多核程序设计与实践期末项目

高维空间中的最近邻

中山大学计算机学院　计算机科学与技术

施天予　　王郅成　　王筝

19335174　　19335202　　19335199

目录

一、背景

最近邻搜索（NNS）又称为“最近点搜索”（Closest point search），是一个在尺度空间中寻找最近点的优化问题。问题描述如下：在尺度空间 M 中给定一个点集 S 和一个目标点 $q \in M$，在 S 中找到距离 q 最近的点。很多情况下，M 为多维的欧几里得空间，距离由欧几里得距离或曼哈顿距离决定。

在本次实验中，我们使用线性查找实现了 1 个 CPU 版本和 8 个 GPU 版本，使用 KD-Tree 实现了 1 个 CPU 版本和 1 个 GPU 版本，使用 Octree 实现了 1 个 CPU 版本和 1 个 GPU 版本，并分析对比了各个算法及其优化的性能。

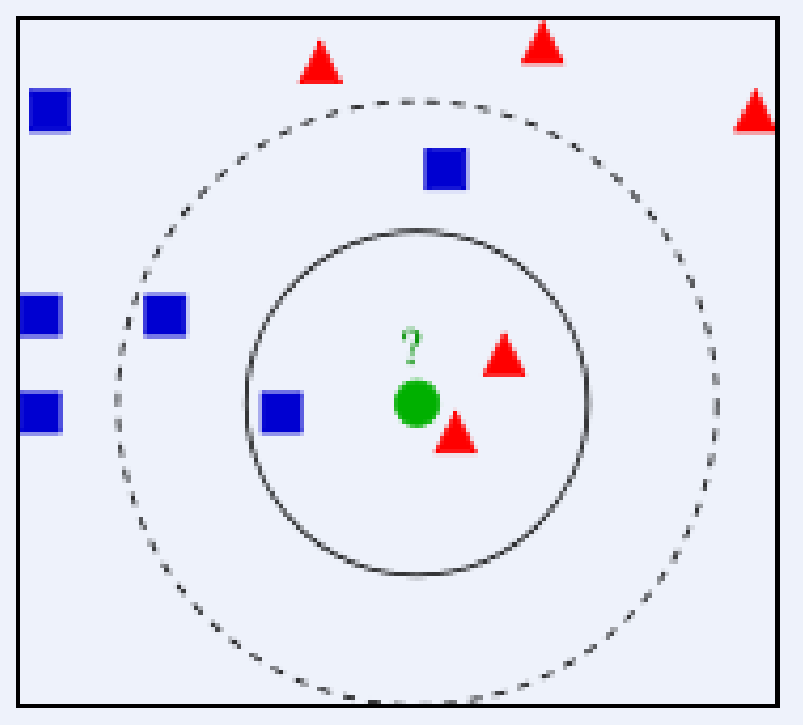

图 1: 最近邻搜索

二、程序逻辑

1. 文件组织

本次实验共有 3 个代码文件，各文件内容如下：

1. main.cu：CUDA 主函数
2. core.cu：CUDA 调用的各种优化版本
3. utils.h：一些通用的库函数

2. 生成随机样例

本次实验计算高维空间中的最近邻，我们首先需要生成所有的随机点样例。getRand 函数用于生成一个随机的浮点数。在 getSample 函数中，k 代表空间维度，m 代表查询点集的数量，n 代表目标点集的数量，s_points 代表查询点集，r_points 代表目标点集。

```
// 生成随机数
float getRand() {
    return rand() / double(RAND_MAX);
}
// 生成随机样例
void getSample(int k, int m, int n, float **s_points, float **r_points) {
    float *tmp;
    tmp = (float*)malloc(sizeof(float) * k * m);
    for (int i = 0; i < k * m; i++)
        tmp[i] = getRand();
    *s_points = tmp;
    tmp = (float*)malloc(sizeof(float) * k * n);
    for (int i = 0; i < k * n; i++)
        tmp[i] = getRand();
    *r_points = tmp;
}
```

3. 硬件热身

CUDA 运行时采用的是惰性初始化策略，有一定的冷启动时间。为便于性能的测试，我们设计了 WarmUP 来在启动真正的计算核函数之前预先运行若干次无关的核函数，以便于吸收这些冷启动的延迟。在实现中，我们将各个版本的核函数预先执行，而由于我们有用到多卡的优化版本，所以需要使用 OpenMP 将 4 张 NVIDIA Tesla V100 充分热身。具体代码如下：

```
struct WarmUP
{
    WarmUP(int k, int m, int n)
    {
        float *s_points = (float *)malloc(sizeof(float) * k * m);
        float *r_points = (float *)malloc(sizeof(float) * k * n);
        #pragma omp parallel
        {
            unsigned int seed = omp_get_thread_num(); //每个线程使用不同的随机数种子
            #pragma omp for
            for (int i = 0; i < k * m; ++i)
                s_points[i] = rand_r(&seed) / double(RAND_MAX); //使用线程安全的随机
                    ↪ 数函数
            #pragma omp for
            for (int i = 0; i < k * n; ++i)
```

```
            r_points[i] = rand_r(&seed) / double(RAND_MAX);
        }
        for (int i = 0; i < 10; ++i)
        {
            int *result;
            v9::cudaCall(k, m, n, s_points, r_points, &result);
            free(result);
        }
        free(s_points);
        free(r_points);
    }
};
static WarmUP warm_up(1, 1, 1 << 15);
```

4. 样例测试

func 函数代表通用的可调用的不同 CUDA 优化版本函数，test 函数对所有不同组的随机样例进行测试。getTime 函数用于计算程序运行的时间。

```
// CUDA通用函数声明
void (*func)(int, int, int, float*, float*, int**);
// 样例测试
void test(int v) {
    srand(seed);
    for (int i = 0; i < total; ++i) {
        int k = samples[3*i];
        int m = samples[3*i+1];
        int n = samples[3*i+2];
        float *s_points, *r_points;
        getSample(k, m, n, &s_points, &r_points);
        int *results;
        st = getTime();
        (*func)(k, m, n, s_points, r_points, &results);
        et = getTime();
        printf("CudaCall %d, %2d, %4d, %5d, %10.3fms\n", v, k, m, n, (et - st) / 1
            ↪ e6);
        free(s_points);
        free(r_points);
    }
}
```

三、CUDA 优化

1. V0：CPU 串行版本

V0 版本是 CPU 的串行版本，也可以看作是验证 CUDA 计算正确性的函数。其逻辑如下：

1. 枚举查询点集中的所有点，每个点下标为 i
2. 初始化当前点之间的最优距离为无限大
3. 枚举目标点集中的所有点，每个点下标为 j
   (a) 计算当前查询点与目标点之间的距离（为减少不必要的计算量，舍去开方运算）
   (b) 若计算距离比当前最优距离更短
      i. 更新当前最优距离
      ii. 记录当前目标点下标
4. 保存最优目标点并返回结果

```
namespace v0
{
    extern void cudaCall(
        int k,           // 空间维度
        int m,           // 查询点数量
        int n,           // 目标点数量
        float *s_points,  // 查询点集
        float *r_points,  // 目标点集
        int **results)   // 最近邻点集
    {
        int *tmp = (int *)malloc(sizeof(int) * m);
        for (int i = 0; i < m; ++i) {
            float minSum = INFINITY;
            int index = 0;
            for (int j = 0; j < n; ++j) {
                float tempSum = 0;
                for (int t = 0; t < k; ++t) {
                    const float diff = s_points[i * k + t] - r_points[j * k + t];
                    tempSum += diff * diff;  // 计算距离
                }
                if (minSum > tempSum) {     // 找出最小点
                    minSum = tempSum;
                    index = j;
                }
            }
            tmp[i] = index;
        }
        *results = tmp;
    }
}
```

2. V1：GPU 计算距离矩阵 + 共享内存树形归约

在 GPU 上设计并行算法需要解决串行算法的循环依赖问题，比如在 V0 中遍历 n 个目标点的 for 循环中，比较当前最优距离具有明显的循环依赖性，因此我们可以将这两个步骤拆成两个阶段：

1. 核函数 get_dis_kernel 计算所有查询点到目标点的大小为 $m \times n$ 的距离矩阵 dis。
2. 对距离矩阵的 m 行分别找出最小值点，并保存对应下标。

另外，get_min_kernel 核函数用共享内存查找最近邻点。每个 block 对应一个查询点的求解。在一个 block 内，其首先使用 BLOCK_DIM=1024（V100 的 block 内最大线程数）的共享内存分别存储距离数组 dis_s 和最近邻点下标数组 ind_s。注意这里单个线程的读取是跳跃的，这样有利于 block 内所有线程的访存连续性。当目标点个数 n 大于 BLOCK_DIM 时，可在线程内部先进行一次归约。在经过一次 ___syncthreads() 的块内同步后，我们可以使用树形归约求出最小值和对应下标。最后由 0 号线程写回结果。

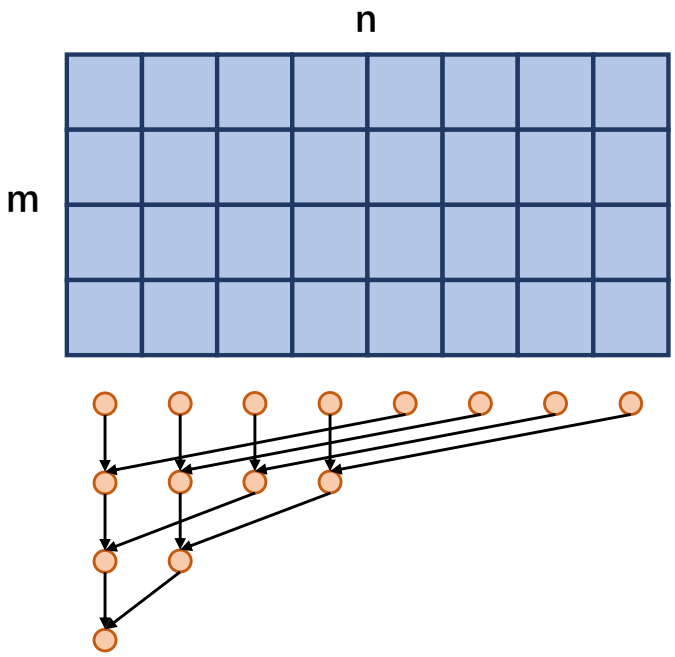

图 2: version1

```
namespace v1
{
    extern __global__ void get_dis_kernel(
        const int k,                        // 空间维度
        const int m,                        // 查询点数量
        const int n,                        // 目标点数量
        const float *__restrict__ s_points,  // 查询点集
```

```
    const float *__restrict__ r_points,  // 目标点集
    float *__restrict__ dis)             // 最近邻点集
{
    const int idn = threadIdx.x + blockIdx.x * blockDim.x;
    const int idm = threadIdx.y + blockIdx.y * blockDim.y;
    if (idn < n && idm < m) {
        float tempSum = 0;
        for (int idk = 0; idk < k; ++idk) {
            const float diff = s_points[idk + idm * k] - r_points[idk + idn * k
                ↪ ];
            tempSum += diff * diff;
        }
        dis[idn + idm * n] = tempSum;  // 计算 m*n 的距离矩阵
    }
}
template <int BLOCK_DIM>
static __global__ void get_min_kernel(  // 共享内存树形归约
    const int m,
    const int n,
    const float *__restrict__ dis,
    int *__restrict__ result)
{
    const int id = blockIdx.x * gridDim.y + blockIdx.y;
    if (id >= m)
        return;
    __shared__ float dis_s[BLOCK_DIM];
    __shared__ int ind_s[BLOCK_DIM];
    dis_s[threadIdx.x] = INFINITY;
    for (int idn = threadIdx.x + blockIdx.x * BLOCK_DIM; idn < n; idn +=
        ↪ gridDim.x * BLOCK_DIM) {
        const float tempSum = dis[idn + blockIdx.y * n];
        if (dis_s[threadIdx.x] > tempSum) {  // 赋值到共享内存
            dis_s[threadIdx.x] = tempSum;
            ind_s[threadIdx.x] = idn;
        }
    }
    __syncthreads();
    for (int offset = BLOCK_DIM >> 1; offset > 0; offset >>= 1) { // 树形归约
        if (threadIdx.x < offset)
            if (dis_s[threadIdx.x] > dis_s[threadIdx.x ^ offset]) {
                dis_s[threadIdx.x] = dis_s[threadIdx.x ^ offset];
                ind_s[threadIdx.x] = ind_s[threadIdx.x ^ offset];
            }
        __syncthreads();
    }
```

```
        if (threadIdx.x == 0)
            result[id] = ind_s[0];
    }
    extern void cudaCall(
        int k,           // 空间维度
        int m,           // 查询点数量
        int n,           // 目标点数量
        float *s_points,  // 查询点集
        float *r_points,  // 目标点集
        int **results)   // 最近邻点集
    {
        float *dis_d, *s_d, *r_d;
        int *results_d;
        CHECK(cudaMalloc((void**)&dis_d, m * n * sizeof(float)));
        CHECK(cudaMalloc((void**)&s_d, k * m * sizeof(float)));
        CHECK(cudaMalloc((void**)&r_d, k * n * sizeof(float)));
        CHECK(cudaMalloc((void**)&results_d, m * sizeof(int)));
        CHECK(cudaMemcpy((void*)s_d, (void*)s_points, k * m * sizeof(float),
            ↪ cudaMemcpyHostToDevice));
        CHECK(cudaMemcpy((void*)r_d, (void*)r_points, k * n * sizeof(float),
            ↪ cudaMemcpyHostToDevice));
        const int BLOCK_DIM_X = 32, BLOCK_DIM_Y = 32; // 设置blockSize
        get_dis_kernel<<<
            dim3(divup(n, BLOCK_DIM_X), divup(m, BLOCK_DIM_Y)),
            dim3(BLOCK_DIM_X, BLOCK_DIM_Y)>>>(k, m, n, s_d, r_d, dis_d);
        *results = (int *)malloc(sizeof(int) * m);
        const int BLOCK_DIM = 1024;  // 设置blockSize
        get_min_kernel<BLOCK_DIM><<<m, BLOCK_DIM>>> (m, n, dis_d, results_d); // 计
            ↪ 算最近邻点
        CHECK(cudaMemcpy((void**)*results, (void*)results_d, m * sizeof(int),
            ↪ cudaMemcpyDeviceToHost));
        CHECK(cudaFree(dis_d));
        CHECK(cudaFree(s_d));
        CHECK(cudaFree(r_d));
        CHECK(cudaFree(results_d));
    }
};
```

3. V2：GPU 使用 Thrust 库函数

在 V2 中我们使用了 Thrust 库用于快速开发出并行应用的原型。Thrust 与 CUDA 的互操作性有利于迭代开发策略，比如使用确定程序瓶颈，使用 CUDA C 实现特定算法并作必要优化，类似于 STL 的高级抽象。thrust::raw_pointer_cast 将设备地址转换为原始 C 指针，原始 C 指针可以调用 CUDA C API 函数，或者作为参数传递到核函数中，比如我们可以将

thrust::device_vector 转为 C 指针传入核函数中。

在找出最近邻点时，我们用一个 for 循环直接调用 thrust::min_element 找出 m 个查询点的最近邻点。cudaCall 函数是外部调用的接口。

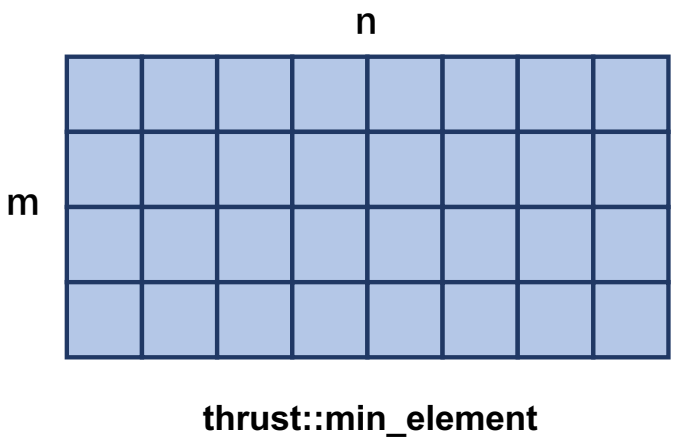

图 3: version2

```
namespace v2
{
    static __global__ void get_dis_kernel(...) {...} // 同v1
    extern void cudaCall(...) // 参数同v1
    {
        thrust::device_vector<float> dis_d(m * n);
        thrust::device_vector<float> s_d(s_points, s_points + k * m);
        thrust::device_vector<float> r_d(r_points, r_points + k * n);
        const int BLOCK_DIM_X = 32, BLOCK_DIM_Y = 32; // 设置blockSize
        get_dis_kernel<<<
            dim3(divup(n, BLOCK_DIM_X), divup(m, BLOCK_DIM_Y)),
            dim3(BLOCK_DIM_X, BLOCK_DIM_Y)>>>(
            k, m, n,
            thrust::raw_pointer_cast(s_d.data()),
            thrust::raw_pointer_cast(r_d.data()),
            thrust::raw_pointer_cast(dis_d.data()));
        *results = (int *)malloc(sizeof(int) * m);
        for (int i = 0; i < m; ++i) // 找出最近邻点
            (*results)[i] = thrust::min_element(dis_d.begin() + n * i, dis_d.begin()
                ↪ + n * i + n) - dis_d.begin() - n * i;
    }
};
```

4. V3：GPU 计算距离并同时归约

注意到 V2 中的 thrust::min_element 函数在求单个查询点时更加高效，但对大量的查询点却性能很差，所以后面我们还是使用基于 V1 的共享内存树形归约方法。在 V1 中我们用两个核函数分别计算距离矩阵和最近邻点，我们当然其实也可以在每个线程计算距离的同时先进行一次线程内归约，从而减少一次核函数启动的开销和距离矩阵的读写开销。在具体实现中，我们将之前的两个核函数合并，让 m 个 block 中的每个线程先计算对应 n 个查询点的距离，并在 n 大于 BLOCK_DIM 时进行线程内的归约，即现在各个线程内先进行一次归约。接下来使用共享内存树形归约找出 m 个查询点的最近邻点。

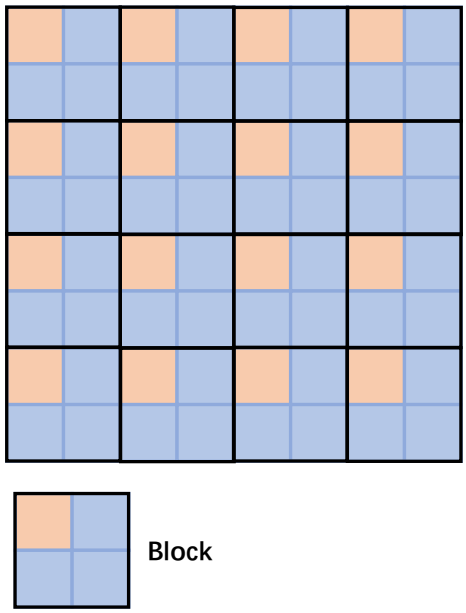

图 4: version3

```
namespace v3
{
    template <int BLOCK_DIM>
    static __global__ void cudaCallKernel(
        const int k,
        const int m,
        const int n,
        const float *__restrict__ s_points,
        const float *__restrict__ r_points,
        int *__restrict__ result)
    {
        const int id = blockIdx.x * gridDim.y + blockIdx.y;
        if (id >= m)
            return;
        __shared__ float dis_s[BLOCK_DIM];
```

```
        __shared__ int ind_s[BLOCK_DIM];
        dis_s[threadIdx.x] = INFINITY;
        for (int idm = blockIdx.y, idn = threadIdx.x + blockIdx.x * BLOCK_DIM; idn
            ↪ < n; idn += gridDim.x * BLOCK_DIM) {
            float tempSum = 0;
            for (int idk = 0; idk < k; ++idk) {  // 计算距离
                const float diff = s_points[idk + idm * k] - r_points[idk + idn * k
                    ↪ ];
                tempSum += diff * diff;
            }
            if (dis_s[threadIdx.x] > tempSum) {
                dis_s[threadIdx.x] = tempSum;
                ind_s[threadIdx.x] = idn;
            }
        }
        __syncthreads();
        for (int offset = BLOCK_DIM >> 1; offset > 0; offset >>= 1) { // 树形归约
            if (threadIdx.x < offset)
                if (dis_s[threadIdx.x] > dis_s[threadIdx.x ^ offset]) {
                    dis_s[threadIdx.x] = dis_s[threadIdx.x ^ offset];
                    ind_s[threadIdx.x] = ind_s[threadIdx.x ^ offset];
                }
            __syncthreads();
        }
        if (threadIdx.x == 0)
            result[id] = ind_s[0];
    }
    extern void cudaCall(...) // 参数同v1
    {
        ... // 显存分配与拷贝
        const int BLOCK_DIM = 1024;  // 设置blockSize
        cudaCallKernel<BLOCK_DIM><<<m, BLOCK_DIM>>> (k, m, n, s_d, r_d, results_d);
            ↪  // 计算最近邻点
        ... // 显存释放
    }
};
```

5. V4: GPU AoS2SoA

开始时的目标点集 r_points 是以一个点的 k 个维度坐标值连续存储的一维数组，这种访存方式有利于 CPU 的连续读写，但在 GPU 上会导致严重的访存不连续性（s_points 不影响，因为一个 block 只对应一个查询点），并且 k 越大问题越严重。因此，我们可以新增一个核函数 mat_inv_kernel 先将矩阵转置，也就是改为第 k 维的 n 个点连续存储。换句话说，就是将点的存储方式从 Array of Structures(AoS) 改为 Structure of Arrays(SoA)，这样就能使 GPU 上

的访存连续了。

cudaCallKernel 函数计算距离并归约求最近邻与 V3 大致相同，只是在目标点的读取上从 r_points[idk + idn * k] 改为 r_points[idk * n + idn]。

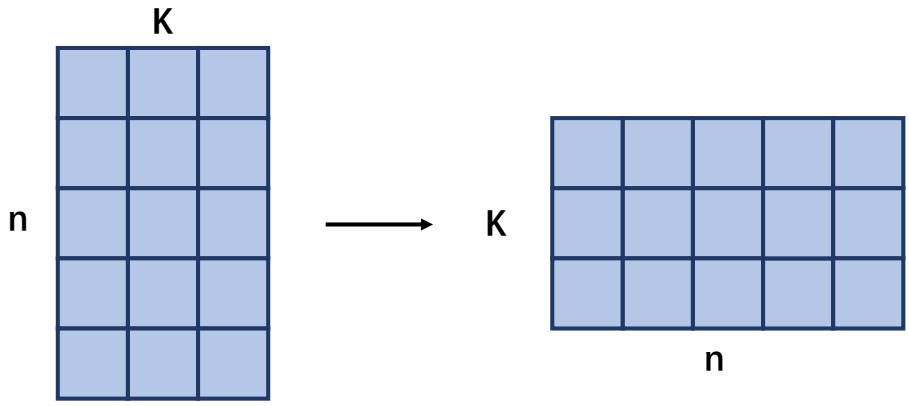

图 5: version4

```
namespace v4 {
    static __global__ void mat_inv_kernel( // 转置矩阵
        const int k,
        const int n,
        const float *__restrict__ input,
        float *__restrict__ output) {
        const int idn = threadIdx.x + blockIdx.x * blockDim.x;
        const int idk = threadIdx.y + blockIdx.y * blockDim.y;
        if (idn < n && idk < k) {
            const float a = input[idn * k + idk];
            output[idn + idk * n] = a;
        }
    }
    template <int BLOCK_DIM>
    static __global__ void cudaCallKernel(...) // 参数同v3
   {
        ... // 同v3
        for (int idm = blockIdx.y, idn = threadIdx.x + blockIdx.x * BLOCK_DIM; idn
            ↪ < n; idn += gridDim.x * BLOCK_DIM) {
            float tempSum = 0;
            for (int idk = 0; idk < k; ++idk) {
                const float diff = s_points[idk + idm * k] - r_points[idk * n + idn
                    ↪ ]; // r_points访存方式改变!
                tempSum += diff * diff;
            }
            if (dis_s[threadIdx.x] > tempSum) {
                dis_s[threadIdx.x] = tempSum;
```

```
                ind_s[threadIdx.x] = idn;
            }
        }
        ... // 同v3
    }
    extern void cudaCall(...)
    {
        ... // 显存分配与拷贝
        const int BLOCK_DIM_X = 32, BLOCK_DIM_Y = 32;
        mat_inv_kernel<<<
            dim3(divup(n, BLOCK_DIM_X), divup(k, BLOCK_DIM_Y)),
            dim3(BLOCK_DIM_X, BLOCK_DIM_Y)>>>(k, n, r_d, rr_d); // AoS 2 SoA
        const int BLOCK_DIM = 1024;
        cudaCallKernel<BLOCK_DIM><<<m, BLOCK_DIM>>> (k, m, n, s_d, rr_d, results_d)
            ↪ ; // 计算最近邻点
        ... // 释放显存
    }
};
```

6. V5：GPU 使用纹理内存存储目标点集

V5 使用 2D 纹理内存存储目标点集，纹理内存适用于线程束访问内存的模式具有空间局部性，从而可以避免一次额外的转置。V5 中的 cudaCallKernel 与 V3 大致相同，只是目标点集 r_points 用纹理对象 texObj 代替。

在外部接口 cudaCall 函数中，由于 2D 纹理内存的最大宽度不能超过 65536，所以当 n 大于 65536 时我们直接调用 V4 的优化版本。使用 2D 纹理内存时，我们将纹理对象绑定到 cudaArray（目标点集）上，并设置为只读模式。

```
namespace v5 {
    template <int BLOCK_DIM>
    static __global__ void cudaCallKernel( // 计算距离并归约
        const int k,
        const int m,
        const int n,
        const int result_size,
        const float *__restrict__ s_points,
        cudaTextureObject_t texObj, //使用纹理对象
        int *__restrict__ result) {
        ... // 同v3
        for (int idm = blockIdx.y, idn = threadIdx.x + blockIdx.x * BLOCK_DIM; idn
            ↪ < n; idn += gridDim.x * BLOCK_DIM) {
            float tempSum = 0;
            for (int idk = 0; idk < k; ++idk) {
                const float diff = s_points[idk + idm * k] - tex2D<float>(texObj,
```

```
                ↪ idk, idn); // 使用纹理对象
            tempSum += diff * diff;
        }
        if (dis_s[threadIdx.x] > tempSum) {
            dis_s[threadIdx.x] = tempSum;
            ind_s[threadIdx.x] = idn;
        }
    }
    ... // 同v3
}
extern void cudaCall(...)
{
    if (n > 65536) { // 纹理内存最大限制
        v4::cudaCall(k, m, n, s_points, r_points, results);
        return;
    }
    cudaArray *cuArray;
    cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc<float>();
    CHECK(cudaMallocArray(&cuArray, &channelDesc, k, n));
    CHECK(cudaMemcpy2DToArray(cuArray, 0, 0, r_points, sizeof(float) * k,
        ↪ sizeof(float) * k, n, cudaMemcpyHostToDevice));

    // 绑定纹理到cudaArray上
    struct cudaResourceDesc resDesc;
    memset(&resDesc, 0, sizeof(resDesc));
    resDesc.resType = cudaResourceTypeArray;
    resDesc.res.array.array = cuArray;

    // 设置纹理为只读
    struct cudaTextureDesc texDesc;
    memset(&texDesc, 0, sizeof(texDesc));
    texDesc.readMode = cudaReadModeElementType;

    // 创建纹理对象
    cudaTextureObject_t texObj = 0;
    CHECK(cudaCreateTextureObject(&texObj, &resDesc, &texDesc, NULL));

    ... // 显存分配与拷贝

    const int BLOCK_DIM = 1024;
    cudaCallKernel<BLOCK_DIM><<<m, BLOCK_DIM>>> (k, m, n, s_d, texObj,
        ↪ results_d); // 计算最近邻点

    // 销毁纹理对象
    CHECK(cudaDestroyTextureObject(texObj));
```

```
        ... // 释放显存
    }
};
```

7. V6：GPU 使用常量内存存储查询点集

V6 在 V4 的基础上使用常量内存优化对查询点的访问，由于每个线程块对应一个查询点，所以用常量内存基于 block id 的广播访问访问非常适合。首先我们声明 64k 的常量内存（最大限制），用于存储查询点集。mat_inv_kernel 函数与 V4 版本相同，用于将目标点集第 k 维的 n 个点连续存储，实现 GPU 上的访存连续。

cudaCall 函数也与 V4 大致相同，只是在计算点之间的距离时用 const_mem 查询点集与转置的目标点集进行计算。另外，在外部接口 cudaCall 中，我们用 cudaMemcpyToSymbol 将查询点集拷贝到常量内存。如果查询点集的大小 $k \times m$ 大于 16834，则会直接调用 V4 版本的 cudaCall 函数。

```
namespace v6 {
    static __constant__ float const_mem[(64 << 10) / sizeof(float)]; // 常量内存最
        ↪ 大限制
    static __global__ void mat_inv_kernel(...) {...} // 同v4
    template <int BLOCK_DIM>
    static __global__ void cudaCallKernel(...)
    {
        ... // 同v3
        for (int idm = blockIdx.y, idn = threadIdx.x + blockIdx.x * BLOCK_DIM; idn
            ↪ < n; idn += gridDim.x * BLOCK_DIM) {
            float tempSum = 0;
            for (int idk = 0; idk < k; ++idk) {
                const float diff = const_mem[idk + idm * k] - r_points[idk * n + idn
                    ↪ ]; // 使用常量内存
                tempSum += diff * diff;
            }
            if (dis_s[threadIdx.x] > tempSum) {
                dis_s[threadIdx.x] = tempSum;
                ind_s[threadIdx.x] = idn;
            }
        }
        ... // 同v3
    }
    extern void cudaCall(...) // 同v4
    {
        if (k * m > (64 << 10) / sizeof(float)) {
            v4::cudaCall(k, m, n, s_points, r_points, results);
            return;
        }
```

```
        CHECK(cudaMemcpyToSymbol(const_mem, s_points, sizeof(float) * k * m)); //
            ↪ 拷贝查询点集到常量内存
        ...    // 同v4
    }
};
```

8. V7：GPU 使用多个 Block 计算单个查询点

由于之前的几个版本仅用一个 block 计算单个查询点的最近邻，当查询点非常少时，启动的 block 也非常少，这将导致很多 SM 空置。V7 中的 cudaCallKernel 核函数和 mat_inv_kernel 核函数与 V4 相同，其在 V4 的基础上，对每个查询点用多个 block 进行查询。cudaCallKernel 核函数启动的 block 数量从 cudaOccupancyMaxActiveBlocksPerMultiprocessor 函数获得，从而保证启动时能够使 SM 满载。当然，mat_inv_kernel 核函数本身就运行很快，所以不需要同时启动大量 block 保证 SM 满载。

对于单个查询点使用多个 block 会计算得到多个结果，这里多个结果拷贝回 CPU 后再次归约。由于实际上需要再次归约的目标点非常少，因此只需在 CPU 上用 for 循环遍历找到最近邻点即可，不用使用 GPU 进行归约从而减少额外的开销。

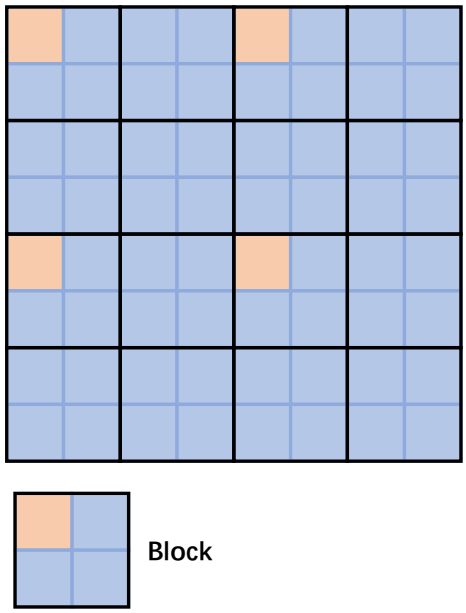

图 6: version7

```
namespace v7 {
    static __global__ void mat_inv_kernel(...) {...} // 同v4
    template <int BLOCK_DIM>
    static __global__ void cudaCallKernel(...) {...} // 同v4
    extern void cudaCall(...)
    {
```

```
thrust::device_vector<float> s_d(s_points, s_points + k * m);
thrust::device_vector<float> r_d(r_points, r_points + k * n);
thrust::device_vector<float> rr_d(k * n);
const int BLOCK_DIM_X = 32, BLOCK_DIM_Y = 32;
mat_inv_kernel<<<
    dim3(divup(n, BLOCK_DIM_X), divup(k, BLOCK_DIM_Y)),
    dim3(BLOCK_DIM_X, BLOCK_DIM_Y)>>>(
    k, n,
    thrust::raw_pointer_cast(r_d.data()),
    thrust::raw_pointer_cast(rr_d.data()));

const int BLOCK_DIM = 1024;
int numBlocks;
CHECK(cudaOccupancyMaxActiveBlocksPerMultiprocessor( // 获取启动的block数量
    &numBlocks,
    cudaCallKernel<BLOCK_DIM>,
    BLOCK_DIM,
    0));
thrust::device_vector<int> results_d(m * divup(numBlocks, m));

cudaCallKernel<BLOCK_DIM><<<dim3(results_d.size() / m, m), BLOCK_DIM>>>(
    k, m, n,
    results_d.size(),
    thrust::raw_pointer_cast(s_d.data()),
    thrust::raw_pointer_cast(rr_d.data()),
    thrust::raw_pointer_cast(results_d.data()));

*results = (int *)malloc(sizeof(int) * m);
if (results_d.size() == m) {
    thrust::copy(results_d.begin(), results_d.end(), *results);
    return;
}
thrust::host_vector<int> results_tmp(results_d);
for (int idm = 0; idm < m; ++idm) { // CPU端归约查找最近邻点
    float minSum = INFINITY;
    int index = 0;
    for (int i = 0; i < results_tmp.size(); i += m) {
        const int idn = results_tmp[i];
        float tempSum = 0;
        for (int idk = 0; idk < k; ++idk) {
            const float diff = s_points[k * idm + idk] - r_points[k * idn +
                ↪ idk];
            tempSum += diff * diff;
        }
        if (minSum > tempSum) {
```

```
                minSum = tempSum;
                index = idn;
            }
        }
        (*results)[idm] = index;
    }
}
};
```

9. V8：GPU 多卡归约目标点集

V8 在 V7 的基础上将目标点集均分到多张显卡上，这样每个显卡执行对目标点集一个子集的求解，然后在对每张显卡的结果再进行一次归约。我们首先用 cudaGetDeviceCount 函数获得显卡数，接下来使用 OpenMP 实现多卡并行。在为每张显卡分配一定的目标点后，经过 mat_inv_kernel 核函数和 cudaCallKernel 核函数，我们使用 OpenMP 中的临界区将多卡结果合并到一个 result_tmp 向量中，这里注意要将每张显卡上分配的目标点 index 转为全局 index。最后再在 CPU 上进行一次归约，得到 m 个查询点的最近邻点。

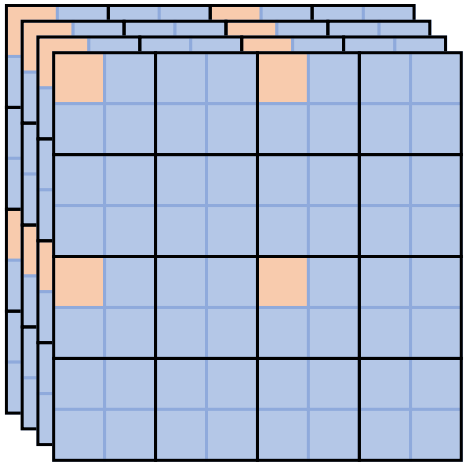

图 7: version8

```
namespace v8
{
    static __global__ void mat_inv_kernel(...) {...} // 同v4
    template <int BLOCK_DIM>
    static __global__ void cudaCallKernel(...) {...} // 同v4
    extern void cudaCall(...)
    {
        thrust::host_vector<int> results_tmp; // 多卡归约后的result
        int num_gpus = 0;
        CHECK(cudaGetDeviceCount(&num_gpus)); // 获得显卡数
        #pragma omp parallel num_threads(num_gpus)  // 多卡并行
```

```
    {
        int thread_num = omp_get_thread_num(); // 显卡id
        int thread_n = divup(n, num_gpus); // 每个显卡分配的目标点个数
        float *thread_r_points = r_points + thread_num * thread_n * k; // 为每张
            ↪ 显卡分配定量的目标点集
        if (thread_num == num_gpus - 1) {
            thread_n = n - thread_num * thread_n;
            if (thread_n == 0) {
                thread_n = 1;
                thread_r_points -= k;
            }
        }
        CHECK(cudaSetDevice(thread_num)); // 选择对应的显卡
        thrust::device_vector<float> s_d(s_points, s_points + k * m);
        thrust::device_vector<float> r_d(thread_r_points, thread_r_points + k *
            ↪ thread_n);
        thrust::device_vector<float> rr_d(k * thread_n);
        const int BLOCK_DIM_X = 32, BLOCK_DIM_Y = 32;

        mat_inv_kernel<<< // 转置目标点集
            dim3(divup(thread_n, BLOCK_DIM_X), divup(k, BLOCK_DIM_Y)),
            dim3(BLOCK_DIM_X, BLOCK_DIM_Y)>>>(
            k, thread_n,
            thrust::raw_pointer_cast(r_d.data()),
            thrust::raw_pointer_cast(rr_d.data()));

        const int BLOCK_DIM = 1024;
        int numBlocks;
        CHECK(cudaOccupancyMaxActiveBlocksPerMultiprocessor( // 获取启动的block
            ↪ 数量
            &numBlocks,
            cudaCallKernel<BLOCK_DIM>,
            BLOCK_DIM,
            0));
        thrust::device_vector<int> results_d(m * divup(numBlocks, m));
        cudaCallKernel<BLOCK_DIM><<<dim3(results_d.size() / m, m), BLOCK_DIM>>>(
            k, m, thread_n,
            results_d.size(),
            thrust::raw_pointer_cast(s_d.data()),
            thrust::raw_pointer_cast(rr_d.data()),
            thrust::raw_pointer_cast(results_d.data()));

        int my_beg, my_end;
        #pragma omp critical  // 临界区将多卡结果合并
        {
```

```
            my_beg = results_tmp.size();
            results_tmp.insert(results_tmp.end(), results_d.begin(), results_d.
                ↪ end());
            my_end = results_tmp.size();
        }
        #pragma omp barrier  // 多卡同步
        for (int offset = (thread_r_points - r_points) / k; my_beg < my_end; ++
            ↪ my_beg)
            results_tmp[my_beg] += offset; // 将每张卡上分配的目标点index转为
                ↪ 全局index
    }
    ... // CPU端归约查找最近邻点（同v7）
  }
};
```

10. V9：GPU 多卡归约 + 完全循环展开

注意到我们的每个测试样例规模都比较大，因此可以事先确定迭代次数，将共享内存树形归约中的 for 循环完全展开。我们只需更改 cudaCallKernel 函数中的归约部分代码，其他部分与 V8 完全一致。

```
namespace v9
{
  static __global__ void mat_inv_kernel(...) {...} // 同v4
  template <int BLOCK_DIM>
  static __global__ void cudaCallKernel(...)
  {
    ... // 同v4
    if (threadIdx.x < 512)
      if (dis_s[threadIdx.x] > dis_s[threadIdx.x ^ 512]) {
        dis_s[threadIdx.x] = dis_s[threadIdx.x ^ 512];
        ind_s[threadIdx.x] = ind_s[threadIdx.x ^ 512];
      }
    __syncthreads();
    if (threadIdx.x < 256)
      if (dis_s[threadIdx.x] > dis_s[threadIdx.x ^ 256]) {
        dis_s[threadIdx.x] = dis_s[threadIdx.x ^ 256];
        ind_s[threadIdx.x] = ind_s[threadIdx.x ^ 256];
      }
    __syncthreads();
    if (threadIdx.x < 128)
      if (dis_s[threadIdx.x] > dis_s[threadIdx.x ^ 128]) {
        dis_s[threadIdx.x] = dis_s[threadIdx.x ^ 128];
        ind_s[threadIdx.x] = ind_s[threadIdx.x ^ 128];
      }
```

```
        __syncthreads();
        if (threadIdx.x < 64)
            if (dis_s[threadIdx.x] > dis_s[threadIdx.x ^ 64]) {
                dis_s[threadIdx.x] = dis_s[threadIdx.x ^ 64];
                ind_s[threadIdx.x] = ind_s[threadIdx.x ^ 64];
            }
        __syncthreads();
        if (threadIdx.x < 32)
            if (dis_s[threadIdx.x] > dis_s[threadIdx.x ^ 32]) {
                dis_s[threadIdx.x] = dis_s[threadIdx.x ^ 32];
                ind_s[threadIdx.x] = ind_s[threadIdx.x ^ 32];
            }
        if (threadIdx.x < 16)
            if (dis_s[threadIdx.x] > dis_s[threadIdx.x ^ 16]) {
                dis_s[threadIdx.x] = dis_s[threadIdx.x ^ 16];
                ind_s[threadIdx.x] = ind_s[threadIdx.x ^ 16];
            }
        if (threadIdx.x < 8)
            if (dis_s[threadIdx.x] > dis_s[threadIdx.x ^ 8]) {
                dis_s[threadIdx.x] = dis_s[threadIdx.x ^ 8];
                ind_s[threadIdx.x] = ind_s[threadIdx.x ^ 8];
            }
        if (threadIdx.x < 4)
            if (dis_s[threadIdx.x] > dis_s[threadIdx.x ^ 4]) {
                dis_s[threadIdx.x] = dis_s[threadIdx.x ^ 4];
                ind_s[threadIdx.x] = ind_s[threadIdx.x ^ 4];
            }
        if (threadIdx.x < 2)
            if (dis_s[threadIdx.x] > dis_s[threadIdx.x ^ 2]) {
                dis_s[threadIdx.x] = dis_s[threadIdx.x ^ 2];
                ind_s[threadIdx.x] = ind_s[threadIdx.x ^ 2];
            }
        if (threadIdx.x == 0)
            result[id] = dis_s[0] > dis_s[1] ? ind_s[1] : ind_s[0];
    }
    extern void cudaCall(...) {...} // 同v8
};
```

11. V10：CPU 使用 KD-Tree 查找

从这里开始，向源代码引入 KD-tree 以加快的查找速度，用 KD-tree 的树形策略替代 $V0 \sim V9$ 线性搜索策略。

建立 KD-tree

建立 KD-tree 的过程如下，整个过程递归实现：

1. 所有目标点作为 KD-tree 的全体查找集合
2. 检查所有目标点的 k 个维度，选择其中方差最大的维度作为划分目标（假定为 d_i ）。
3. 以 d_i 为目标，所有目标点从小到大排序，选择排在中间（假定为 x_i ）的作为树根。在 d_i 这个维度比 x_i 小的是 KD-tree 的左子树，其他是右子树。
4. 当前子树是否还能继续划分，是跳 2 ，否则当前位置跳出构建树过程。

因为直接移动目标点的开销过大，所有过程通过指针实现。建树过程调整一个一维数组，数组存放这个目标点的下标。

```
void build(thrust::host_vector<int>::iterator beg, thrust::host_vector<int>::
    ↪ iterator end, int rt = 1)
{
    if (beg >= end)
        return;
    float sa_max = -INFINITY;
    // 这个循环找到当前目标点子集方差最大的维度
    for (int idk = 0; idk < k; ++idk)
    {
        float sum = 0, sa = 0;
        for (thrust::host_vector<int>::iterator it = beg; it != end; ++it)
        {
            float val = r_points[(*it) * k + idk];
            sum += val, sa += val * val;
        }
        sa = (sa - sum * sum / (end - beg)) / (end - beg);
        if (sa_max < sa)
            sa_max = sa, dim[rt] = idk;
    }
    // 排序找到目标维度中间点
    thrust::host_vector<int>::iterator mid = beg + (end - beg) / 2;
    std::nth_element(beg, mid, end, DimCmp{dim[rt]});
    p[rt] = *mid;
    // 递归实现左子树和右子树构建
    build(beg, mid, rt << 1);
    build(++mid, end, rt << 1 | 1);
}
```

查询过程

查询是一个自底向上的过程，对查询点，按照 KD-tree 分类到叶子节点，计算查询点和叶子节点上的目标点的欧式距离（假设为 d ）。如果 d 比查询点到另一个子树的距离还远，那就需要在另一个子树内查询。

1. 计算 KD-tree 节点和查询点的欧式距离

2. 根据 KD-tree 的划分策略，查找右子树（左子树）
3. 如果到 KD-tree 的划分边界比当前计算的最近欧式距离短，那么还需要进入对应的子树继续查找

```
thrust::pair<float, int> ask(int x, thrust::pair<float, int> ans = {INFI
NITY, 0}, int rt = 1)
{
    if (dim[rt] < 0)
        return ans;
    float d = s_points[x * k + dim[rt]] - r_points[p[rt] * k + dim[rt]], tmp = 0;
    // 这里统计和 KD-tree 节点的位置
    for (int idk = 0; idk < k; ++idk)
    {
        float diff = s_points[x * k + idk] - r_points[p[rt] * k + idk];
        tmp += diff * diff;
    }
    int w = d > 0;
    ans = ask(x, min(ans, {tmp, p[rt]}), (rt << 1) ^ w);
    if (ans.first > d * d - 1e-6)
        ans = ask(x, ans, (rt << 1) ^ w ^ 1);
    return ans;
}
```

外部接口

通过上面两个步骤，得到对应的 KD-tree ，在主要函数上只需要调用查询过程即可。

```
extern void cudaCall(
    int k,          // 空间维度
    int m,          // 查询点数量
    int n,          // 目标点数量
    float *s_points, // 查询点集
    float *r_points, // 目标点集
    int **results) // 最近邻点集，结果集合
{
    if (k > 16)
        return v0::cudaCall(k, m, n, s_points, r_points, results);
    v9::k = k;
    v9::s_points = s_points;
    v9::r_points = r_points;
    KDTreeCPU kd(n);
    *results = (int *)malloc(sizeof(int) * m);
    for (int i = 0; i < m; ++i)
    // 在这一行找到解
        (*results)[i] = kd.ask(i).second;
}
```

12. V11：GPU 使用 KD-Tree 查找

挪用 $V11$ 版本的 KD-tree 过程，使用 CUDA 实现并行查询。在 CPU 上创建好 KD-tree，然后拷贝到 GPU 内存中，在 GPU 上进行查找操作。CPU 版本的 ask 函数核函数，稍微调整适应 GPU 的访问方式即可。

```
__device__ thrust::pair<float, int> ask_device(
    float *s_d,
    float *r_d,
    int *dim,
    int *p,
    int k,
    int x,
    thrust::pair<float, int> ans = {INFINITY, 0},
    int rt = 1)
{
    int dimrt = dim[rt];
    if (dimrt < 0)
        return ans;
    int prt = p[rt];
    if (prt < 0)
        return ans;
    float d = s_d[x * k + dimrt] - r_d[prt * k + dimrt], tmp = 0;
    for (int kInd = 0; kInd < k; ++kInd)
    {
        float diff = s_d[x * k + kInd] - r_d[prt * k + kInd];
        tmp += diff * diff;
    }
    int w = d > 0;
    ans = ask_device(s_d, r_d, dim, p, k, x, thrust::min(ans, {tmp, prt}), (rt <<
        ↪ 1) ^ w);
    if (ans.first > d * d - 1e-6)
        ans = ask_device(s_d, r_d, dim, p, k, x, ans, (rt << 1) ^ w ^ 1);
    return ans;
}
__global__ void range_ask_kernel(
    float *s_d,
    float *r_d,
    int *dim,
    int *p,
    int k,
    int m,
    int *results)
{
    int global_id = blockIdx.x * blockDim.x + threadIdx.x;
```

```
    if (global_id >= m)
        return;
    results[global_id] = ask_device(s_d, r_d, dim, p, k, global_id).second;
}

struct DimCmp
{
    int dim;
    bool operator()(int lhs, int rhs) const
    {
            return r_points[lhs * k + dim] < r_points[rhs * k + dim];
        }
    };
    void range_ask(int m, int *results)
    {
        thrust::device_vector<int> results_d(m);
        int minGridSize, blockSize;
        CHECK(cudaOccupancyMaxPotentialBlockSize(
            &minGridSize,
            &blockSize,
            range_ask_kernel));
        range_ask_kernel<<<
            divup(m, blockSize),
            blockSize>>>(
            thrust::raw_pointer_cast(s_d.data()),
            thrust::raw_pointer_cast(r_d.data()),
            thrust::raw_pointer_cast(dim_d.data()),
            thrust::raw_pointer_cast(p_d.data()),
            k,
            m,
            thrust::raw_pointer_cast(results_d.data()));
        thrust::copy(results_d.begin(), results_d.end(), results);
    }
```

13. V12：CPU 使用 Octree 查找

Octree 受制于算法本身的性质，只允许使用在 $k = 3$ 的测试样例，对 3 维的数据可以建树，因此对其它维度的测试样例，将会调用 V0-baseline 版本运行。

建立 Octree

建立 Octree 的过程如下，整个过程递归实现

1. 对需要建树的数据集，先检查三个维度的最大值和最小值，根据三个维度的最大值和最小值得到当前树根中心点位置和相应的半径（或者说正方体的边长 1/2）
2. 建立树根的 8 棵子树（八角树名称由来），遍历当前需要建树的数据集，依次判断这个数

据和树根中心的各个维度的大小关系，放入对应的子树

3. 检查每一个子树包括的数据点数量，小于等于 1 的子树是叶子节点；否则递归调用建立子树（回到 1）

因为直接移动目标点的开销过大，所有过程通过指针实现。建树过程调整一个一维数组，数组存放这个目标点的下标。对一个数据点分配到对应的子树，这里引入条件赋值，加入样本点 a 在 x 维度比树根中心点大，给 pos 赋值 0 否则赋值 1，y 维度赋值 0 否则 2，z 维度赋值 0 否则 4，这样使用 3 个条件赋值语句可以确定一个样本点属于哪个子树。之后查询也会使用到这个设定。

```
Node build(std::vector<int>::iterator beg, std::vector<int>::iterator end, int
    ↪ depth)
{
    if (beg >= end)
        return Node(depth);

    float x_min = INFINITY, x_max = -x_min, y_min = INFINITY, y_max = -y_min, z_min
        ↪  = INFINITY, z_max = -z_min;
    // 找到当前点集三个维度最大和最小值
    for (std::vector<int>::iterator i = beg; i != end; i++)
    {
        float *point = &r_points[(*i)];
        x_min = std::min(x_min, point[0]);
        x_max = std::max(x_max, point[0]);
        y_min = std::min(y_min, point[1]);
        y_max = std::max(y_max, point[1]);
        z_min = std::min(z_min, point[2]);
        z_max = std::max(z_max, point[2]);
    }
    float r = std::max((x_max - x_min) / 2, std::max((y_max - y_min) / 2, (z_max -
        ↪ z_min) / 2));
    Node root(depth);
    root.setC((x_min + x_max) / 2, (y_max + y_min) / 2, (z_max + z_min) / 2, r);
    root.subtree.resize(8, Node(root.depth + 1)); // 建立 8 课子树
    for (std::vector<int>::iterator i = beg; i != end; i++)
    {
        float *point = &r_points[(*i)];
        // 这里引入或运算，使用条件赋值语句。并得到数据点合适的子树位置
        int pos = (point[0] > root.x_c) ? 0 : 1; // 如果x维度比中心点大取 0 否则 1
        pos |= (point[1] > root.y_c) ? 0 : 2; // 如果y维度比中心点大取 0 否则 2
        pos |= (point[2] > root.z_c) ? 0 : 4; // 如果z维度比中心点大取 0 否则 4
        root.subtree[pos].incl.push_back((*i));
    }
    root.incl.clear();
```

```
    for (int i = 0; i < 8; i++)
    {
        if (root.subtree[i].depth > 9 || root.subtree[i].incl.size() <= 1)
        {
            root.subtree[i].pos = -1; // 划分为 叶子节点
        }
        else
        {
            root.subtree[i] = build(root.subtree[i].incl.begin(), root.subtree[i].
                ↪ incl.end(), depth + 1);
            root.subtree[i].incl.clear();
            root.subtree[i].pos = i;
        }
    }
    return root;
}
```

查询过程

查询是一个自底向上的过程，对查询点，按照 Octree 分类到叶子节点，计算查询点和叶子节点上的目标点的欧式距离（假设为 d ）。如果 d 比查询点到另一个子树的距离还远，那就需要在另一个子树内查询。

上面提到过建立子树，分配子树位置时，x 维度对应 1，y 维度对应 2，z 维度对应 4，在对应的位置使用异或运算，就可以找到当前子树在这个维度上的兄弟节点。比如我要找子树 7 在 z 维度上的兄弟节点，只需要和 4 进行异或运算得到 3 ，子树 3 就是子树 7 在 z 维度上的兄弟节点，同理其他，减少大量判断语句。

检查到兄弟节点的位置，判断当前最小距离（假设是 r）是否与兄弟节点所在位置相交，也就是判断在这个维度上一测试点为中心 r 为半径的球与兄弟节点所在的正方体是否有交点。存在交点就需要进入这棵子树查询，否则忽略这棵子树。

```
std::pair<float, int> ask(Node &root, float *s_point, std::pair<float, int> ans =
    ↪ {INFINITY, 0})
{
    if (root.pos == -1 && root.incl.size() == 0)
        return ans; // 空节点
    std::pair<float, int> localAns = ans;
    if (root.incl.size() == 0)
    {
        // 非叶子节点
        std::pair<float, int> tmp(INFINITY, 0);
        int pos = (s_point[0] > root.x_c) ? 0 : 1;
        pos |= (s_point[1] > root.y_c) ? 0 : 2;
        pos |= (s_point[2] > root.z_c) ? 0 : 4;
```

```
        tmp = ask(root.subtree[pos], s_point, localAns);  // 按照树查询下去
        localAns = tmp.first > localAns.first ? localAns : tmp; // 取小的一项
        Node *rt = &(root.subtree[pos ^ 4]);
        //使用 异或 运算替换大量判断语句，这里查找在 z 维度的兄弟节点
        if (localAns.first > std::min(std::abs(s_point[2] - rt->z_c - rt->radius),
            ↪ std::abs(s_point[2] - rt->z_c + rt->radius)))
        {
            tmp = ask(*rt, s_point, localAns);
            localAns = tmp.first > localAns.first ? localAns : tmp;
        }
        rt = &(root.subtree[pos ^ 2]);
        // y 维度的兄弟节点
        if (localAns.first > std::min(std::abs(s_point[1] - rt->y_c - rt->radius),
            ↪ std::abs(s_point[1] - rt->y_c + rt->radius)))
        {
            tmp = ask(*rt, s_point, localAns);
            localAns = tmp.first > localAns.first ? localAns : tmp;
        }
        rt = &(root.subtree[pos ^ 1]);
        // x 维度的兄弟节点
        if (localAns.first > std::min(std::abs(s_point[0] - rt->x_c - rt->radius),
            ↪ std::abs(s_point[0] - rt->x_c + rt->radius)))
        {
            tmp = ask(*rt, s_point, localAns);
            localAns = tmp.first > localAns.first ? localAns : tmp;
        }
    }

    // 非空的叶子节点
    for (std::vector<int>::iterator i = root.incl.begin(); i != root.incl.end(); i
        ↪ ++)
    {
        float *r_point = &r_points[(*i)];
        float dis = std::pow((r_point[0] - s_point[0]), 2);
        dis += std::pow((r_point[1] - s_point[1]), 2);
        dis += std::pow((r_point[2] - s_point[2]), 2);
        if (dis < localAns.first)
        {
            localAns.first = dis;
            localAns.second = *i;
        }
    }

    return localAns;
}
```

14. V13：GPU 使用 Octree 查找

挪用 $V12$ 版本的 Octree 过程，使用 CUDA 实现并行查询。在 CPU 上创建好 Octree，然后拷贝到 GPU 内存中，在 GPU 上进行查找操作。CPU 版本的 ask 函数核函数，稍微调整适应 GPU 的访问方式即可。

```
__device__ thrust::pair<float, int> ask_device(
    Node root = Node(0),
    float *s_point = nullptr,
    thrust::pair<float, int> ans = {INFINITY, 0},
    float *r_points = nullptr,
    int rt = 1)
{
    if (root.pos == -1 && root.incl.size() == 0)
        return ans; // 空节点
    thrust::pair<float, int> localAns = ans;
    if (root.incl.size() == 0)
    {
        // 非叶子节点
        thrust::pair<float, int> tmp(INFINITY, 0);
        int pos = (s_point[0] > root.x_c) ? 0 : 1;
        pos |= (s_point[1] > root.y_c) ? 0 : 2;
        pos |= (s_point[2] > root.z_c) ? 0 : 4;
        tmp = ask_device(root.subtree[pos], s_point, localAns); // 按照树查询下去
        localAns = tmp.first > localAns.first ? localAns : tmp; // 取小的一项
        Node *rt = &(root.subtree[pos ^ 4]);
        if (localAns.first > thrust::min(std::abs(s_point[2] - rt->z_c - rt->radius
            ↪ ), std::abs(s_point[2] - rt->z_c + rt->radius)))
        {
            tmp = ask_device(*rt, s_point, localAns);
            localAns = tmp.first > localAns.first ? localAns : tmp;
        }
        rt = &(root.subtree[pos ^ 2]);
        if (localAns.first > thrust::min(std::abs(s_point[1] - rt->y_c - rt->radius
            ↪ ), std::abs(s_point[1] - rt->y_c + rt->radius)))
        {
            tmp = ask_device(*rt, s_point, localAns);
            localAns = tmp.first > localAns.first ? localAns : tmp;
        }
        rt = &(root.subtree[pos ^ 1]);
        if (localAns.first > thrust::min(std::abs(s_point[0] - rt->x_c - rt->radius
            ↪ ), std::abs(s_point[0] - rt->x_c + rt->radius)))
        {
            tmp = ask_device(*rt, s_point, localAns);
            localAns = tmp.first > localAns.first ? localAns : tmp;
```

```
        }
    }
    // 非空的叶子节点

    for (thrust::host_vector<int>::iterator i = root.incl.begin(); i != root.incl.
        ↪ end(); i++)
    {
        float *r_point = &r_points[(*i)];
        float dis = std::pow((r_point[0] - s_point[0]), 2);
        dis += std::pow((r_point[1] - s_point[1]), 2);
        dis += std::pow((r_point[2] - s_point[2]), 2);
        if (dis < localAns.first)
        {
            localAns.first = dis;
            localAns.second = *i;
        }
    }

    return localAns;
}
__global__ void range_ask_kernel(
    Node root = Node(0),
    float *s_point = nullptr,
    float *r_points = nullptr,
    int m = 0,
    int *results = nullptr)
{
    int global_id = blockIdx.x * blockDim.x + threadIdx.x;
    if (global_id > m)
    {
        return;
    }
    thrust::pair<float, int> ans = {INFINITY, 0};
    results[global_id] = ask_device(root, s_point, ans, r_points);
}
```

四、实验结果

1. 实验环境

CPU	Intel(R) Xeon(R) Gold 6242 CPU @ 2.80GHz
GPU	NVIDIA Tesla V100
CUDA	Cuda compilation tools, release 10.2, V10.2.89

表 1: 软硬件环境

2. 编译运行

```
$ nvcc -Xcompiler -fopenmp -arch=sm_70 main.cu -o main
$ ./main
```

3. 测试数据

由于过小的测试数据会受到系统噪声的影响,我们设置了 4 组较大的测试,用于方便测试线性查找的加速比。5 和 6 的小样例用于测试对比各个方法的性能（算例过大 KD-Tree 和 Octree 建树极其耗时）。如表2所示，k 代表空间维度，m 代表查询点数量，n 代表目标点数量。

id	k	m	n
1	3	1	16777216
2	16	1	16777216
3	3	1024	1048576
4	16	1024	1048576
5	3	1024	65536
6	16	1024	65536

表 2: 测试数据

4. 性能分析

(i) 线性查找

设置不同参数，每组参数测试 5 次取均值。为了公平比较，包含了显存拷贝的开销。计算结果如下（单位 ms）:

id	V0	V1	V2	V3	V4	V5	V6	V7	V8	V9
1	217.659	71.519	68.312	77.431	158.297	114.476	103.508	73.057	50.997	38.612
2	992.546	441.212	318.814	433.462	347.600	530.248	359.411	334.903	195.009	180.473
3	12248.817	137.139	444.331	135.974	95.110	109.187	95.472	30.037	15.060	12.787
4	56173.148	333.739	533.652	244.009	173.665	174.367	107.556	93.974	31.884	27.767

表 3: 线性查找测试结果

对上述结果做可视化，横轴对应各个版本，纵轴对应时间，不同颜色的折线代表 4 组不同的测试。由于 V0 是 CPU 上的串行版本，部分测试时间特别久，超过了纵轴范围，就没有显示。

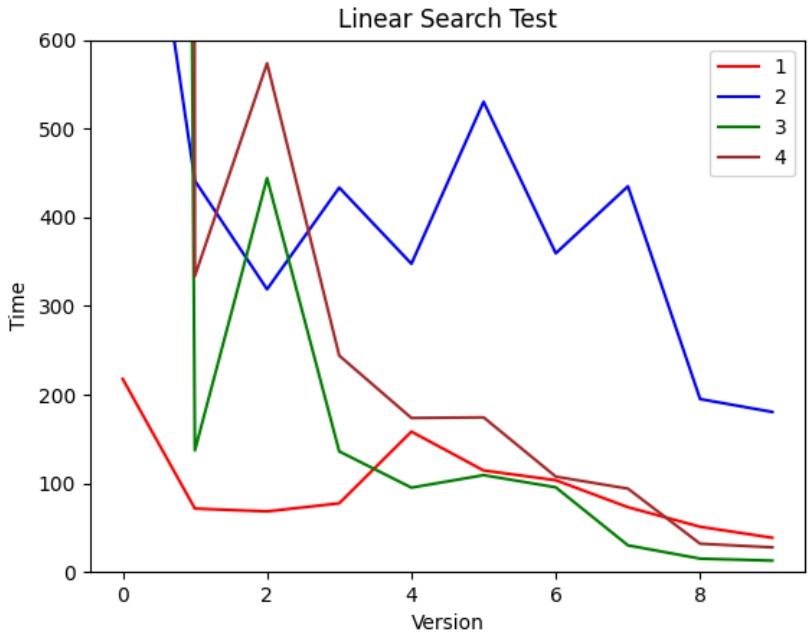

图 8: 线性查找测试时间（单位：ms）

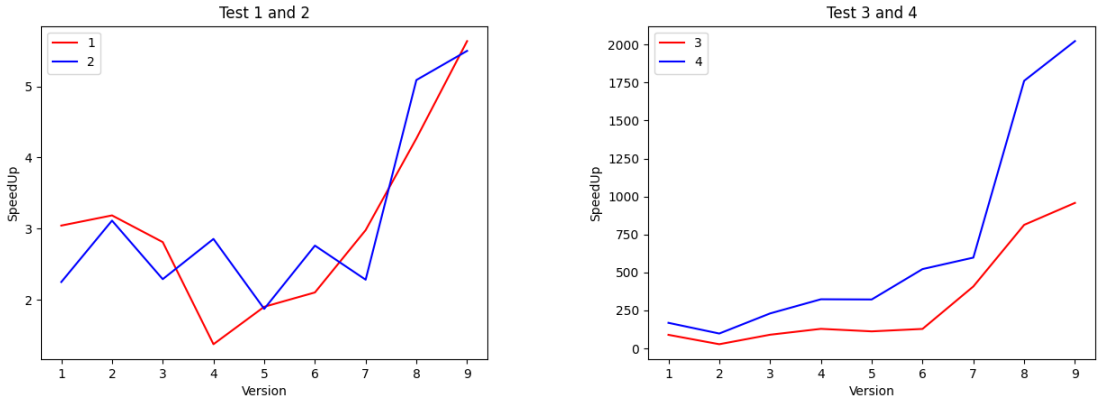

图 9: 线性查找加速比

从如上的实验结果可以看出，无论哪一组测试数据，V9 的结果都是最优的，在 Test4 的数据集上加速比甚至达到了惊人的 2023！在多个查询点中，每个优化版本的加速比都很高，然而单个查询点的加速效果并不理想，这可能是因为多个查询点本身适合并行的特性，各个查询之间并不影响。从各个优化版本的角度分析，我们可以得出一些结论：

- V1 先计算距离矩阵再用共享内存归约

  在单个查询点时，由于 V2 只启动了一个 block 进行归约，过多的目标点会先在同一线程内进行一次归约，所以会导致很多 SM 空置，性能不是特别好。V1 的性能可以作为 GPU 的基线性能，供后几组目标。

- V2 使用 Thrust 库函数

  同样先计算距离矩阵，但使用 thrust::min_element 查找最近邻点。可以发现，在一个查询点时，V2 的速度相较于其它单卡的优化版本来说是最快的。但是当查询点个数达到 1024 时，V2 的结果是 GPU 中最差的。因为 thrust::min_element 本身是线程不安全的特性，直接强制使用 OpenMP 并行会发生报错。

- V3 计算距离并同时归约

  相较于 V1，V3 的性能还是有所提升的。这也是因为将两个核函数合并后，减少了核函数启动的开销和部分显存拷贝的开销。

- V4 AoS2SoA

  可以发现，在 Test2，Test3，Test4 上 V4 相比 V3 性能都有所提升。然而在 Test1 时 V4 的结果并不理想，这可能是因为只启动了一个 block，并且本身改变目标点集访存方式的核函数也有一定开销。

- V5 使用纹理内存存储目标点集

  除了在 Test1 的数据集上 V5 性能优于 V4，其它结果都不如 V4。这说明尽管纹理内存具

有更优的内存访问局部性，但仍然不如直接优化 CUDA 访存的全局内存。

- V6 使用常量内存存储查询点集

  使用常量内存后，其性能相比 V4 还是有略微提升的。使用常量内存我们基于 block 可以有效地进行广播访问，无需串行访问。

- V7 使用多个 block 处理单个查询点

  使用多个 block 后，在单个查询点对 SM 的利用率更高，所以在几组测试上性能相比之前都提升了不少。

- V8 多卡归约目标点集

  分散了每个查询点的对应目标点，在各个数据集上的性能都大大提升。当然，这也是充分利用了 4 张 V100 和每张卡的 SM 的结果，性能的提升也显而易见。

- V9 多卡归约 + 完全循环展开

  将共享内存用于归约的 for 循环完全展开后，V9 的性能相比 V8 依然提升了一些。V9 同时也是性能最优的版本。

(ii) 综合对比

我们将线性查找，Octree，KD-Tree 的结果进行对比。为公平比较，我们取线性查找 CPU 串行版本和 GPU 单卡最快版本 V7 进行对比。另外，由于 Octree 和 KD-Tree 需要大量的建树时间，因此我们也应该将建树和查询的时间分开来对比。我们用 Test5 和 Test6 相对较小的算例（大算例建树极其耗时）进行测试，结果如表4所示（Octree 只适用 3 维测试样例中）：

Time / Version	Test5(k=3)		Test6(k=16)	
	Search Time	Total Time	Search Time	Total Time
V0(Linear Search CPU)	174.035	174.035	731.452	731.452
V7(Linear Search GPU)	1.117	1.117	3.442	3.442
V10(KD-Tree CPU)	1.239	19.351	2612.250	2658.916
V11(KD-Tree GPU)	0.443	18.373	24.877	60.398
V12(Octree CPU)	20.439	140.368	-	-
V13(Octree GPU)	0.978	121.373	-	-

表 4: 综合对比（单位：ms）

可以发现，在维数比较低的时候，无论是在 CPU 上还是 GPU 上 KD-Tree 和 Octree 都能有效减少查询时间，且 KD-Tree 的效果比 Octree 要更好。然而在高维空间中，KD-Tree 的空间划分显得十分不实用，由于大部分的点都会被查询，且难以触发剪枝条件，从而导致其效率低下。尽管 Octree 只能受限于 3 维空间中，但其高维拓展的 KD-Tree 在过高的维度空间中的表现也可以说是“维数灾难”了。

五、总结

本次实验我们从最近邻搜索的基础算法线性查找出发，分别实现了 1 个 CPU 版本和 9 个 GPU 版本，包括共享内存归约、Thrust 库函数的使用、纹理内存常量内存的使用、多卡归约等。另外我们实现了 KD-Tree 的 1 个 CPU 版本和 1 个 GPU 版本，Octree 的 1 个 CPU 版本和 1 个 GPU 版本。我们发现在维数较低时，使用 KD-Tree 和 Octree 相比于线性查找拥有更好的性能，但在高维空间中，KD-Tree 过于繁琐而不实用，使得整体查询时间反而不如暴力的线性查找。

总的来说，这次实验加深了我们对 CUDA 异构计算的理解，并运用到了多种课内讲述的 CUDA 优化方法，同时增加了利用不同类型 CUDA 内存甚至多卡归约进行优化的经验，受益匪浅。感谢老师和助教一学期的辛勤付出，希望在将来我们还有机会在 CUDA 的并行计算领域继续探索遨游！

一、附录

1. 分工

施天予

1. 线性查找多种优化实现
2. 性能分析
3. 实验报告撰写

王郅成

1. 线性查找基础版本实现
2. 框架规划
3. 展示视频和图片制作

王筝

1. KD-tree 的 CPU 和 GPU 版本实现
2. Octree 的 CPU 和 GPU 版本实现
3. 数据结构部分的报告、实验和展示内容制作

2. 个人感想

施天予

在本次大作业中，我主要完成了最近邻搜索中的线性查找的 GPU 优化版本的实现。我从最开始的共享内存树形归约，到尝试 thrust 库函数进行归约，再到优化 GPU 方寸读写，使用纹理内存、常量内存，到最后多卡归约，完全展开循环等，可以说用到了老师课件中的各种优

化方法。另外，我也分析比较了线性查找版本的各个性能，进行对比分析，总结出不同 CUDA 版本优化的优势和特点。

王郅成

在本次作业中, 我主要实现了最近邻搜索的 CPU 串行版本和最基础的 GPU 版本, 此外我负责了展示 PPT 以及视频的制作以及各个优化版本演示图片的绘制. 另外我统筹了实验推进的安排, 完善了实验的架构, 规划了展示的框体架构. 在本次实验中, 我们配合良好, 实验推进比较顺利, 完成了预期的实验项目.

王筝

在本次作业中, 我主要实现最近邻搜索中利用数据结构优化查找速度的部分, 包括 KD-Tree 和 Octree 的 CPU 版本和 GPU 版本实现。实现建树到结构复制到 GPU 进行查询优化。另外，分析比较了适用数据结构优化查询的方法和 baseline 的线性查找方法之间的性能差距，总结得到这些数据结构在特定的查询条件下可以加速查询过程，在更加一般的情况下，这些树结构的查询速度和 baseline 相比并没有显著的差异。