CUDA作业2-计算高维空间最近邻

姓名学号： 王程钥 17341146
联系方式： wangchy56@mail2.sysu.edu.cn
日期： 2020-7-29

1 简介

本次作业的任务如下

> 给定高维空间中的两个点集，一个点集为参考点集，另一个点集为询问点集。对于询问点集中的每个节点，在参考点集中找到它的最近点

基于这个任务，我尝试从存储器的使用，询问点集大小，参考点集大小，空间维度大小这几个维度进行优化，使用了基于多线程的一套方法，大大降低了程序的运行时间。最终，运行完所有测试数据用时仅为8.763ms，最慢的一组也只要4.556ms。

2 方法与优化

2.1 串行法

串行法就是简单的三重循环。最外层循环枚举询问点集，第二层枚举参考点集，第三层枚举空间维度，计算前两层枚举的询问点和参考点之间的距离。方法比较简单，不过多赘述。

该方法的时间复杂度时$O(N*M*K)$，接下来我将针对N（询问点集维度），M（参考点集维度），K（空间维度）这三个维度分别进行优化。

2.2 基于询问点集维度 (N) 的并行优化

实验数据中存在三个 $N=1024$ 的测试点，参考4.2节的表格中串行程序的运行时间可以看到，这三个测试点的耗时非常长。我们可以分析一下运行时间。测试点test_5的运行时间是测试点test_2的约850倍，而前者的N是后者的1024倍。这显然就是串行导致的性能损失。因此可以认为串行算法的性能瓶颈在 N 这个维度。

因此我们尝试对 N 这个维度进行优化。不同询问点之间的操作时独立的，所以可以直接对这个维度进行并行。创建线程的时候将 N 个测试点分配到 N 个block中，每个block创建一个线程，对一个测试点进行操作。之后的优化都是基于本算法进行的，我们设定本算法为Baseline。

另外，在本文中优化算法并不区分 $N=1$ 或 $N=1024$ ，因为根据本文的优化算法，两者之间其实只是存在单线程和多线程的差别，这种差别多开几个block，多开几个线程就可以基本忽略不计。

2.3 基于参考点集维度 (M) 的多线程优化

对于 M 这个维度的优化，可以采用多线程+共享内存的形式。在2.2节的算法中，一个询问点对 M 个参考点求最近点对的操作，其实就是枚举 M 个参考点，分别计算这个询问点和参考点的距离。不同参考点计算和询问点的距离的操作是独立的，这部分是可以并行计算的。将 M 个参考点分配给 DM 个线程，每个线程线性计算询问点和一部分参考点的距离，并记录最近点信息。

与N维度不同，对于M维度的优化还需要考虑不同线程信息的合并。参考倍增算法，以2为底倍增枚举一个stride。假设这部分有DM个线程，第i个线程处理的区间的最近点为 pos_i ，距离为 dis_i ，将这些信息存在共享内存里，这样一个块的不同线程就可以访问。对于第i个线程，如果线程 $i+stride$ 在 $[0,DM)$ 范围内就将 dis_i 和 $dis_{i+stride}$ 进行比较，并更新 pos_i 和 dis_i。具体可以参考下面的伪代码

```
for(int stride=(DM>>1); stride>0; stride>>=1) {
    if(ThreadId + stride < DM) {
        if(dis[ThreadId] > dis[ThreadId + stride]) {
            dis[ThreadId] = dis[ThreadId + stride];
            pos[ThreadId] = pos[ThreadId + stride];
        }
    }
    __syncthreads();
}
if(ThreadId==0) ans = pos[0];
```

2.4 基于空间维度 (K) 的多线程优化

对 K 这个维度的优化和对 M 的优化类似，也是采用多线程计算并合并的方法。K 维度其实做的就是一个多项求和的事情，先多线程分块求和，再二进制分解合并。不过因为本次实验数据中 K 较小，在这个维度上优化用到的线程数也不会很多，所以在求和的时候直接做线性加法，不进行倍增优化。

2.5 存储器的选择

基于baseline的算法，所有内存都采用动态全局内存。在一个块内，每个线程对一个询问向量执行操作，访问M遍存储该向量的全局内存，访问一遍存储所有参考向量的全局内存，总共需要访问全局内存 $2*M*K$ 次。对于存储询问向量的内存部分，存在M次重复访问，因此可以考虑使用共享内存将询问向量存起来，这样就只要访问 $(M+1)*K$ 次全局内存。另外如果采用2.3节和2.4节的优化，每个线程只需要将询问点信息的一部分录入到共享内存中，又可以提升程序的运行速度。

3 代码实现

3.1 Kernal函数

这一节主要介绍内核函数的实现。我会将整个kernel函数分成多个部分进行介绍。完整的代码见 `core.cu` 中的 `kernal` 函数。

内核函数传入的参数有以下几个，n, m, D 为测试数据的三个维度，$partNum$ 将M维切分成的块数，$DimNum$为将K维切分成的块数。qry_d 为询问点集，$refer\_d$ 为参考点集，$output\_d$ 为答案数组，需要返回最近点的编号。

```
__global__ void kernal(int n, int m, int D, int PartNum, int DimNum, float
*qry_d, float *refer_d, int *output_d)
```

首先根据线程编号和块编号确定该线程锁对应的询问点编号 idx，M 维被分到的块编号 $part$，K 维被分到的块编号 dim。计算 M 维的块大小 $BlkSize$，K 维的块大小 $DimSize$，并根据这两个块大小计算该线程在这两个维度所对应的区间 $[L, R]$，$[\_l, \_r]$。

```
int idx = blockIdx.x;
int ThreadId = threadIdx.x;
int part = ThreadId / DimNum;
int dim = ThreadId % DimNum;
int BlkSize = (n+PartNum-1)/PartNum;
int DimSize = (D+DimNum-1)/DimNum;
int L = BlkSize*part, R = min(L+BlkSize, n);
int _l = DimSize*dim, _r = min(_l+DimSize, D);
```

共享内存方面，因为需要开多个动态共享内存数组，而CUDA只支持传一个 $size$ 参数，所以先把所有需要的共享内存空间都传到临时数组 Str 中，后根据需要分配给三个共享内存数组 $Info, sum, Cur$。`kernal` 函数部分的代码如下。

```
extern __shared__ int Str[];
int *Share = Str;
info *Info = (info*)&Share[PartNum];
float *sum = (float*)&Share[PartNum * DimNum];
float *Cur = (float*)&Share[D];
```

其中$Info$数组存储一个$part$内的最近点编号和最近点距离，sum数组是D维度计算的中间数组，Cur数组用于存储当前询问点的相关信息。$info$类的相关定义如下。

```
struct info{
    float dis;
    int pos;
};
```

`cudaCallback` 函数部分调用 `kernal` 的代码如下。先确定三个维度的大小，创建 `BloNum` 个块，每个块有 `PartNum*DimNum` 个线程。先统计好 `kernal` 需要的三个共享内存数组所需的空间大小，放入调用函数内。

```
int BlkNum = m;
int PartNum = min(dim1, n);
int DimNum = min(dim2, (k+3)/4);
int ThreadNum = PartNum * DimNum;
int SizeInfo = sizeof(info) * PartNum *2;
int SizeSum = sizeof(float) * (PartNum * DimNum) * 2;
int SizeCur = sizeof(float) * k;
kernal<<< BlkNum, ThreadNum, SizeInfo + SizeSum + SizeCur >>>(n, m, k,
PartNum, DimNum, qry_d, refer_d, output_d);
```

随后将当前询问点的信息存入Cur数组中。因为有n个part，只在第0个part进行操作，其它part的线程等待即可。

```
if(part==0){
    for(int i=_l;i<_r;i++) Cur[i] = qry_d[idx*D+i];
}
__syncthreads();
```

接下来就是在 $[L, R]$ 区间内找最近点了。枚举这个区间的每一个参考点，分别计算这个参考点和询问点的距离。

计算方法是这样的，先计算该参考点和询问点在 $[\_l, \_r]$ 区间这几个维度上的距离，并将其存在 $sum[ThreadId]$ 中。随后，对于 $dim == 0$ 的线程，枚举$DimNum$个$part$，将这些点计算的sum值加起来。得到的和就是在该参考点和询问点的距离。

对于每个参考点都这么处理，这样就可以得到 $[L, R]$ 区间内所有参考点到询问点的距离，取最小存入 $Info[part]$ 中。

```
float ans = -1;
int pos = -1;
for(int i=L;i<R;i++)
{
```

```
    float dis = 0, diff;
    for(int j=_l;j<_r;j++) {
        diff = Cur[j] - refer_d[i*D+j];
        dis += diff * diff;
    }
    sum[ThreadId] = dis;
    __syncthreads();

    if(dim==0){
        for(int j=1;j<DimNum;j++) dis += sum[ThreadId+j];
        if(ans<0 || dis<ans) ans=dis, pos=i;
    }
}
if(!dim) Info[part] = (info){ans, pos};
__syncthreads();
```

最后我们需要将$PartNum$ 个$part$的结果合并起来。这里的合并算法在2.3节已经描述过了，不过多赘述。最后在每个$block$的0号线程，将计算得到的最近点编号写入 $output\_d$ 数组，结束操作。

```
if(!dim) {
    for(int stride=(PartNum>>1);stride>0;stride>>=1) {
        if(part+stride<PartNum){
            if(Info[part].dis > Info[part+stride].dis)
                Info[part] = Info[part+stride];
        }
        __syncthreads();
    }
}
if(!ThreadId) output_d[idx] = Info[0].pos;
```

3.2 cudaCallback函数

这个函数的构成比较简单，先开设动态的全局内存，后将输入数据传入内存，后调用 $kenral$ 函数，最后将 $kernal$ 返回的结果传入 $results$ 数组即可。相关代码如下。

```
extern void cudaCallback(int k, int m, int n, float *searchPoints,
                         float *referencePoints, int **results,
                         int dim1, int dim2) {
    float *qry_d, *refer_d;
    int *output_d;
    int size_m = m*k;
    int size_n = n*k;

    CHECK(cudaMalloc((void **)&qry_d, sizeof(float)*size_m));
    CHECK(cudaMalloc((void **)&refer_d, sizeof(float)*size_n));
    CHECK(cudaMalloc((void **)&output_d, sizeof(int)*m));
    CHECK(cudaMemcpy(qry_d, searchPoints, sizeof(float)*size_m,
cudaMemcpyHostToDevice));
    CHECK(cudaMemcpy(refer_d, referencePoints, sizeof(float)*size_n,
cudaMemcpyHostToDevice));

    int BlkNum = m;
    int PartNum = min(dim1, n);
    int DimNum = min(dim2, (k+3)/4);
    int ThreadNum = PartNum * DimNum;
    int SizeInfo = sizeof(info) * PartNum *2;
```

```
    int SizeSum = sizeof(float) * (PartNum * DimNum) * 2;
    int SizeCur = sizeof(float) * k;
    kernal<<< BlkNum, ThreadNum, SizeInfo + SizeSum + SizeCur >>>(n, m, k, PartNum, DimNum, qry_d, refer_d, output_d);

    *results = (int *)malloc(sizeof(int)*m);
    CHECK(cudaMemcpy(*results, output_d, sizeof(int)*m, cudaMemcpyDeviceToHost));
    CHECK(cudaFree(qry_d));
    CHECK(cudaFree(refer_d));
}
```

4 实验结果与分析

4.1 数据集与相关说明

本次实验提供的测试数据共有八组，其中两组为样例。八组数据的坐标点均为$[0,1]$范围内的随机小数。数据集具体的大小见下表。

数据编号	询问点集维度（N）	参考点集维度（M）	空间维度（K）
sample_0	1	2	3
sample_1	2	8	3
test_0	1	1024	3
test_1	1	65536	3
test_2	1	65536	16
test_3	1024	1024	3
test_4	1024	65536	3
test_5	1024	65536	16

本次实验的所有结果都是基于新版的样例代码进行修改，在集群上测试得到的。

4.2 并行与串行

我们首先尝试比较串行算法和并行算法在性能上的差距。并行算法只对询问点集分块，每个块只运行一个线程暴力查找。如下表可以看到两种算法差距明显。

可以看到对于 $N=1024$ 的三组数据，并行算法下的运行时间远远小于串行算法下的运行时间，甚至已经和 $N=1$ 的三组数据相近。由此可见基于Block的并行效率还是非常高的。

不过对于 $N=1$ 的三组，算法的性能则出现了明显的下降。这主要是因为对 N 并行对于这三组数据在时间复杂度上并没有优化，同时内存拷贝等操作耗费了大量的时间。

观察下表可以发现 $N=1024$ 的三组数据性能与 $N=1$ 的三组相近，说明此时性能瓶颈已经不在 N 这维上了。此时可以观察一下 test_0和test_1 以及 test_3和test_4 这两对的运行时间。 test_1的运行时间是 test_0 的50倍，test_4的运行时间是 test_3 的55倍。这两组数据的不同点主要在 M 维上，前者是后者的64倍。这与它们的性能差异基本吻合，因此可以认为此时算法的性能瓶颈在 M 维上。

方法	test_0	test_1	test_2	test_3	test_4	test_5	total
Serial	0.035	2.187	4.125	11.8805	766.255	3506.953	4291.826
Parallel	0.507	25.213	43.693	0.505	27.730	43.165	140.814

4.3 多线程优化

4.3.1 基于M维度

我们尝试枚举在参考点集维度进行优化的线程数（DM），并探究线程数与性能的关系。特别地，如果数据集中 $M < DM$ ，则设置 $DM = M$。

根据下表的数据，可以发现随着线程数的增加，模型的性能越来越好，在线程数为128时达到最佳。因为线程调度存在时间消耗，所以当线程数大于128时，模型的性能随着线程数的增加略微下降。因为CUDA一个block的线程数最多为1024，所以枚举的最大空间即1024。

我们可以观察一下当 M 维的性能瓶颈被解除之后的性能瓶颈。取 $DM = 128$ 一行为例，test_2的运行时间约是test_1的4.01倍，test_5运行时间更是test_4的5.17倍。考虑到两组数据中，K的比值都是 $16:3$，即5.33倍，这个倍数差和运行时间差基本吻合。因为我们认为此时模型的性能瓶颈主要在 K 维中。

DM	DK	test_0	test_1	test_2	test_3	test_4	test_5	total
1	1	0.507	25.213	43.693	0.505	27.730	43.165	140.814
2	1	0.231	11.579	20.634	0.263	12.823	24.514	70.043
4	1	0.144	5.965	10.775	0.151	6.142	14.571	37.747
8	1	0.108	2.938	6.178	0.109	3.397	12.566	25.297
16	1	0.077	1.646	3.918	0.094	2.394	11.719	19.849
32	1	0.070	1.027	2.746	0.096	2.202	10.771	16.912
64	1	0.055	0.688	2.338	0.087	2.123	10.838	16.128
128	1	0.059	0.531	2.131	0.079	2.143	11.068	16.010
256	1	0.048	0.481	2.053	0.076	2.419	11.748	16.824
512	1	0.057	0.449	1.981	0.084	2.421	11.702	16.695
1024	1	0.055	0.433	1.988	0.107	2.599	12.218	17.400

4.3.2 基于K维度

随后，我们又尝试在固定$DM = 128$的情况下枚举空间维度线程数（DK），探究线程数与性能的关系。另外，如果数据集中 $K/2 < DK$ ，则设置 $DK = K/2$。设置 $K/2$ 的主要原因是，如果一个线程的工作量太小（如只完成一对数字的运算），的线程调度的开销会超过多线程并行带来的收益。具体可以看下表中 $DK = 1$ 和 $DK = 2$ 时测试机test_1和test_4的表现。在DK更大的情况下，两个数据集的表现并没有更好。

如下表可以看到，随着线程数的增加，模型的性能会先变好后变差。具体原因和上一段中设置 $DK = K/2$ 的原因相同，不再赘述。根据下表，取DK=4的时候性能最好，此时对于$M = 3$的数据在K维度使用2线程，对于$M = 16$的数据在K维度使用4线程。

DM	DK	test_0	test_1	test_2	test_3	test_4	test_5	total
128	1	0.059	0.531	2.131	0.079	2.143	11.068	16.010
128	2	0.053	0.542	1.847	0.081	1.919	6.909	11.352
128	4	0.057	0.542	1.700	0.083	1.826	4.556	8.763
128	8	0.053	0.544	1.741	0.083	1.922	6.848	11.191

4.3.3 最优参数

由以上几个表格的数据可以发现，当 $DM = 128, DK = 4$ 时性能最好。我们尝试在这两个值附近枚举DM和DK，寻找最优参数。

如下表，当 $DM = 128, DK = 4$ 的时候，模型的性能最好，只要8.763ms就可以跑完所有测试数据，即使是最慢的一组也只需要4.556ms。

DM	DK	test_0	test_1	test_2	test_3	test_4	test_5	total
64	4	0.055	0.735	1.951	0.085	1.713	4.708	9.247
64	8	0.060	0.730	1.994	0.086	1.729	6.747	11.345
128	4	0.057	0.542	1.700	0.083	1.826	4.556	8.763
256	4	0.059	0.467	1.590	0.087	1.977	4.689	8.870

4.4 存储优化

以上实验对于询问点，都将其先拷贝到共享内存。下面做对照实验，尝试直接访问全局内存，并比较性能。这里我分别基于baseline算法和上一节调出的最优参数，对是否进行存储优化进行控制变量的对照试验。可以看到在做了存储优化后性能还是有一定提升的。使用了存储优化后我们得到了更优的性能。

DM*DK	UseShare	test_0	test_1	test_2	test_3	test_4	test_5	total
1*1	N	0.466	27.141	42.153	0.529	29.898	46.076	146.262
1*1	Y	0.507	25.213	43.693	0.505	27.730	43.165	140.814
128*4	N	0.073	0.556	1.710	0.086	1.865	4.803	9.093
128*4	Y	0.057	0.542	1.700	0.083	1.826	4.556	8.763

5 总结

本次实验我尝试基于多线程从N，M，K三个维度对算法进行优化，同时还对模型的存储方式进行了优化。通过各种优化，最终算法在测试集上运行的总时间可以从140.814ms降低到8.763ms，最大测试集的运行时间也从43.165ms降低到4.556ms。由此可见善用CUDA的多线程真的能很好地提升模型的性能。