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 24 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

```
float get_query_dis(float *query1, float *query2, int vecdim){
    float32x4_t sum_vec = vdupq_n_f32(0);
    for(int k=0;k<vecdim;k+=4){
        float32x4_t vec_1 = vld1q_f32(query1+k);
        float32x4_t vec_2 = vld1q_f32(query2+k);
        float32x4_t mul_vec = vmulq_f32(vec_1,vec_2);
        sum_vec = vaddq_f32(sum_vec,mul_vec);
    }
    float tmp[4];
    vst1q_f32(tmp,sum_vec);
    float dis = tmp[0]+tmp[1]+tmp[2]+tmp[3];
    dis = 1-dis;
    return 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

```
void sq_compress_base(float* base, uint8_t* compressed_base, size_t base_number,
    size_t vecdim){
    int byte_number = base_number*vecdim;

    float32x4_t min_vec = vdupq_n_f32(INFINITY);
    float32x4_t max_vec = vdupq_n_f32(-INFINITY);

    for(int i=0;i+4<=byte_number;i+=4){
```

```
        float32x4_t vec = vld1q_f32(base+i);
        min_vec = vminq_f32(min_vec,vec);
        max_vec = vmaxq_f32(max_vec,vec);
    }
    float32x4_t temp1 = vminq_f32(min_vec, vrev64q_f32(min_vec));
    float32x2_t temp2 = vpmin_f32(vget_low_f32(temp1), vget_high_f32(temp1));
    float min_val = vget_lane_f32(temp2, 0);
    temp1 = vmaxq_f32(max_vec, vrev64q_f32(max_vec));
    temp2 = vpmax_f32(vget_low_f32(temp1), vget_high_f32(temp1));
    float max_val = vget_lane_f32(temp2, 0);
    for(int i=(byte_number/4)*4;i<byte_number;i++){
        if(base[i]<min_val) min_val = base[i];
        if(base[i]>max_val) max_val = base[i];
    }

    if(min_val==max_val){
        uint8x16_t zero_vec = vdupq_n_u8(0);
        for(int i=0;i<=byte_number-16;i+=16){
            vst1q_u8(compressed_base+i,zero_vec);
        }
        for(int i=(byte_number/16)*16;i<byte_number;i++){
            compressed_base[i] = 0;
        }
        return;
    }

    float scale = 255.0f/(max_val-min_val);
    float32x4_t vmin = vdupq_n_f32(min_val);
    float32x4_t vscale = vdupq_n_f32(scale);

    for(int i=0;i+16<=byte_number;i+=16){
        float32x4_t x0 = vld1q_f32(base+i);
        float32x4_t x1 = vld1q_f32(base+i+4);
        float32x4_t x2 = vld1q_f32(base+i+8);
        float32x4_t x3 = vld1q_f32(base+i+12);

        x0 = vsubq_f32(x0, vmin);
        x1 = vsubq_f32(x1, vmin);
        x2 = vsubq_f32(x2, vmin);
        x3 = vsubq_f32(x3, vmin);
        x0 = vmulq_f32(x0, vscale);
        x1 = vmulq_f32(x1, vscale);
        x2 = vmulq_f32(x2, vscale);
        x3 = vmulq_f32(x3, vscale);
        x0 = vmaxq_f32(vminq_f32(x0, vdupq_n_f32(255.0f)), vdupq_n_f32(0.0f));
        x1 = vmaxq_f32(vminq_f32(x1, vdupq_n_f32(255.0f)), vdupq_n_f32(0.0f));
        x2 = vmaxq_f32(vminq_f32(x2, vdupq_n_f32(255.0f)), vdupq_n_f32(0.0f));
        x3 = vmaxq_f32(vminq_f32(x3, vdupq_n_f32(255.0f)), vdupq_n_f32(0.0f));
        int32x4_t y0 = vcvtnq_s32_f32(x0);
```

```
        int32x4_t y1 = vcvtnq_s32_f32(x1);
        int32x4_t y2 = vcvtnq_s32_f32(x2);
        int32x4_t y3 = vcvtnq_s32_f32(x3);
        int16x4_t y0_16 = vmovn_s32(y0);
        int16x4_t y1_16 = vmovn_s32(y1);
        int16x4_t y2_16 = vmovn_s32(y2);
        int16x4_t y3_16 = vmovn_s32(y3);
        int16x8_t y01 = vcombine_s16(y0_16, y1_16);
        int16x8_t y23 = vcombine_s16(y2_16, y3_16);
        uint8x8_t y01_u8 = vqmovun_s16(y01);
        uint8x8_t y23_u8 = vqmovun_s16(y23);
        vst1q_u8(compressed_base+i,vcombine_u8(y01_u8,y23_u8));
    }
    for(int i=(byte_number/16)*16;i<byte_number;i++){
        float x = base[i];
        float y = (x-min_val)*scale;
        y = std::max(0.0f,std::min(y,255.0f));
        compressed_base[i] = static_cast<uint8_t>(y+0.5f);
    }
}
```

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

```
float sq_l2_distance(uint8_t* query1, uint8_t* query2, size_t vecdim){//平方距离
    uint32x4_t sum_vec = vdupq_n_u32(0);
    for(int i=0;i<vecdim;i+=16){
        uint8x16_t x = vld1q_u8(query1+i);
        uint8x16_t y = vld1q_u8(query2+i);
        uint8x16_t diff = vabdq_u8(x,y);
        uint8x8_t diff_lo = vget_low_u8(diff);
        uint8x8_t diff_hi = vget_high_u8(diff);
        uint16x8_t diff2_lo = vmull_u8(diff_lo,diff_lo);
        uint16x8_t diff2_hi = vmull_u8(diff_hi,diff_hi);

        sum_vec = vaddq_u32(sum_vec,vpaddlq_u16(diff2_lo));
        sum_vec = vaddq_u32(sum_vec,vpaddlq_u16(diff2_hi));
    }
    uint32_t sum = vaddvq_u32(sum_vec);
    return 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

```
std::priority_queue<std::pair<float, uint32_t> > sq_search(uint8_t* base, uint8_t*
    query, size_t base_number, size_t vecdim, size_t k) {
    std::priority_queue<std::pair<float, uint32_t> > q;

    for(int i = 0; i < base_number; ++i) {
        float dis = 0;
        uint8_t* base_query = base+i*vecdim;

        dis = sq_l2_distance(query,base_query,vecdim);

        if(q.size() < k) {
            q.push({dis, i});
        } else {
            if(dis < q.top().first) {
                q.push({dis, i});
                q.pop();
            }
        }
    }
    return q;
}
```

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

```
void kmeans_compress_base(float *base, size_t base_number, size_t vecdim){
    const int K = 256;
    uint8_t *compressed_base = new uint8_t[base_number*4];
    float *centers = new float[vecdim*K];

    for(int i=0;i<4;i++){
        float *tmp_base = new float[base_number*vecdim/4];
        for(int j=0;j<base_number;j++){
            for(int k=0;k<vecdim/4;k++){
                tmp_base[j*vecdim/4+k] = base[i*vecdim/4+j*vecdim+k];
            }
        }

        uint8_t *tmp_labels = new uint8_t[base_number];
        float* tmp_centers = new float[vecdim/4*K];

        kmeans(tmp_base,tmp_labels,tmp_centers,base_number,vecdim/4,K);

        for(int j=0;j<base_number;j++){
            compressed_base[j*4+i] = tmp_labels[j];
        }

        for(int j=0;j<K;j++){
            for(int k=0;k<vecdim/4;k++){
                centers[(i*K + j)*vecdim/4 + k] = tmp_centers[j*vecdim/4 + k];
            }
        }
        delete[] tmp_base;
        delete[] tmp_labels;
        delete[] tmp_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

```
std::priority_queue<std::pair<float, uint32_t> > pq_search(uint8_t* compressed_base,
    float *centers, float *centers_dis, float* query, size_t base_number, size_t
    vecdim, size_t k_top) {
    const int K=256;
    // int query_labels[4];
    float query_centers_dis[4][256];

    for(int i=0;i<4;i++){
        float min_dis = INFINITY;
```

```
        for(int j=0;j<256;j++){
            float *center = centers+i*K*vecdim/4+j*vecdim/4;
            float *sub_query = query+i*vecdim/4;
            float32x4_t sum_vec = vdupq_n_f32(0);
            for(int k=0;k<vecdim/4;k+=4){
                float32x4_t vec_1 = vld1q_f32(center+k);
                float32x4_t vec_2 = vld1q_f32(sub_query+k);
                float32x4_t diff = vsubq_f32(vec_1, vec_2);
                float32x4_t squared_diff  = vmulq_f32(diff,diff);
                sum_vec = vaddq_f32(sum_vec,squared_diff);
            }
            float sum = vaddvq_f32(sum_vec);
            // if(dis<min_dis){
            //     min_dis = dis;
            //     query_labels[i] = j;
            // }
            query_centers_dis[i][j] = sum;
        }
    }

    std::priority_queue<std::pair<float, uint32_t>> q;

    for(int i = 0; i < base_number; i++) {
        float dis = 0;
        for(int j=0;j<4;j++){
            // dis+=centers_dis[j*256*256+compressed_base[i*4+j]*256+query_labels[j]];
            dis+=query_centers_dis[j][compressed_base[i*4+j]];
        }
        if(q.size() < k_top) {
            q.push({dis, i});
        } else {
            if(dis < q.top().first) {
                q.push({dis, i});
                q.pop();
            }
        }
    }
    return q;
}
```

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

```
void ivfpq_compress_base(float *base, uint8_t* ivf_base, float* ivf_centers, size_t
    base_number, size_t vecdim){
    const int K = 256;//聚类中心数
    uint8_t *compressed_base = new uint8_t[base_number*4];
    float *centers = new float[vecdim*K];

    for(int i=0;i<4;i++){
        float *tmp_base = new float[base_number*vecdim/4];
        for(int j=0;j<base_number;j++){
            for(int k=0;k<vecdim/4;k++){
                tmp_base[j*vecdim/4+k] =
                    base[j*vecdim+i*(vecdim/4)+k]-ivf_centers[ivf_base[j]*vecdim+i*vecdim/4+k];
            }
        }

        uint8_t *tmp_labels = new uint8_t[base_number];
        float* tmp_centers = new float[vecdim/4*K];

        kmeans(tmp_base,tmp_labels,tmp_centers,base_number,vecdim/4,K);

        for(int j=0;j<base_number;j++){
            compressed_base[j*4+i] = tmp_labels[j];
        }

        for(int j=0;j<K;j++){
            for(int k=0;k<vecdim/4;k++){
                centers[(i*K + j)*vecdim/4 + k] = tmp_centers[j*vecdim/4 + k];
            }
        }
        delete[] tmp_base;
        delete[] tmp_labels;
        delete[] tmp_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

```
float get_query_dis(float *query1, float *query2, int vecdim){
    float32x4_t sum_vec = vdupq_n_f32(0);
    for(int k=0;k<vecdim;k+=4){
        float32x4_t vec_1 = vld1q_f32(query1+k);
        float32x4_t vec_2 = vld1q_f32(query2+k);
        float32x4_t diff = vsubq_f32(vec_1, vec_2);
        float32x4_t squared_diff  = vmulq_f32(diff,diff);
        sum_vec = vaddq_f32(sum_vec,squared_diff);
    }
    float sum = vaddvq_f32(sum_vec);
    return 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

```
std::priority_queue<std::pair<float, uint32_t>> ivfpq_search(uint8_t*
    compressed_base, float *centers, float *ivf_centers,std::vector<int>
    *inverted_kmeans, float* query, size_t base_number, size_t vecdim, size_t k_top) {
    const int K=256;

    int m=16;
    int *query_ivf_labels = new int[m];

    std::priority_queue<std::pair<float, uint32_t>> ivf_q;

    for(int i = 0; i < 256; i++) {
        float dis = get_query_dis(ivf_centers+i*vecdim, query, vecdim);
        if(ivf_q.size() < m) {
            ivf_q.push({dis, i});
        } else {
            if(dis < ivf_q.top().first) {
                ivf_q.push({dis, i});
                ivf_q.pop();
            }
        }
    }

    for(int i=0;i<m;i++){
        query_ivf_labels[i] = ivf_q.top().second;
```

```
        ivf_q.pop();
    }

    int *size = new int[m];
    int **candidate_base = new int*[m];
    float **residual = new float*[m];
    for(int i=0; i<m; i++){
        size[i] = inverted_kmeans[query_ivf_labels[i]].size();
        candidate_base[i] = inverted_kmeans[query_ivf_labels[i]].data();
        residual[i] = new float[vecdim];

        float* center_ptr = &ivf_centers[query_ivf_labels[i]*vecdim];
        float* residual_ptr = residual[i];

        for(int j=0; j<vecdim; j+=4){
            float32x4_t center = vld1q_f32(center_ptr + j);
            float32x4_t qvec = vld1q_f32(query + j);
            float32x4_t residual_vec = vsubq_f32(qvec, center);
            vst1q_f32(residual_ptr + j, residual_vec);
        }
    }

    std::priority_queue<std::pair<float, uint32_t>> q;

    for(int i = 0; i < m; i++) {
        for(int j=0;j<size[i];j++){
            int candidate_query = candidate_base[i][j];
            float dis = 0;
            for(int k=0;k<4;k++){
                dis+=get_query_dis(residual[i]+k*vecdim/4,centers+k*256*vecdim/4+compressed_base[c
                    vecdim/4);
            }
            if(q.size() < k_top) {
                q.push({dis, candidate_query});
            } else {
                if(dis < q.top().first) {
                    q.push({dis, candidate_query});
                    q.pop();
                }
            }
        }
    }
    return q;
}
```

2 性能分析

所有实验结果都是在服务器上取得的。其中 PQ 的正确率只有 30%，IVFPQ 也只有 40%，说实话我也觉得太低了。但我已经被这个正确率硬控 5 天了，代码怎么也找不出来哪里出了问题，我只能默认这个结果是正常的了。

latency 取 10 次实验结果的平均值。默认 test_number 为 2000。k_top 默认为 100。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。

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，并没有和经过 neon 并行优化后的 flat 拉开太大的差距。PQ 虽然速度上有了很大的提高，但准确率大大降低。IVF-PQ 则是在速度进一步提升的同时更好地维持了准确率。

表 1: 各算法性能

指标	flat	SQ	PQ	IVF-PQ
recall@100	1.000000	0.809763	0.315720	0.40342
latency (μs)	6300.170	3022.926	571.951	248.692

```
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: 0.809763
average latency (us): 3031.12
```

图 2.2: SQ

```
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: 0.31572
average latency (us): 571.253
```

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: 0.40342
average latency (us): 258.848
```

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。

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。

```
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: 0.22845
average latency (us): 1417.8
```

图 2.5: SDC

```
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: 0.31572
average latency (us): 571.253
```

图 2.6: ADC

2.2 k_top

k_top 的高低在 latency 上影响较小，因为它只会影响维护最大堆消耗的时间。而无论是 SQ、PQ 还是 IVF-PQ，它们遍历向量的数量都和 k_top 无关。所以 latency 就不画图了。

k_top 在 recall@k 上的影响就非常大了。毕竟 k_top 越小，搜索结果的范围越来越小，召回率自然下降。

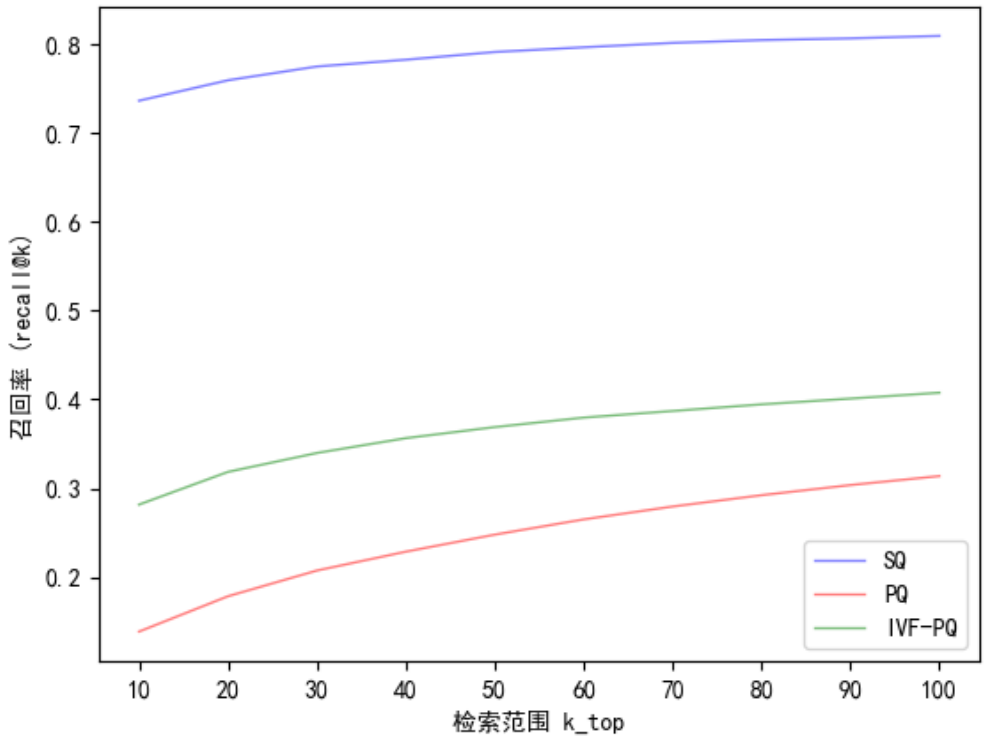

图 2.7: k_top

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。

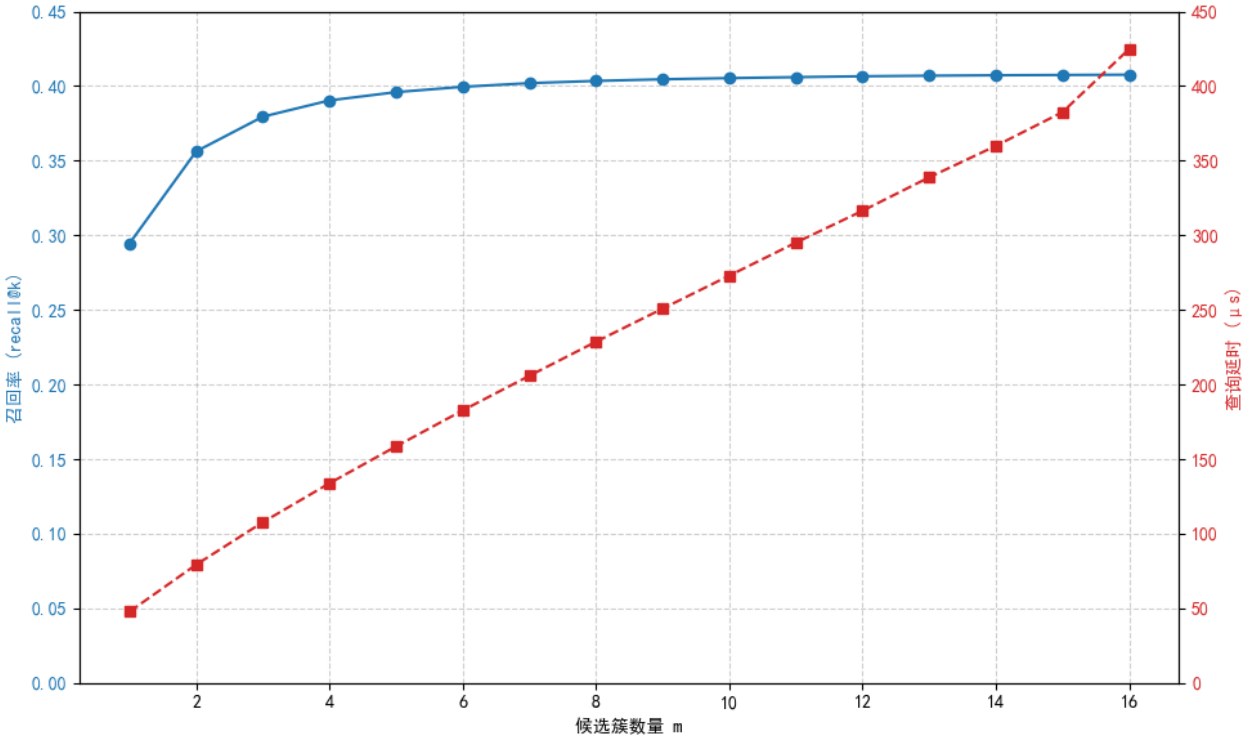

图 2.8: 候选簇数量 m

可以看到召回率在候选簇数量到达 4 时涨幅就不明显了，但查询时间却是线性地在增长。如果追求速度的话候选簇数量 m 可以设置为 4。但考虑到正确率，我选择设置的候选簇数量为 8。

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。尤其是在正确率上纠结了很长时间，期间换过距离计算公式，对聚类结果进行标准化，但完全没用。结果就是更多探索已经没时间实现了。考虑到这次实验的重点是 SIMD 编程，应该已经完成任务了吧（?）

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。