
南開大學
Nankai University

计 算 机 学 院

**并行程序设计实验报告**

---

# SIMD 编程

---

张奥喆

年级：2023 级

专业：计算机科学与技术

指导教师：王刚

2025 年 4 月 27日

# 摘要

本实验围绕高维最近邻搜索（NNS）与近似最近邻搜索（ANNS）技术展开，通过 SIMD 指令集优化对应算法的性能。在 NNS 任务中，基于 NEON 指令的 SIMD 并行化策略实现了最高 4 倍加速比，并通过实验验证了并行度与内存带宽的权衡关系：当并行度超过一定限度时，内存带宽限制导致性能回退。在 ANNS 任务中，本文首先实现 PQ 算法，并利用跨向量 SIMD 的方法优化性能。此外，成功复现 PQ-Fastscan 技术进一步性能，实现 4 倍加速比，与原论文中的效果一致。实验表明，PQ-Fastscan 将内存读次数削减近 90%，突破传统 PQ 的访存瓶颈。围绕该算法还做了内存对齐与不对齐的性能分析、原生（AVX）/非原生（NEON）gather 指令的性能对比，以及 rerank 数量（latency）和 recall 的权衡，验证了算法参数的实际工程意义。



**关键字：ANN；SIMD 优化；NEON 指令集；PQ；PQ-Fastscan**


# 目录

# 一、 NNS

## (一) 问题描述

最近邻搜索（NNS）的核心任务是高效检索高维空间中与查询向量最邻近的数据点。得益于深度学习对非结构化数据的向量化表征能力——通过保持相似内容在嵌入空间中的邻近性，NNS已成为推荐系统、搜索引擎和大语言模型的核心技术。然而面对千万级向量规模与百维以上的高维特性，传统精确算法因计算复杂度呈指数增长难以实用，这使得近似最近邻搜索（ANNS）成为主流解决方案，其通过在可接受的精度损失范围内实现高效检索，有效平衡了准确率与计算效率。在接下来的实验中，首先对 NNS 算法进行 SIMD 优化。

## (二) 串行平凡实现

### 1. 实现思路

平凡实现遍历整个数据库中的每个向量，计算它们与查询向量之间的内积距离（先计算内积相似度，再用 1 减去结果转换为距离），然后使用最大堆（优先队列）维护当前找到的 k 个最近邻点，每当发现比堆顶更近的点时就替换掉堆顶元素，最终返回这个包含 k 个最近邻的优先队列。

### 2. 关键代码

```
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];
        }
        dis = 1 - dis;
        if(q.size() < k) {
            q.push({dis, i});
        } else {
            if(dis < q.top().first) {
                q.push({dis, i});
                q.pop();
            }
        }
    }
return q;
```

### 3. 算法理论分析

平凡实现的空间复杂度为 $O(k)$。对于时间复杂度来说，有两层循环，外层循环遍历所有基向量，执行 $base\_number$ 次，复杂度为 $O(n)$，内层循环执行 $vecdim$ 次，复杂度为 $O(d)$，其中这个循环对每个基向量都执行一次，因此这部分的时间复杂度为 $O(n \times d)$，堆的插入和删除操作复杂度为 $O(logk)$，最坏情况下，每个基向量都需要对队列进行操作，总复杂度为 $O(n \times logk)$。综合以上分析，总的时间复杂度为：$O(n \times (d + logk))$。

通常情况下，$n >> d$ 且 $n >> k$，所以这个算法的**瓶颈在于必须遍历所有 n 个基向量**，在真实的应用场景中，n 会非常大，这种方法简单直观但计算复杂度高，不适用于大规模数据集。

## (三) 使用 SIMD 加速内积计算（NEON）

### 1. 实现思路

平凡算法中，由于 n 是一个很大的数字，逐个计算非常耗时。因此在一部分引入 128 位的 NEON 寄存器，每个寄存器可以容纳 4 个 32 位浮点数，程序可以使用两个 NEON 寄存器实现每次处理 8 个浮点数 [4]。

### 2. 关键代码

```
float32x4_t sum1 = vdupq_n_f32(0); // 第一个累加寄存器
float32x4_t sum2 = vdupq_n_f32(0); // 第二个累加寄存器
// 每次处理8个浮点数(使用2个NEON寄存器)
for (size_t i = 0; i + 7 < vecdim; i += 8)
{
    // 加载第一个向量的4个float
    float32x4_t v1_1 = vld1q_f32(b1 + i);
    float32x4_t v1_2 = vld1q_f32(b1 + i + 4);
    // 加载第二个向量的4个float
    float32x4_t v2_1 = vld1q_f32(b2 + i);
    float32x4_t v2_2 = vld1q_f32(b2 + i + 4);
    // 执行乘法并累加到sum寄存器
    sum1 = vmlaq_f32(sum1, v1_1, v2_1);
    sum2 = vmlaq_f32(sum2, v1_2, v2_2);
}
```

### 3. 算法理论分析

设输入向量长度为 $n$，以 NEON 每次迭代并行处理 8 个 `float`为例。

- **循环次数** $n/8$
- **计算量** 每元素执行一次乘加，故总 FLOPs 为 $2n$ ($\mathcal{O}(n)$)
- **访存量** 两个向量各读 $n$ 个 `float`，共 $8n$ B，故空间复杂度 $\mathcal{O}(n)$
- **指令分解（每 8 元素）**

$$4\,\texttt{vld1q} + 2\,\texttt{vfmaq} + 1\,\text{loop ctl} \approx 7 \text{ instructions}$$

- **算术强度** $\dfrac{2 \text{ FLOPs}}{8 \text{ B}} = 0.25$ FLOP/Byte，远低于缓存屋顶线，故**性能主要受内存带宽限制**

这个 SIMD 优化的版本中，每次处理 8 个浮点数（可以调整为更高的并行度），理论上内积计算速度可以提升 8 倍。但是并行部分可能存在额外的开销，比如当维度无法被 8 整除时，尾部元素就需要使用标量计算处理剩余元素（不超过 7 个），还有实际内存带宽的限制、服务器压力等，实测加速比达不到 8 倍。

## (四) 实验

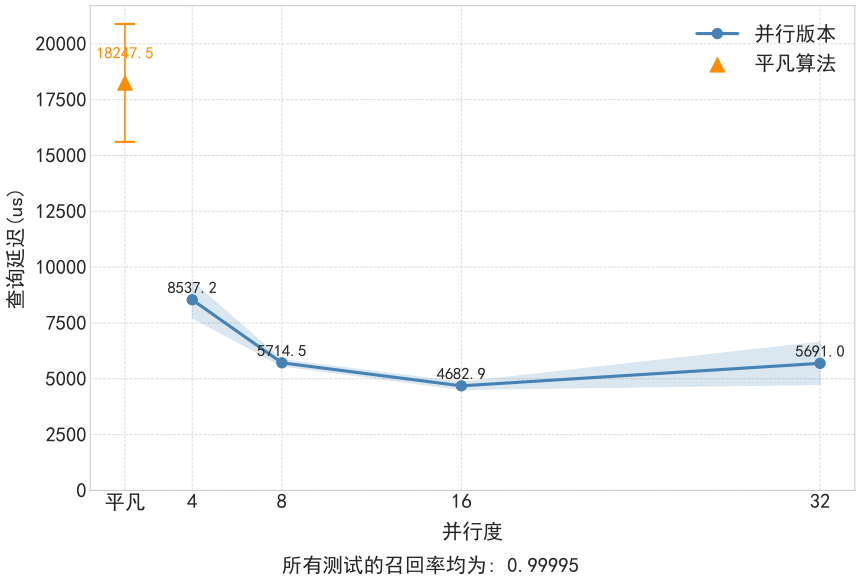
### 1. 查询延时与加速比

在 SIMD 的编写过程中，我意识到并行度是可以自己调整的，因此可以探究不同并行度下的性能，设置了 4 组，分别是每次处理 4、8、16 和 32 个浮点数。测试规模固定在前 2000 条

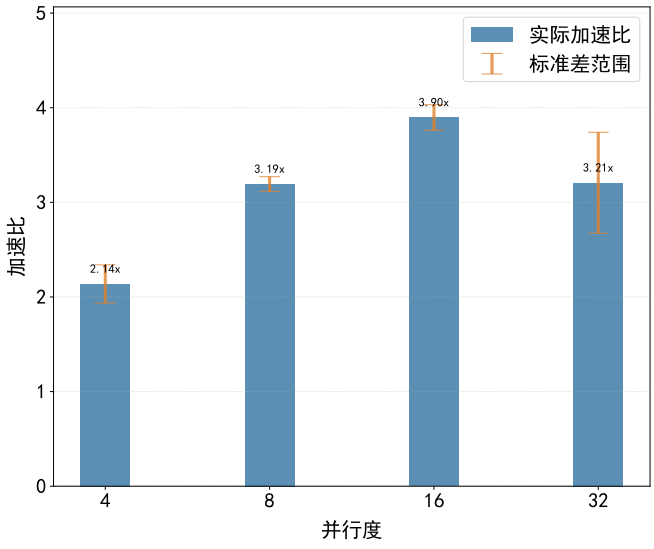
(经过我的验证，测试数量对于 average latency 几乎没有影响，服务器波动影响更大一点)。由于服务器的波动较大，每组测试了 5 次，结果如图1所示。

首先，由于优化的代码只是用了 SIMD 来加速平凡实现，所以 recall 不会下降。平凡算法和 SIMD 优化后的 recall 都是 1（0.99995 是由于计算机浮点计算误差导致的）。相比于平凡算法，使用了 Neon 寄存器进行 SIMD 进行 SIMD 后，都实现了**两倍**以上的加速比。此外更重要的是：

- 从一次算 4 个浮点数到一次算 16 个浮点数，尽管查询延迟在降低，加速比在上升，但是实测加速比远达不到理论的加速比，使用更多的寄存器并不会带来线性的效率提升。
- **性能拐点出现在并行度 16，可以得到接近 4 倍的加速比**：并行度 8 和 16 的标准差较小，表明这些设置下系统性能更稳定；但是此时已经出现收益递减了，查询效率的提升不如之前并行度 4 至 8 的提升；当并行度到 32 时，发现性能竟然下降了，并且性能波动较大，这反映了**一味地提高寄存器的数量并不会一直带来性能的提升**，甚至适得其反，在之后的实验中也会探讨出现这种情况的原因。




(a) SIMD 优化下不同并行度的查询延时




(b) 不同并行度下的加速比

图 1: SIMD 优化平凡算法性能分析

## 2. Perf 剖析——SIMD 并行化的优势 & 为何一味增加并行度会适得其反?

为了分析 SIMD 优化相比于平凡算法的优势，同时为了找到并行度增长的瓶颈，在服务器上使用 Perf 工具进行剖析，重点关注 Instructions Retired、CPI 和 Memory Bound。得到的性能数据如图2所示。

- **SIMD 大幅减少了 Instructions Retired，这是并行加速的主要来源**。从平凡算法到一次算 4 个浮点数，指令完成数的确减少了 $\frac{3}{4}$，相比于并行度 4，并行度 8 在此基础上减少 50%，目前的收益的确是线性的，但是从并行度 8 到 16 再到 32，这样的收益就边际递减了，这一部分指令数减少的比例并不与线程数成正比，表明存在并行开销（比如高并发下的额外管理）。

- CPI 随着并行度的上升而增高。CPI 值越低表示处理器效率越高，当 CPI 超过 1 时，意味着平均每条指令需要多于一个时钟周期执行，处理器效率显著下降。注意到，从 16 线程到 32 线程，CPI 陡然增加了 87.5%，也超过了 1（1.35），这是并行度 32 状态下，性能下降的直接证明之一。
- **Memory-bound 逐渐升高**。从平凡算法的 17% 一直逐步上升（基本是线性的）到 34 %、45 %、64 %、87 % , 瓶颈逐渐转变为内存带宽的限制。在并行度 32 时, 87 % 的 Memory-bound 基本宣告带宽被榨干，这意味着处理器大部分时间都在等待内存操作完成。

综合以上的分析，可以得出结论，总体耗时随并行度呈“先降后升”的倒 U 形：SIMD 在 4 至 8 并行度阶段带来近乎线性的收益——指令完成数锐减，而 CPI 和 Memory-bound 仍处于低位，因此总时间快速下降；当并行度扩展到 16 线程，**指令数的递减进入边际报酬递减区，而同步、调度等并行开销开始显现**，CPI 抬升到 0.72、Memory-bound 跃至 64 %，耗时降幅明显变缓；继续扩展到 32 线程时，虽然指令数再降 37 %，但 **87 % 的 Memory-bound 使内存带宽基本饱和**，CPI 飙升到 1.35——CPU 大部分周期都在等待数据——并行开销反客为主，致使总耗时反而回弹。也就是说，性能拐点出现在 16 线程：此前耗时主要受计算量驱动，此后则主要受内存吞吐与线程管理开销主导。

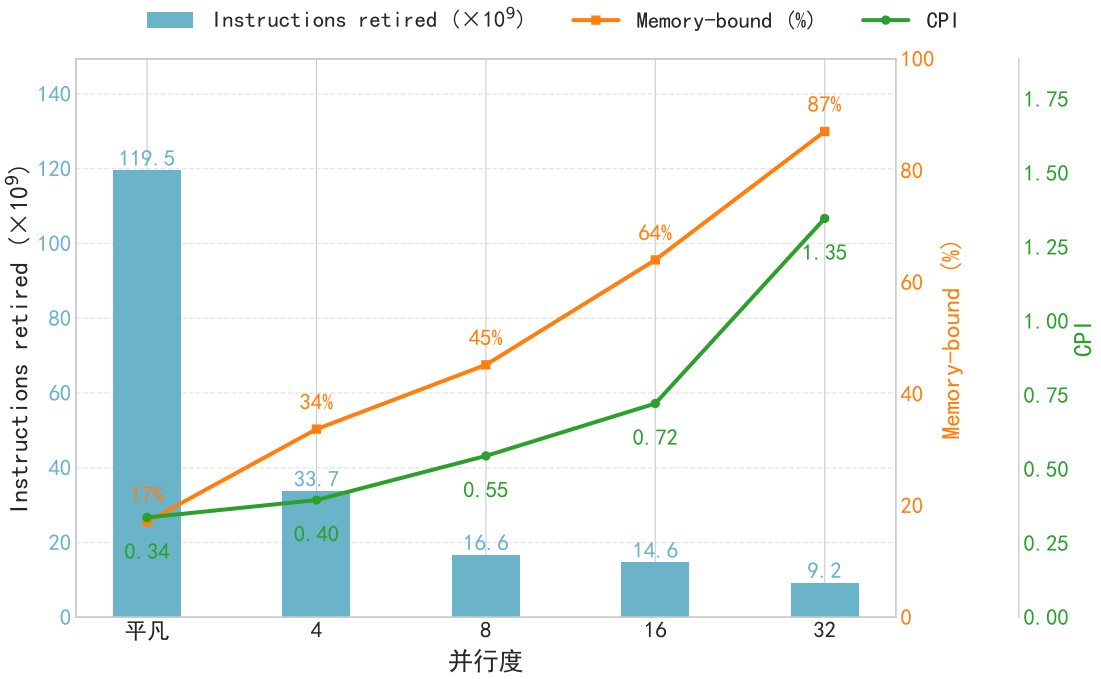



图 2: Instructions Retired、CPI 和 Memory Bound

# 二、 ANNS

## (一) 问题描述

ANNS 使用“可以接受”的精度损失换取查询效率的进一步提升。在这一部分我重点关注了 Product Quantization（PQ）加速方法。

## (二) 平凡 PQ

### 1. PQ 索引构建

1. 原理：PQ 的核心思想是把高维向量 $\mathbf{x} \in \mathbb{R}^D$ 拆分为 $M$ 个低维子向量

$$\mathbf{x} = \left[\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(M)}\right], \quad \mathbf{x}^{(m)} \in \mathbb{R}^{D/M},$$

并在每个子空间独立聚类训练一个 $K$ 质心（codewords）的子代码簿：

$$\mathcal{C}^{(m)} = \left\{\mathbf{c}_1^{(m)}, \mathbf{c}_2^{(m)}, \ldots, \mathbf{c}_K^{(m)}\right\}.$$

对整条向量来说，最终编码是 $M$ 个子代码索引的串联 $\mathbf{q}(\mathbf{x}) = \left[i^{(1)}, i^{(2)}, \ldots, i^{(M)}\right]$，累计码长为 $M \log_2 K$ bits，压缩率可达数十倍。

2. 实现：

(a) **向量切分**：按维度均匀划分成 $M$ 份。

(b) **子空间 K-means**：对第 $m$ 个子空间全部样本 $\{\mathbf{x}^{(m)}\}$ 运行 K-means，得到 $\mathcal{C}^{(m)}$。

训练成本：$O(NDM/M)$（近似 $O(ND)$），一次性；离线编码成本：$O(ND)$。

### 2. 平凡 PQ 查询流程

1. 实现思路:

(a) **子向量切分：** 将 $\mathbf{q}$ 与训练阶段一致地切成 $\left\{\mathbf{q}^{(m)}\right\}$。

(b) **构建查表（LUT）：** 先把查询向量在每个子空间里到所有聚类质心的距离都算好，放进一张小表，后面扫库时直接查这张表，而不再做乘减。对每个子空间：

$$d_k^{(m)} = \left\|\mathbf{q}^{(m)} - \mathbf{c}_k^{(m)}\right\|_2^2, \qquad k = 1, \ldots, K.$$

(c) **线性扫描并累加距离：** 对库中每条压缩码 $\mathbf{q}(\mathbf{x}) = \left[i^{(1)}, \ldots, i^{(M)}\right]$：

$$\widetilde{d}(\mathbf{q}, \mathbf{x}) = \sum_{m=1}^{M} d_{i^{(m)}}^{(m)},$$

(d) **维护 Top-$L$ 堆并精确重排：** 使用大小为 $L$ 的小顶堆实时保留当前最优 $L$（$L > k$）个候选。得到候选后对其原始向量做精确计算，重新排序以提高 Recall。

2. 关键代码

**Algorithm 1** 预计算查询–质心查找表（LUT）——对应 `build_lut` 函数

**Input:** 查询向量 $q$；子空间数 $M$；每个子空间质心数 $K$；子向量维度 $d$；质心集合 $C$
**Output:** 填充完毕的查找表 $LUT$ $(M \times K)$

```
if LUT 为空 then
    申请长度 M × K 的数组 LUT
end if
for m ← 0 to M − 1 do
    q_s ← q[m · d . . . (m + 1) · d − 1]                ▷ 取出第 m 段查询子向量
    for k ← 0 to K − 1 do
        c ← C[m][k]                                      ▷ 第 m 子空间第 k 个质心
```

$s \leftarrow 0$ ▷ 距离累加器
**for** $t \leftarrow 0$ **to** $d-1$ **do**
$diff \leftarrow \mathbf{q_s}[t] - \mathbf{c}[t]$
$s \leftarrow s + diff^2$
**end for**
$LUT[m \times K + k] \leftarrow s$ ▷ 记录距离
**end for**
**end for**
**return** $LUT$

构建距离查表 (LUT) 有三层循环,总共有 $M \times K \times d$ 次浮点运算,时间复杂度为 $O(M \times K \times d) = O(KD)$，实际上这个开销是常数级别的，和本问题的规模 N 关系不大，当 K 和 D 都确定了，那么构建 LUT 的开销也就确定了。

```
for (uint32_t id = 0; id < code.size(); ++id){
            float d2 = 0.f;
            for (int m = 0; m < M; ++ m)                    // O(M) 查表
                d2 += LUT[m * K + code[id][m]];
            if ((int)heap.size() < k) heap.emplace(d2,id);
            else if (d2 < heap.top().first){ heap.pop(); heap.emplace(d2,id);
                 }
        }
        std::vector<Pair> res;
        while (!heap.empty()){ res.push_back(heap.top()); heap.pop(); }
        std::reverse(res.begin(), res.end());           // 升序
        return res;
```

线性扫描库中所有压缩向量需 $M$ 次查表和 $(M-1)$ 次加法。复杂度 $O(NM)$，是查询主耗时。

### 3. 算法复杂度分析

平凡 PQ 查询流程把「高维乘加」变为「低维查表」，用 $O(M)$ 操作近似代替 $O(D)$ 距离计算，相比于 NNS 的算法，有效降低查询延迟，但是由于不够精确，recall 也会降低，为了提高 recall 采用对 L 个候选向量进行精确的距离计算，得到 Top-K 结果。由于 $L \ll N$，在计算复杂度的时候可以不用重点考虑，但是 L 过大也会影响查询延时，这里存在权衡，接下来的实验也会讨论这个权衡。

$$\underbrace{KD}_{\text{LUT}} + \underbrace{NM}_{\text{查表累加}} + \underbrace{N \log L}_{\text{堆}} = O\big(N(M + \log L)\big) \approx O(NM), \text{其中 } K, M \text{ 为常数且 } L \ll N$$

## (三) SIMD 加速 PQ 距离计算

### 1. 实现思路与核心难点

乘积量化使用预计算的查找表 (LUT) 来加速距离估算。标量计算方式为 $\delta(x,q) \approx \sum_{k=1}^{m} LUT_{k,o_k}$，其中 $o_k$ 是数据库向量 $x$ 的第 $k$ 个子空间的码字索引，$LUT_{k,i}$ 是查询向量 $q_k$ 与第 $k$ 子空间第 $i$ 个聚类中心的预计算距离。

直接向量化对单个向量 $x$ 的 $m$ 次 LUT 查找与累加) 效率低下。主要原因是码字 $o_k$ 的值通常是随机的，导致访问 $LUT_{k,o_k}$ 时内存访问模式是**分散**的，缺乏数据局部性。

因此，这里我采用**跨向量向量化**策略：一次性计算 $W$ 个向量与查询向量 $q$ 的距离 [4]。该策略的核心步骤是在处理第 $k$ 个子空间时：

1. **加载 $W$ 个码字:** 高效加载 $W$ 个向量的第 $k$ 个码字 $o_k^1, \ldots, o_k^W$ 。
2. **SIMD 聚集:** 使用一条 SIMD gather 指令，根据这 $W$ 个码字索引，并行地从 $LUT_k$ 中查找并加载 $W$ 个对应的预计算距离值 $[LUT_{k,o_k^1}, \ldots, LUT_{k,o_k^W}]$。
3. **SIMD 加法:** 使用一条 SIMD 加法指令，将加载的 $W$ 个距离值分别累加到 $W$ 个向量对应的累加器中。

通过迭代所有子空间 $k = 1 \ldots m$，最终得到 $W$ 个向量的 PQ 距离。这种方法将 gather 指令的开销**摊销**到 $W$ 个向量上，并充分利用了 SIMD 的并行算术运算能力，从而实现加速。

### 2. 关键代码

ARM 平台上的 NEON 使用 128 位寄存器，以下代码片段展示了处理 $W = 4$ 个向量时，针对单个子空间 $k$ 的核心 SIMD 计算:

```
// 1. Gather: 从 LUT_k 加载 4 个距离值
lut_vals = vsetq_lane_f32(val0, lut_vals, 0);
lut_vals = vsetq_lane_f32(val1, lut_vals, 1);
lut_vals = vsetq_lane_f32(val2, lut_vals, 2);
lut_vals = vsetq_lane_f32(val3, lut_vals, 3);
float32x4_t lut_vals = {d0, d1, d2, d3}; // 构建 SIMD 向量
// 2. Accumulate: 并行累加 4 个距离值
acc = vaddq_f32(acc, lut_vals);
```

### 3. 复杂度与加速分析

采用跨向量 SIMD 策略后，每次处理 $W$ 个向量。处理 $N$ 个向量需要 $N/W$ 轮。这只是相当理想的情况，实际上由于并行化额外开销，**朴素的 SIMD 优化效果并不好**。并且通过查阅资料发现，**ARM 的 NEON 并不支持原生的 gather 指令** [2]，我的关键代码中的 gather 实现实际上是手动执行“查找 + 构建”这个过程，这大大降低了 SIMD 的优化效果。此外，我在实验中也编写了 AVX 的跨向量 SIMD 优化版本（X86 平台），使用了原生的 AVX2 的 gather 指令，探索非原生和原生 gather 指令之间的性能差距。为了进一步探索优化的空间，我将目光转向 PQ-Fastscan。

## (四) PQ-Fastscan

PQ FastScan 是在 4-bit Product-Quantization 基础上做的一种“寄存器内查表”扫描技术：先把每个查询向量对所有子量化器构造出的 16×16 距离表量化成 8-bit 后整块放入 SIMD 寄存器；再把数据库编码重排成“同一子量化器、连续 16 条向量的码字”连续布局，这样一次 128bit（NEON 的寄存器是 128bit）加载即可获得 16 个（省略了 unpack 的步骤）4-bit 索引。核心用 ARM NEON 的 `vqtbl1q_u8` 指令在寄存器里直接完成「索引 → 距离」映射，随后用 `vmovl_u8` 扩展到 16-bit 并在寄存器中累加，多子量化器循环结束即可得到 16 条向量的近似距离。全过程只顺序读取 code 字节、顺序写回累加结果，彻底消除了传统 PQ 扫描中最耗时的随机 LUT 访存和分支 [1]。

## 1. 实现思路

1. **4-bit 子码（$K$=16）+ 8-bit LUT：**每个子向量只需 4 bit 保存编号；查表项量化到 $u8$ [3]，16 个距离恰好放进 128 bit NEON 寄存器。距离积累使用 16-bit 无符号整数，不会溢出。注意在我的平凡 PQ 算法中，$K$=256，也就是聚了 256 个类，这样精度会比 $K$=16 更高。由于寄存器的限制导致聚类数量减少，为了保证 recall，需要适当提高精确重排的个数 $L$。

2. **Block-of-16 转置布局：**以 16 条向量为一个块，将同一子量化器的码字收拢：

$$\text{codes\_fs}[\,blk\cdot(M\times 16)+m\times 16+pos\,]\leftarrow \text{code}_{id}^{(m)}$$

   这样 `vld1q_u8` 一次即可载入 16 个码字，避免散读。

3. **整数化 LUT：**对查询向量构建 $M\times K$ 距离后线性量化到 $u8$：$d\_mk^{\mathrm{q8}}=\mathrm{round}(d\_mk/\alpha)$，$\alpha=\max d\_mk/255$，量化误差控制在 $<0.4\%$ [3]。

4. **SIMD 扫描核：**每处理 16 向量仅用 4 条指令：`vld1q_u8` (codes) → `vqtbl1q_u8` (查表) → `vmovl_u8` (扩位) → `vaddq_u16` (累加)；累加 $M$ 轮后得到 16 个 16-bit 近似距离。

5. **Re-ranking：**在扫描过程中维护大小为 `rerank_L` 的最大堆；再对堆中向量做一次精确内积重排，提高 recall。

## 2. 关键代码

```
/* ---- 布局转换：把 codes[i][m] 转成 block-of-16 ---- */
for(size_t i=0;i<N;++i){
    size_t blk=i/16,pos=i%16;
    for(int m=0;m<M;++m)
        codes_fs[ blk*M*16 + m*16 + pos ] = codes[i][m];
}
/* ---- LUT 量化 ---- */
float maxv = *std::max_element(dist.begin(), dist.end());
float alpha = maxv/255.f;
for(int t=0;t<M*K;++t) LUT_q8[t] = uint8_t(dist[t]/alpha + 0.5f);
/* ---- FastScan 核心：一次处理 16 向量 ---- */
uint16x8_t acc0=vdupq_n_u16(0), acc1=acc0;
for(int m=0;m<M;++m){
    uint8x16_t lut  = vld1q_u8(LUT_q8 + m*16);          // 16×u8
    uint8x16_t code = vld1q_u8(blk_ptr + m*16);         // 16×u8
    uint8x16_t d    = vqtbl1q_u8(lut, code);            // 查表
    acc0 = vaddq_u16(acc0, vmovl_u8(vget_low_u8 (d)));  // 0..7
    acc1 = vaddq_u16(acc1, vmovl_u8(vget_high_u8(d)));  // 8..15
}
vst1q_u16(acc, acc0);   vst1q_u16(acc+8, acc1);     // 写回 16 距离
```

## 3. 算法复杂度

设数据规模 $N$、子空间数 $M=16$：LUT 构建 $KD$（常数级）；扫描期每 16 条向量只累加 $M$ 次，等价于把平凡 PQ 的常数项再除以 16；堆维护 $N\log k$ 对 Top-$k$ ($k\le 100$) 几乎无感；但是最后精确重排 rerank 的数量会影响最终的查询延迟，rerank 部分也可以使用 SIMD 进行优化（和朴素 SIMD 优化类似）。

$$\underbrace{KD}_{\text{LUT}}+\underbrace{\frac{NM}{16}}_{\substack{\text{Fast-Scan}\\ \text{block=16}}}+\underbrace{N\log k}_{\text{堆}}\approx O\Big(N\frac{M}{16}\Big)$$

事实上，PQ-Fastscan 最关键的改进并不能直接展示在复杂度的公式当中——访存优化，接下来的实验会重点探讨它的优势。

## （五） 实验

### 1. 查询延时与加速比

PQ 会损失精度，我调整 rerank 参数，使 recall=90.2%——一般来说这是一个可以接受的精度损失。由于服务器波动，每种策略运行 5 次以避免偶然性。得到的结果如图3所示。

- **引入 PQ 算法之后立即获得显著的查询加速**，即使是对比用 SIMD 优化得比较好的 NNS 算法，ANNS 使用串行的策略就可以超越它（40% 的提升），体现了 PQ 的有效性与高效性。
- 跨向量 SIMD 并行优化后的 PQ，**加速比为 1.18**，效果没有想象中的好，因为它仍然执行了大量的表查找，无法充分利用现代 CPU 的能力，这在之后的实验中会进一步研究。我认为直接使用 SIMD 应该还有一定的提升空间，但是为了得到更好的效果，我直接把重点放在了 Fastscan 的实现上。
- **PQ-Fastscan 实现了 4 倍的加速比**，查询时间缩短到了百微秒级别，并且很稳定，说明我的 fastscan 算法实现良好，基本复现了论文的效果（论文中 PQ-Fastscan 的加速比在 3~6 之间）。

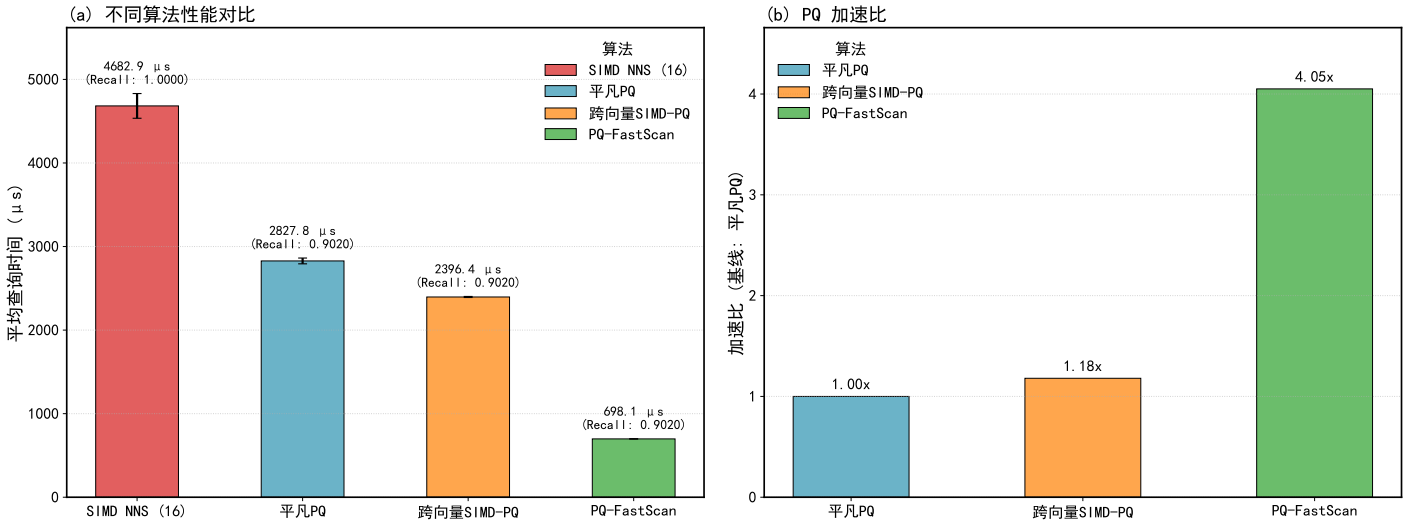



图 3: SIMD 优化 PQ 算法性能分析

### 2. Perf 剖析——Fastscan 的优势

- Memory Load：朴素 PQ 与跨向量 SIMD-PQ 都在 $6\times10^9$ 次左右，但跨向量 SIMD 的内存读入减少约 8%；PQ-FastScan 仅 $0.7\times10^9$ 次，**内存读量削减 90% 左右**，这就是 Fastscan 的主要优化来源，**将 LUT 直接存储到 SIMD 寄存器中，在计算时便不再需要访问内存**。访存不再成为瓶颈，大大加速了查询效率。
- Instructions Retired：平凡 PQ 的指令执行数最多，跨向量 SIMD-PQ 减少了 **46%** 的指令执行数，体现了跨向量 SIMD 的优化效果；PQ-Fastscan 相比平凡 PQ 减少了约 **78%** 的指令执行量，效果显著。

- CPI：PQ-Fastscan 和跨向量 SIMD 的 CPI 值略微升高，表明每条指令平均需要更多的时钟周期，这符合 SIMD 的优化规律，但考虑到其指令数大幅减少，总体性能仍有显著提升。

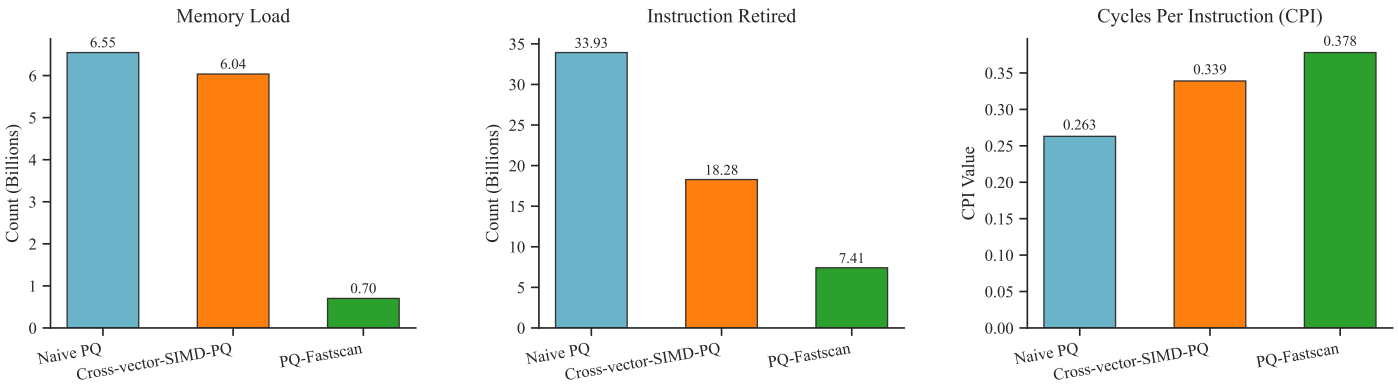



图 4: PQ 优化性能分析

- 缓存命中：FastScan 的 L1_HIT 数量由原来的 $8.0\times10^9$ 降至 $1.9\times10^9$，下降 76%；L1_MISS 也下降 54%。可见，**将 LUT 直接存储到 SIMD 寄存器中，这样连 cache 都不需要访问了**。

- Fastscan 的各级缓存命中率都有显著提升，跨向量 SIMD-PQ 相较于平凡算法有轻微的提升，但是并不明显，这说明**简单向量化并不能缓解大量查表带来的访存压力**，这也是跨向量 SIMD 效果一般的症结所在。

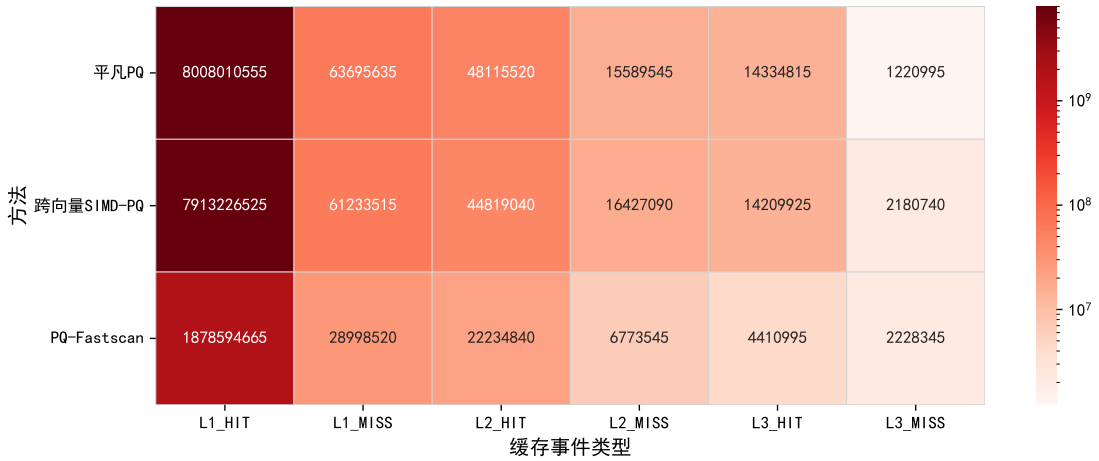



图 5: 缓存命中次数

跨向量化 PQ 通过引入 SIMD 并行执行，减少了约 46% 的指令数和部分访存次数，提升了一定的计算效率，但由于未能根本缓解访存压力，整体加速效果有限。相比之下，PQ-FastScan 通过将查询用 LUT 直接存入 SIMD 寄存器，显著减少了内存和缓存访问，有效突破了内存瓶颈，指令数也下降约 78%，带来了 4 倍性能提升。

### 3. rerank：recall-latency tradeoff

由于使用聚类中心来表示附近的向量存在误差，PQ 这个方法依赖 rerank 来提高 recall。rerank 使用的是原始向量计算距离，rerank 数量越多开销越大，耗时也越长（尽管可以使用 SIMD 来优化这一部分），因此需要权衡 rerank 的数量和 recall 的精度。实验结果如图6所示。

- latency 和 rerank 的数量基本成**线性关系**，rerank 越多，查询延时越长。
- recall 曲线随 rerank 数量的增加而单调上升，但是**收益逐渐递减**：当 rerank 数量（n）小于 200 时，recall 上升最陡，单位候选能换来 $> 0.003$ 的 recall 提升；大约在 $(n \approx 290)$ 处出现“膝点”，此时 recall 0.906，延迟 696 µs；继续把 rerank 数量从 300 提到 1000，只额外提升 6 个百分点的 recall（此时 recall=0.985），但时延却翻了近一倍（700 µs → 1300 µs），典型的“边际收益快速递减”。
- 直观从图中看出，我实现的 fastscan 算法的 rerank“甜点”大约就在 n=290 的样子，此时 recall=0.906。

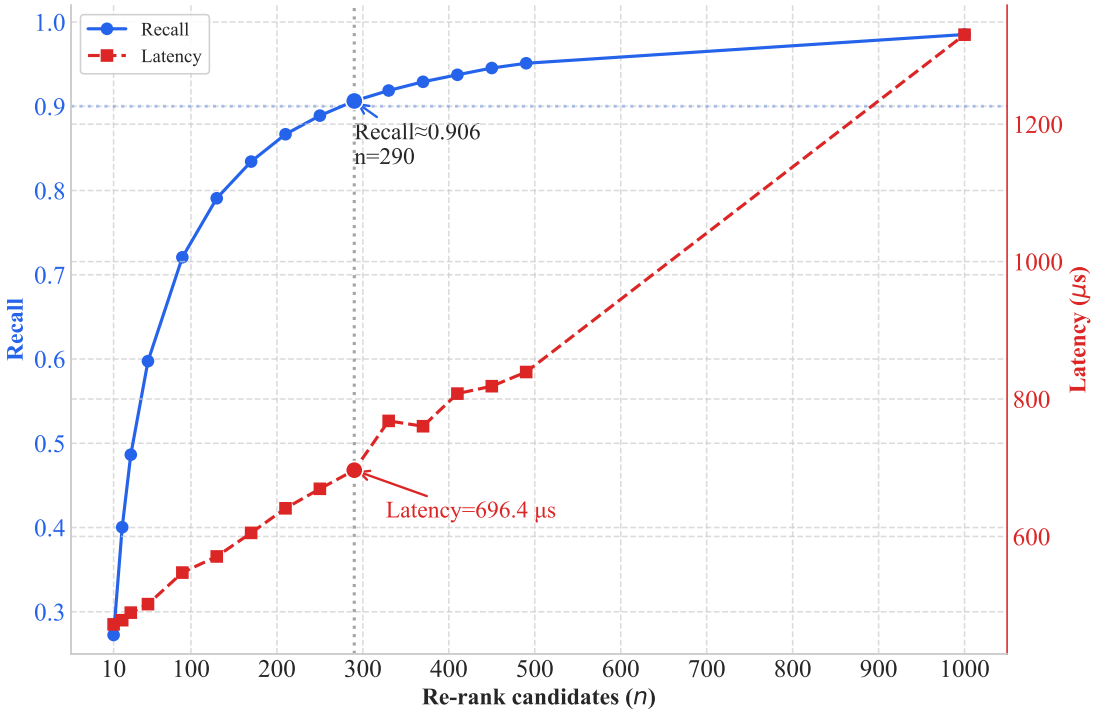



图 6: recall-latency tradeoff

### 4. 指令集差距——原生/非原生 gather 指令影响

在之前的实验中已经看到，由于 ARM 的 NEON 并不支持原生的 gather 指令，实测加速的效果一般。而 x86 平台上的 AVX 指令集**有原生 gather 指令**，因此将相关代码移植到 x86 上，并使用 AVX2 的原生 gather 指令从距离查找表（LUT）中获取距离。实验的其他条件与前文保持一致，recall=0.902，每个代码运行 5 次以避免偶然性，实验结果如图7。

从图可以直观地看出，~~我的轻薄本 CPU 比服务器 CPU 的单核性能更强，~~由于 AVX2 提供了原生的 `gather` 指令，**加速比达到了 1.48**，这一结果明显优于 NEON 的 1.18：

1. 跨向量 SIMD 加速 PQ 查询也可以得到较好效果，但是这依赖于指令集的支持程度。

2. NEON 需要手动模拟 gather 操作，其加速效果相对有限。

3. 资料显示，原生 gather 指令的核心优势在于**利用硬件**并行性来高效处理分散内存读取，减少了指令数量和开销。**在进行优化方案选择时，也要考虑到实际硬件对指令的支持度**。

实验数据详情

| | 1 | 2 | 3 | 4 | 平均值 ± 标准差 | 加速比 |
|---|---|---|---|---|---|---|
| AVX Flat | 1129.79 | 1138.28 | 1143.90 | 1097.76 | 1127.43 ± 17.85 | - |
| AVX SIMD | 802.56 | 752.86 | 751.56 | 750.97 | 764.49 ± 21.99 | 1.48x |
| NEON Flat | 2824.07 | 2783.20 | 2889.45 | 2814.32 | 2827.76 ± 38.68 | - |
| NEON SIMD | 2391.53 | 2408.27 | 2392.97 | 2392.64 | 2396.35 ± 6.90 | 1.18x |

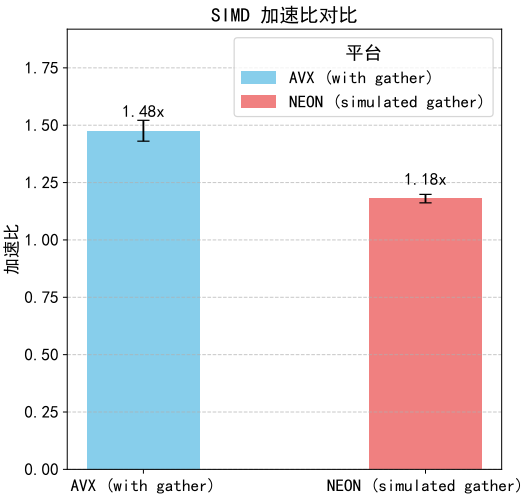



图 7: AVX 与 NEON SIMD 性能 (gather) 对比分析

### 5. Perf 剖析——内存对齐与不对齐的影响

接下来探究并行化时对齐访问和非对齐访问在性能上的差异，以 PQ-Fastscan 为例，我分别编写了对齐并行程序和不对齐并行程序，并利用 Perf 进行性能分析，实验结果如图8所示。

- 查询延迟：**对齐访问的查询延迟快 10%**。非对齐访问不仅平均延迟明显更高，波动也更大（标准差更大），说明其执行更慢且可预测性更差。
- Cache 命中率：在 L1、L2、L3 各级缓存中，**对齐访问的命中率始终更高**。
- Instructions Retired：完成同样任务，**非对齐访问需要执行更多指令**。

综合来看，结果强烈表明：内存对齐访问具有优势。

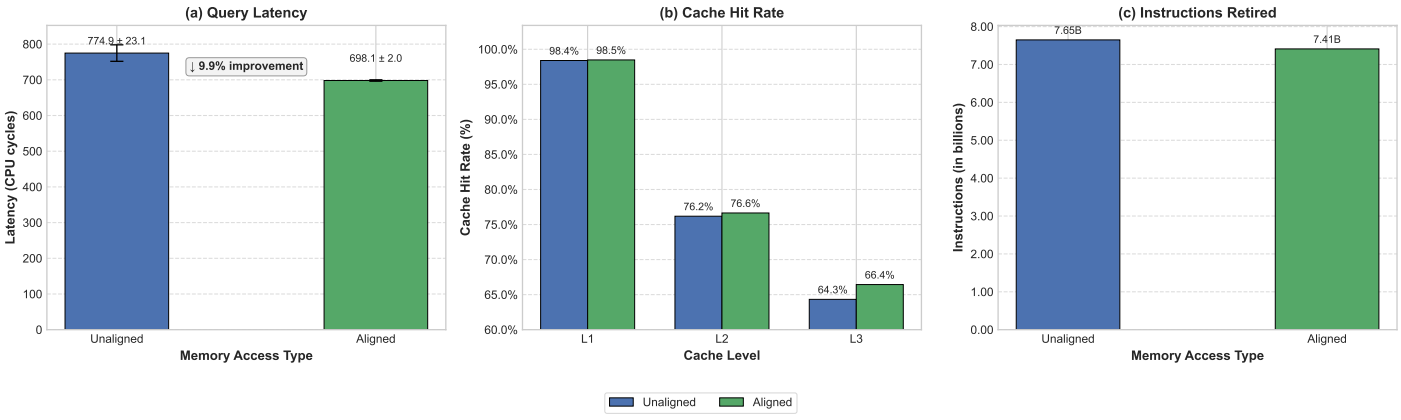



图 8: 内存对齐性能分析

# 三、 总结

本次实验探索了 SIMD 在高维向量检索中的优化路径。在 NNS 任务中，NEON 指令通过单周期 16 浮点并行计算实现最高 4 倍加速，但并行度超过 16 后因内存带宽饱和导致性能下降与不稳定，这揭示了硬件资源约束下的并行设计原则。在 ANNS 任务中，PQ 算法通过聚类和 LUT 的思想显著降低了查询时的计算量，却也因此带来了问题——直接的 SIMD 向量化非常困难。好在有 Fastscan 技术利用寄存器内查表，大幅降低访存开销，在这次实验任务中实现**百微秒级查询效率**，实测加速比达到 4。此外，本实验对我还有如下的启发：

1. **算法-硬件协同优化**：SIMD 加速需匹配数据局部性（如 Fastscan 的 Block-of-16 布局），并且应该根据实际的 CPU 情况（如内存、cache 大小等）优化并行化的相关参数，尽量挖掘 CPU 的潜能。注意 CPU 对指令集的支持度也是重要的指标。

2. **精度-效率权衡**：ANNS 难以达到 NNS 的精度，毕竟 ANNS 的效率就是用精度的损失换来的，当精度达到一定水平时，边际效应显现，rerank 候选数从 200 增至 1000 时 rerank 提升 6%，但时延翻倍。因此在实际应用中应该根据业务需求选择” 膝点” 参数，实现权衡；

3. **并行化边际效应**：SIMD 并行度超过阈值后，指令数减少的收益被内存带宽限制、CPI 等抵消，此时需要通过 Perf 等工具定量分析相关指标，找到平衡点，实现最大的加速比。

跳转至：源代码地址，主要结构如下：

| 类别 | 文件 |
|---|---|
| 精确 NNS 相关代码 | `main.cc` |
| | `simd.h` |
| | `flat_scan.h` |
| 近似 ANNS 相关代码 | `main_fastscan.cc`（包含索引构建相关代码） |
| | `pqflat.h` |
| | `pqsimd.h` |
| | `pqfastscan.h` |
| 预处理数据文件 | `pq.index1616` |

# 参考文献

[1] Fabien André, Anne-Marie Kermarrec, and Nicolas Le Scouarnec. Cache locality is not enough: High-performance nearest neighbor search with product quantization fast scan. In 42nd International Conference on Very Large Data Bases, volume 9, page 12, 2016.

[2] Arm Limited. Arm c language extensions (acle) for neon intrinsics. https://developer.arm.com/architectures/instruction-sets/simd-isas/neon, 2025.

[3] FAISS contributors. Fast accumulation of pq and aq codes (fastscan). https://github.com/facebookresearch/faiss/wiki/Fast-accumulation-of-PQ-and-AQ-codes-(FastScan), 2025.

[4] 杜忱莹, 安祺, 徐一帆, 曹珉浩, 华志远, 张逸非, and 孔德嵘. 并行程序设计实验指导书——simd 编程实验, 2025. SIMD 编程实验指导书.