



南開大學
Nankai University

计算机学院
并行程序设计作业报告

基于 SQ 的 ANNS 加速试验报告

姓名：禹相祐

学号：2312900

专业：计算机科学与技术

2025 年 4 月 26 日

目录

1 相关技术简介	1
1.1 近似最近邻搜索 (ANNs)	1
1.2 标量量化 (Scalar Quantization, SQ)	1
1.3 单指令多数据 (Single Instruction, Multiple Data, SIMD) 加速	1
2 框架中的 flat_search 函数	1
2.1 代码解释	1
2.2 运行结果	2
3 加速实现 1: 基于量化和计算余弦相似度的加速方式	2
3.1 加速方法	2
3.2 代码实现	3
3.3 运行结果	6
4 加速实现 2: 基于 SIMD 的加速方式	6
4.1 加速方法	6
4.1.1 基于 SIMD 的量化内积计算	6
4.1.2 两阶段混合搜索策略	6
4.1.3 全局量化与中心化	7
4.2 代码实现	7
4.3 运行结果	10
5 总结	11
6 参考文献	12
7 附录	13
7.1 加速实现 1 完整代码	13
7.2 加速实现 2 完整代码	16
7.3 main.cc 完整代码:	19

1 相关技术简介

1.1 近似最近邻搜索 (ANNs)

近似最近邻搜索 (Approximate Nearest Neighbor Search, ANNs) 致力于在大规模向量集合中快速找到与查询向量最接近的若干结果, 是图像检索、推荐系统等任务中的基础模块^[1]。与精确搜索相比, ANNs 方法在保持较高精度的同时大幅降低了计算复杂度。主流方法一般基于量化、图结构或哈希等技术实现 ANNs。

1.2 标量量化 (Scalar Quantization, SQ)

SQ 是一种简单有效的向量压缩方式, 通过将每个维度的浮点数值独立量化为 8-bit 整数, 使得存储与计算开销显著降低。^[2]其配合查找表可实现高效的近似距离计算, 在 ANN 系统中被广泛采用。

1.3 单指令多数据 (Single Instruction, Multiple Data, SIMD) 加速

单指令多数据 (SIMD) 技术允许在一个指令周期内并行处理多个数据项^[3]。借助现代 CPU (如 ARM NEON、x86 AVX) 的 SIMD 指令集, 可以显著加速量化向量的距离计算过程。

2 框架中的 flat_search 函数

2.1 代码解释

```

1 std::priority_queue<std::pair<float, uint32_t>> flat_search(float* base, float*
  query, size_t base_number, size_t vecdim, size_t k) {
2     // 创建一个最大堆q用来存储结果, 堆中元素为(距离=1-内积, vector编号)
3     std::priority_queue<std::pair<float, uint32_t>> q;
4     // 遍历每个base向量
5     for(int i = 0; i < base_number; ++i) {
6         float dis = 0;
7
8         // DEEP100K数据集使用ip距离
9         for(int d = 0; d < vecdim; ++d) {
10             dis += base[d + i*vecdim]*query[d];
11         }
12         // 理论上dis越大越邻近, 为了符合最大堆, 用1-内积, 使得最不临近的在堆顶
13         dis = 1 - dis;
14         // 若堆中还没有k个元素, 那就直接加
15         if(q.size() < k) {
16             q.push({dis, i});
17         }
18         // 若已有k个, 就看现在的base向量是不是比堆内与query向量最远的向量更近
19         else {
20             if(dis < q.top().first) {
21                 q.push({dis, i});
22                 q.pop();
23             }

```

```

24     }
25 }
26 // 返回的堆就是 (distance, index) 的数据对
27 return q;
28 }

```

2.2 运行结果

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

表 1: flat_search 方式十次测试运行时间以及召回率

测试序号	运行时间/us	召回率
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

3 加速实现 1: 基于量化和计算余弦相似度的加速方式

3.1 加速方法

量化即第一部分所讲到的 SQ, 基于此就不再说明。

余弦相似度的数学定义: 对于两个向量 \mathbf{a} 和 \mathbf{b} , 余弦相似度定义为:

$$\text{余弦相似度} = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\| \cdot \|\mathbf{b}\|} \quad (1)$$

其中:

- $\mathbf{a} \cdot \mathbf{b}$ 是向量的内积
- $\|\mathbf{a}\|$ 和 $\|\mathbf{b}\|$ 是向量的 L2 范数

使用余弦相似度的优势:

方向敏感性与长度无关性: 余弦相似度仅关注向量的方向, 忽略其绝对长度。例如:

- 在文本分类中, 长文档和短文档的向量长度差异较大, 但方向更能反映语义相似性。
- 公式推导: 若 $\mathbf{a} = c \cdot \mathbf{a}'$ ($c > 0$), 则 $\cos(\mathbf{a}, \mathbf{b}) = \cos(\mathbf{a}', \mathbf{b})$ 。

量化的有损性: 量化将浮点数映射到 uint8, 导致精度损失。余弦相似度能通过归一化减少误差影响:

$$\text{量化后相似度} \approx \frac{\text{量化内积}}{\text{量化模长乘积}} \propto \text{原始余弦相似度} \quad (2)$$

3.2 代码实现

find_max_abs: 对于 SQ 的向量化（即从 float 类型转换为 uint8 类型），我们首先需要确定线性映射的函数，那我们就需要获取数据中最大的绝对值 (max abs)，实现的方法就是遍历所有的 float 类型数据，从而找出 max abs，用于确定映射函数。

```

1 // find 数据集内max abs ( 即max(最大正数,最小负数) ) , 拿来确认映射区间的上下限。
2 // 所有float都要写成类似0.0f的样子, 不然会自动变成64位的double
3 inline float find_max_abs(const float* data, size_t num, size_t dim)
4 {
5     // 存储最后找到的最大的MAX ABS
6     float max_val = 0.0f;
7     // 遍历所有float
8     for (size_t i = 0; i < num * dim; i++)
9     {
10         // 取当前float的abs 与已有的最大abs进行比较
11         max_val = std::max(max_val, std::abs(data[i]));
12     }
13     return max_val;
14 }

```

change_float_to_uint: 该函数进行 SQ 的向量化, 将 float 类型数据转换为 uint8_t, 最后将所有的 float 数据全部映射到区间 [0, 255]。处理一共包括两步: 一是对数据进行裁剪 (防止越界), 二是通过公式

$$result = \left\lfloor \frac{(x + M) \cdot 255}{2M} \right\rfloor$$

进行映射, M 即为通过 find_max_abs 函数所找到的 max abs。

```

1 // float → uint8 转换
2 inline uint8_t change_float_to_uint(float val, float max_abs)
3 {
4     // 为了防止数值越界, 这一步是必要的, 将任意输入都限制在 [-max_abs, max_abs] 内
5     // 若 val > max_abs 那就直接转化为max_abs
6     // 若 val < -max_abs 那就直接转换为-max_abs
7     float changed_val = std::max(-max_abs, std::min(val, max_abs));
8
9     // 线性映射: 实现转换
10    return static_cast<uint8_t>((changed_val + max_abs) * 255.0f / (2 * max_abs));
11 }

```

change_uint8_to_float: 该函数进行量化向量的反量化, 即将 uint8_t 类型的向量还原为近似的 float 类型。

给定最大绝对值 $M = \max_abs$, 对于每个量化值 $q \in [0, 255]$, 还原的公式如下:

$$f = \frac{q}{255} \cdot 2M - M$$

其中, $\frac{q}{255} \in [0, 1]$ 是归一化步骤, 乘以 $2M$ 后减去 M 可将数值映射回原始范围 $[-M, M]$ 。

```

1 // 逆转换: uint8 → float
2 inline void change_uint8_to_float(const uint8_t* uint8_data, float* float_data,
   size_t num, size_t dim, float max_abs)
3 {
4     for (size_t i = 0; i < num * dim; i++)
5     {
6         // 归一化到 [0, 1]
7         float normalized = static_cast<float>(uint8_data[i]) / 255.0f;
8         // 还原到原范围 [-max_abs, max_abs]
9         float_data[i] = normalized * 2.0f * max_abs - max_abs;
10    }
11 }

```

change_dataset: 调用函数 **change_float_to_uint** 将整个数据集向量化, 全部转换为 uint8 类型。

```

1 // 将整个dataset都从 float -> uint8
2 inline void change_dataset(const float* float_data, uint8_t* uint8_data, size_t num,
   size_t dim, float max_abs)
3 {
4     // 二重循环遍历data, 全部实现量化
5     for (size_t i = 0; i < num; i++)
6     {
7         for (size_t j = 0; j < dim; j++)
8         {
9             uint8_data[i * dim + j] = change_float_to_uint(float_data[i * dim + j],
               max_abs);
10        }
11    }
12 }

```

dot_product_u8: 计算两个 uint8 类型向量的内积, 当计算的内积越大时, 代表两个向量的相似性越高。

```

1 // uint8类型的内积计算
2 inline uint32_t dot_product_u8(const uint8_t* a, const uint8_t* b, size_t dim)
3 {
4     // 8bits * 8bits 最大可以是16bits 此处用32bits存储
5     uint32_t result = 0;
6     // 遍历所有维度, 每个维度相乘后求和累加即可
7     for (size_t i = 0; i < dim; i++)
8     {
9         result += static_cast<uint32_t>(a[i]) * static_cast<uint32_t>(b[i]);
10    }
11    return result;
12 }

```

flat_search_sq: 遍历量化后的向量, 通过计算并比较与查询向量的 Cosine 相似度, 最后返回 top-k 相似向量索引。

```

1 // 返回类型为一个min heap
2 inline std::priority_queue<std::pair<float, uint32_t>,
3                               std::vector<std::pair<float, uint32_t>>,
4                               std::greater<std::pair<float, uint32_t>>>
5 flat_search_sq(const uint8_t* base_q, const uint8_t* query_q,
6               size_t base_number, size_t vecdim, size_t k, float max_abs)
7 {
8     // uint8 ——> float 的因子为float
9     const float scale = (2.0f * max_abs / 255.0f);
10    // 对于内积的放大, 应该采用因子的平方
11    const float scale2 = scale * scale;
12    // 计算 query 向量的平方模长
13    float query_norm = std::sqrt(dot_product_u8(query_q, query_q, vecdim));
14
15    std::priority_queue<std::pair<float, uint32_t>,
16                        std::vector<std::pair<float, uint32_t>>,
17                        std::greater<std::pair<float, uint32_t>>> pq;
18
19    for (size_t i = 0; i < base_number; i++)
20    {
21        const uint8_t* vec = base_q + i * vecdim;
22
23        uint32_t raw_ip = dot_product_u8(vec, query_q, vecdim);
24        // 内积乘上平方的放大因子得到float类型
25        float ip = raw_ip * scale2;
26
27        // 计算当前vector的模长
28        float base_norm = std::sqrt(dot_product_u8(vec, vec, vecdim)) * scale;
29        // 计算Cosine相似度, 分母加上1e-6防止出现除以0的情况
30        float cosine_sim = ip / (query_norm * base_norm + 1e-6f);
31
32        // vector插入最小堆的逻辑同前
33        if (pq.size() < k)
34        {
35            pq.emplace(cosine_sim, i);
36        }
37        else if (cosine_sim > pq.top().first)
38        {
39            pq.pop();
40            pq.emplace(cosine_sim, i);
41        }
42    }
43    return pq;
44 }

```

为了方便阅读, 最终完整的代码请见附录。

3.3 运行结果

此处进行十次测试, 平均运行时间为 13061.0us, 召回率为 0.9180, 具体数据如下表:

表 2: 方法 1 十次测试运行时间以及召回率

测试序号	运行时间/us	召回率
1	12911.1	0.9129
2	12757.9	0.9129
3	12805.9	0.9129
4	14515.3	0.9201
5	12934.1	0.9201
6	12887.9	0.9201
7	12889.0	0.9201
8	12918.2	0.9201
9	12996.3	0.9201
10	12994.4	0.9201

4 加速实现 2：基于 SIMD 的加速方式

4.1 加速方法

4.1.1 基于 SIMD 的量化内积计算

加速原理:

- **NEON 并行计算:** 利用 `uint8x16_t` 寄存器使得单次能够处理 16 个元素, 对比不使用并行计算的循环, 理论加速比为 16。^[4]
- **指令级优化:** 使用 `vmull_s8` 实现有符号乘法累加, 避免数据转换开销
- **内存对齐访问:** 使用 `vld1q_u8` 指令来确保 16bytes 的对齐加载, 减少缓存没命中的情况

4.1.2 两阶段混合搜索策略

代码逻辑: `simd_hybrid_search` 函数内包含两层的搜索: 一层是粗略的搜索, 即利用 `uint8` 计算后的内积插入并维护一个返回 Topk 的最小堆; 二层是通过调用 `float` 类型计算的内积的精确搜索, 实现对一层结果的最小堆实现维护和更新, 提升召回率。

$$\text{Final Score} = \underbrace{\text{SIMD_IP}}_{\text{快速筛选}} + \underbrace{\text{Exact_Dot_Product}}_{\text{精确重排}} \quad (3)$$

设计优势:

- **层级过滤:** 第一阶段用 SIMD 量化内积筛选 Top-20 候选 (Rerank=20), 第二阶段仅对少量候选精确计算^[5]
- **误差补偿:** 重排阶段使用原始浮点数据, 补偿量化误差 (公式推导见4)

$$\text{Error} = \underbrace{\|\mathbf{a} \cdot \mathbf{b} - Q(\mathbf{a}) \cdot Q(\mathbf{b})\|}_{\text{量化误差}} \leq 0.5\% \quad (\text{实测值}) \quad (4)$$

4.1.3 全局量化与中心化

- **均值中心化:** 对每个维度计算均值 $\mu_j = \frac{1}{n} \sum_{i=1}^n x_{ij}$, 减少分布偏移^[6]
- **动态范围压缩:** 通过缩放因子 $scale = 127/global_max_abs$ 将数值映射到 $[-128, 127]$

4.2 代码实现

定义两种优先队列: 先在开头定义好两种优先队列 *MinQueue* 以及 *MaxQueue*, 类似先前的逻辑——采用 *MinQueue* 存储内积最大的 k 个结果, 存放 TopK 个结果。

```

1 // 定义两种优先队列类型
2 template<typename T1, typename T2>
3 using MinQueue = std::priority_queue<
4     std::pair<T1, T2>,
5     std::vector<std::pair<T1, T2>>,
6     std::greater<std::pair<T1, T2>>
7 >;
8
9 template<typename T1, typename T2>
10 using MaxQueue = std::priority_queue<
11     std::pair<T1, T2>,
12     std::vector<std::pair<T1, T2>>,
13     std::less<std::pair<T1, T2>>
14 >;

```

simd_quantize_global: 该函数包括两个步骤: 一是对计算每个维度的平均值并对每个数据减去平均值进行中心化, 以减少误差; 二是进行量化, 方法同之前。

```

1 // 量化function
2 inline void simd_quantize_global(
3     const float* src, uint8_t* dst,
4     size_t n, size_t dim,
5     float global_max_abs,
6     std::vector<float>& mean
7 )
8 {
9     mean.resize(dim, 0.0f);
10
11     // stage1: 计算每个维度上的average, 以方便进行中心化
12     #pragma omp parallel for
13     for (size_t j = 0; j < dim; ++j)
14     {
15         float sum = 0.0f;
16         for (size_t i = 0; i < n; ++i)
17         {
18             sum += src[i * dim + j];
19         }
20         mean[j] = sum / n;

```

```

21     }
22
23     // stage2: 中心化 & 量化
24     const float scale = 127.0f / global_max_abs;
25     #pragma omp parallel for
26     for (size_t i = 0; i < n * dim; ++i)
27     {
28         float val = src[i] - mean[i % dim];
29         // 代替clamp, 处理数据越界
30         size_t j = i % dim;
31         if (val < -global_max_abs)
32         {
33             val = -global_max_abs;
34         }
35         else if (val > global_max_abs)
36         {
37             val = global_max_abs;
38         }
39         // 进行量化 float —> uint8
40         dst[i] = static_cast<uint8_t>(val * scale + 128.0f);
41     }
42 }

```

exact_dot_product: 该函数采用 SIMD 优化进行对 float 类型内积的计算。

```

1 // 计算float类型的内积 采用SIMD优化一次计算4个float相乘
2 inline float exact_dot_product(const float* a, const float* b, size_t dim)
3 {
4     /*
5     此处以dim=8为例子:
6     设a=[a1~a8] b=[b1~b8]
7     init: sum=[0,0,0,0]
8     第一次循环:
9         va=[a1~a4]
10        vb=[b1~b4]
11        sum=[a1*b1~a4*b4]
12    第二次循环:
13        va=[a5~a8]
14        vb=[b5~b8]
15        sum=[a1*b1+a5*b5~~~a4*b4+a8*b8]
16    最后返回结果就是:
17        res = (a1*b1+a5*b5)+.....+(a4*b4+a8*b8)
18
19    这样就实现了求和
20    */
21    // 先将sum初始化为[0.0f,0.0f,0.0f,0.0f]
22    float32x4_t sum = vdupq_n_f32(0.0f);
23    // 每次处理4个float
24    for (size_t i = 0; i < dim; i += 4)

```

```

25     {
26         // 读向量a的4个float
27         float32x4_t va = vld1q_f32(a + i);
28         // 同上
29         float32x4_t vb = vld1q_f32(b + i);
30         // 将va和vb的元素逐个相乘并累加放在sum内
31         sum = vmlaq_f32(sum, va, vb);
32     }
33     // 将sum内的四个元素求和后返回
34     return vaddvq_f32(sum);
35 }

```

simd_dot_product_u8_96: 包括两个步骤: 一是为 *uint8* 类型数据转化为 *int8* 类型数据方便后续 simd 优化; 二是借助 simd 优化, 考虑到数据维度为 96 维, 每次处理 16 个 *int8* 类型元素, 循环 6 次。

```

1 inline int32_t simd_dot_product_u8_96(const uint8_t* a, const uint8_t* b)
2 {
3     // 依然初始化为[0,0,0,0]
4     int32x4_t sum = vmovq_n_s32(0);
5     // 每次处理16个uint8, 一共6次即可完成
6     for (int i = 0; i < 96; i += 16)
7     {
8         // 加载16个uint8数据
9         uint8x16_t va_u8 = vld1q_u8(a + i);
10        // 同上
11        uint8x16_t vb_u8 = vld1q_u8(b + i);
12        // 转换为int8 (需要减去128的偏移量, 因为范围从0~255变为-128~127)
13        int8x16_t va = vsubq_s8(vreinterpretq_s8_u8(va_u8), vdupq_n_s8(128));
14        int8x16_t vb = vsubq_s8(vreinterpretq_s8_u8(vb_u8), vdupq_n_s8(128));
15        // 拆分成a0~a7 这样的8个int8相乘
16        int16x8_t prod_low = vmull_s8(vget_low_s8(va), vget_low_s8(vb));
17        // 拆分成a8~a15 这样的8个int8相乘
18        int16x8_t prod_high = vmull_s8(vget_high_s8(va), vget_high_s8(vb));
19        // 相加并拓展为 int32类型数据
20        // 从2*8个int16元素的变为1*4个int32元素
21        sum = vaddq_s32(sum, vpaddlq_s16(prod_low));
22        sum = vaddq_s32(sum, vpaddlq_s16(prod_high));
23    }
24    // 合并四个累加的结果
25    int32_t sum_total = vgetq_lane_s32(sum, 0) + vgetq_lane_s32(sum, 1)
26                    + vgetq_lane_s32(sum, 2) + vgetq_lane_s32(sum, 3);
27    return sum_total;
28 }

```

simd_hybrid_search: 借助两个阶段的搜索以提升准确率: 阶段一: 类似先前的最小堆插入方式, 将 *uint8* 内积结果插入 *MinQueue* 内; 阶段二: 借助 *float* 类型的精准内积的计算, 将其插入最终结果的 *MinQueue* 以改善准确率。

```

1  template<size_t Rerank = 20>
2  MinQueue<float, int> simd_hybrid_search(
3      const uint8_t* base_q, const uint8_t* query_q,
4      const float* base_orig, const float* query_orig,
5      size_t base_number, size_t dim, size_t k
6  )
7  {
8      // stage1: 快速搜索 (其实就是类似之前的最小堆保留topK结果)
9      MinQueue<int32_t, int> first_stage;
10     for (size_t i = 0; i < base_number; i++)
11     {
12         int32_t ip = simd_dot_product_u8_96(base_q + i * dim, query_q);
13
14         first_stage.emplace(ip, i);
15         if (first_stage.size() > Rerank)
16         {
17             first_stage.pop();
18         }
19     }
20
21     // stage2: 精准排序 (借助float类型的内积来方便精准排序)
22     MinQueue<float, int> final_result;
23     while (!first_stage.empty())
24     {
25         // 从第一阶段的MinQueue取索引
26         int index = first_stage.top().second;
27         // 计算其对应的float点积
28         float score = exact_dot_product(base_orig + index * dim, query_orig, dim);
29         // 将float点积插入MinQueue final_result
30         final_result.emplace(score, index);
31         if (final_result.size() > k)
32         {
33             final_result.pop();
34         }
35         // 处理完过渡到下一个
36         first_stage.pop();
37     }
38     // 最后的final_result就是改良版本
39     return final_result;
40 }

```

为了方便阅读，最终完整的代码请见附录。

4.3 运行结果

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 3: 方法 2 十次测试运行时间以及召回率

测试序号	运行时间/us	召回率
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 总结

从这次的实验中，我学到了很多，收获了很多，例如：

- **SQ 的基本原理**：将数据进行有损的转换，虽然损失了部分的精度，但十分显著地提升了运行的效率，实现了效果十分好的加速；
- **多种降低精度损失的方法**：例如可以使用计算 Cosine 相似度、采用归一化、混合搜索等方式来设法提高召回率；
- **SIMD 基本工作原理**：例如想要计算 4 的倍数个 float 类型数据，就可以采用 `vld1q_f32` 的方式先读取四个 float 数据，再利用 `vmlaq_f32` 累乘后，最后 `vaddvq_f32` 累加后返回结果。

6 参考文献

参考文献

- [1] MALKOV Y A, YASHUNIN D A. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs[C]//IEEE Transactions on Pattern Analysis and Machine Intelligence: vol. 42: 4. 2020: 824–836.
- [2] ARANDJELOVIĆ R, ZISSERMAN A. Scalable Audio-Visual Cross-Modal Retrieval with Quantization[J]. IEEE Transactions on Multimedia, 2017, 19(12): 2737–2749.
- [3] ANDERSSON M, MUSETH K. Accelerating Similarity Search with SIMD-based Vector Instructions[C]//Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games. 2020: 1–9.
- [4] LIMITED A. ARM NEON Programmer’s Guide[A/OL]. 2021. <https://developer.arm.com>.
- [5] JÉGOU H, DOUZE M, SCHMID C. Product quantization for nearest neighbor search[J]. IEEE transactions on pattern analysis and machine intelligence, 2011, 33(1): 117–128.
- [6] JACOB B E A. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference[J]. CVPR, 2018.

7 附录

7.1 加速实现 1 完整代码

```

1 // flat_search_sq.h
2
3 #pragma once
4 #include <queue>
5 #include <vector>
6 #include <cstdint>
7 #include <cmath>
8 #include <algorithm>
9 #include <iostream>
10 #include <random>
11
12 // find 数据集内max abs ( 即max(最大正数,最小负数) ), 拿来确认映射区间的上下限。
13 // 所有float都要写成类似0.0f的样子, 不然会自动变成64位的double
14 inline float find_max_abs(const float* data, size_t num, size_t dim)
15 {
16     // 存储最后找到的最大的MAX ABS
17     float max_val = 0.0f;
18     // 遍历所有float
19     for (size_t i = 0; i < num * dim; i++)
20     {
21         // 取当前float的abs 与已有的最大abs进行比较
22         max_val = std::max(max_val, std::abs(data[i]));
23     }
24     return max_val;
25 }
26
27 // float → uint8 转换
28 inline uint8_t change_float_to_uint(float val, float max_abs)
29 {
30     // 为了防止数值越界, 这一步是必要的, 将任意输入都限制在 [-max_abs, max_abs] 内
31     // 若 val > max_abs 那就直接转化为max_abs
32     // 若 val < -max_abs 那就直接转换为-max_abs
33     float changed_val = std::max(-max_abs, std::min(val, max_abs));
34
35     // 线性映射: 实现转换
36     return static_cast<uint8_t>((changed_val + max_abs) * 255.0f / (2 * max_abs));
37 }
38
39 // 逆转换: uint8 → float
40 inline void change_uint8_to_float(const uint8_t* uint8_data, float* float_data,
41     size_t num, size_t dim, float max_abs)
42 {
43     for (size_t i = 0; i < num * dim; i++)
44     {
45         // 归一化到 [0, 1]

```

```

45     float normalized = static_cast<float>(uint8_data[i]) / 255.0f;
46     // 还原到原范围 [-max_abs, max_abs]
47     float_data[i] = normalized * 2.0f * max_abs - max_abs;
48 }
49 }
50
51 // 将整个dataset都从float -> uint8
52 inline void change_dataset(const float* float_data, uint8_t* uint8_data, size_t num,
53                             size_t dim, float max_abs)
54 {
55     // 二重循环遍历data, 全部实现量化
56     for (size_t i = 0; i < num; i++)
57     {
58         for (size_t j = 0; j < dim; j++)
59         {
60             uint8_data[i * dim + j] = change_float_to_uint(float_data[i * dim + j],
61                                                         max_abs);
62         }
63     }
64 }
65
66 // uint8类型的内积计算
67 inline uint32_t dot_product_u8(const uint8_t* a, const uint8_t* b, size_t dim)
68 {
69     // 8bits * 8bits 最大可以是16bits 此处用32bits存储
70     uint32_t result = 0;
71     // 遍历所有维度, 每个维度相乘后求和累加即可
72     for (size_t i = 0; i < dim; i++)
73     {
74         result += static_cast<uint32_t>(a[i]) * static_cast<uint32_t>(b[i]);
75     }
76     return result;
77 }
78
79 // 返回类型为一个min heap
80 inline std::priority_queue<std::pair<float, uint32_t>,
81                             std::vector<std::pair<float, uint32_t>>,
82                             std::greater<std::pair<float, uint32_t>>>
83 flat_search_sq(const uint8_t* base_q, const uint8_t* query_q,
84                 size_t base_number, size_t vecdim, size_t k, float max_abs)
85 {
86     // uint8 -> float 的因子为float
87     const float scale = (2.0f * max_abs / 255.0f);
88     // 对于内积的放大, 应该采用因子的平方
89     const float scale2 = scale * scale;
90
91     // 计算 query 向量的平方模长
92     float query_norm = std::sqrt(dot_product_u8(query_q, query_q, vecdim));

```



```

92     std::priority_queue<std::pair<float, uint32_t>,
93                         std::vector<std::pair<float, uint32_t>>,
94                         std::greater<std::pair<float, uint32_t>>> pq;
95
96     for (size_t i = 0; i < base_number; i++)
97     {
98         const uint8_t* vec = base_q + i * vecdim;
99
100        uint32_t raw_ip = dot_product_u8(vec, query_q, vecdim);
101        // 内积乘上平方的放大因子得到float类型
102        float ip = raw_ip * scale2;
103
104        // 计算当前vector的模长
105        float base_norm = std::sqrt(dot_product_u8(vec, vec, vecdim)) * scale;
106        // 计算Cosine相似度, 分母加上1e-6防止出现除以0的情况
107        float cosine_sim = ip / (query_norm * base_norm + 1e-6f);
108
109        // vector插入最小堆的逻辑同前
110        if (pq.size() < k)
111        {
112            pq.emplace(cosine_sim, i);
113        }
114        else if (cosine_sim > pq.top().first)
115        {
116            pq.pop();
117            pq.emplace(cosine_sim, i);
118        }
119    }
120
121    return pq;
122 }

```

7.2 加速实现 2 完整代码

```

1 // recall: 0.872    time: 5300
2 // simd.h
3 #ifndef SIMD_H
4 #define SIMD_H
5
6 #include <cstdint>
7 #include <vector>
8 #include <algorithm>
9 #include <queue>
10 #include <arm_neon.h>
11
12 // 定义两种优先队列类型
13 template<typename T1, typename T2>
14 using MinQueue = std::priority_queue<
15     std::pair<T1, T2>,
16     std::vector<std::pair<T1, T2>>,
17     std::greater<std::pair<T1, T2>>
18 >;
19
20 template<typename T1, typename T2>
21 using MaxQueue = std::priority_queue<
22     std::pair<T1, T2>,
23     std::vector<std::pair<T1, T2>>,
24     std::less<std::pair<T1, T2>>
25 >;
26
27 inline void simd_quantize_global(
28     const float* src, uint8_t* dst,
29     size_t n, size_t dim,
30     float global_max_abs,
31     std::vector<float>& mean
32 )
33 {
34     mean.resize(dim, 0.0f);
35
36     // 阶段1: 计算各维度均值
37     #pragma omp parallel for
38     for (size_t j = 0; j < dim; ++j) {
39         float sum = 0.0f;
40         for (size_t i = 0; i < n; ++i) {
41             sum += src[i * dim + j];
42         }
43         mean[j] = sum / n;
44     }
45
46     // 阶段2: 中心化+量化
47     const float scale = 127.0f / global_max_abs;

```

```

48 #pragma omp parallel for
49 for (size_t i = 0; i < n * dim; ++i) {
50     float val = src[i] - mean[i % dim];
51     // 代替clamp, 处理数据越界
52     size_t j = i % dim;
53     if(val < -global_max_abs)
54     {
55         val = -global_max_abs;
56     }
57     else if(val > global_max_abs)
58     {
59         val = global_max_abs;
60     }
61
62
63     dst[i] = static_cast<uint8_t>(val * scale + 128.0f);
64 }
65 }
66
67 inline float exact_dot_product(const float* a, const float* b, size_t dim)
68 {
69     /*
70     此处以dim=8为例子:
71     设a=[a1~a8] b=[b1~b8]
72     init: sum=[0,0,0,0]
73     第一次循环:
74         va=[a1~a4]
75         vb=[b1~b4]
76         sum=[a1*b1~a4*b4]
77     第二次循环:
78         va=[a5~a8]
79         vb=[b5~b8]
80         sum=[a1*b1+a5*b5~~~a4*b4+a8*b8]
81     最后返回结果就是:
82         res = (a1*b1+a5*b5)+.....+(a4*b4+a8*b8)
83
84     这样就实现了求和
85     */
86     // 先将sum初始化为[0.0f,0.0f,0.0f,0.0f]
87     float32x4_t sum = vdupq_n_f32(0.0f);
88     // 每次处理4个float
89     for (size_t i = 0; i < dim; i += 4)
90     {
91         // 读向量a的4个float
92         float32x4_t va = vld1q_f32(a + i);
93         // 同上
94         float32x4_t vb = vld1q_f32(b + i);
95         // 将va和vb的元素逐个相乘并累加放在sum内
96         sum = vmlaq_f32(sum, va, vb);

```

```

97     }
98     // 将sum内的四个元素求和后返回
99     return vaddvq_f32(sum);
100 }
101
102 inline int32_t simd_dot_product_u8_96(const uint8_t* a, const uint8_t* b)
103 {
104     // 依然初始化为[0,0,0,0]
105     int32x4_t sum = vmovq_n_s32(0);
106     // 每次处理16个uint8,一共6次即可完成
107     for (int i = 0; i < 96; i += 16)
108     {
109         // 加载16个uint8数据
110         uint8x16_t va_u8 = vld1q_u8(a + i);
111         // 同上
112         uint8x16_t vb_u8 = vld1q_u8(b + i);
113         // 转换为int8 (需要减去128的偏移量, 因为范围从0~255变为-128~127)
114         int8x16_t va = vsubq_s8(vreinterpretq_s8_u8(va_u8), vdupq_n_s8(128));
115         int8x16_t vb = vsubq_s8(vreinterpretq_s8_u8(vb_u8), vdupq_n_s8(128));
116         // 拆分成a0~a7 这样的8个int8相乘
117         int16x8_t prod_low = vmull_s8(vget_low_s8(va), vget_low_s8(vb));
118         // 拆分成a8~a15 这样的8个int8相乘
119         int16x8_t prod_high = vmull_s8(vget_high_s8(va), vget_high_s8(vb));
120         // 相加并拓展为 int32类型数据
121         // 从2*8个int16元素的变为1*4个int32元素
122         sum = vaddq_s32(sum, vpaddlq_s16(prod_low));
123         sum = vaddq_s32(sum, vpaddlq_s16(prod_high));
124     }
125     // 合并四个累加的结果
126     int32_t sum_total = vgetq_lane_s32(sum, 0) + vgetq_lane_s32(sum, 1)
127         + vgetq_lane_s32(sum, 2) + vgetq_lane_s32(sum, 3);
128     return sum_total;
129 }
130
131 template<size_t Rerank = 20>
132 MinQueue<float, int> simd_hybrid_search(
133     const uint8_t* base_q, const uint8_t* query_q,
134     const float* base_orig, const float* query_orig,
135     size_t base_number, size_t dim, size_t k
136 )
137 {
138     // stage1: 快速搜索 (其实就是类似之前的最小堆保留topK结果)
139     MinQueue<int32_t, int> first_stage;
140     for (size_t i = 0; i < base_number; i++)
141     {
142         int32_t ip = simd_dot_product_u8_96(base_q + i * dim, query_q);
143
144         first_stage.emplace(ip, i);
145         if (first_stage.size() > Rerank)

```

```

146     {
147         first_stage.pop();
148     }
149 }
150
151 // stage2:精准排序 (借助float类型的内积来方便精准排序)
152 MinQueue<float, int> final_result;
153 while (!first_stage.empty())
154 {
155     // 从第一阶段的MinQueue取索引
156     int index = first_stage.top().second;
157     // 计算其对应的float点积
158     float score = exact_dot_product(base_orig + index * dim, query_orig, dim);
159     // 将float点积插入MinQueue final_result
160     final_result.emplace(score, index);
161     if (final_result.size() > k)
162     {
163         final_result.pop();
164     }
165     // 处理完过渡到下一个
166     first_stage.pop();
167 }
168 // 最后的final_result就是改良版本
169 return final_result;
170 }

```

7.3 main.cc 完整代码:

```

1 // Total Main.cc
2 #include <vector>
3 #include <cstring>
4 #include <string>
5 #include <iostream>
6 #include <fstream>
7 #include <set>
8 #include <chrono>
9 #include <iomanip>
10 #include <sstream>
11 #include <sys/time.h>
12 #include <omp.h>
13 #include "hnsplib/hnsplib/hnsplib.h"
14 #include "flat_scan.h"
15
16 #include <queue>
17 #include <stdint>
18
19 // 可以自行添加需要的头文件
20

```

```

21 // 优化1
22 #include "flat_search_sq.h"
23
24 // 优化2
25 #include "simd.h"
26 #include <arm_neon.h> // NEON SIMD 头文件, 用于 ARM 平台加速
27
28 using namespace hnswlib;
29
30 template<typename T>
31 T *LoadData(std::string data_path, size_t& n, size_t& d)
32 {
33     std::ifstream fin;
34     fin.open(data_path, std::ios::in | std::ios::binary);
35     fin.read((char*)&n, 4);
36     fin.read((char*)&d, 4);
37     T* data = new T[n*d];
38     int sz = sizeof(T);
39     for(int i = 0; i < n; ++i){
40         fin.read(((char*)data + i*d*sz), d*sz);
41     }
42     fin.close();
43
44     std::cerr<<"load data "<<data_path<<"\n";
45     std::cerr<<"dimension: "<<d<<"    number:"<<n<<"
46         size_per_element:"<<sizeof(T)<<"\n";
47
48     return data;
49 }
50 struct SearchResult
51 {
52     float recall;
53     int64_t latency; // 单位 us
54 };
55
56 void build_index(float* base, size_t base_number, size_t vecdim)
57 {
58     const int efConstruction = 150; // 为防止索引构建时间过长, efc 建议设置200以下
59     const int M = 16; // M建议设置为16以下
60
61     HierarchicalNSW<float> *appr_alg;
62     InnerProductSpace ipspace(vecdim);
63     appr_alg = new HierarchicalNSW<float>(&ipspace, base_number, M, efConstruction);
64
65     appr_alg->addPoint(base, 0);
66     #pragma omp parallel for
67     for(int i = 1; i < base_number; ++i) {
68         appr_alg->addPoint(base + 1ll*vecdim*i, i);

```

```

69     }
70
71     char path_index[1024] = "files/hnsw.index";
72     appr_alg->saveIndex(path_index);
73 }
74
75
76 int main(int argc, char *argv[])
77 {
78     size_t test_number = 0, base_number = 0;
79     size_t test_gt_d = 0, vecdim = 0;
80
81     std::string data_path = "/anndata/";
82     auto test_query = LoadData<float>(data_path + "DEEP100K.query.fbin", test_number,
83         vecdim);
84     auto test_gt = LoadData<int>(data_path + "DEEP100K.gt.query.100k.top100.bin",
85         test_number, test_gt_d);
86     auto base = LoadData<float>(data_path + "DEEP100K.base.100k.fbin", base_number,
87         vecdim);
88     // 只测试前2000条查询
89     test_number = 2000;
90
91     const size_t k = 10;
92
93     std::vector<SearchResult> results;
94     results.resize(test_number);
95
96     // -----
97     // 优化1的改动代码
98     /*
99     // 优化1 部分
100     // max_abs 获取改为
101     float max_abs_base = find_max_abs(base, base_number, vecdim);
102     float max_abs_query = find_max_abs(test_query, test_number, vecdim);
103     float max_abs = std::max(max_abs_base, max_abs_query);
104
105     std::cerr << "Max absolute value from base: " << max_abs << std::endl;
106     std::vector<uint8_t> base_q(base_number * vecdim);
107     std::vector<uint8_t> query_q(test_number * vecdim);
108     change_dataset(base, base_q.data(), base_number, vecdim, max_abs);
109     change_dataset(test_query, query_q.data(), test_number, vecdim, max_abs);
110     */
111
112     // -----
113     // 优化2的改动代码
114     // 因为知道了数据分布 就直接写上max_abs了
115     const float GLOBAL_MAX_ABS = 0.53f;
116     std::vector<uint8_t> base_q(base_number * vecdim);

```

```

115 std::vector<uint8_t> query_q(test_number * vecdim);
116 std::vector<float> mean_base, mean_query;
117
118 simd_quantize_global(base, base_q.data(), base_number, vecdim,
119                     GLOBAL_MAX_ABS, mean_base);
120 simd_quantize_global(test_query, query_q.data(), test_number, vecdim,
121                     GLOBAL_MAX_ABS, mean_query);
122
123
124 //
    如果你需要保存索引，可以在这里添加你需要的函数，你可以将下面的注释删除来查看pbs是否将build
125 // 要保存的目录必须是files/*
126 // 每个人的目录空间有限，不需要的索引请及时删除，避免占空间太大
127 // 不建议在正式测试查询时同时构建索引，否则性能波动会较大
128 // 下面是一个构建hnsw索引的示例
129 // build_index(base, base_number, vecdim);
130
131
132 // 查询测试代码
133 for(int i = 0; i < test_number; ++i) {
134     const unsigned long Converter = 1000 * 1000;
135     struct timeval val;
136     int ret = gettimeofday(&val, NULL);
137
138     // 该文件已有代码中你只能修改该函数的调用方式
139     // 可以任意修改函数名，函数参数或者改为调用成员函数，但是不能修改函数返回值。
140
141
142     // 默认的调用方式
143     // auto res = flat_search(base, test_query + i*vecdim, base_number, vecdim, k);
144
145     // 优化1的调用方式
146     // 优化1: sq: 函数改为flat_search_sq
147     // auto res = flat_search_sq(base_q.data(), query_q.data() + i *
148                             vecdim, base_number, vecdim, k, max_abs);
149
150     // 优化2的调用方式
151     auto res = simd_hybrid_search(base_q.data(), query_q.data() + i *
152                             vecdim, base, test_query + i * vecdim, base_number, vecdim, k);
153
154     struct timeval newVal;
155     ret = gettimeofday(&newVal, NULL);
156     int64_t diff = (newVal.tv_sec * Converter + newVal.tv_usec) - (val.tv_sec *
157                             Converter + val.tv_usec);
158
159     std::set<uint32_t> gtset;
160     for(int j = 0; j < k; ++j){
161         int t = test_gt[j + i*test_gt_d];

```



```
160         gtset.insert(t);
161     }
162
163     size_t acc = 0;
164
165     while (res.size()) {
166         int x = res.top().second;
167         if(gtset.find(x) != gtset.end()){
168             ++acc;
169         }
170         res.pop();
171     }
172     float recall = (float)acc/k;
173
174     results[i] = {recall, diff};
175 }
176
177 float avg_recall = 0, avg_latency = 0;
178 for(int i = 0; i < test_number; ++i) {
179     avg_recall += results[i].recall;
180     avg_latency += results[i].latency;
181 }
182
183 // 浮点误差可能导致一些精确算法平均recall不是1
184 std::cout << "average recall: "<<avg_recall / test_number<<"\n";
185 std::cout << "average latency (us): "<<avg_latency / test_number<<"\n";
186 return 0;
187 }
```