CONTENT

I. Project Discription

# DBSCAN

DBSCAN requires two parameters: ε (eps) and the minimum number of points required to form a dense region(minPts). It starts with an arbitrary starting point that has not been visited. This point's ε-neighborhood is retrieved, and if it contains sufficiently many points, a cluster is started. Otherwise, the point is labeled as noise. Note that this point might later be found in a sufficiently sized ε-environment of a different point and hence be made part of a cluster.

If a point is found to be a dense part of a cluster, its ε-neighborhood is also part of that cluster. Hence, all points that are found within the ε-neighborhood are added, as is their own ε-neighborhood when they are also dense. This process continues until the density-connected cluster is completely found. Then, a new unvisited point is retrieved and processed, leading to the discovery of a further cluster or noise.
--[Wikipedia](http://en.wikipedia.org/wiki/DBSCAN)

II.Analysis of Sample given

1. dbscan.h:

Given a data structure Point and some assignments of functions.

```
struct Point{
        double x;
        double y;
        int lable;    // -1 unvisited, 0 noise, >0 cluster index
};
```

2. 3spiral.txt

Data set given as Point.

3.main.cpp

Data input to vector<Point> dataset;
Call function dbscan();
Evaluate CPU time used;

4.dbscan.cpp

Function **int dbscan(vector<Point> &dataset, double eps, int min_pts)** do dbscan, given radius and minimum number of neigborhoods.

For each point **p** in dataset, we call **region_query()** to search its neighborhood point which is legal and then push it into its queue called **neighborhood.** Then we decide whether this point **p** is a noise point. Is not we mark it a new cluster and expand this cluster which means push other legal point into this cluster.

Function **expand_cluster()** is an algorithm which just like Breadth-First-Search.

5.Time Evaluation

We have run this sample and it costs **150 second.**

6. Which part can be parallelized?

Let's get started from a small part!
We found that we can do better in calculating Euclidean distance between points and query if this point belong to other points' neighborhood by parallelization.
Implement and other parts of parallelizations' implements will be discuss below.

III. Implements of Parallelizations(CUDA)

1 .We found that we can do better in calculating Euclidean distance between points and query if this point belong to other points' neighborhood by parallelization likely.

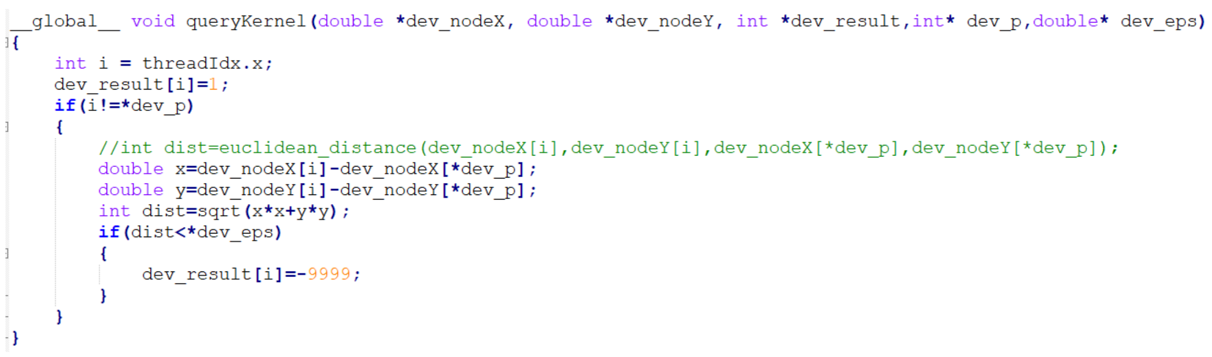
Firstly in fuction **region_query()**
We initialized data structure **nodeX[ ], nodeY[ ]** which mark every point's location x, y.
Memory initialization: cudaMalloc, cudaMemcpy(cudaMemcpyHostToDevice).

```
int size=dataset.size();

double* dev_nodeX;
double* dev_nodeY;
int* dev_result;
int* dev_p;
double* dev_eps;

int *result=(int*)malloc(size*sizeof(int));
double *nodeX = (double*)malloc(size*sizeof(double));
double *nodeY = (double*)malloc(size*sizeof(double));
int i;
for(i=0;i<size;i++)
{
    nodeX[i]=dataset[i].x;
    nodeY[i]=dataset[i].y;
}

cudaMalloc((void**)&dev_nodeX,size*sizeof(double));
cudaMalloc((void**)&dev_nodeY,size*sizeof(double));
cudaMalloc((void**)&dev_result,size*sizeof(int));
cudaMalloc((void**)&dev_p,sizeof(int));
cudaMalloc((void**)&dev_eps,sizeof(double));
cudaMemcpy(dev_nodeX,nodeX,size*sizeof(double),cudaMemcpyHostToDevice);
cudaMemcpy(dev_nodeY,nodeY,size*sizeof(double),cudaMemcpyHostToDevice);
cudaMemcpy(dev_p,&p,sizeof(int),cudaMemcpyHostToDevice);
cudaMemcpy(dev_eps,&eps,sizeof(double),cudaMemcpyHostToDevice);
```

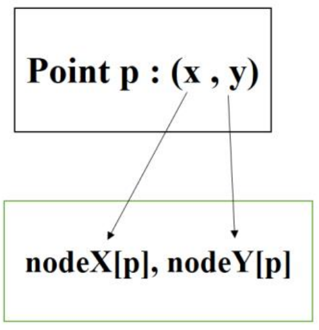
Call a query function on GPU:

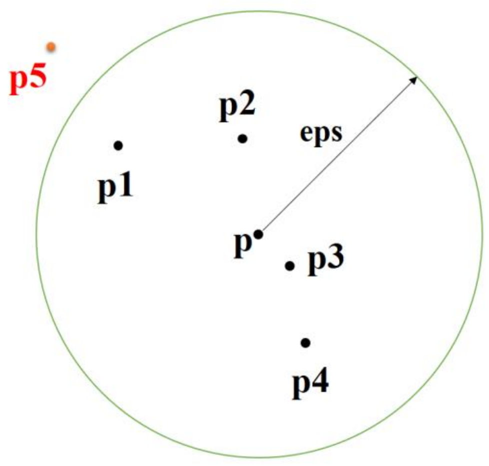
```
queryKernel<<<1,size>>>(dev_nodeX,dev_nodeY,dev_result,dev_p,dev_eps);
```

Which is defined as below:



```
__global__ void queryKernel(double *dev_nodeX, double *dev_nodeY, int *dev_result,int* dev_p,double* dev_eps)
{
    int i = threadIdx.x;
    dev_result[i]=1;
    if(i!=*dev_p)
    {
        //int dist=euclidean_distance(dev_nodeX[i],dev_nodeY[i],dev_nodeX[*dev_p],dev_nodeY[*dev_p]);
        double x=dev_nodeX[i]-dev_nodeX[*dev_p];
        double y=dev_nodeY[i]-dev_nodeY[*dev_p];
        int dist=sqrt(x*x+y*y);
        if(dist<*dev_eps)
        {
            dev_result[i]=-9999;
        }
    }
}
```





Result[1]=-9999, Result[2]=-9999, Result[3]=-9999, Result[4]=-9999
Result[5]=1,

If dev_result[i] is marked as **-9999** which means they are neighborhood between this point **p** and point **i.** If not , it will be marked as **1.**

Then we copy this result set back to host and update points' neighborhood:

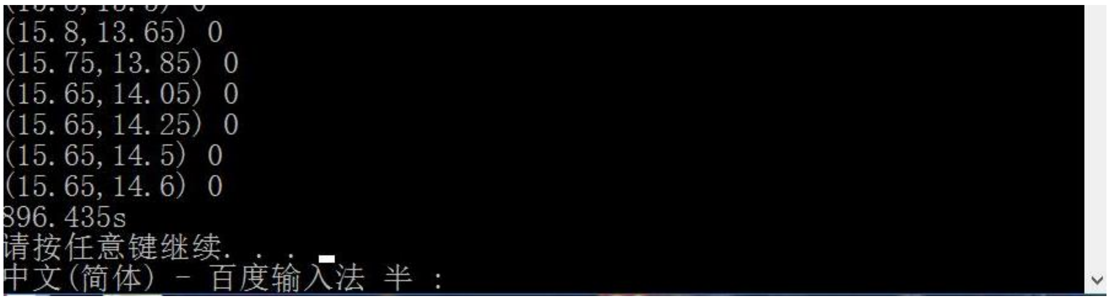
```
cudaThreadSynchronize();
cudaMemcpy(result,dev_result,size*sizeof(int),cudaMemcpyDeviceToHost);
```

```
for(i=0;i<size;i++)
{
    if(result[i]==-9999)
    {
        neighborhood.push(i);
        //printf("%d ",i);
    }

}
```

Time evaluation:

We found this parallel algorithm is much slower than serial algorithm about **6 times slower** than sample’s.

//screenshot



```
(15.8,13.65) 0
(15.75,13.85) 0
(15.65,14.05) 0
(15.65,14.25) 0
(15.65,14.5) 0
(15.65,14.6) 0
896.435s
请按任意键继续. . .
中文(简体) - 百度输入法 半 :
```

Speedup: S = 150/896.435 = 0.167

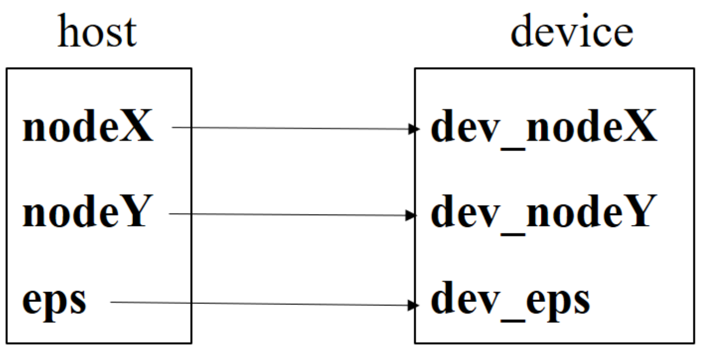
CPU：Intel core i7-6700HQ，GPU: NVIDIA GTX950M

We thought that it costs much of CPU and GPU time on memcpy between host and device. Because every time we call function **region_query()** we do memcopy.

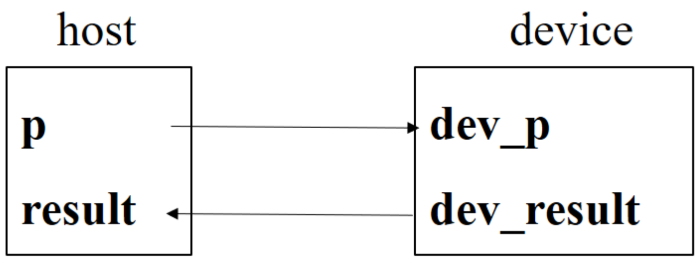
So we work out another solution described below.

2. one time malloc, memcpy and free between host and device. We made malloc, memcpy and free between host and device as a series of functions.

Dbscan( ):



Region_query( ):



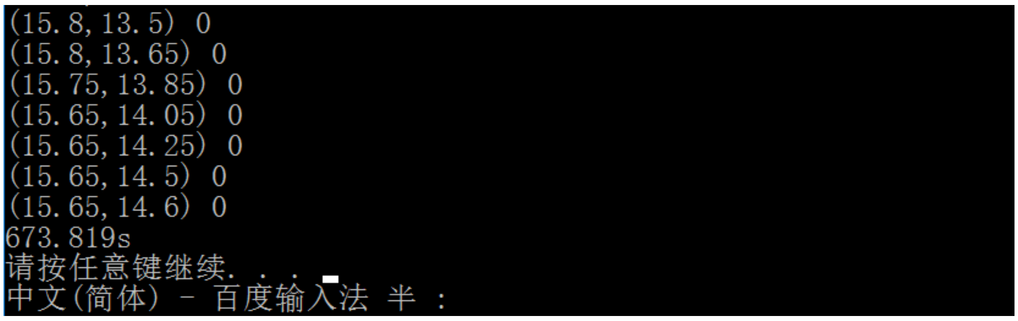
```
void allocate_data_init(vector<Point> &dataset, double eps) {
    // device memory allocate
    int size = dataset.size();
    double *nodeX = (double*)malloc(size * sizeof(double));
    double *nodeY = (double*)malloc(size * sizeof(double));

    for (int i = 0; i<size; i++)
    {
        nodeX[i] = dataset[i].x;
        nodeY[i] = dataset[i].y;
    }

    cudaMalloc((void**)&dev_nodeX, size * sizeof(double));
    cudaMalloc((void**)&dev_nodeY, size * sizeof(double));
    cudaMalloc((void**)&dev_eps, sizeof(double));
    cudaMemcpy(dev_nodeX, nodeX, size * sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(dev_nodeY, nodeY, size * sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(dev_eps, &eps, sizeof(double), cudaMemcpyHostToDevice);

    cudaMalloc((void**)&dev_result, size * sizeof(int));
    cudaMalloc((void**)&dev_p, sizeof(int));

    free(nodeX); free(nodeY);
    //  host memory allocate
    result = (int*)malloc(size * sizeof(int));
}

void allocate_data_free() {
    // device memory free
    cudaFree(dev_nodeX); cudaFree(dev_nodeY);
    cudaFree(dev_result); cudaFree(dev_p); cudaFree(dev_eps);
    // host memory free
    free(result);
}
```

This functions will be only called for one time in function dbscan() and copy all data including nodeX/Y and eps into device one time for all.

Time evaluation:

We found this parallel algorithm(version2) is still slower than serial algorithm about **4 to 5 times slower** than sample's. we are not satisfied about that.

//screenshot



```
(15.8,13.5) 0
(15.8,13.65) 0
(15.75,13.85) 0
(15.65,14.05) 0
(15.65,14.25) 0
(15.65,14.5) 0
(15.65,14.6) 0
673.819s
请按任意键继续. . .
中文(简体) - 百度输入法 半 :
```

Speedup: S = 150/673.819 = 0.223

CPU：Intel core i7-6700HQ，GPU: NVIDIA GTX950M

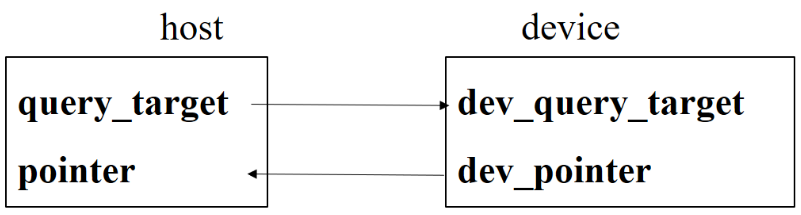
We thought that every time we call function expand_cluster() we will call another function region_query() at least 2 times. Actually, it is not necessary to call function region_query() so many time in function expand_cluster().

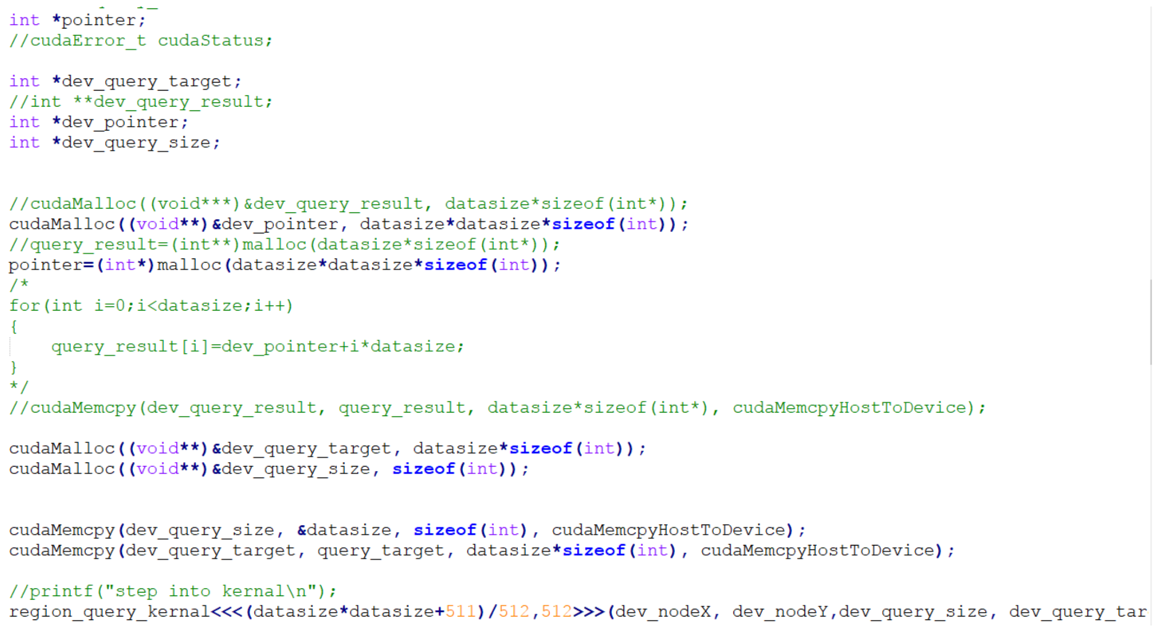
3.optimization for function region_query() and get the result once for all

Just as mentioned above, it is not necessary to call function **region_query()** so many time in function **expand_cluster().** So we optimize function **region_query()** and get the query result once for all**.**
We already parallelized calculating Euclidean distance between points.
And now we will parallelize **region_query()**!!!

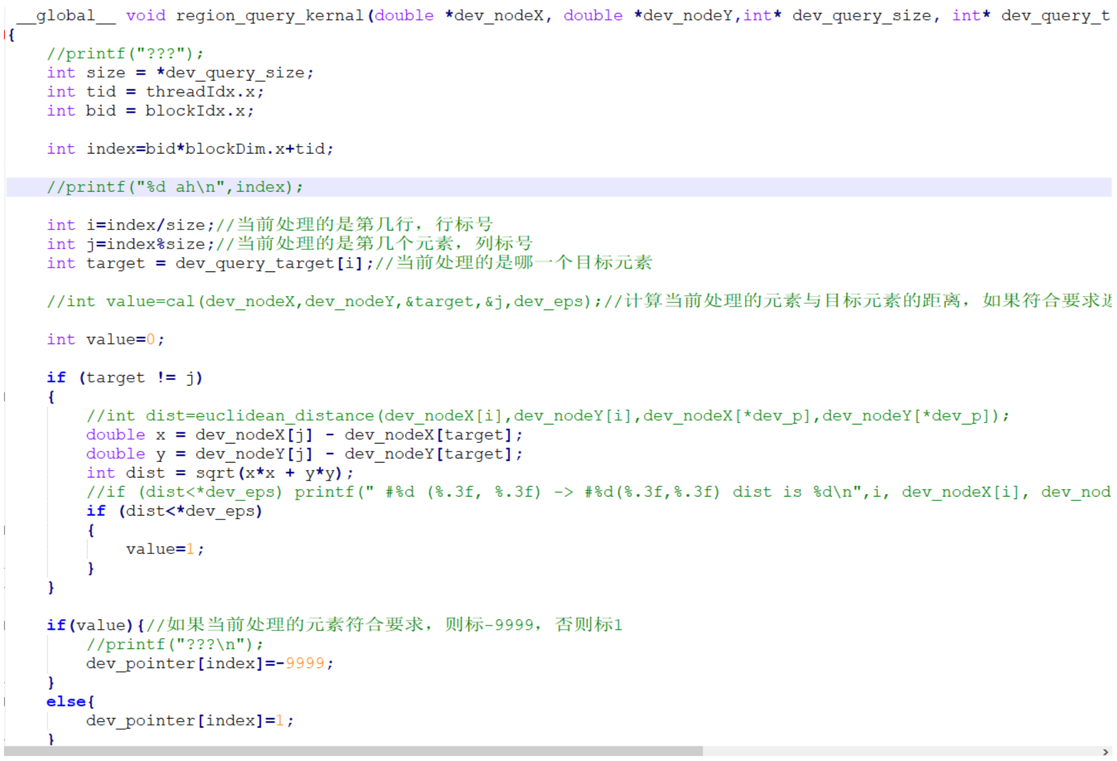
We do memcpy from host to device for region query in function **pral_query(int*query_target,int**total_query_result, double eps)** and call **region_query_kernal()** on device(GPU).

pral_query( ):



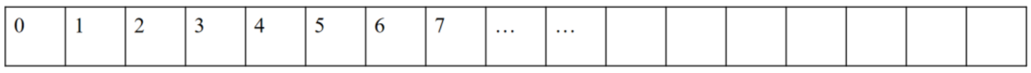

```
int *pointer;
//cudaError_t cudaStatus;

int *dev_query_target;
//int **dev_query_result;
int *dev_pointer;
int *dev_query_size;


//cudaMalloc((void***)&dev_query_result, datasize*sizeof(int*));
cudaMalloc((void**)&dev_pointer, datasize*datasize*sizeof(int));
//query_result=(int**)malloc(datasize*sizeof(int*));
pointer=(int*)malloc(datasize*datasize*sizeof(int));
/*
for(int i=0;i<datasize;i++)
{
    query_result[i]=dev_pointer+i*datasize;
}
*/
//cudaMemcpy(dev_query_result, query_result, datasize*sizeof(int*), cudaMemcpyHostToDevice);

cudaMalloc((void**)&dev_query_target, datasize*sizeof(int));
cudaMalloc((void**)&dev_query_size, sizeof(int));


cudaMemcpy(dev_query_size, &datasize, sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(dev_query_target, query_target, datasize*sizeof(int), cudaMemcpyHostToDevice);

//printf("step into kernal\n");
region_query_kernal<<<(datasize*datasize+511)/512,512>>>(dev_nodeX, dev_nodeY,dev_query_size, dev_query_tar
cudaThreadSynchronize();

cudaMemcpy(pointer, dev_pointer, datasize*datasize*sizeof(int), cudaMemcpyDeviceToHost);

for(int p=0;p<datasize*datasize;p++)
{
    int i=p/datasize;
    int j=p%datasize;
    total_query_result[i][j]=pointer[p];
}

//free(query_result);
free(pointer);
cudaFree(dev_query_target);
//cudaFree(dev_query_result);
cudaFree(dev_query_size);cudaFree(dev_pointer);
```

Function **region_query_kernal()** is define as below:

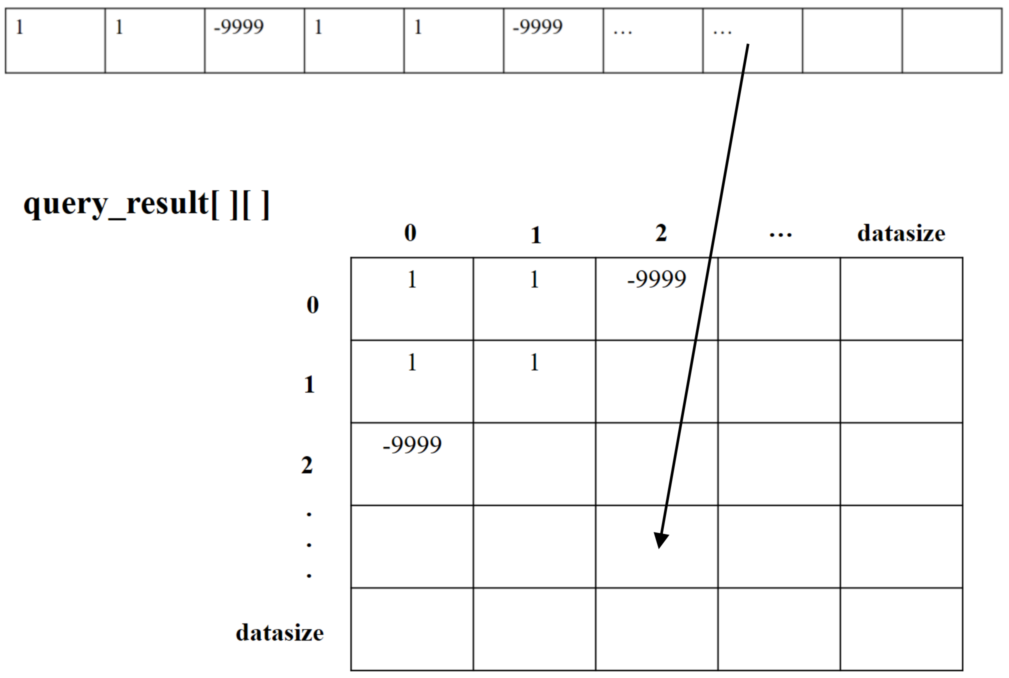


```
__global__ void region_query_kernal(double *dev_nodeX, double *dev_nodeY,int* dev_query_size, int* dev_query_t
{
    //printf("???");
    int size = *dev_query_size;
    int tid = threadIdx.x;
    int bid = blockIdx.x;

    int index=bid*blockDim.x+tid;

    //printf("%d ah\n",index);

    int i=index/size;//当前处理的是第几行，行标号
    int j=index%size;//当前处理的是第几个元素，列标号
    int target = dev_query_target[i];//当前处理的是哪一个目标元素

    //int value=cal(dev_nodeX,dev_nodeY,&target,&j,dev_eps);//计算当前处理的元素与目标元素的距离，如果符合要求让

    int value=0;

    if (target != j)
    {
        //int dist=euclidean_distance(dev_nodeX[i],dev_nodeY[i],dev_nodeX[*dev_p],dev_nodeY[*dev_p]);
        double x = dev_nodeX[j] - dev_nodeX[target];
        double y = dev_nodeY[j] - dev_nodeY[target];
        int dist = sqrt(x*x + y*y);
        //if (dist<*dev_eps) printf(" #%d (%.3f, %.3f) -> #%d(%.3f,%.3f) dist is %d\n",i, dev_nodeX[i], dev_nod
        if (dist<*dev_eps)
        {
            value=1;
        }
    }

    if(value){//如果当前处理的元素符合要求，则标-9999，否则标1
        //printf("???\n");
        dev_pointer[index]=-9999;
    }
    else{
        dev_pointer[index]=1;
    }
```

index=blockIdx.x*blockDim.x+threadIdx.x

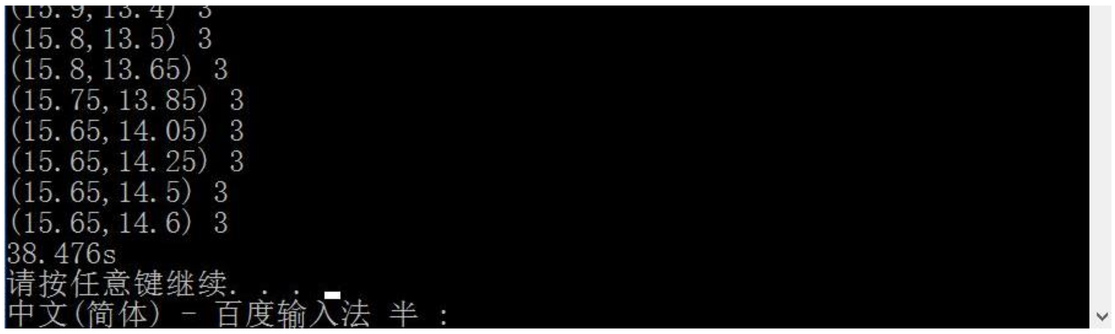


pointer[ ]:



We copy those point to memory of device and compute the distance between every two points. If point **i** and point **j** are neighborhood we will mark in matrix **query_result[i][j]** as **-9999**. If not, it will be marked as **1.**
Then in function dbscan() , we will check **query_result[i][j]** to cnnfirm whether point I and point j are neighborhood.

Time evaluation:
We found this parallel algorithm(version3) is faster than serial algorithm about **4 times faster** than sample's.
//screenshot



```
(15.9, 13.4) 3
(15.8, 13.5) 3
(15.8, 13.65) 3
(15.75, 13.85) 3
(15.65, 14.05) 3
(15.65, 14.25) 3
(15.65, 14.5) 3
(15.65, 14.6) 3
38.476s
请按任意键继续. . .
中文(简体) - 百度输入法 半 :
```

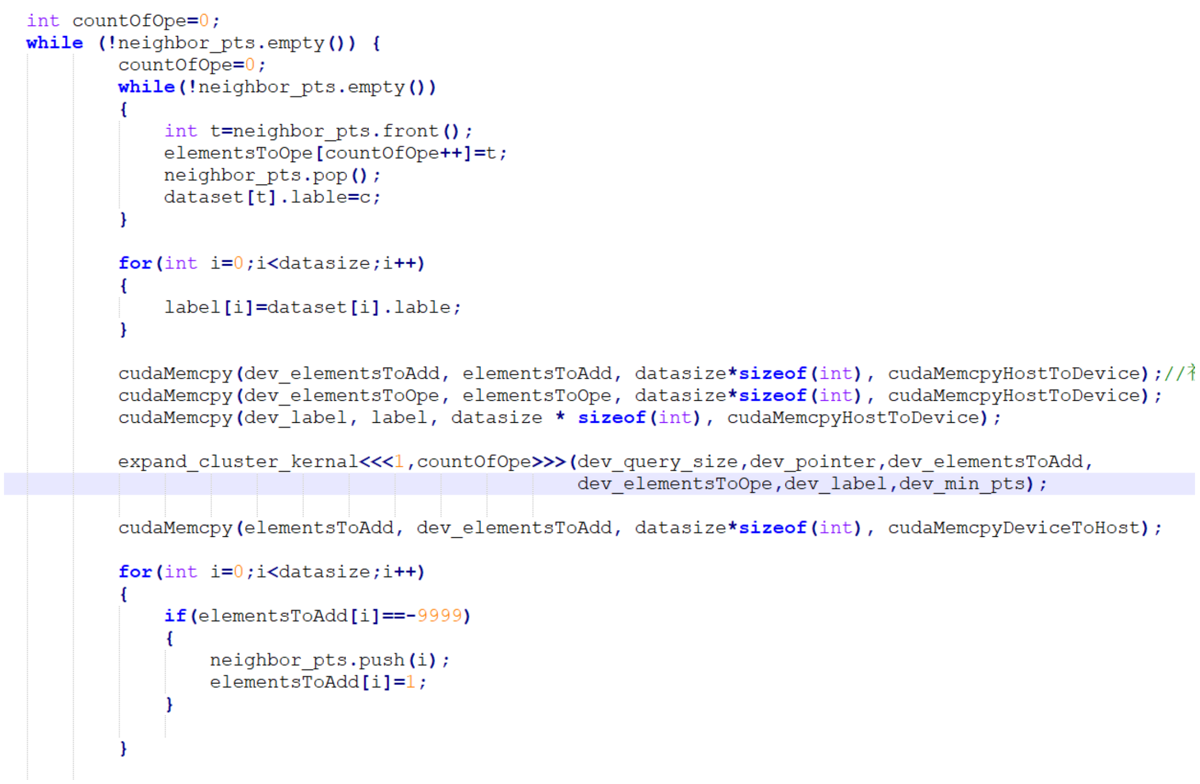
Speedup: S = 150/38.476 = 3.899

CPU：Intel core i7-6700HQ, GPU: NVIDIA GTX950M

4.parallelize part of expand_cluster()

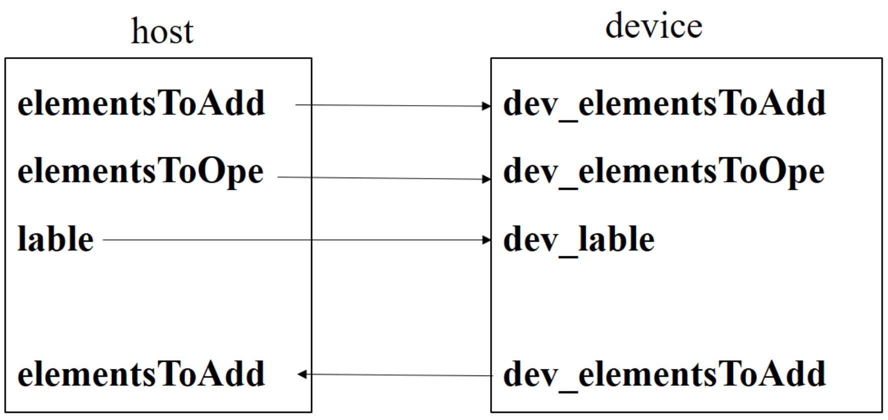
In function **expand_cluster()** we will find which points or element to push in queue by parallelization.

elementsToOpe[ ] means elements which need to be operated, and we find those elements' neighborhood and save them to **elementsToAdd[ ]** by function **expand_cluster_kernal().** We push it into a neighbor queue when we get the elementToAdd[].



```
int countOfOpe=0;
while (!neighbor_pts.empty()) {
        countOfOpe=0;
        while(!neighbor_pts.empty())
        {
            int t=neighbor_pts.front();
            elementsToOpe[countOfOpe++]=t;
            neighbor_pts.pop();
            dataset[t].lable=c;
        }

        for(int i=0;i<datasize;i++)
        {
            label[i]=dataset[i].lable;
        }

        cudaMemcpy(dev_elementsToAdd, elementsToAdd, datasize*sizeof(int), cudaMemcpyHostToDevice);//
        cudaMemcpy(dev_elementsToOpe, elementsToOpe, datasize*sizeof(int), cudaMemcpyHostToDevice);
        cudaMemcpy(dev_label, label, datasize * sizeof(int), cudaMemcpyHostToDevice);

        expand_cluster_kernal<<<1,countOfOpe>>>(dev_query_size,dev_pointer,dev_elementsToAdd,
                                                dev_elementsToOpe,dev_label,dev_min_pts);

        cudaMemcpy(elementsToAdd, dev_elementsToAdd, datasize*sizeof(int), cudaMemcpyDeviceToHost);

        for(int i=0;i<datasize;i++)
        {
            if(elementsToAdd[i]==-9999)
            {
                neighbor_pts.push(i);
                elementsToAdd[i]=1;
            }

        }
```

expand_cluster( ):



Function **expand_cluster_kernal()** is defined as below:

```
__global__ void expand_cluster_kernal(int *dev_query_size,int *dev_pointer,
        int* dev_elementsToAdd,int *dev_elementsToOpe,int *dev_label,int *dev_min_pts)
{
    int index=threadIdx.x;
    int size=*dev_query_size;
    int target=dev_elementsToOpe[index];
    int counter=0;
    for(int i=0;i<size;i++)
    {
        int location = target*size+i;
        if(dev_pointer[location]==-9999)
            counter++;
    }
    if(counter>=*dev_min_pts-1)
    {
        for(int i=0;i<size;i++)
        {
            int location = target*size+i;
            if(dev_pointer[location]==-9999&&dev_label[i]==-1)
            {
                dev_elementsToAdd[i]=-9999;
            }
        }
    }

}
```

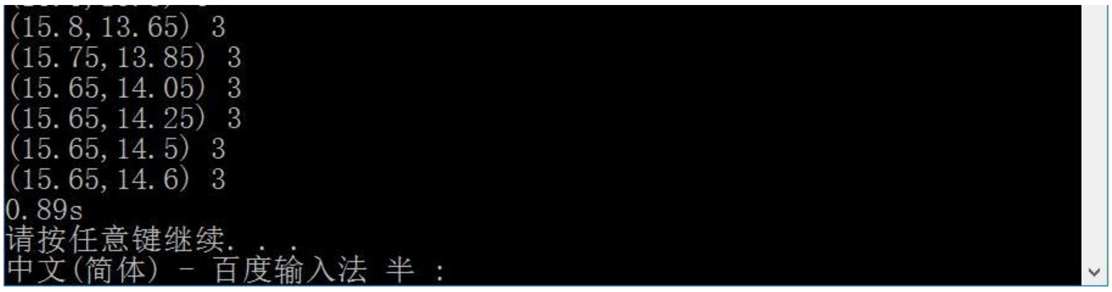
If the number of its neighbors is more than **min_pst** and the neighbor dose not belongs to any cluster, we mark the neighbor **elementToAdd** and set the value **-9999.**

Time evaluation:

We found this parallel algorithm(version4) is faster than serial algorithm about **169 times faster** .

We are satisfied about that.

//screenshot



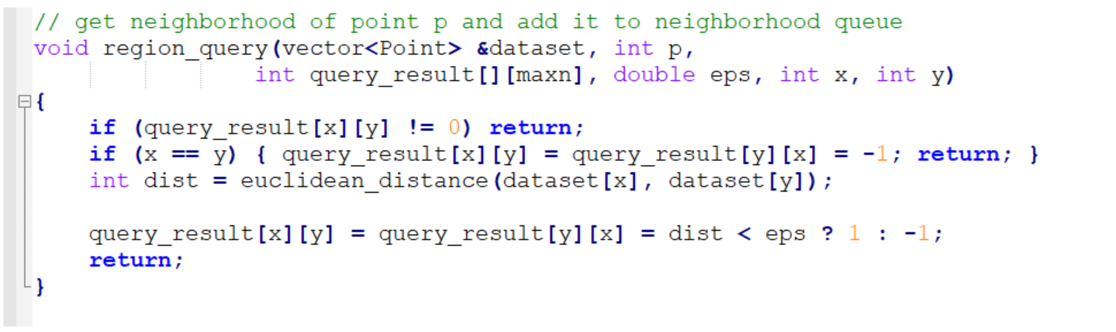
```
(15.8,13.65) 3
(15.75,13.85) 3
(15.65,14.05) 3
(15.65,14.25) 3
(15.65,14.5) 3
(15.65,14.6) 3
0.89s
请按任意键继续. . .
中文(简体) - 百度输入法 半 :
```

Speedup:　S = 150/0.89 = 168.539

CPU：Intel core i7-6700HQ，GPU:　NVIDIA GTX950M

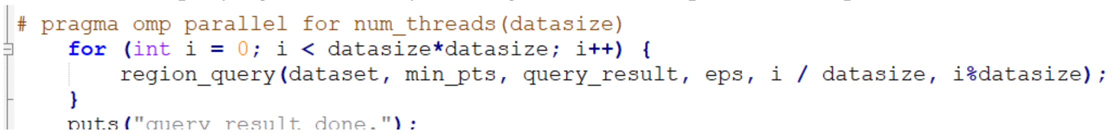
IV. Implements of Parallelizations(OpenMP)

We set a matrix **query_result[ ][ ]** which holds the “distance” between points for the same reason above. If the distance between point **i** and point **j** have not been computed **query_result[i][j]** and **query_result[j][i]** will be marked as **0**. If point **i** and point **j** are neighbors **query_result[i][j]** and **query_result[j][i]** will be marked as **1**. If not it will be marked as **-1**.



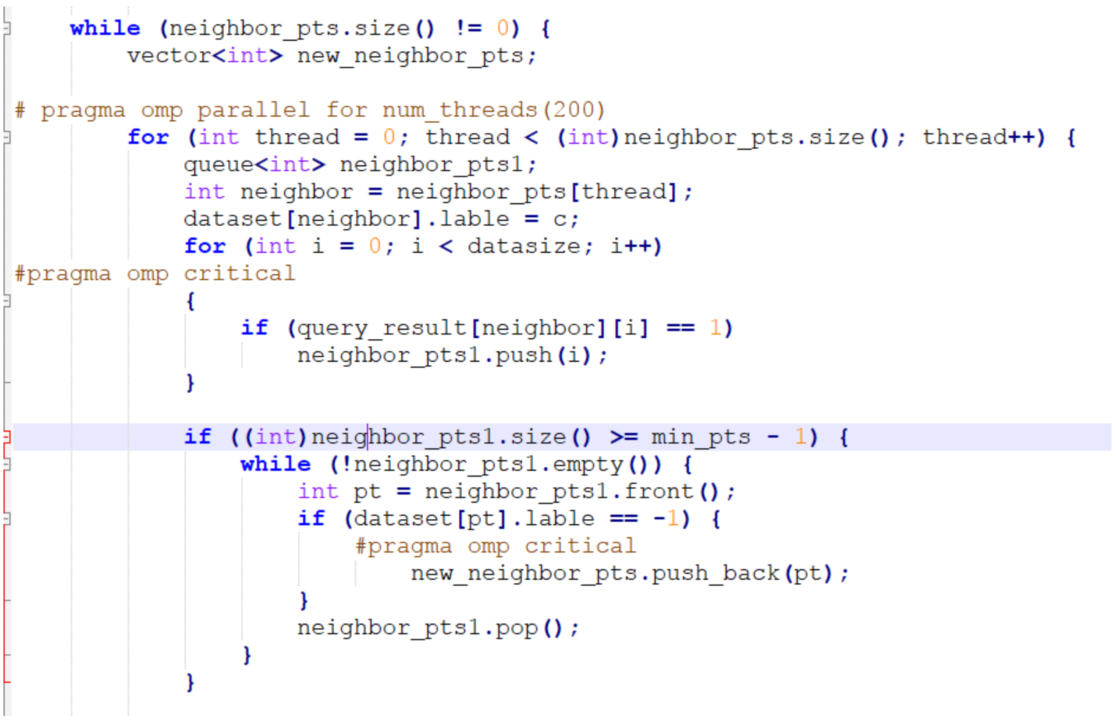
```
// get neighborhood of point p and add it to neighborhood queue
void region_query(vector<Point> &dataset, int p,
                int query_result[][maxn], double eps, int x, int y)
{
    if (query_result[x][y] != 0) return;
    if (x == y) { query_result[x][y] = query_result[y][x] = -1; return; }
    int dist = euclidean_distance(dataset[x], dataset[y]);

    query_result[x][y] = query_result[y][x] = dist < eps ? 1 : -1;
    return;
}
```

The section of querying whether they are beighbors between points can be parallelized as below:



```
# pragma omp parallel for num_threads(datasize)
    for (int i = 0; i < datasize*datasize; i++) {
        region_query(dataset, min_pts, query_result, eps, i / datasize, i%datasize);
    }
    puts("query result done.");
```

Parallelization of **for** loop in BFS:



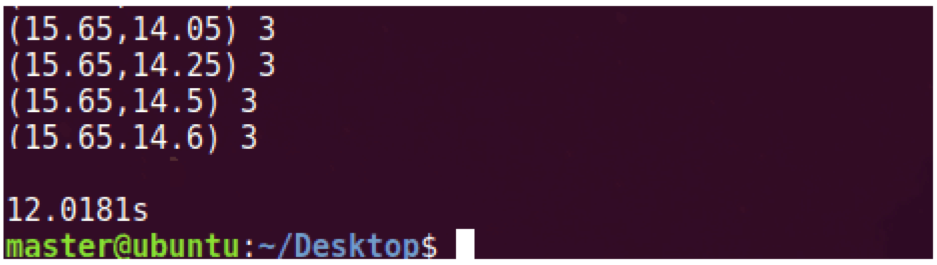
```
    while (neighbor_pts.size() != 0) {
        vector<int> new_neighbor_pts;

# pragma omp parallel for num_threads(200)
        for (int thread = 0; thread < (int)neighbor_pts.size(); thread++) {
            queue<int> neighbor_pts1;
            int neighbor = neighbor_pts[thread];
            dataset[neighbor].lable = c;
            for (int i = 0; i < datasize; i++)
#pragma omp critical
            {
                if (query_result[neighbor][i] == 1)
                    neighbor_pts1.push(i);
            }

            if ((int)neighbor_pts1.size() >= min_pts - 1) {
                while (!neighbor_pts1.empty()) {
                    int pt = neighbor_pts1.front();
                    if (dataset[pt].lable == -1) {
                        #pragma omp critical
                            new_neighbor_pts.push_back(pt);
                    }
                    neighbor_pts1.pop();
                }
            }
```

Critical section:

The operation to queue **neighbor_pts1** and **new_neighbor_pts.**

Time evaluation:

We found this parallel algorithm(OpenMP) is faster than serial algorithm about **12.5 times faster** .



```
(15.65,14.05) 3
(15.65,14.25) 3
(15.65,14.5) 3
(15.65.14.6) 3

12.0181s
master@ubuntu:~/Desktop$
```

Speedup: S = 150/12 = 12.5

VM Ubuntu, core: 4

V. Conclusion

We found that it's more complex to use CUDA to implement a parallel program because of memory copy between host and device and other reasons. But the speedup of CUDA is awesome.

OpenMP is also a nice way to implement parallelization. The most important thing is OpenMP is much easier than CUDA and you can check the code line to find that. But it's a bit slower than CUDA.