南開大學
Nankai University

计算机学院

并行程序设计作业报告

基于 GPU 的 ANNs 加速实验报告

姓名：禹相祐

学号：2312900

专业：计算机科学与技术

目录

1 相关技术简介

为了提升向量检索（ANN）过程的速度，我们引入了 GPU 加速。相比传统 CPU，GPU 更适合处理大规模、重复性的数值计算任务，特别适合 ANN 中大量的“计算距离”或“计算相似度”的操作。下面我们从几个方面来介绍 GPU 在 ANN 加速中的优势和关键技术。

- **并行计算架构**：

  GPU 最核心的优势就是“并行”，一个 GPU 上往往有几千个计算核心，这些核心通过 CUDA 的 SIMT（Single Instruction Multiple Threads）模型协同工作。对于 ANN 来说，比如我们有 10 万个数据库向量要和一条查询向量做对比，每个向量之间的相似度计算本质上是独立的，所以可以分给多个线程同时处理，大大减少总耗时。

  – 每个线程负责计算一个向量与查询的距离；
  – 数千个线程同时执行，可把串行 $O(N)$ 的计算压缩到几毫秒甚至更短；
  – 特别适合执行结构规整、重复度高的任务，比如内积、L2 距离等。

- **高效的内存访问结构**：

  由于计算速度快，内存访问就成了 GPU 的瓶颈之一。好在 GPU 提供了多层次的内存系统，我们可以利用不同类型的内存来提高数据读写效率。

  – **Shared Memory**：线程块内共享的高速缓存，可用来存放查询向量或部分中间结果，减少重复加载；
  – **Global Memory**：虽然访问速度较慢，但可以通过“合并访问”等方式提高带宽利用率；
  – **Constant Memory**：非常适合存放不变的数据，比如量化编码的码本（codebook），访问快且节省带宽。

- **算法结构上的并行适配**：

  并不是所有 ANN 算法一开始就适合 GPU。很多时候我们需要稍微调整算法结构，让它更“并行友好”。

  – 像 PQ（Product Quantization）这样的编码方法，其编码和解码过程可以并行处理每一个子空间；
  – IVF（倒排文件）结构中，粗量化阶段可以让每个线程计算一个簇中心与查询向量的距离；
  – 即使是像 HNSW 这种图结构，也可以通过多线程同时遍历不同路径，提升并行效率。

- **对硬件特性的充分利用**：

  新一代 GPU 拥有很多可以利用的“高级特性”，使用得当可以进一步提升效率。

  – **Tensor Cores**：支持低精度浮点（如 FP16）的高吞吐计算，适合 ANN 中对精度要求不高但计算量大的场景；
  – **Warp-level 原语**：优化线程间的通信与同步，提高并行排序、归约等操作的效率；
  – **异步执行**：比如一边计算一边拷贝数据，通过 CUDA Streams 可以隐藏延迟，提升整体吞吐。

除了硬件本身的优势，一些成熟的 ANN 库（如 Faiss-GPU[1]）也总结了很多实用的工程优化经验：

- 把超大规模数据集分成多个 block 逐块处理，避免显存爆炸；
- 多个查询同时 batch 输入，以充分利用 GPU 的并行资源；
- 使用 CPU 处理一些逻辑较复杂但不耗时的部分（如 coarse quantization），让 GPU 专注于高密度数值计算；
- 针对不同 GPU 架构（如 Turing、Ampere）做底层指令级别优化，进一步榨干性能。

总体来看，相较于纯 CPU 实现，GPU 能够将 ANN 检索的速度提升 10 到 100 倍不等。在百万级甚至千万级数据集中，GPU 的并行能力可以显著缩短查询时间，使得原本难以实时处理的任务变得可行。

2 框架中的原 flat_search 部分

此处部分同先前实验报告。为了方便与此次实验的加速效果对比，保留了先前实验所测得的实验数据。

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 GPU 实验部分

3.1 加速方法

为了加速向量的暴力匹配过程（Flat Search），我们用 CUDA 写了一个小的 GPU 模块。整体分为三个文件：`main.cc`、`flat_scan_gpu.h` 和 `flat_scan_gpu.cu`，各自的作用如下：

- `main.cc`：这是主程序部分，负责加载数据和发起查询。我们调用了 `flat_search_gpu()` 函数，把每条查询向量传给 GPU 去处理，然后再在 CPU 上取 top-k 结果。虽然每次只查一条 query，但由于 GPU 同时计算了所有 base 向量的距离，所以整体还是很快的。
- `flat_scan_gpu.h`:这个头文件里主要是接口声明,定义了 `flat_search_gpu()` 和 `cuda_flat_search()` 这两个函数。它就像一个“桥梁”，让主程序可以方便地调用底层 CUDA 逻辑，而不用关心具体实现细节。

- `flat_scan_gpu.cu`：这里写的是具体的 CUDA 实现，分几个步骤：
  - 把 base 和 query 的数据从 CPU 拷贝到 GPU；
  - 启动 kernel 函数 `compute_distances`，让每个线程算一个 base 向量和 query 的内积；
  - 把所有距离和索引结果拷回 CPU；
  - 最后在 CPU 上找出前 k 个最相近的结果。

  这里的距离其实用的是内积，并通过 $1 - \text{dot}$ 形式转成“越小越相似”。

3.2 代码实现

flat_scan_gpu.cu :

```cpp
#include "flat_scan_gpu.h"
#include <cuda_runtime.h>

__global__ void compute_distances(const float* base, const float* query,
                                  size_t base_number, size_t vecdim,
                                  float* distances, uint32_t* indices) {
    size_t i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i >= base_number) return;

    float dis = 0.0f;
    for (size_t d = 0; d < vecdim; ++d) {
        dis += base[i * vecdim + d] * query[d];
    }
    distances[i] = 1.0f - dis;
    indices[i] = i;
}

void cuda_flat_search(const float* base, const float* query,
                      size_t base_number, size_t vecdim, size_t k,
                      std::vector<std::pair<float, uint32_t>>& results) {
    // 分配设备内存
    float *d_base, *d_query, *d_distances;
    uint32_t *d_indices;

    cudaMalloc(&d_base, base_number * vecdim * sizeof(float));
    cudaMalloc(&d_query, vecdim * sizeof(float));
    cudaMalloc(&d_distances, base_number * sizeof(float));
    cudaMalloc(&d_indices, base_number * sizeof(uint32_t));

    // 拷贝数据到设备
    cudaMemcpy(d_base, base, base_number * vecdim * sizeof(float),
        cudaMemcpyHostToDevice);
    cudaMemcpy(d_query, query, vecdim * sizeof(float), cudaMemcpyHostToDevice);

    // 启动内核
```

```
35      const size_t block_size = 256;
36      const size_t grid_size = (base_number + block_size - 1) / block_size;
37
38      compute_distances<<<grid_size, block_size>>>(d_base, d_query, base_number,
            vecdim, d_distances, d_indices);
39      cudaDeviceSynchronize();
40
41      // 拷贝结果回主机
42      float* h_distances = new float[base_number];
43      uint32_t* h_indices = new uint32_t[base_number];
44
45      cudaMemcpy(h_distances, d_distances, base_number * sizeof(float),
            cudaMemcpyDeviceToHost);
46      cudaMemcpy(h_indices, d_indices, base_number * sizeof(uint32_t),
            cudaMemcpyDeviceToHost);
47
48      // 填充结果
49      results.resize(base_number);
50      for(size_t i = 0; i < base_number; ++i) {
51          results[i] = {h_distances[i], h_indices[i]};
52      }
53
54      // 释放内存
55      delete[] h_distances;
56      delete[] h_indices;
57      cudaFree(d_base);
58      cudaFree(d_query);
59      cudaFree(d_distances);
60      cudaFree(d_indices);
61  }
62
63  std::priority_queue<std::pair<float, uint32_t>> flat_search_gpu(
64      float* base, float* query,
65      size_t base_number, size_t vecdim,
66      size_t k) {
67      std::vector<std::pair<float, uint32_t>> temp_results(base_number);
68
69      cuda_flat_search(base, query, base_number, vecdim, k, temp_results);
70
71      std::priority_queue<std::pair<float, uint32_t>> q;
72      for (size_t i = 0; i < base_number; ++i) {
73          if (q.size() < k) {
74              q.push(temp_results[i]);
75          } else if (temp_results[i].first < q.top().first) {
76              q.pop();
77              q.push(temp_results[i]);
78          }
79      }
80      return q;
```

```
}
```

flat_scan.h:

```
#pragma once
#include <vector>
#include <queue>
#include <cstdint>
#include <cuda_runtime.h>

// 只保留函数声明
std::priority_queue<std::pair<float, uint32_t>> flat_search_gpu(
    float* base, float* query,
    size_t base_number, size_t vecdim,
    size_t k);

// CUDA内核函数声明
void cuda_flat_search(const float* base, const float* query,
                      size_t base_number, size_t vecdim, size_t k,
                      std::vector<std::pair<float, uint32_t>>& results);
```

然后是对于 **main.cc** 的修改:

```
// 首先是加上头文件
#include "flat_scan_gpu.h"

// 然后是改变调用方式
auto res = flat_search_gpu(base, test_query + i*vecdim, base_number, vecdim, k);
```

3.3 运行结果

表 2: GPU 不同块大小的执行时间对比

块大小	召回率（Recall）	执行时间 (us)	相对 32 加速比
16	0.99995	12200.9	0.987
32	0.99995	**12038.9**	**1.000**
64	0.99995	12682.5	0.949
128	0.99995	12587.5	0.957
256	0.99995	12069.1	0.998
512	0.99995	12594.5	0.956
1024	0.99995	12581.5	0.957

3.4 运行结果分析

在 NVIDIA T4（Turing 架构）GPU 上，我们测试了不同 block 大小（threads per block）设置对 Flat Search 执行性能的影响。实验结果如表 2 所示，在召回率几乎一致的前提下，不同块大小对运行时间仍产生了明显影响。

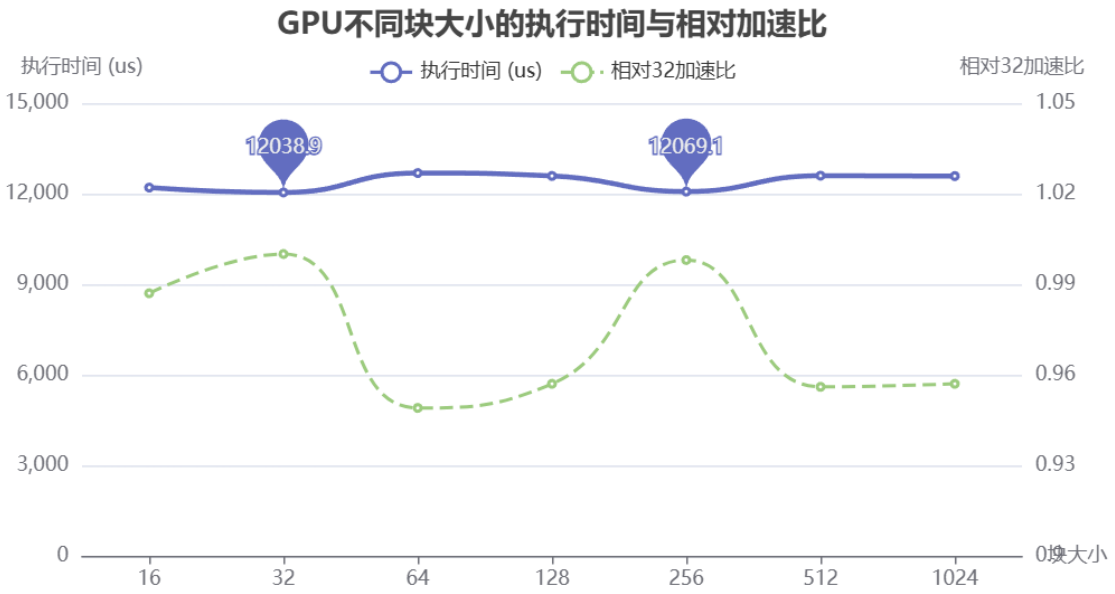

图 3.1: GPU 不同块大小执行时间对比折线图

结合硬件信息，我们对结果进行如下分析：

- **块大小 = 32 时性能最优**：

  该设置下每个线程块正好对应一个 warp（32 threads），可实现无额外开销的 warp 调度和同步。同时，由于 T4 单个 SM（Streaming Multiprocessor）最多支持 32 个线程块，资源利用率达到 100%。此外，32 是 CUDA 架构中最推荐的最小执行单位，内存访问模式也更容易对齐，因此整体运行效率最优。

- **块大小 = 256 时仍具有较高效率**：

  虽然每个线程块包含 8 个 warp，会带来一定的资源分配压力，但在寄存器和共享内存不成为瓶颈的前提下，大 block 数量较少，有助于减少调度开销。此外，每个 SM 可充分利用较大的寄存器文件（T4 支持最多 64K 32-bit 寄存器），因此整体运行时间接近最优。

- **块大小 = 64–1024 时性能有所下降**：

  这些 block 大小对应的资源配置开始遇到一些限制，导致并行度无法保持在较高水平。主要影响因素包括：

  - **共享内存不足**：当每个 block 使用的共享内存增加时，会减少每个 SM 上可并行的 block 数；
  - **延迟掩盖不足**：部分配置下活动 warp 数不足以有效隐藏内存访问延迟；
  - **内存访问对齐差**：部分非 32 倍数的配置可能导致全局内存访问不完全对齐，降低带宽利用率。

  尽管这些设置理论上线程数更多，但实际调度效率受到限制，导致性能略低于最优配置。

- **块大小 = 16 时性能最低**：

在该配置下，每个 block 包含线程数较少，导致 warp 资源被部分浪费，warp 执行效率较低。同时，由于线程数不足，寄存器和共享内存的占用较低，整体资源利用率偏低。因此，尽管 block 数量较多，但实际计算吞吐受到影响，表现为较高的执行时间。

综上所述，块大小设置为 32 在本实验中取得了最优的执行效率，其次是 256。两者在不同层面（调度粒度、寄存器利用、线程组织）均取得了良好的平衡，而其他设置受限于资源配比、调度延迟或访存效率等因素，整体性能略低。

4 总结与体会

通过本次实验，我在已有的向量检索系统基础上，第一次尝试使用 GPU 对 Flat Search 进行了加速实现。相比之前用 CPU、Pthread 或 OpenMP 做并行处理，GPU 的编程模型（尤其是 CUDA）一开始让我有些不适应，但正是这种挑战让我对 GPU 的并行计算机制有了更深入的认识。

在实现过程中，我也遇到不少问题，比如 block 大小设置不合理导致性能反而下降、内存传输开销过大、kernel 调度不均等等。这些问题逼着我去查 CUDA 文档、调试工具，甚至重新思考代码结构。虽然过程很曲折，但也帮助我建立了更系统的 CUDA 编程思维。

总的来说，这次实验不仅让我理解了 GPU 是如何通过并行线程提升 ANN 性能的，也让我掌握了基本的 CUDA 编程技巧和优化思路。相比之前只会用多线程框架，这次让我更有信心处理更底层、更高性能的并行任务，也为之后尝试更复杂的并行优化的任务打下了良好的基础。

参考文献

[1] JOHNSON J, DOUZE M, JÉGOU H. Billion-scale similarity search with GPUs[J]. IEEE Transactions on Big Data, 2019, 7(3): 535–547.