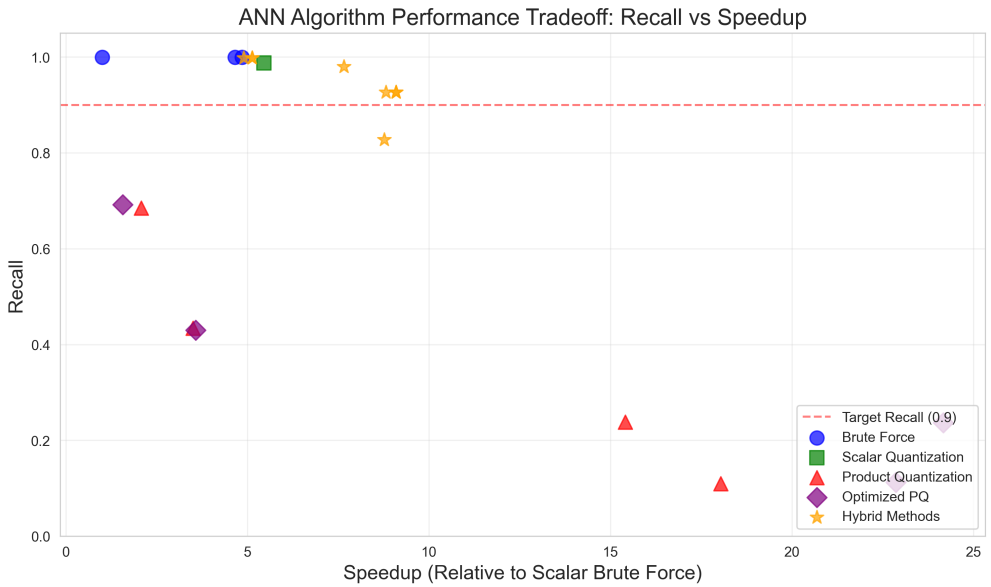

图 7: ANN 算法性能权衡：召回率 vs 加速比

表 2: Neon 指令集加速的 ANN 算法性能对比

算法	召回率	延迟 (微秒)	加速比
暴力串行	**0.99995**	16147.30	1.00×
暴力并行（Neon）	**0.99995**	1015.31	**15.90**×
SQ_Neon(openmp)	0.97965	997.68	16.18×
SQ_Neon	0.97965	7909.72	2.04×
PQ8	0.23865	1210.82	13.34×
PQ16+rerank(openmp)	0.97565	**580.73**	**27.80**×
PQ16+rerank	0.92711	2683.44	6.02×

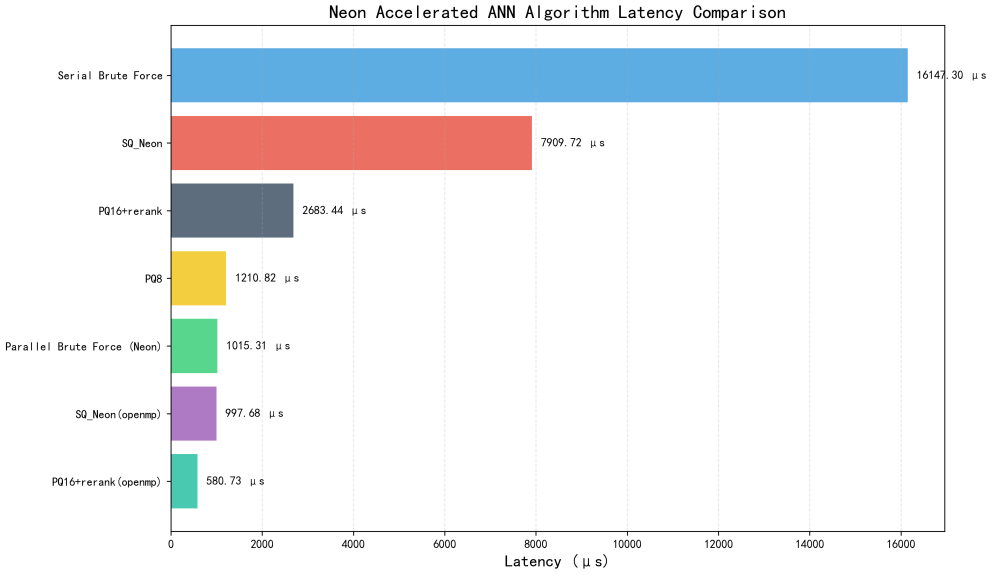

图 8: Neon 迁移后延迟对比

在 neon 平台中我们可以看到实验结论与我们在本地所设想结论也是大差不差的，可以佐证实验的结论的正确性。由于在 SQ 与 PQ 中都存在归并，进行过对于算法的 openmp 优化尝试，可以发现 openmp 中无论是 recall 还是 latency 都有很大提升，原因在于 openmp 中计算的精度损失比较小，多线程下并行归并存在对串行或是循环展开的“降维打击”，这可能是我们之后主要的优化方向之一。

六、 服务器 perf 分析：针对 PQ16+rerank

我们对 PQ16+rerankNeon 实现进行了性能分析，使用 perf 工具收集了关键性能指标。表3总结了主要结果。

表 3: PQ16+rerankNeon 性能计数器统计

缓存性能		指令执行		执行时间	
缓存引用	8,070,560,668	周期数	14,494,959,960	总执行时间	5.78s
缓存缺失	29,209,298 (0.362%)	指令数	32,057,535,844	用户时间	5.65s
		IPC	2.21	系统时间	0.04s

从分析结果看，PQ16+rerankNeon 实现表现出色：

- 缓存缺失率低 (0.362%) 数据局部性良好
- 每周期指令数 (IPC) 达到 2.21，显示出良好的指令级并行性和 SIMD 指令利用

七、 总结

总的来说，在本次实验中我们对 ANN 及其优化算法实现了 SIMD 实现，对于 recall 与 latency 的 tradeoff 有一定的了解，在同样使用并行或串行的情况下，我们需要根据需求选择速度和准确率达到要求平衡的方法实现。在 DEEP100K 数据集上进行对比测试后，我们可以归纳：

- **SIMD 并行**：基于 AVX2/SSE 指令集的向量化内积计算，将暴力搜索的延迟由 9.30 ms 提升至 1.92 ms（约 4.85× 加速），在保证 100% 召回率的前提下实现了显著的性能提升。
- **标量量化（SQ）**：将浮点向量映射至 8 位整数，并结合 SIMD 加速，延迟降至 1.70 ms，召回率达 98.78%，实现了低存储开销与高效率的平衡。
- **乘积量化（PQ/OPQ）**：随着子空间数量 M 增大，PQ 和 OPQ 的召回率从约 10% 提升至 70% 以上，但延迟也显著上升。OPQ 在低维划分（$M = 4, 8$）时相比 PQ 可进一步降低延迟并略微提高召回，但高维划分下二者性能差异不大。
- **混合方法（PQ+rerank）**：先通过 PQ 快速筛选候选集，再对候选进行精确重排序，实现了在 1.02 ms 延迟下 92.65% 召回率的最优平衡，综合性能优于单纯 PQ/OPQ 。

此外，在本次实验中我们还熟练了有关服务器的使用、对于编译器和指令脚本也有了更深入的了解。[4]

[4] **AI 使用情况：** 在本次实验中我们在接口的调用，不同平台的 simd 指令迁移和 debug 时使用了 AI 进行辅助，对于不是很了解的 bash 和编译器也通过 AI 减小了学习成本。