多核程序设计

作业二：计算高维空间中的最近邻

数据科学与计算机学院　17大数据与人工智能

17341015　陈鸿峥

目录

一、 题目描述

计算高维空间中的最近邻(nearest neighbor)

- 输入：查询点集，参考点集，空间维度k
- 输出：对每个查询点，输出参考点集中最近邻的序号

回答以下问题：

1. 介绍程序整体逻辑，包含的函数，每个函数完成的内容。（10分）
    - 对于核函数，应该说明每个线程块及每个线程所分配的任务
2. 解释程序中涉及哪些类型的存储器（如全局内存、共享内存等），并通过分析数据的访存模式及该存储器的特性说明为何使用该种存储器。（15分）
3. 针对查询点集为1个点及1024个点给出两个版本，并说明设计逻辑（例如任务分配方式）的异同。如两个版本使用同一逻辑，需说明原因。（35分）
4. 请给出一个基础版本（baseline）及至少一个优化版本。并分析说明每种优化对性能的影响。（40分）
    - 具体得分根据优化难度（技巧及工作量）而定
5. 选做：使用空间划分数据结构加速查询（如KD-Tree，BVH-Tree）。（20分）

二、 问题简答

1. 参见第3节，每个线程所做的任务均用加粗字体标注。
2. 参见第3节GPU共享内存、Warp原语版本等，不同类型的存储器均使用不同颜色字体进行标注。
3. 参见第3节，除了GPU Block归约版本，其他GPU优化版本对于查询点集为1个点和1024个点的任务分配方式均相同，但具体每个线程做的任务有所不同，可见下文加粗字体。基本的分配策略都是开设m个Block给m个查询点使用，而每个Block内的每个线程则用于计算每个参考点到该查询点的距离。这样的好处在于查询点的坐标可以直接存储在共享内存中，供所有参考点使用；并且所有的并行归约操作都限制在一个Block内，可以直接通过共享内存传输数据，而无需通过全局内存，大大降低了访存的开销。
4. 参见第3节，一共实现了五个版本的GPU程序进行优化与比较，实验结果见第5节。
5. 参见第4节，利用KD树加速查询。

三、 具体实现

本次实验我总共实现了七个版本的程序，包括CPU版本、GPU基线版本、GPU共享内存版本、GPU并行归约版本、GPU Warp原语版本、GPU Block归约版本、KD树加速优化的版

本。

1. CPU版本

最基础的实现即助教提供的CPU版本程序，依次遍历所有的查询点，对于每一个查询点，遍历所有的参考点，并（遍历每个维度）计算每个查询点和参考点之间的距离。从这些距离中选取最小的，其下标即为所求的最近邻。

```
/*!
 * Naive CPU implementation
 *   used to test the correctness of the results
 * \param k The dimension size of the points
 * \param m The nubmer of search points
 * \param n The number of reference points
 * \param searchPoints
 * \param referencePoints
 * \param results
 * \return void. Results will be put in result.
 */
extern void cudaCallbackCPU(int k, int m, int n, float *searchPoints,
                            float *referencePoints, int **results) {

    int *tmp = (int*)malloc(sizeof(int)*m);
    int minIndex;
    float minSquareSum, diff, squareSum;

    // Iterate over all search points
    for (int mInd = 0; mInd < m; mInd++) {
        minSquareSum = -1;
        // Iterate over all reference points
        for (int nInd = 0; nInd < n; nInd++) {
            squareSum = 0;
            for (int kInd = 0; kInd < k; kInd++) {
                diff = searchPoints[k*mInd+kInd] - referencePoints[k*nInd+kInd];
                squareSum += (diff * diff);
            }
            if (minSquareSum < 0 || squareSum < minSquareSum) {
                minSquareSum = squareSum;
                minIndex = nInd;
            }
        }
        tmp[mInd] = minIndex;
    }

    *results = tmp;
    // Note that you don't have to free searchPoints, referencePoints, and
```

```
    // *results by yourself
}
```

2. GPU基线版本

GPU基线(baseline)版本即将CPU版本的最外层循环进行并行。 **每个线程处理一个查询点**，同样是遍历所有的参考点并计算查询点与参考点之间的距离。每个线程都各自计算出所有参考点的最小距离，并将结果直接写入全局内存中。核函数如下所示。

```
/*!
 * Core execution part of CUDA
 *   that calculates the nearest neighbor of each search point.
 * \param k The dimension size of the points
 * \param m The nubmer of search points
 * \param n The number of reference points
 * \param searchPoints
 * \param referencePoints
 * \param output
 * \return void. Results will be put in output
 */
__global__ void kernel(int k, int m, int n, float* searchPoints, float*
    ↪ referencePoints, int* output) {
    int pid = threadIdx.x;
    int minIdx;
    float minSquareSum = -1;
    float diff, squareSum;
    // Iterate over all reference points
    if (pid < m) {
        for (int nInd = 0; nInd < n; nInd++) { // ref points
            squareSum = 0;
            for (int kInd = 0; kInd < k; kInd++) { // dimension
                diff = searchPoints[k * pid + kInd]
                     - referencePoints[k * nInd + kInd];
                squareSum += (diff * diff);
            }
            if (minSquareSum < 0 || squareSum < minSquareSum) {
                minSquareSum = squareSum;
                minIdx = nInd;
            }
        }
        output[pid] = minIdx;
    }
}
```

GPU调用的接口如下所示，主要分为几个步骤：

1. 为查询点坐标、参考点坐标、输出位置分配GPU内存
2. 将查询点坐标、参考点坐标从主机端(CPU)传到设备端(GPU)
3. 调用核函数，可以看到开设了1个Block，并且有m个线程。
4. 将结果从GPU运回CPU
5. 释放GPU内存

```
/*!
 * Wrapper of the CUDA kernel
 *   used to be called in the main function
 * \param k The dimension size of the points
 * \param m The nubmer of search points
 * \param n The number of reference points
 * \param searchPoints
 * \param referencePoints
 * \param results
 * \return void. Results will be put in result.
 */
extern void cudaCallbackGPU_baseline(int k, int m, int n, float *searchPoints,
                                     float *referencePoints, int **results) {
    float *searchPoints_d, *referencePoints_d;
    int* output_d;

    // Allocate device memory and copy data from host to device
    CHECK(cudaMalloc((void **)&searchPoints_d, sizeof(float)*m*k));
    CHECK(cudaMalloc((void **)&referencePoints_d, sizeof(float)*n*k));
    CHECK(cudaMalloc((void **)&output_d, sizeof(int)*m));
    CHECK(cudaMemcpy(searchPoints_d, searchPoints, sizeof(float)*m*k,
        ↪ cudaMemcpyHostToDevice));
    CHECK(cudaMemcpy(referencePoints_d, referencePoints, sizeof(float)*n*k,
        ↪ cudaMemcpyHostToDevice));

    // Invoke the device function
    kernel<<< 1, m >>>(k, m, n, searchPoints_d, referencePoints_d, output_d);
    cudaDeviceSynchronize();

    // Copy back the results and de-allocate the device memory
    *results = (int *)malloc(sizeof(int)*m);
    assert(results != NULL);
    CHECK(cudaMemcpy(*results, output_d, sizeof(int)*m, cudaMemcpyDeviceToHost));

    // int *cpu_results;
    // cudaCallbackCPU(k, m, n, searchPoints, referencePoints, &cpu_results);
    // for (int i = 0; i < m; ++i)
    //     assert(cpu_results[i] == (*results)[i]);
```

```
37
38      CHECK(cudaFree(searchPoints_d));
39      CHECK(cudaFree(referencePoints_d));
40      CHECK(cudaFree(output_d));
41  }
```

注释中的部分最开始用于核验CPU版本的程序与GPU版本程序的正确性。

3. GPU共享内存版本

注意到由于每次计算距离时都会涉及到**查询点集的大量内存访问**，因此一个优化思路即将查询点集放置在共享内存中；至于参考点集则仍放置在全局内存中，因为每个点坐标只会被访问一次，将其读取至共享内存也不会减少访问时间。

因此这里**每个Block计算一个查询点的最近邻，每个Thread分别计算q个参考点与查询点的距离**。由于使用的GPU每个Block的最大线程数为1024，故$q = \lceil m/1024 \rceil$。具体流程如下：

- 每个Block先开取k个线程，读取查询点的k维坐标向量到共享内存`s_mem`中
- 创建共享内存`dist`和`dist_idx`分别存储该Block内的最小距离值及其下标。注意到每个线程处理q个参考点，因此`dist`和`dist_idx`中的元素对应的也是这q个点的最小距离值及其下标（由于同个Block中的所有线程处理的是同个查询点，因此距离数组无需在Block之间传递，而可以采用共享内存方式实现）。
- 使用单一线程进行归约操作，对`dist`数组进行遍历，寻找其中的最小值
- 每个Block将最终结果（距离最小值对应坐标）写入全局数组`output`中

完整核函数代码如下所示。

```
1  __global__ void kernel_sharedmem(int k, int m, int n, float* searchPoints, float*
       ↪ referencePoints, int* output) {
2      int bid = blockIdx.x; // each block, one search point
3      int tid = threadIdx.x;
4      int n_points = ((n % 1024 == 0) ? n / 1024 : n / 1024 + 1);
5      int searchId = bid;
6      int referenceId = tid; // [tid,tid+n_points]
7      int minIdx;
8      float diff, squareSum;
9      __shared__ float s_mem[16];
10     if (tid < k) {
11         s_mem[tid] = searchPoints[k * searchId + referenceId];
12     }
13     __syncthreads();
14     __shared__ float dist[1024];
15     __shared__ int dist_idx[1024];
```

```
    float minSquareSum = -1;
    for (int i = 0; i < n_points; ++i) {
        squareSum = 0;
        int refId = referenceId * n_points + i;
        for (int kInd = 0; kInd < k; kInd++) { // dimension
            diff = s_mem[kInd] - referencePoints[k * refId + kInd];
            squareSum += (diff * diff);
        }
        if (minSquareSum < 0 || squareSum < minSquareSum) {
            minSquareSum = squareSum;
            minIdx = refId;
        }
    }
    dist[referenceId] = minSquareSum;
    dist_idx[referenceId] = minIdx;
    __syncthreads();
    if (referenceId == 0) {
        float minSquareSum = -1;
        int bound = min(n,1024);
        for (int i = 0; i < bound; ++i) {
            squareSum = dist[i];
            if (minSquareSum < 0 || squareSum < minSquareSum) {
                minSquareSum = squareSum;
                minIdx = dist_idx[i];
            }
        }
        output[searchId] = minIdx;
    }
}
```

4. GPU并行归约版本

在上述共享内存版本的基础上，添加并行归约的功能。需要在1024个元素中找最小值，这个可以采用树形的方式进行归约，如图1所示。

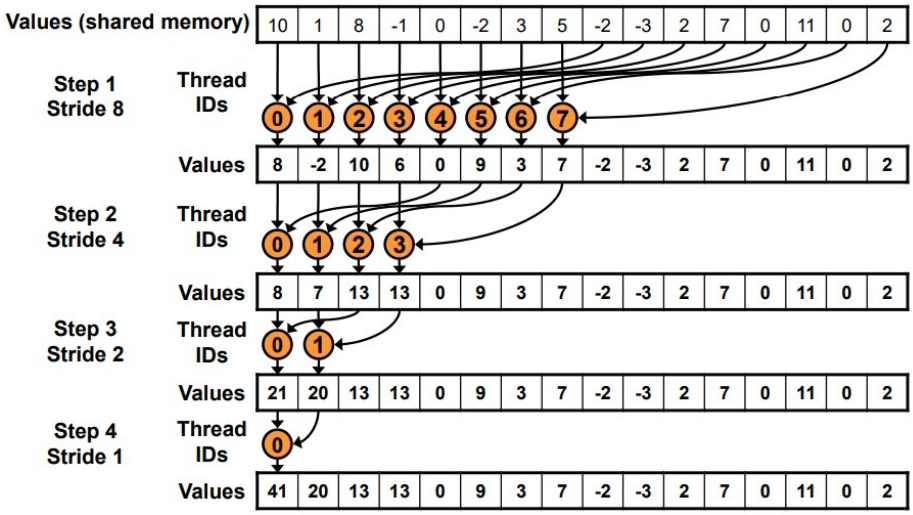

图 1: 树形方式归约

注意这里读取的都是连续内存，通过加步长来消除存储体冲突。每个线程计算两个元素间的最小值，最终1024个元素的最小值会被存放在数组第一个元素中。由于全程的操作都在共享内存中完成，因此速度非常快。最后只有一次全局内存的写操作，即将最终结果写入output中。

核函数并行归约部分代码如下所示。

```
// parallel reduction
// only support power of 2 at this time
for (int stride = 512; stride > 0; stride >>= 1) {
    if (n > stride && tid < stride) {
        if (dist[tid] > dist[tid + stride]) {
            dist[tid] = dist[tid + stride];
            dist_idx[tid] = dist_idx[tid + stride];
        }
    }
    __syncthreads();
}
if (tid == 0)
    output[searchId] = dist_idx[0];
```

为确保参考点数目小于1024时也能够正常执行，这里的判断条件还添加了对n与步长关系的判断。不过这个程序一个小缺陷在于只能支持2的指数次幂的归约。

5. GPU Warp原语版本

这里同样是对归约部分进行优化，由于采用树形归约方式，当线程数少于32时，就只剩第一个线程束(warp)还在工作了。因此这时候可以直接采用线程束内的并行机制，在课件中可以通过展开最后一个线程束而无需添加__syncthreads来实现并行归约求和。但是对于求最小值

这种涉及到判断语句的归约，则没有办法简单通过上述方法实现自动同步，否则会导致结果运算错误。

故解决方案是采用warp层面的原语操作，如图2所示，CUDA 9引入了几个新的原语，使得并行归约操作变得更加高效便捷。对于一个线程束内的32个线程，__shfl_down_sync可以实现线程之间的数据传输直接通过寄存器进行。通过指定掩码、传输变量及偏移量，即可在线程束内实现快速高效的数据交换，进而达成归约的目的。

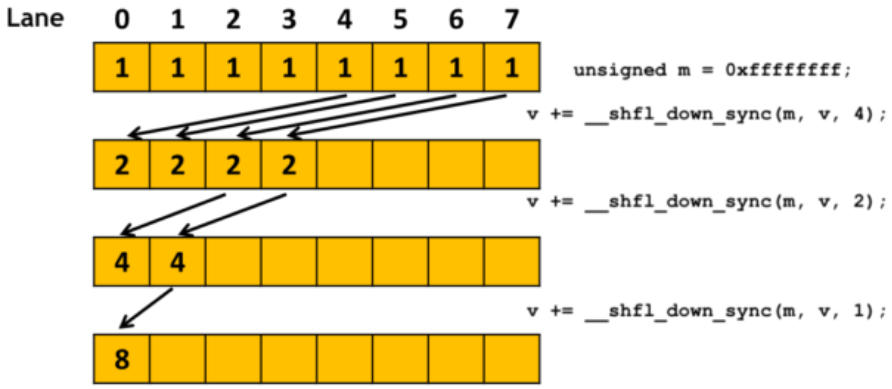

图 2: __shfl_down_sync原语操作

具体到本问题来说，我将最后一个线程束进行包裹，实现了warpReduceMin函数，同样是树形归约，但直接将高位线程数据搬移至低位线程求最小值，而不用再计算对应数组地址。

```
__inline__ __device__ void warpReduceMin(float& val, int& idx)
{
    for (int stride = warpSize / 2; stride > 0; stride >>= 1) {
        float tmpVal = __shfl_down_sync(FULL_MASK, val, stride);
        int tmpIdx = __shfl_down_sync(FULL_MASK, idx, stride);
        if (tmpVal < val) {
            val = tmpVal;
            idx = tmpIdx;
        }
    }
}
```

注意到这里无需显式同步，而且数据直接通过寄存器交换，速度也会大幅提升。

核函数部分修改为下面代码。

```
    // parallel reduction
    // only support power of 2 at this time
    for (int stride = 512; stride >= 32; stride >>= 1) {
        if (n > stride && tid < stride) {
            if (dist[tid] > dist[tid + stride]) {
                dist[tid] = dist[tid + stride];
```

```
            dist_idx[tid] = dist_idx[tid + stride];
        }
    }
    __syncthreads();
}
if (tid < 32)
    warpReduceMin(dist[tid], dist_idx[tid]);
if (tid == 0)
    output[searchId] = dist_idx[0];
```

6. GPU Block归约版本

有了warp内的归约求最值，更进一步，可以将一个block内的所有warp都按照这种方式实现，这样将大幅减少共享内存的访问，同时也减少线程之间的同步。

故有下列blockReduceMin函数，如果一个block内有1024个线程，那么将有32个wrap。对每个wrap先调用warpReduceMin求出每个wrap的最小值，然后再对这32个最小值求全局最小。最后0号线程的结果即为全局最小值。

```
__inline__ __device__ void blockReduceMin(float& val, int& idx)
{
    static __shared__ float values[32];
    static __shared__ int indices[32];
    int lane = threadIdx.x % warpSize;
    int wid = threadIdx.x / warpSize;

    warpReduceMin(val, idx); // partial reduction for each warp in a block

    if (lane == 0) { // write back to shared mem
        values[wid] = val;
        indices[wid] = idx;
    }

    __syncthreads();

    // read from shared memory only if that warp existed
    if (threadIdx.x < blockDim.x / warpSize) {
        val = values[lane];
        idx = indices[lane];
    } else {
        val = INT_MAX;
        idx = 0;
    }

    if (wid == 0) {
        warpReduceMin(val, idx); // final reduce within first warp
```

```
28      }
29  }
```

注意如果参考点集小于1024个，则会回滚回Warp原语的版本，即只对最后一个线程束进行归约求最小值。

四、 KD树加速查询

KD树其实可以看作高维空间中的二叉搜索树，循环用垂直于坐标轴的超平面将k维空间进行划分。

构建KD树的过程是一个递归的过程：假设k维空间的查询点为$T = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}$，其中$\mathbf{x}_i = (x_i^{(1)}, x_i^{(2)}, \ldots, x_i^{(k)})^{\mathrm{T}}$

1. 构造根节点：选择所有样本$x^{(1)}$坐标的中位数作为切分点，将根节点对应的超矩形区域划分维两个子区域。切分由通过切分点并与坐标轴$x^{(1)}$垂直的超平面实现。由根节点生成深度为1的左、右子结点，左子结点对应坐标$x^{(1)}$小于切分点的子区域，右子结点对应于坐标$x^{(1)}$大于切分点的子区域。
2. 对深度为j的结点，选择$x^{(l)}$为切分的坐标轴，$l = j \mod k + 1$，以该结点区域中所有示例的$x^{(l)}$坐标的中位数作为切分点，将该结点对应的超矩形区域切分为两个子区域。
3. 直到两个子区域都没有样本存在时停止，从而构建得到KD树

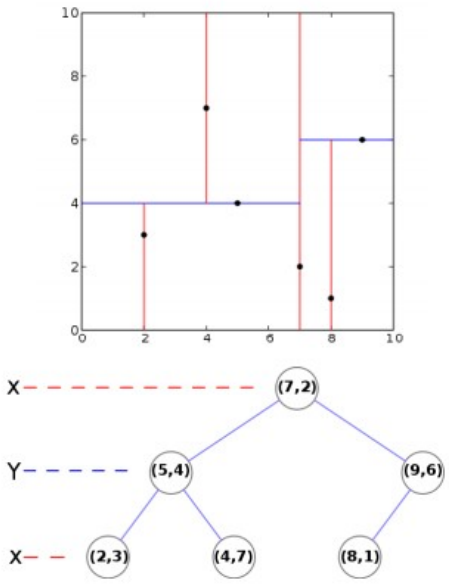

图 3: KD树示例

因此只需对每组测试样例，只需对参考点建立起一棵KD树，即可快速实现对查询点的最近邻查找。

1. 在KD树中找出包含目标点**x**的叶结点：从根结点出发，递归向下访问KD树。若目标点**x**当前维坐标小于切分点的坐标，则移动到左子结点，否则移动到右子结点，知道子结点为叶结点为止
2. 将此叶结点作为当前最近点，并递归向上回退，在每个结点执行下列操作
   (a) 若该结点保存的参考点比当前最近点距离目标点更近，则将该参考点作为当前最近点
   (b) 检查子结点的父节点的另一子结点对应区域是否有更近的点，即检查另一子结点对应的区域是否与以查询点为球心、以查询点与当前最近点间的距离为半径的超球体相交
      - 若相交，则可能在另一个子结点对应的区域内存在离目标点更近的点；移动到另一个子结点再递归进行最近邻搜索
      - 若不相交，向上回退

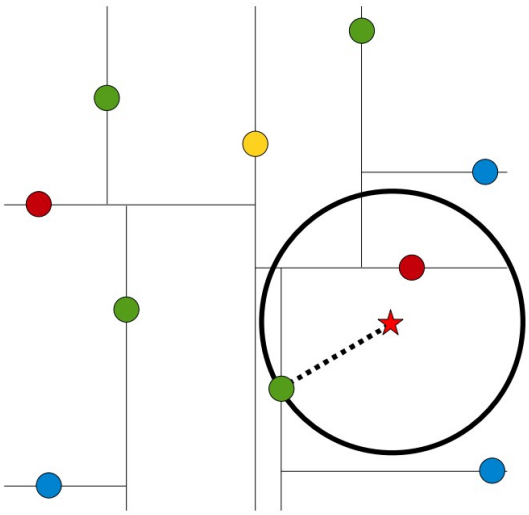

图 4: KD树搜索最近邻例子

   (c) 当回退到根结点时，搜索结束，最后的最近点即为所求

由上述算法过程知查找最近邻的时间复杂度为$O(\log n)$，远优于直接线性搜索。

具体实现上则是创建一个`KdTree`的类，并用数组存储每个结点对应的参考向量。建树和查询过程同上述算法描述。

```
1 class KdTree{
2 public:
3     KdTree() {
4         parent = leftChild = rightChild = NULL;
5     }
```

```
    bool isEmpty() {
        return root.empty();
    }
    bool isLeaf() {
        return (!root.empty()) && rightChild == NULL && leftChild == NULL;
    }
    bool isRoot() {
        return (!isEmpty()) && parent == NULL;
    }
    bool isLeft() {
        return parent->leftChild->root == root;
    }
    bool isRight() {
        return parent->rightChild->root == root;
    }
    vector<double> root;
    KdTree* parent;
    KdTree* leftChild;
    KdTree* rightChild;
};
```

较为遗憾的是KD树的实现在提交报告之时还只能在CPU上运行，如果要将其迁移至GPU上则需对数据结构进行重新组织，用Array of Structure (AoS)的方式进行存储。

五、 实验设置与结果

1. 环境设置

为了避免占用学院集群环境的大量资源，我采用了另外一台服务器进行测试。该服务器上装备了2个Intel Xeon Gold 5118处理器（24物理核/48逻辑核），另有一块Titan V GPU。操作系统为Ubuntu 18.04 LTS，编译器采用gcc 7.4.0和nvcc 10.2。同时，我也在学院集群上进行了简单测试，运行时间均与下述在服务器上的测试差异不大。

所有版本的正确性均已测试通过，下列所有实验均为运行多次取平均的结果。

2. 实验结果

实验结果如图5所示，可以看到CPU版本的程序在样例较小时表现都比较好，但是一旦查询/参考点的数目提升了，其性能就下降得很快，与GPU优化后的版本差了2个数量级。

另外，可以看到共享内存的使用给GPU的性能带来了极大的提升，至少是1个数量级的优化。 GPU Baseline第一组数据的突起是由于冷启动造成的，后面GPU程序的运行则没有这种问题。

由于本次实验提供的测试样例规模依然较小，因此采用了不同归约优化方式，性能提升在图中看并不明显。但是从完整实验数据（见附录A）中可以看出无论是并行归约，还是采

用warp原语，对GPU性能的影响依然是非常正面的。基本上每添加一种优化运行时间都会缩短一些，至于优化起作用的原因在前文中已经阐述。

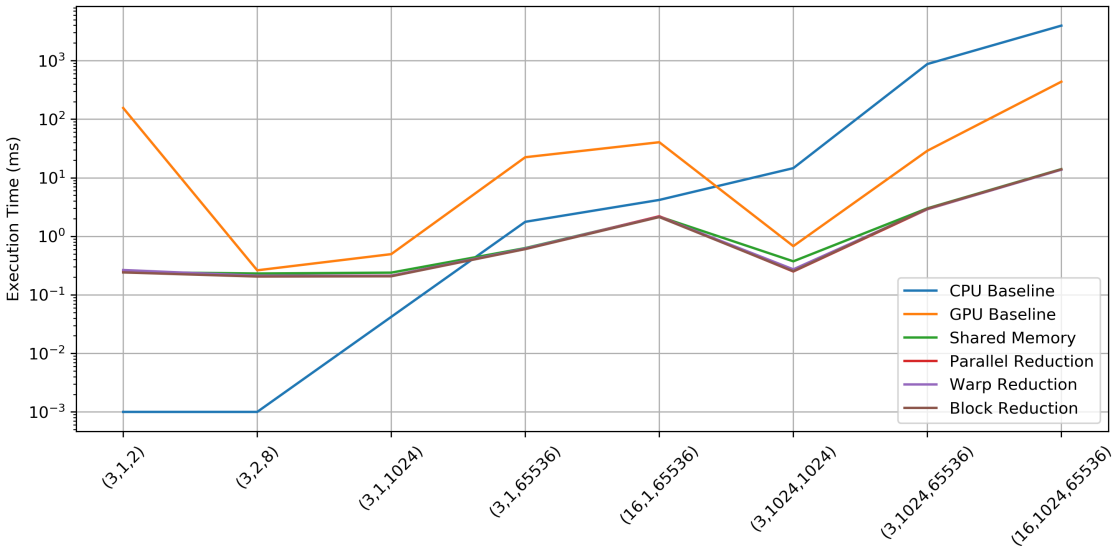

图 5: CUDA不同优化版本比较

参考文献

[1] CUDA Warp-Level Primitives, https://developer.nvidia.com/blog/using-cuda-warp-level-primitives/

[2] 李航，《统计机器学习》，第3.3节《k近邻法的实现：kd树》

[3] KD Tree， http://stanford.edu/class/archive/cs/cs106l/cs106l.1162/handouts/assignment-3-kdtree.pdf

附录 A. 完整实验数据

表 1: 不同规模数据下不同版本程序的运行时间(ms)

	(3,1,2)	(3,2,8)	(3,1,1024)	(3,1,65536)	(16,1,65536)	(3,1024,1024)	(3,1024,65536)	(16,1024,65536)
CPU基线	0.001	0.001	0.042	1.764	4.171	14.559	872.909	3955.051
GPU基线	155.364	0.262	0.495	22.362	40.349	0.678	28.838	436.137
GPU共享内存	0.245	0.231	0.239	0.628	2.168	0.373	2.997	14.024
GPU并行归约	0.253	0.212	0.212	0.608	2.196	0.250	2.891	13.815
GPU Warp原语	0.265	0.212	0.211	0.616	2.149	0.272	2.909	13.696
GPU Block归约	0.241	0.205	0.207	0.603	2.126	0.253	2.956	13.892