南開大學
Nankai University

计算机学院

并行程序设计实验报告

SIMD 编程实验–ANNS

姓名：杨宇翔

学号：2312506

专业：计算机科学与技术

2025 年 4 月 29 日

目录

1 算法设计

1.1 暴力算法 flat

直接计算查询向量和其他所有向量的内积距离，取最小的 k 个作为返回结果。内积距离的计算我们可以用 neon 并行进行优化。将原向量内积拆成 4 个部分，同时计算 4 个内积和，最后水平求和，得到总的内积和。

向量内积距离

```
函数 计算距离(query1, query2, vecdim):
    初始化 sum = 0.0

    对于 k 从 0 到 vecdim-1 步长 4:
        取出 query1 中从k开始的4个元素 → vec1
        取出 query2 中从k开始的4个元素 → vec2
        计算对应元素乘积 → products = [vec1[i] * vec2[i] for i in 0..3]
        将products累加到sum → sum += products[0]+products[1]+products[2]+products[3]

    距离 dis = 1.0 - sum
    返回 dis
```

1.2 SQ

我们已经把原始的内积和拆成 4 个部分并行计算了，能不能拆得再多一点呢？注意到原本一个 float 占 32 位，一个 128 位的向量寄存器只能存 4 个，即只能 4 位并行。于是我们可以考虑把 float 变成只占 8 位的 uint8_t 类型，这样子一个向量寄存器就可以存 16 个，即 16 位并行，进一步加速计算。尽管这样会丢失精度，但在接受范围内。

1.2.1 标量量化

将原本向量里的所有值都映射到 [0,255] 这个范围内，这样就可以用 uint8_t 来装了。具体映射方法就是确定所有向量原始值所在的区间，即确定最大值和最小值，用 255 除以这个区间长度从而得到缩放因子 scale。同时减去最小值得使负数映射到正区间。映射函数为 y = scale*(x - minval)。

这个过程同样可以用 neon 并行大幅加速计算。

标量量化

```
函数 压缩数据(base数组, compressed_base数组, base_number向量数, vecdim向量维度):
    总元素数 = 基向量数 × 向量维度

    min_val = 最大浮点数
    max_val = 最小浮点数

    对于 i 从 0 到 总元素数-4 步长4:
        加载base[i..i+3] → current_vec
        min_val = 取当前min_val与current_vec的最小值
        max_val = 取当前max_val与current_vec的最大值

```

```
    对于 i 从 (总元素数/4)*4 到 总元素数-1:
        min_val = 取min_val与base[i]的较小值
        max_val = 取max_val与base[i]的较大值

    若 min_val == max_val:
        将compressed_base所有元素置0
        返回

    缩放系数 = 255.0 / (max_val - min_val)

    对于 i 从 0 到 总元素数-16 步长16:
        batch = 加载base[i..i+15]

        减去最小值: batch = batch - min_val
        缩放: batch = batch × 缩放系数
        截断到[0,255]区间

        四舍五入转整数 → int_batch
        将int_batch转换为uint8类型
        存储到compressed_base[i..i+15]

    对于 i 从 (总元素数/16)*16 到 总元素数-1:
        y = (base[i] - min_val) × 缩放系数
        y = 限制在0-255之间
        四舍五入取整 → compressed_base[i]
```

1.2.2 欧式距离并行加速

由于原向量经过了标量量化，显然没有办法再用内积距离。所以我们直接用欧式距离计算。这个过程用 neon 并行算法优化。

欧氏距离并行加速

```
函数 计算平方距离(query1数组, query2数组, 向量维度vecdim):
    初始化 sum = 0

    对于 i 从 0 到 vecdim-1 步长16:
        加载 query1[i..i+15] → 向量x
        加载 query2[i..i+15] → 向量y

        差值向量 = 绝对值(x - y)
        拆分前8个差值 → 低部差值
        拆分后8个差值 → 高部差值

        低部平方和 = 逐元素平方(低部差值)
        高部平方和 = 逐元素平方(高部差值)

        sum += 水平相加(低部平方和)
        sum += 水平相加(高部平方和)
```

```
17
18      返回 sum
```

1.2.3 SQ 搜索

本质和暴力算法相同，还是遍历所有向量算出距离，得到 k_top 个向量。但速度有显著的提升。

平凡算法

```
1  函数 SQ搜索(库数据base, 查询向量query, 库向量总数base_number, 向量维度vecdim,
       返回数量k):
2      创建最大堆优先级队列q（存储<距离, 索引>对）
3
4      对于每个向量索引i 从0到base_number-1:
5          当前向量 = base + i×vecdim
6          计算距离 = 计算平方距离(query, 当前向量, vecdim)
7
8          如果 队列大小 < k:
9              将(距离, i)加入队列
10         否则:
11             如果 当前距离 < 队列最大距离:
12                 加入新距离对
13                 弹出当前最大距离对
14
15     返回最终优先级队列
```

1.3 PQ

虽然 SQ 将原向量压缩成 uint8_t 后并行效率得到了很大提高，但遍历所有向量的做法还是太暴力了。如果向量规模继续扩大，遍历千万甚至亿的数量级显然不现实。于是 PQ 提出了将原向量切割成多个子空间，分别聚类，把向量替换成其所属簇中心的方法。

我们将原向量空间切成 4 个子空间，各自聚类得到 256 个类中心，总共 1024 个中心。当我们要计算查询向量与某个向量的距离时，我们只用直接计算到它所属中心的距离就可以了。

文档中提出的对查询向量进行同样的切分和量化编码，于是查询距离被替换成中心与中心的距离，即 SDC 的做法，显然是没道理的。首先查询向量没办法提前切分量化，而量化的过程又避免不了遍历一遍所有聚类中心并计算查询向量与聚类中心的距离。那为什么不直接拿这个距离用呢？还能避免查询向量量化产生的误差。所以我使用的是直接构建查询向量与聚类中心的距离表的方法，即 ADC。当然 SDC 的代码我也保留了，提前计算了类中心之间的距离，后续性能分析上会有比较。

1.3.1 切分量化

kmeans 函数省略，切分量化函数也有部分省略。由于是离线计算，所以没有特地做并行优化。

切分量化

```
函数 k均值压缩(base数组, 基向量数base_number, 向量维度vecdim):
    设 K = 256
    创建压缩基数组 compressed_base[base_number×4]
    创建聚类中心数组 centers[4×K×vecdim/4]

    对于 每个子空间索引i 从0到3:
        创建子空间数据 tmp_base[base_number×(vecdim/4)]
        对于 每个向量j 从0到base_number−1:
            对于 每个维度k 在子空间i中:
                tmp_base[j][k] = base[j][原向量位置i对应的维度段k]

        执行 k均值算法:
            输入: tmp_base, 聚类数K, 子空间维度vecdim/4
            输出: 标签数组tmp_labels, 聚类中心tmp_centers

        对于 每个向量j 从0到base_number−1:
            compressed_base[j][i] = tmp_labels[j]

        对于 每个聚类中心c 从0到K−1:
            将tmp_centers[c] 复制到 centers[i×K + c] 的位置

    返回 compressed_base 和 centers
```

1.3.2 PQ 搜索

注释掉的即为 SDC。在计算查询向量与聚类中心的距离时进行了 neon 并行优化。

PQ 搜索

```
函数 乘积量化搜索(压缩库compressed_base, 聚类中心centers, 查询向量query,
    库向量数base_number, 向量维度vecdim, 返回数k_top):
    K = 256

    创建查询中心距离表 query_centers_dis[4][256]

    对于 每个子空间i 从0到3:
        子查询 = 查询向量在第i子空间的维度段
        对于 每个聚类中心j 从0到255:
            聚类中心向量 = centers中第i组第j个中心
            距离 = 0.0

            对于 每4个维度k 从0到vecdim/4−1 步长4:
                加载中心向量的4个元素 → vec1
                加载子查询的4个元素 → vec2
                计算差值平方和 → sum += Σ(vec1−vec2)^2

            记录距离到 query_centers_dis[i][j]

    创建最大堆优先级队列q (存储<距离, 索引>)
```

```
20
21      对于 每个库向量i 从0到base_number-1:
22          总距离 = 0.0
23          对于 每个子空间j 从0到3:
24              获取该向量在第j子空间的标签 → label = compressed_base[i*4+j]
25              累加距离 → 总距离 += query_centers_dis[j][label]
26
27          如果 堆大小 < k_top:
28              入堆(总距离, i)
29          否则:
30              如果 总距离 < 堆顶距离:
31                  替换堆顶元素
32
33      返回最终堆
```

1.4 IVF-PQ

在原本的 PQ 基础上，我们加上了一层粗聚类，即 IVF 操作。通过粗聚类和倒排表，我们可以快速锁定查询向量相关的搜索范围，而不用像 PQ 一样遍历所有的聚类中心。

除此之外，IVF-PQ 在进行 PQ 量化之前，还将所有向量减去其所属的粗聚类中心，得到残差，最后对这个残差进行 PQ 量化。残差的动态范围更小，能更精确地表示向量间的局部差异，显著提高了正确率。

1.4.1 IVF 粗聚类

IVF 就是在切分量化之前先做一遍聚类，构建倒排表并计算残差。这些全部保存成二进制文件供后续处理。

IVF 粗聚类

```
1   函数 IVF-PQ压缩(原始数据base, 输出ivf_base, 粗聚类中心ivf_centers, 数据量base_number,
        维度vecdim):
2       K = 256
3       分配压缩数据 compressed_base[base_number×4]
4       分配PQ聚类中心 centers[4×K×(vecdim/4)]
5
6       对于每个子空间i 从0到3:
7           创建子空间残差数据 tmp_base[base_number×(vecdim/4)]
8           对于每个向量j 从0到base_number-1:
9               获取该向量的粗聚类中心索引 → cluster_idx = ivf_base[j]
10              提取粗聚类中心的第i子空间部分 → center_part =
                    ivf_centers[cluster_idx][i的子空间维度]
11              计算残差 → tmp_base[j] = base[j][i的子空间维度] - center_part
12
13          执行k-means聚类:
14              输入: tmp_base (残差数据), 聚类数K, 子空间维度vecdim/4
15              输出: 标签数组tmp_labels, 聚类中心tmp_centers
```

```
        记录子空间i的标签 → compressed_base[所有向量][i] = tmp_labels
        存储子空间i的聚类中心 → centers[i×K..(i+1)×K] = tmp_centers

    返回 compressed_base 和 centers
```

1.4.2 欧氏距离并行加速

虽然之前写过一个欧式距离了，但那个是针对 uint8_t 类型的并行加速，float 类型还是得重新写一个。

欧式距离并行加速

```
函数 计算平方距离(query1数组, query2数组, 维度vecdim):
    初始化累加器 sum = 0.0

    对于 k 从 0 到 vecdim−1 步长4:
        加载 query1[k..k+3] → 向量vec1
        加载 query2[k..k+3] → 向量vec2

        差值向量 = vec1 − vec2
        平方向量 = 差值向量 * 差值向量  # 逐元素相乘

        sum += 平方向量的所有元素之和 (SIMD通道内求和)

    返回 sum
```

1.4.3 IVF-PQ 搜索

中间经过一系列处理后，我们最后拿到了粗聚类的 ivf_base 信息，倒排表，以及残差 PQ 量化后的 pqivf_base 信息。有了这些后我们就可以进行搜索了。拿到查询向量后遍历一遍粗聚类中心，得到 m 个相距最近的簇，然后遍历这些簇中的向量，计算查询向量与簇中心的残差到候选向量的残差的距离，从而得到 k_top 的向量。

其中计算查询向量与簇中心的残差部分也可以做 neon 并行加速。

IVFPQ 搜索

```
函数 IVF−PQ搜索(压缩库compressed_base, PQ聚类中心centers, IVF聚类中心ivf_centers,
    倒排索引inverted_kmeans, 查询向量query, 库向量数base_number, 维度vecdim,
    返回数k_top):
    m = 16
    创建粗聚类候选列表 query_ivf_labels[m]

    创建最大堆ivf_q (存储<距离, 聚类索引>)
    对于每个IVF聚类中心i 从0到255:
        计算距离 = query与ivf_centers[i]的L2距离
        维护堆ivf_q保持前m个最小距离

```

```
从ivf_q中逆序取出m个最小距离的聚类索引 → query_ivf_labels

创建候选集信息:
    size[m] ← 各候选聚类的向量数量
    candidate_base[m] ← 各候选聚类的向量索引列表
    residual[m] ← 各候选聚类对应的查询残差

对于每个候选聚类i in 0..m-1:
    聚类中心 = ivf_centers[query_ivf_labels[i]]
    残差 = query - 聚类中心 (向量化计算)

创建结果最大堆q (存储<总距离, 向量ID>)

对于每个候选聚类i in 0..m-1:
    对于该聚类中的每个向量j in 0..size[i]-1:
        向量ID = candidate_base[i][j]
        总距离 = 0

        对于每个子空间k in 0..3:
            获取该向量的PQ标签 → label = compressed_base[向量ID][k]
            子空间中心 = centers[k][label]
            子距离 = 残差[i]在该子空间与子空间中心的距离
            总距离 += 子距离

        如果 堆大小 < k_top 或 总距离 < 堆顶距离:
            插入/替换堆元素

返回最终结果堆q
```

1.5 重排

以上所有算法都只是粗排，得到的结果都要再经过一次精排。流程就是先设置一个比 top_k 较大的 rerank，经过优化算法快速得到前 rerank 的值后再做精排，这样可以极大幅度提高召回率。因为优化算法虽然快速，但模糊了相邻向量之间的差异，直接拿优化算法的结果测试，召回率将惨不忍睹。

精排就是经过并行优化后的暴力算法，代码同暴力算法 flat。

2 性能分析

所有实验结果都是在服务器上取得的。默认 test_number 为 2000。k_top 默认为 10。PQ 默认切割 4 个子空间，每个子空间 256 个聚类中心，采用 ADC。IVF-PQ 默认 256 个粗聚类中心，残差 PQ 结构与常规 PQ 结构相同，默认候选簇数量为 8。除了 flat 以外，优化算法的其他参数均取能使召回率刚过 0.9 的值。latency 取十次实验平均值。

从测试结果上可以看到，SQ 因为被查询向量的标量量化操作拖慢了速度所以较慢,PQ 在速度上有很大的提高,IVF-PQ 则是在维持准确率的同时进一步提升了速度。

表 1: 各算法性能

指标	flat	SQ	PQ	IVF-PQ
recall@10	1.000000	0.903353	0.903054	0.902904
latency (μs)	6300.170	3437.100	1011.580	631.425

```
load data /anndata/DEEP100K.query.fbin
dimension: 96  number:10000  size_per_element:4
load data /anndata/DEEP100K.gt.query.100k.top100.bin
dimension: 100  number:10000  size_per_element:4
load data /anndata/DEEP100K.base.100k.fbin
dimension: 96  number:100000  size_per_element:4
average recall: 1
average latency (us): 5425.13
```

图 2.1: flat

```
load data /anndata/DEEP100K.query.fbin
dimension: 96  number:10000  size_per_element:4
load data /anndata/DEEP100K.gt.query.100k.top100.bin
dimension: 100  number:10000  size_per_element:4
load data /anndata/DEEP100K.base.100k.fbin
dimension: 96  number:100000  size_per_element:4
成功读入sq_compressed_base.bin!
average recall: 0.903353
average latency (us): 2831.76
```

图 2.2: SQ

```
[s2312506@master_ubss1 ann]$ bash test.sh 1 1
Submitted job with ID: 13080.master_ubss1

成功读入pq_kmeans_base.bin!
成功读入pq_kmeans_centers.bin!
成功读入pq_kmeans_centers_distance.bin!
average recall: 0.903054
average latency (us): 1020.52
```

图 2.3: PQ

```
load data /anndata/DEEP100K.query.fbin
dimension: 96  number:10000  size_per_element:4
load data /anndata/DEEP100K.gt.query.100k.top100.bin
dimension: 100  number:10000  size_per_element:4
load data /anndata/DEEP100K.base.100k.fbin
dimension: 96  number:100000  size_per_element:4
成功读入inverted_kmeans.bin!
成功读入ivf_centers.bin!
成功读入ivf_base.bin!
成功读入ivfpq_kmeans_base.bin!
成功读入ivfpq_kmeans_centers.bin!
average recall: 0.902903
average latency (us): 607.341
```

图 2.4: IVFPQ

2.1 SDC 和 ADC

计算查询向量与原始向量距离时, SDC 计算的是两个类中心之间的距离, 而 ADC 计算的是查询向量与类中心的距离。显然我们这次实验的测试查询向量没办法提前量化，量化的时间应该纳入 latency 中。所以使用了 ADC。二者的性能我也做了对比。

两次实验的 rerank 都设置为 600, 其他参数都相同。

```
load data /anndata/DEEP100K.query.fbin
dimension: 96  number:10000  size_per_element:4
load data /anndata/DEEP100K.gt.query.100k.top100.bin
dimension: 100  number:10000  size_per_element:4
load data /anndata/DEEP100K.base.100k.fbin
dimension: 96  number:100000  size_per_element:4
成功读入pq_kmeans_base.bin!
成功读入pq_kmeans_centers.bin!
成功读入pq_kmeans_centers_distance.bin!
average recall: 0.705901
average latency (us): 1918.82
```

图 2.5: SDC

```
[s2312506@master_ubss1 ann]$ bash test.sh 1 1
Submitted job with ID: 13080.master_ubss1

成功读入pq_kmeans_base.bin!
成功读入pq_kmeans_centers.bin!
成功读入pq_kmeans_centers_distance.bin!
average recall: 0.903054
average latency (us): 1020.52
```

图 2.6: ADC

可以看到 SDC 在正确率上远低于 ADC。甚至反而因为 SDC 需要大量偏移访问 pq_kmeans_centers 里的数据，速度严重落后于 ADC。所以选择 ADC 更优。

2.2 rerank

由于粗排的召回率很低，所以我们需要对一个大致范围的 rerank 做精细重排。rerank 若太低则召回率低，rerank 太大又会带来不必要的时间损耗。所以需要确定最优的 rerank 值。

2.2.1 SQ

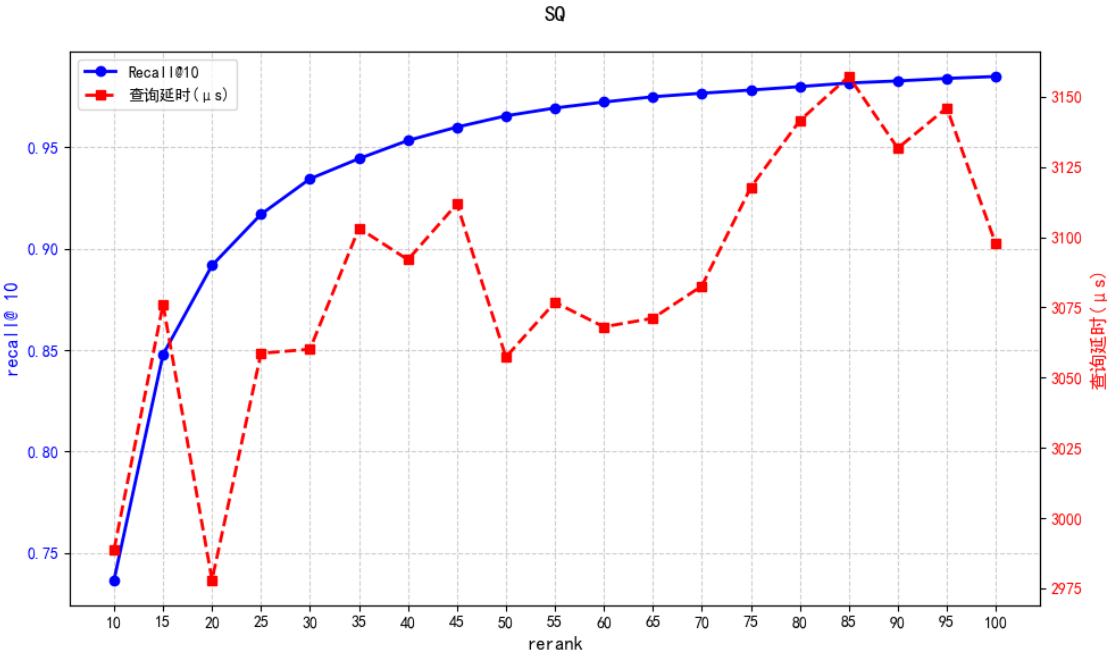

图 2.7: SQ_rerank

SQ 的图已经是取十次平均的值了，由于是最后重新补的实验，服务器已经乱七八糟了，而且 rerank 之间的差距很小，所以图是一坨，将就看罢。

2.2.2 PQ

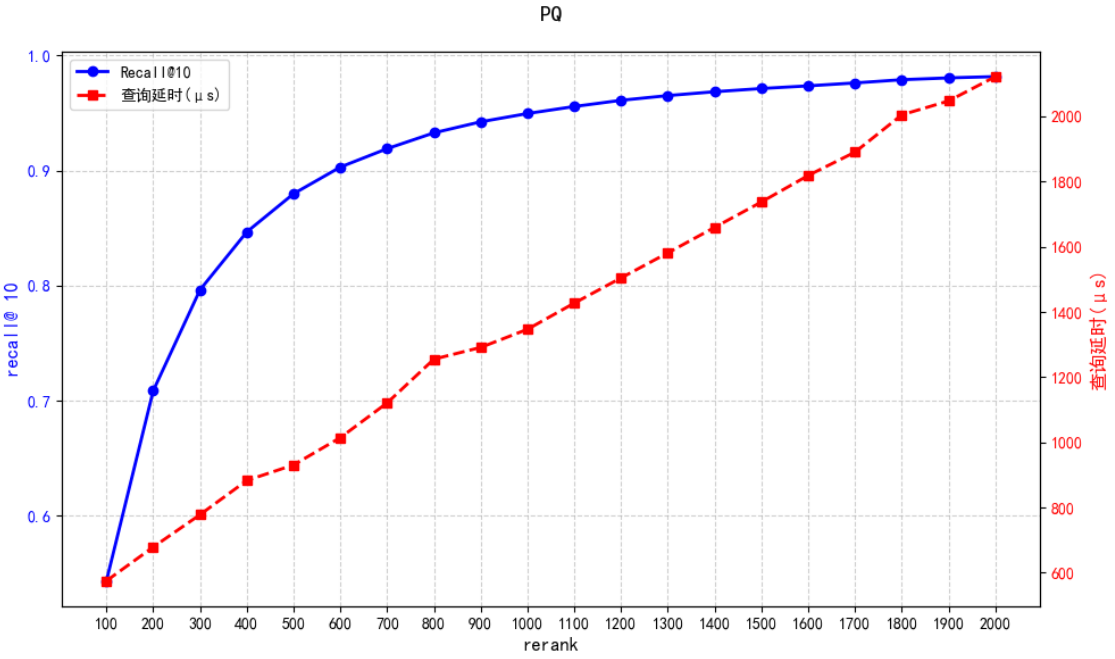

图 2.8: PQ_rerank

2.2.3 IVFPQ

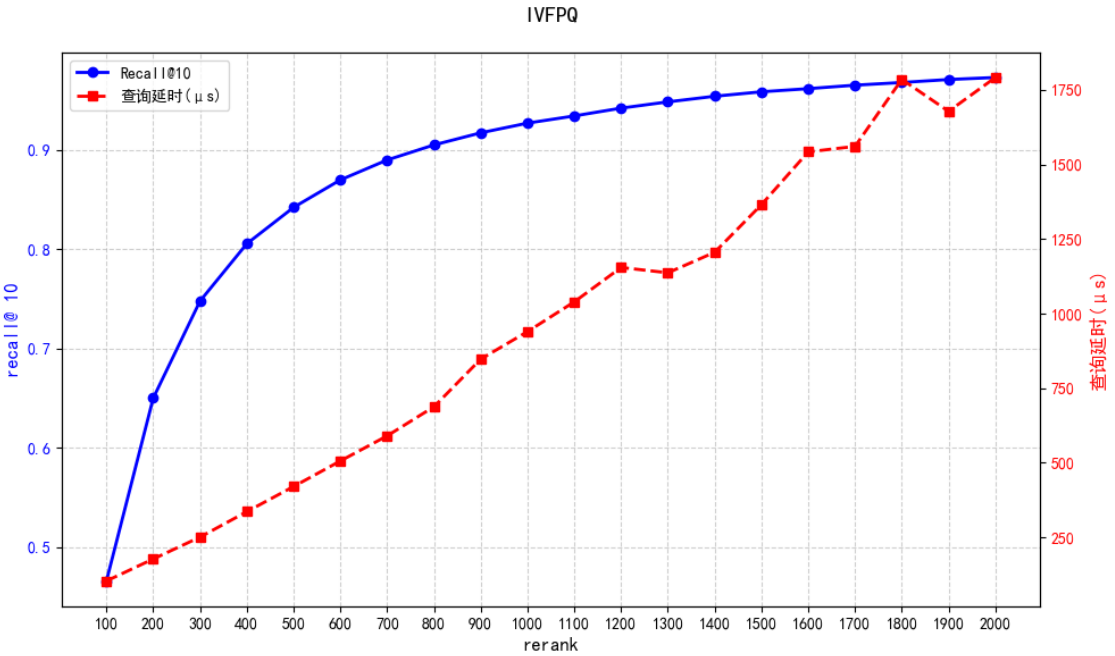

图 2.9: IVFPQ_rerank

2.3 IVF-PQ 候选簇数量

虽然 IVF-PQ 通过粗簇筛选掉了大量冗余数据，但这同时也可能会漏掉真实的最近邻向量。假设搜索时检索 m 个候选簇，对于这 m 个候选簇我们不但要遍历其中的所有向量分别计算距离，而且由于我们 PQ 量化的是残差，我们还要先分别算出查询向量和各候选簇的残差才能用来计算距离。这个计算开销是巨大的。

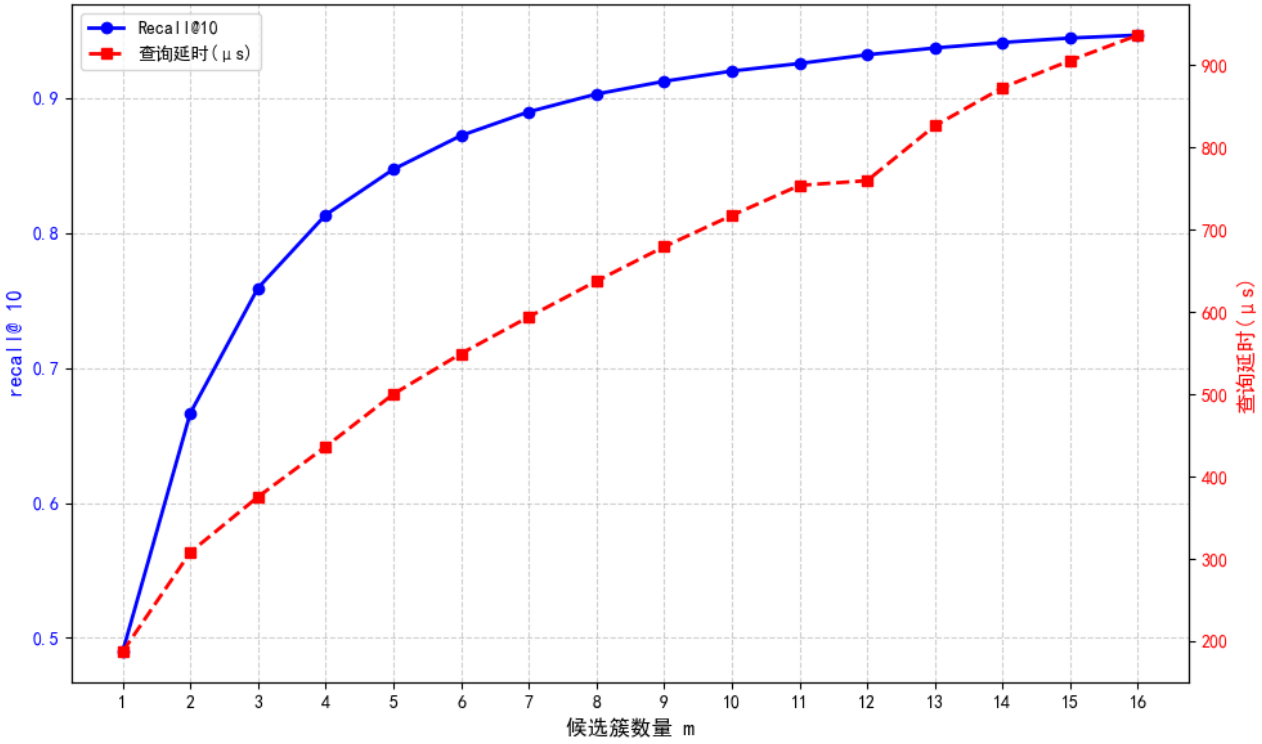

图 2.10: 候选簇数量

可以看到召回率随候选簇数量的涨幅就越来越小，但查询时间却是几乎线性地在增长。由于候选簇数量 m 和 rerank 的设置是围绕召回率等于 0.9 设置的，所以二者平衡即可。

2.4 其他参数

还有一些其他的参数比如 test_number，PQ 的聚类中心数量，IVF 粗聚类中心数量等需要考虑。但尝试修改测试过后发现**影响很小**。例如 IVF 粗聚类中心设置为 256 和 1024，对应的设置候选簇数量为 16 和 256 时，二者的 recall@k 几乎没有区别，后者反而因为计算开销导致 latency 升高。

3 总结

这次实验虽然是 SIMD 编程实验，但我几乎全都在忙着实现各种算法。尤其是我在没反应过来要做精排的时候，光拿着 PQ 算法算出来的结果去测召回率，只有可怜的 30%，让我一度怀疑人生以为是算法实现的有问题。明明提前一周就做完的工作，直到最后一天，才从老师发的”调整 rerank 来提高召回率”中恍然大悟要再做一次精排。导致各种实验数据都得重新跑，但服务器已经变成一坨根本运行不了。难得偶尔得到一次结果，延时数据也是乱七八糟的不稳定。烂完了。

这次实验我用 neon 进行 SIMD 编程，对欧式距离、内积距离的计算，标量量化过程，搜索时的查询向量归类过程做了并行优化，实现了 arm 平台上查询速度的显著提高。

更多的 ANNS 算法实现和并行优化就留到期末报告里实现罢。

Git 项目链接：https://github.com/SheepSpaceFly/NKHW/tree/main/并行程序设计/SIMD 编程实验