南開大學
Nankai University

计 算 机 学 院

并行程序设计实验报告

基于 GPU 的高维向量最近邻搜索并行优化

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 7 月 2 日

摘要

本实验深入探究了利用图形处理器（GPU）的强大并行计算能力，对高维向量相似度搜索进行加速的多种策略。首先，针对精确的暴力搜索（NNS），实验通过将其核心的距离计算过程重构为大规模矩阵乘法，并调用高度优化的 NVIDIA cuBLAS 库，与 CPU 串行基准相比，实现了最高达 1648.6 倍的性能加速，验证了算法与硬件协同设计的巨大潜力。

其次，为了解决暴力搜索的计算开销问题，本实验进一步对基于 IVF 的近似搜索算法进行了 GPU 并行化。我设计并实现了一套包含并行粗筛、基于簇重合度的查询分组和并行精筛的三阶段的优化方案，相较于 CPU 串行 IVF 版本，取得了 47x 至 73x 的显著加速。此外，本实验引入了 Faiss-GPU 库作为对比。实验结果表明，Faiss 凭借其极致的底层优化，性能远超手动实现，加速比高达 345x 至 564x 。本次实验不仅成功实现了对向量搜索的大幅性能提升，也通过多层次方案的对比，揭示了在高性能计算中，算法设计、访存优化以及善用专业工具库的重要性。

关键字：ANN；GPU 加速；CUDA；IVF；Faiss

目录

一、 实验背景

在之前的实验中，我已经完成了 SIMD、多线程和多进程的实验，他们都是在 CPU 上进行并行化的。图形处理器（GPU）拥有数以千计的计算核心，为大规模并行计算提供了硬件基础：

1. **大规模并行计算能力：** GPU 拥有数千个计算核心，支持同时执行大量线程，适合处理高度数据并行的任务，如大规模向量相似度计算。相比之下，CPU 核心数量有限，更适合控制流密集的串行任务。
2. **更高的内存带宽：** GPU 具备远高于 CPU 主存的显存带宽，有效缓解数据传输瓶颈，提升大规模向量加载与批量计算的效率。
3. **更高的吞吐率：**在单位时间内，GPU 可以完成更多的浮点运算（FLOPS），特别适用于欧氏距离、内积等核函数的大规模并行计算任务。

本实验的目标即是利用 GPU 的这些特性，我将在 Nividia GPU 进行并行加速。

二、 并行设计与实现

(一) 串行平凡算法分析

串行暴力搜索算法的逻辑简单直观：对于每一个查询向量 q，遍历数据库中的全部 N 个基准向量 x_i。在遍历过程中，计算 q 与每一个 x_i 的距离（本实验采用内积距离 $1 - q \cdot x_i$），并使用一个大小为 k 的最大堆（或优先队列）来动态维护当前距离最小的 k 个向量的索引和距离。遍历结束后，堆中存储的即为最终的 Top-K 结果。

该算法的瓶颈显而易见：$N \times d$ 次乘法和加法运算构成了其主要的计算开销。当 N 和 d 很大时，总计算量极为庞大，导致查询延迟很高。

(二) GPU 并行化核心思想

为了利用 GPU 进行加速，必须摒弃“逐个查询、逐个比较”的串行思维，转向“**批量处理、并行计算**”的模式，也就是将距离计算重构为矩阵乘法。

假设我们有 m 个查询向量组成的查询批次，每个查询向量维度为 d。我们可以将其组织成一个 $m \times d$ 的矩阵 Q。整个数据库的 N 个向量可以组织成一个 $N \times d$ 的矩阵 B。我们希望计算出每个查询向量与每个数据库向量的内积，这恰好可以由矩阵乘法 $D = Q^T \times B$ 得到。结果将是一个 $m \times N$ 的矩阵 D_{ist}，其中每个元素 $D_{\text{ist}}[i][j]$ 就等于第 i 个查询向量与第 j 个数据库向量的内积。

通过这种方式，原本 $m \times N$ 次独立的内积计算（每次计算包含 d 次乘加）被统一成了一次大规模的矩阵乘法。这正是 GPU 最擅长的计算类型，能够将成千上万的计算核心全部调动起来，实现并行加速。

(三) GPU 实现流程与关键代码

整个 GPU 加速流程可以分为数据准备、核心计算和结果取回三个阶段 [3]。

1. 数据准备与 GPU 内存分配

在开始计算前，需要将主机（CPU）内存中的数据传输到 GPU 的显存中。我们首先为数据库向量、查询批次、距离矩阵和结果索引分配好对应的显存空间。其中，数据库向量 `d_base` 只需传输一次，而查询向量 `d_queries` 则在处理每个批次时进行更新。

```
float *d_base, *d_queries, *d_distances;
int *d_indices;

// 为数据库、查询批次、距离矩阵和结果索引分配显存
CUDA_CHECK(cudaMalloc(&d_base, base_number * vecdim * sizeof(float)));
CUDA_CHECK(cudaMalloc(&d_queries, BATCH_SIZE * vecdim * sizeof(float)));
CUDA_CHECK(cudaMalloc(&d_distances, base_number * BATCH_SIZE * sizeof(float)));
CUDA_CHECK(cudaMalloc(&d_indices, BATCH_SIZE * k * sizeof(int)));

// 将整个数据库向量从CPU拷贝到GPU（一次性操作）
CUDA_CHECK(cudaMemcpy(d_base, base, base_number * vecdim * sizeof(float),
    cudaMemcpyHostToDevice));

// 创建cuBLAS句柄
cublasHandle_t handle;
CUBLAS_CHECK(cublasCreate(&handle));
```

Listing 1: GPU 显存分配与数据初始化

2. 核心计算：矩阵乘法与 Top-K 筛选

这是算法的核心部分，在一个循环中按批次处理所有查询。

1. **cuBLAS 矩阵乘法** [2]:我调用 cublasSgemm 来计算内积矩阵。通过将查询矩阵 `d_queries` 进行转置（`CUBLAS_OP_T`），与原始的数据库矩阵 `d_base` 相乘，即 $D = Q^T \times B$，可以得到一个 `current_batch_size` × `base_number` 大小的距离矩阵 `d_distances`。矩阵中的每个元素 D_{ij} 即为第 i 个查询与第 j 个基准向量的内积。

```
// 计算内积: C(m,n) = op(A)_m*k * op(B)_k*n
// op(A) = d_queries^T, 维度(vecdim x batch_size)
// op(B) = d_base,      维度(vecdim x base_number)
// C = d_distances,   维度(batch_size x base_number)
const float alpha = 1.0f, beta = 0.0f;
CUBLAS_CHECK(cublasSgemm(handle,
            CUBLAS_OP_T, CUBLAS_OP_N,
            current_batch_size, base_number, vecdim,
            &alpha,
            d_queries, vecdim,
            d_base, vecdim,
            &beta,
            d_distances, current_batch_size));
```

Listing 2: 调用 cuBLAS sgemm 进行批量内积计算

2. **并行 Top-K 筛选核函数**: sgemm 高效地生成了距离矩阵，但如何从这个巨大的矩阵中为每个查询找出 Top-K 的条目，是下一个挑战。为此，我们设计了 `find_topk_heap` 核函数，利用共享内存来加速筛选过程。

```cpp
int threads = 128;
int blocks = (current_batch_size + threads - 1) / threads;
// 计算每个线程块需要的共享内存大小 (k个距离 + k个索引)
size_t shared_mem_size = threads * k * (sizeof(float) + sizeof(int));

// 启动核函数, 并将共享内存大小作为参数传入
find_topk_heap<<<blocks, threads, shared_mem_size>>>(
    d_distances, d_indices, base_number, current_batch_size, k);

// 错误检查与同步
CUDA_CHECK(cudaGetLastError());
CUDA_CHECK(cudaDeviceSynchronize());
```

Listing 3: Top-K 筛选核函数启动

在 `find_topk_heap` 核函数内部，每个线程块负责一个查询，块内线程协同，利用高速的共享内存构建大小为 k 的堆，从而高效地完成比较和筛选，避免了大量对慢速全局显存的访问。

3. 结果取回

最后，将 GPU 上计算好的 Top-K 结果索引 `d_indices` 拷贝回 CPU 内存中，以便进行后续的 Recall 计算和结果验证。

```cpp
std::vector<int> batch_indices(current_batch_size * k);
CUDA_CHECK(cudaMemcpy(batch_indices.data(), d_indices,
           current_batch_size * k * sizeof(int), cudaMemcpyDeviceToHost));
```

Listing 4: 将结果从 GPU 拷贝回 CPU

三、 IVF 的 GPU 并行加速

(一) IVF 查询瓶颈与 GPU 优化思路

IVF 算法通过将数据集聚类，将全量搜索缩小为对少数几个簇（由 nprobe 参数指定）的搜索，从而大幅减少了计算量 [4]。但其查询过程包含两个计算密集型阶段，也比较适合并行加速。

1. **粗筛瓶颈**: 对于一个查询，需要计算它与所有 N_c 个簇质心的距离，以确定最近的 nprobe 个簇。当查询数量很大时，这部分计算量不容忽视，可以并行。
2. **精筛瓶颈**: 在确定了目标簇之后, 需要在这些簇包含的所有向量中进行暴力搜索, 这是 GPU 并行的典型场景，在上一个部分已经详细的讨论过。
3. **访存挑战**: 不同的查询会访问不同的簇，这会导致矩阵乘法中有很多计算其实是浪费的，因此在查询的时候尽量让 batch 中的向量的最近簇重合度较高。

(二) GPU 并行实现细节

1. 阶段一：并行化的粗筛

与暴力搜索的优化思路一致，我们将粗筛阶段也重构为矩阵乘法。将所有簇的质心向量视为一个“质心数据库”，然后利用 cuBLAS 的 sgemm 函数，一次性计算一个查询批次中所有查

询向量与所有质心向量的距离。随后，通过一个并行的 Top-K 核函数，为每个查询高效地选出 nprobe 个最近的质心。这一步将粗筛过程完全并行化。

2. 阶段二：基于簇重合度的查询分组

在划分 batch 的时候，不再像上一个部分那样随机划分：获得每个查询需要访问的簇列表后，不在 GPU 上立即处理，而是在 CPU 端执行一个预处理步骤：**查询分组**。

该策略的动机是，一个查询批次中的不同查询，它们的目标簇列表可能存在很高的重合度。如果我们将这些查询分为一组，就可以为这一整组一次性加载它们所需的所有簇的并集到 GPU 显存中，从而避免了为每个查询反复加载相同的数据 [5]。ivf_gpu.cu 中的 GroupQueriesByClusterOverlap 函数正是这一思想的体现，它通过计算查询间目标簇的交集大小，将簇重合度高的查询（例如，重合度超过 50%）划分到同一个 `QueryGroup` 中。

Algorithm 1 基于簇重合度的查询分组策略伪代码

Input: 所有查询的目标簇列表 `query_clusters`, 查询总数 `num_queries`
Output: 查询组列表 `groups`

processed[num_queries] ← {false}
for $i \leftarrow 0$ **to** `num_queries-1` **do**
  if processed[i] **then continue**
  end if
  创建新组 `new_group`, 将查询 i 加入
  `base_clusters` ← 查询 i 的目标簇集合
  for $j \leftarrow i+1$ **to** `num_queries-1` **do**
    if processed[j] **or** 组已满 **then continue**
    end if
    `other_clusters` ← 查询 j 的目标簇集合
    `overlap` ← | `base_clusters` ∩ `other_clusters` |
    if `overlap` ≥ 阈值 **then**
      将查询 j 加入 `new_group`
      `base_clusters` ← `base_clusters` ∪ `other_clusters`
      processed[j] ← true
    end if
  end for
  将 `new_group` 加入 `groups`
end for

通过这种方式，极大地提升了数据局部性和访存效率。但是实际发现，这个阈值的控制需要小心，如果太大那么很可能每个 batch 只有几个向量，这就体现不出批处理的优点了，甚至会负优化；太小则重合度低，导致这个策略的收益又不是特别理想。

3. 阶段三：分组化的并行精筛

在 CPU 完成分组后，GPU 开始对每个查询组进行处理，这一部分和暴力搜索的 GPU 优化类似，此处略讲：

1. **数据加载**: 对于一个查询组，首先确定其需要访问的所有簇的并集。然后，仅将这些簇包含的向量数据从全局 IVF 索引中拷贝到 GPU 的一块连续的临时显存中。

2. **并行搜索**: 启动一个专门为分组搜索设计的 CUDA 核函数。该核函数接收查询组的信息（包含哪些查询）和刚刚加载到显存中的簇数据。

3. **核函数内部逻辑**: 在核函数中，每个线程或线程块被分配给组内的一个特定查询。该线程只在加载到显存的这部分数据中进行搜索，并根据查询信息精确地计算其与目标簇内向量的距离。最终的 Top-K 结果筛选同样可以利用共享内存进行加速。

这个阶段实现了计算和访存的双重优化：不仅计算被并行化，而且由于分组策略，GPU 核函数工作时的数据访问具有了良好的空间局部性。

(三) Faiss-GPU 的 IVF 实现

在完成上述的工作后,搜集资料发现 Faiss 有专门对 GPU 优化的加速工具。Faiss 是 Facebook AI 研究院开源的、用于高效稠密向量相似度搜索的库，其 GPU 模块提供了业界顶尖的性能 [1]。使用 Faiss 可以让我们将精力集中在算法流程上，而将底层的 CUDA 优化细节交给高度工程化的库来处理。我也编写了调用 Faiss-GPU 库的并行代码，作为对比。

1. Faiss-GPU 工作流程

使用 Faiss-GPU 实现 IVF 搜索的流程被极大地简化了，主要分为索引构建和搜索两个阶段。

1. **索引构建**: Faiss 提供了一个便捷的函数 `faiss::gpu::index_cpu_to_gpu`，它可以将一个在 CPU 上训练好并填充了数据的 IVF 索引，“克隆”到 GPU 上。在这个过程中，Faiss 会自动处理所有的数据传输、显存分配，并对数据布局进行优化（例如转置存储）以适应 GPU 的访存模式。

2. **搜索**: 搜索过程只需要调用 GPU 索引对象的 `search` 方法。所有复杂的并行操作，包括查询批处理、粗筛、精筛、多 Stream 异步执行等，都由 Faiss 在内部自动完成。

2. 关键代码解析

Faiss 的高度封装使得代码非常简洁。

1. 索引构建与 GPU 克隆 首先在 CPU 上构建一个 `IndexIVFFlat` 对象，对其进行训练和数据添加。然后，使用 `GpuCloner` 将其转换为 GPU 索引。

```
// 1. 在CPU上创建和训练索引
auto quantizer = new faiss::IndexFlatL2(dimension);
auto cpu_index = std::make_unique<faiss::IndexIVFFlat>(quantizer, dimension,
    num_clusters);
cpu_index->train(n, training_data);
cpu_index->add(n, training_data);

// 2. 设置GPU克隆选项并转移
faiss::gpu::GpuClonerOptions options;
options.storeTransposed = true; // 优化访存
auto gpu_index = faiss::gpu::index_cpu_to_gpu(
    gpu_resources.get(), 0, cpu_index.get(), &options);
```

Listing 5: 使用 Faiss 将 CPU 索引克隆至 GPU

2. 搜索接口调用 搜索时，只需简单地调用 GPU 索引的 search 方法，传入查询数据和参数即可。

```
// 设置要探索的簇数量
gpu_index->nprobe = nprobe;

// 分配结果存储空间
std::vector<float> distances(nq * k);
std::vector<faiss::idx_t> indices(nq * k);

// 执行搜索，Faiss内部处理所有并行细节
gpu_index->search(nq, queries, k, distances.data(), indices.data());
```

Listing 6: 调用 Faiss-GPU 进行搜索

四、 实验结果与分析

(一) 暴力搜索——GPU 优化

为验证 GPU 并行方案的有效性，我将其与一个优化的单线程 CPU 串行暴力搜索算法进行了对比。测试均在 DEEP100K 数据集上进行，Top-K 的 k 值设为 10。在实验中，我调整 batch 的大小，得到的 QPS 和加速比如表1所示。

从表1结果可以看出，GPU 并行方案在查询吞吐量（QPS）方面相比 CPU 基准方案有了显著提升。具体而言，当 batch size 为 10000 时，GPU 方案的 QPS 达到 173140.5，约为 CPU 基准方案的 **1648.6 倍**，显示了大规模并行处理在向量搜索任务中的**巨大**优势。

- 性能的提升与 batch 大小呈正相关关系，并且比较线性。随着 batch size 增大，GPU 的硬件资源利用率逐渐提高，内核切换和内存访问延迟等开销在整体执行时间中的占比降低，从而进一步释放了吞吐潜力。
- 需注意的是，随着 batch size 继续扩大，受限于显存容量和 PCIe 传输带宽等硬件因素，性能增益可能会逐渐趋于饱和，本次实验的数据规模不是非常大，因此没有体现出性能拐点。
- 综合来看，基于 GPU 的并行加速方案在高并发场景下表现出优异的扩展性和处理能力，能够有效满足大规模向量检索系统对实时性和吞吐率的需求。

表 1: 不同方案下的性能对比（Recall=1）

算法方案	QPS (queries/sec)	加速比
flat (基准)	105.0 ± 0.2	1.0x
GPU_500	13 809.0 ± 115.8	130.3x
GPU_1000	27 861.1 ± 461.3	261.6x
GPU_2000	50 568.2 ± 211.7	483.9x
GPU_5000	107 893.8 ± 120.8	1027.5x
GPU_10000	173 140.5±2177.3	**1648.6**x

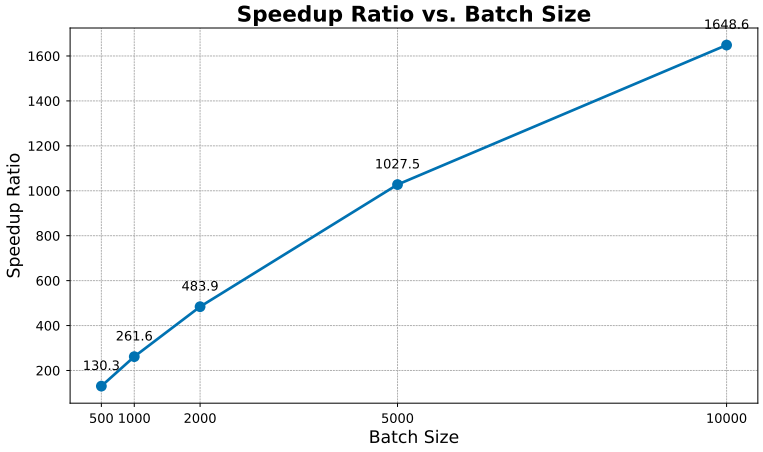

图 1: Batch Size 和加速比的关系

(二) IVF——GPU 优化

为了验证 GPU 对 IVF 算法的加速效果，我对自己实现的 GPU 版本（`IVF_gpu`）、业界知名的 Faiss 库的 GPU 版本（`IVF_faiss_gpu`）以及纯 CPU 串行版本（`IVF_cpu`）的性能进行了对比测试。测试主要关注在不同召回率下的每秒查询率（QPS），实验结果如图2所示。

从 recall-latency tradeoff 曲线中可以直观地看出，两种基于 GPU 的实现方案在性能上都远超 CPU 基线。由于 QPS 坐标轴采用了对数尺度，CPU 串行版本和 `IVF_cpu` 曲线之间的巨大差距，实际上代表了**几个数量级的性能提升**。这初步证明了将 IVF 算法移植到 GPU 平台进行计算的巨大优势。

表2详细量化了加速效果。在不同的召回率设定下：

- IVF 串行版本的性能已经逼近 GPU 优化的暴力搜索（batch size=500）的性能，这凸显了近似最近邻搜索（ANNS）的核心价值，即通过牺牲一定的召回率来换取查询性能的巨大提升，从而在 CPU 上实现了比肩 GPU 暴力搜索的效率。
- 我自己实现的 `IVF_gpu` 版本相较于 CPU 版本，实现了约 **47**× 到 **73**× 的性能提升。
- 而 `IVF_faiss_gpu` 版本的表现则更为出色，其加速比达到了惊人的 **345**× 到 **564**×。这也体现了 faiss 库的成熟和先进，其在底层 CUDA 核函数的设计、内存访问模式以及并行计算调度上都进行了深度优化，从而实现了极致的性能表现。

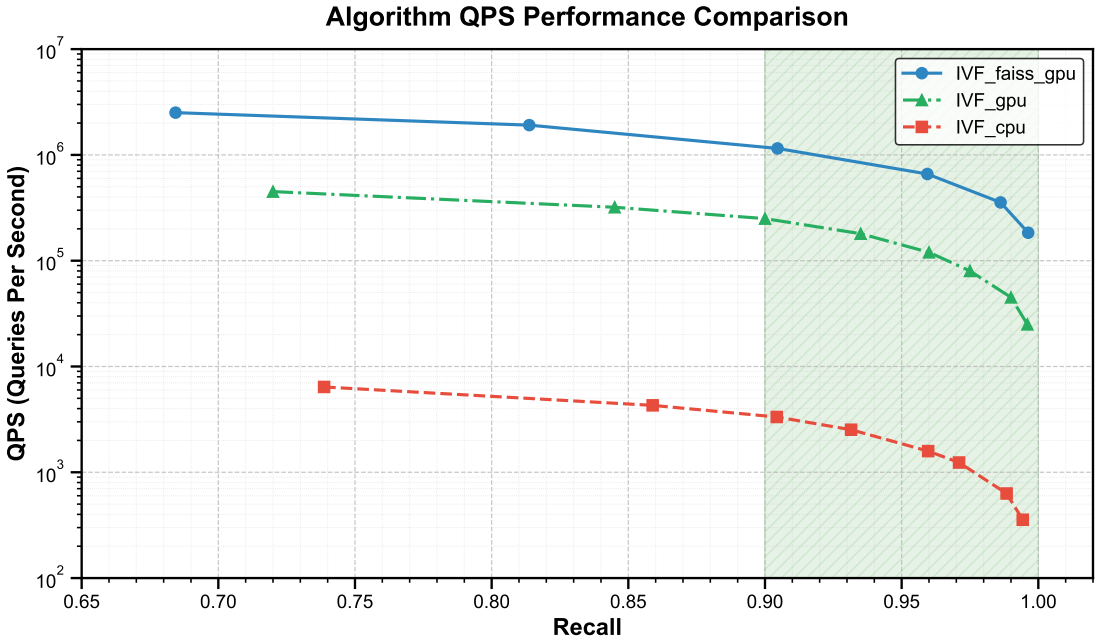

图 2: GPU 优化 IVF 的 QPS

表 2: 不同召回率 (Recall) 下的加速比 (Speedup) 对比

方法	**召回率 (Recall)**					
	0.68	0.81	0.9	0.95	0.986	0.995
IVF_gpu	47.07x	69.91x	58.52x	66.24x	73.01x	49.18x
IVF_faiss_gpu	363.49x	425.91x	345.59x	416.41x	**564.14x**	516.38x

五、 总结

在这次实验中, 我深入探究了如何利用 GPU 的并行计算能力来加速高维向量的相似度搜索, 成功设计、实现并评估了两种核心的 GPU 优化策略：暴力搜索的并行化和 IVF 近似搜索算法的并行化。

1. **暴力搜索 GPU 优化**：为了充分利用 GPU 的大规模并行能力，我将传统的串行搜索重构为批量的矩阵乘法运算 ($Q \times B^T$)。实验证明，该方法取得了巨大成功，与单线程 CPU 基准相比，实现了最高 1648.6 倍的惊人加速比。

2. **IVF 算法 GPU 优化**：为了解决暴力搜索的巨大计算开销，我进一步对 IVF 近似搜索算法进行了 GPU 并行化。一系列设计实现了计算与访存的双重优化，与 CPU 版本相比，我的实现取得了 47 倍到 73 倍的显著加速。

   我也下载并调用了 faiss-gpu 库去实现 IVF 的 GPU 并行加速，并将其与我的实现进行了对比。实验结果表明，Faiss-GPU 版本的性能更为出色，加速比达到了惊人的 345 倍到 564 倍。一个高度工程化的专业库在底层 CUDA 设计和内存访问模式上的深度优化，能够实现极致的性能表现。

除此之外我还有以下启发：

- **算法与硬件的协同设计**：要释放 GPU 的潜能，必须在算法层面进行适配。将搜索问题巧妙地“翻译”成 GPU 最擅长的矩阵运算，是并行化成功的关键。
- **专业工具库的价值**: 通过与 Faiss 库的对比, 我深刻体会到“不要重复造轮子”的原则。Faiss 将复杂的底层 CUDA 优化、内存管理和异步执行等细节封装起来，让开发者能更专注于上层算法逻辑，同时获得顶尖的性能，这在工程实践中极具价值。
- **批处理（Batching）的力量**：在之前的优化中我都关注的是单个查询的耗时，但为了充分利用 GPU 数以千计的核心，我们必须提供足够多的并行任务，这使得“批处理”成为所有 GPU 应用开发中的一个基础且关键的概念。

跳转至：源代码地址，主要结构如下：

类别	文件/目录

GPU 相关代码	`ivf_gpu.cu`
	`faiss_gpu_ivf.cpp`
	`main_flat_gpu.cu`

CPU 相关代码	`flat_cpu.cpp` `ivf_cpu.cpp`
安装及依赖项	`install_faiss.sh` `cmake-3.27.7-linux-x86_64.tar.gz`

参考文献

[1] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, 7(3):535–547, 2019.

[2] NVIDIA Corporation. *cuBLAS Library User's Guide*, 2024. CUDA Toolkit 12.4.

[3] NVIDIA Corporation. *CUDA C++ Programming Guide*, 2024. Version 12.4.

[4] Josef Sivic and Andrew Zisserman. Video Google: A text retrieval approach to object matching in videos. In *Proceedings of the Ninth IEEE International Conference on Computer Vision*, volume 2, pages 1470–1477, 2003.

[5] 李世阳, 孙辉, 华志远, 孔德嵘, and 张逸非. *GPU* **编程实验指导书**, 2024. 并行程序设计实验指导书.