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 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

```
std::priority_queue<std::pair<float, uint32_t>> flat_search(float* base, float*
    query, size_t base_number, size_t vecdim, size_t k) {
    // 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)
    std::priority_queue<std::pair<float, uint32_t>> q;
    // 遍历每个base向量
    for(int i = 0; i < base_number; ++i) {
        float dis = 0;

        // DEEP100K数据集使用ip距离
        for(int d = 0; d < vecdim; ++d) {
            dis += base[d + i*vecdim]*query[d];
        }
        // 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
        dis = 1 - dis;
        // 若堆中还没有k个元素，那就直接加
        if(q.size() < k) {
            q.push({dis, i});
        }
        // 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
        else {
            if(dis < q.top().first) {
                q.push({dis, i});
                q.pop();
            }
```

```
        }
    }
    // 返回的堆就是(distance,index)的数据对
    return q;
}
```

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：

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

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：

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

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。

```
// 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。
// 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
inline float find_max_abs(const float* data, size_t num, size_t dim)
{
    // 存储最后找到的最大的MAX ABS
    float max_val = 0.0f;
    // 遍历所有float
    for (size_t i = 0; i < num * dim; i++)
    {
        // 取当前float的abs 与已有的最大abs进行比较
        max_val = std::max(max_val, std::abs(data[i]));
    }
    return max_val;
}
```

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。

```
// float → uint8 转换
inline uint8_t change_float_to_uint(float val, float max_abs)
{
    // 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
    // 若val > max_abs 那就直接转化为max_abs
    // 若val < -max_abs 那就直接转换为-max_abs
    float changed_val = std::max(-max_abs, std::min(val, max_abs));

    // 线性映射：实现转换
    return static_cast<uint8_t>((changed_val + max_abs) * 255.0f / (2 * max_abs));
}
```

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 $M = \texttt{max\_abs}$，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 $[-M, M]$。

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

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。

```cpp
// 将整个dataset都从float -> uint8
inline void change_dataset(const float* float_data, uint8_t* uint8_data, size_t num,
    size_t dim, float max_abs)
{
    // 二重循环遍历data，全部实现量化
    for (size_t i = 0; i < num; i++)
    {
        for (size_t j = 0; j < dim; j++)
        {
            uint8_data[i * dim + j] = change_float_to_uint(float_data[i * dim + j],
                max_abs);
        }
    }
}
```

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。

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

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。

```cpp
// 返回类型为一个min heap
inline std::priority_queue<std::pair<float, uint32_t>,
                           std::vector<std::pair<float, uint32_t>>,
                           std::greater<std::pair<float, uint32_t>>>
flat_search_sq(const uint8_t* base_q, const uint8_t* query_q,
               size_t base_number, size_t vecdim, size_t k, float max_abs)
{
    // uint8 ——> float 的因子为float
    const float scale = (2.0f * max_abs / 255.0f);
    // 对于内积的放大，应该采用因子的平方
    const float scale2 = scale * scale;
    // 计算 query 向量的平方模长
    float query_norm = std::sqrt(dot_product_u8(query_q, query_q, vecdim));

    std::priority_queue<std::pair<float, uint32_t>,
                        std::vector<std::pair<float, uint32_t>>,
                        std::greater<std::pair<float, uint32_t>>> pq;

    for (size_t i = 0; i < base_number; i++)
    {
        const uint8_t* vec = base_q + i * vecdim;

        uint32_t raw_ip = dot_product_u8(vec, query_q, vecdim);
        // 内积乘上平方的放大因子得到float类型
        float ip = raw_ip * scale2;

        // 计算当前vector的模长
        float base_norm = std::sqrt(dot_product_u8(vec, vec, vecdim)) * scale;
        // 计算Cosine相似度，分母加上1e-6防止出现除以0的情况
        float cosine_sim = ip / (query_norm * base_norm + 1e-6f);

        // vector插入最小堆的逻辑同前
        if (pq.size() < k)
        {
            pq.emplace(cosine_sim, i);
        }
        else if (cosine_sim > pq.top().first)
        {
            pq.pop();
            pq.emplace(cosine_sim, i);
        }
    }
    return pq;
}
```

为了方便阅读，最终完整的代码请见附件。

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。

$$\text{Final Score} = \underbrace{\text{SIMD\_IP}}_{\text{快速筛选}} + \underbrace{\text{Exact\_Dot\_Product}}_{\text{精确重排}} \tag{3}$$

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）

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

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 $scale = 127/\text{global\_max\_abs}$ 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。

```
// 定义两种优先队列类型
template<typename T1, typename T2>
using MinQueue = std::priority_queue<
    std::pair<T1, T2>,
    std::vector<std::pair<T1, T2>>,
    std::greater<std::pair<T1, T2>>
>;

template<typename T1, typename T2>
using MaxQueue = std::priority_queue<
    std::pair<T1, T2>,
    std::vector<std::pair<T1, T2>>,
    std::less<std::pair<T1, T2>>
>;
```

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。

```
// 量化function
inline void simd_quantize_global(
    const float* src, uint8_t* dst,
    size_t n, size_t dim,
    float global_max_abs,
    std::vector<float>& mean
)
{
    mean.resize(dim, 0.0f);

    // stage1: 计算每个维度上的average, 以方便进行中心化
    #pragma omp parallel for
    for (size_t j = 0; j < dim; ++j)
    {
        float sum = 0.0f;
        for (size_t i = 0; i < n; ++i)
        {
            sum += src[i * dim + j];
        }
        mean[j] = sum / n;
```

```
    }

    // stage2: 中心化 & 量化
    const float scale = 127.0f / global_max_abs;
    #pragma omp parallel for
    for (size_t i = 0; i < n * dim; ++i)
    {
        float val = src[i] - mean[i % dim];
        // 代替clamp, 处理数据越界
        size_t j = i % dim;
        if(val < -global_max_abs)
        {
            val = -global_max_abs;
        }
        else if(val > global_max_abs)
        {
            val = global_max_abs;
        }
        // 进行量化 float ---> uint8
        dst[i] = static_cast<uint8_t>(val * scale + 128.0f);
    }
}
```

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

```
// 计算float类型的内积 采用SIMD优化一次计算4个float相乘
inline float exact_dot_product(const float* a, const float* b, size_t dim)
{
    /*
    此处以dim=8为例子:
      设a=[a1~a8] b=[b1~b8]
      init: sum=[0,0,0,0]
      第一次循环:
          va=[a1~a4]
          vb=[b1~b4]
          sum=[a1*b1~a4*b4]
      第二次循环:
          va=[a5~a8]
          vb=[b5~b8]
          sum=[a1*b1+a5*b5~~~a4*b4+a8*b8]
      最后返回结果就是:
          res = (a1*b1+a5*b5)+.....+(a4*b4+a8*b8)

    这样就实现了求和
    */
    // 先将sum初始化为[0.0f,0.0f,0.0f,0.0f]
    float32x4_t sum = vdupq_n_f32(0.0f);
    // 每次处理4个float
    for (size_t i = 0; i < dim; i += 4)
```

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

simd_dot_product_u8_96: 包括两个步骤：一是为 *uint*8 类型数据转化为 *int*8 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 *int*8 类型元素，循环 6 次。

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

simd_hybird_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。

```cpp
template<size_t Rerank = 20>
MinQueue<float, int> simd_hybrid_search(
    const uint8_t* base_q, const uint8_t* query_q,
    const float* base_orig, const float* query_orig,
    size_t base_number, size_t dim, size_t k
)
{
    // stage1: 快速搜索（其实就是类似之前的最小堆保留topK结果）
    MinQueue<int32_t, int> first_stage;
    for (size_t i = 0; i < base_number; i++)
    {
        int32_t ip = simd_dot_product_u8_96(base_q + i * dim, query_q);

        first_stage.emplace(ip, i);
        if (first_stage.size() > Rerank)
        {
            first_stage.pop();
        }
    }

    // stage2:精准排序（借助float类型的内积来方便精准排序）
    MinQueue<float, int> final_result;
    while (!first_stage.empty())
    {
        // 从第一阶段的MinQueue取索引
        int index = first_stage.top().second;
        // 计算其对应的float点积
        float score = exact_dot_product(base_orig + index * dim,query_orig,dim);
        // 将float点积插入MinQueue final_result
        final_result.emplace(score, index);
        if (final_result.size() > k)
        {
            final_result.pop();
        }
        // 处理完过渡到下一个
        first_stage.pop();
    }
    // 最后的final_result就是改良版本
    return final_result;
}
```

为了方便阅读，最终完整的代码请见附件。

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.