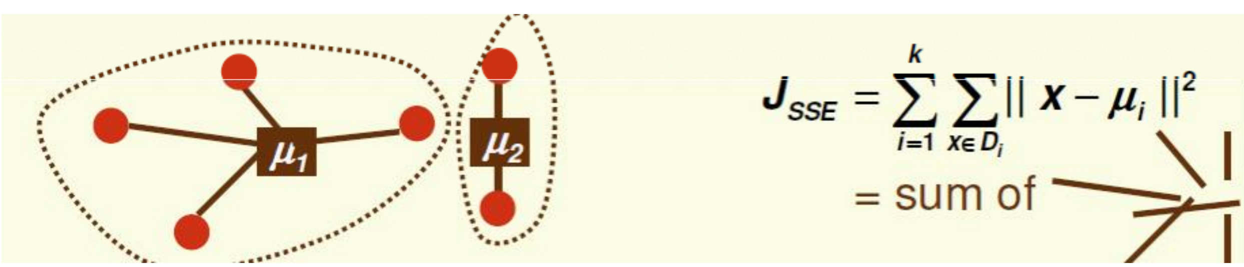
K-Means Clustering

Introduction

Clustering is a method of unsupervised learning that partitions a set of data objects in to clusters. The K-Means algorithm is one of the most popular analysis. The running time of K-Means algorithm grows with the increase of the size and also the dimensions of the data set. Hence clustering large-scale data sets is usually a time-consuming task. Parallelizing K-Means is a promising approach to overcoming the challenge of the huge computational requirement.

Traditional K-Means algorithm

1. Initialize: Pick k cluster centers arbitrarily, assign each example to closest center.
2. Compute sample means for each cluster.



3. Reassign all samples to the closest mean.

$$\frac{\partial}{\partial z}\sum_{x\in D_i}\frac{1}{2}\|x-z\|^2=\frac{\partial}{\partial z}\sum_{x\in D_i}\frac{1}{2}\left(\|x\|^2-2x^t z+\|z\|^2\right)=\sum_{x\in D_i}(-x+z)=0$$

$$\Rightarrow z=\frac{1}{n_i}\sum_{x\in D_i}x$$

4. If cluster changed at step 3, go to step 2.

Suitability for GPU ACCELERATION

The distance between two points is independent, and permanent.

More than 95% of the K-Means calculation parts can be done on the GPU.

Distance is the core computation in k-means algorithm.

It is suitable for GPU computation.

GPU implements

1. Random pick k centers in array cluster's centers.
2. Calculate all the distance from each point to each center.
3. Update the cluster's index calculated by step 2.
4. Calculate the new center with the cluster index.

Data structure:

float *data, *centers store the points data in a 1D float array,

Point:

D0	D1	D2	D3	D4	D5	D6..	dimension

P0	P1	P2	P3	P4	P5	P6	…

Step2: calculate all distance.

I use a 2d structure to calculate the distance, the x index is the cluster number k, and the y index is the points number n, each thread calculates the distance from cluster center[x] to point[y]. In the first implement, I used the global memory, then I try to use shared memory. At first, I store all the centers into the shared memory, but when the size of the data increase, the result is incorrect. Then, I use tiled calculation, each block finds the address of the data and the center's data and store them to the shared memory. The reason why I chose 2D structure is the x parameter can find the index of the cluster centers, and the y parameters can find the index of the points, and use the parameter dimension to find the data I need. Each block thread will store to point share and the cluster center share dimension/16 times.

	0	1	2	3	4…	k
0	0,0	1,0	2,0	3,0	4,0	…
1	1,0					
2	2,0					
…n	…					

```
The cal_distance part of GPU

__global__ void cal_distance(float *dev_obj_data,float *dev_center_data,
        float *dev_disntance, int obj_num, int cluster_num, int dimension){
    int col = blockDim.x * blockIdx.x + threadIdx.x;
    int row = blockDim.y * blockIdx.y + threadIdx.y;
    if(row < obj_num && col < cluster_num){
        int idx = col + row * cluster_num;
        dev_disntance[idx] = distance(dev_obj_data + row * dimension,
dev_center_data + col * dimension, dimension);
    }
}


__device__ float distance(float *dev_obj_data, float *dev_center_data, int
dimension){
    float distance = 0;
    for (int i = 0; i < dimension; ++i) {
        float tmp = dev_obj_data[i] - dev_center_data[i];
        distance = distance + tmp * tmp;
    }
    return distance;
}
__global__ void cal_distance_share(float *dev_obj_data, float *dev_center_data,
        float *dev_distance, int obj_num, int cluster_num, int dimension){
    int col = blockDim.x * blockIdx.x + threadIdx.x;
    int row = blockDim.y * blockIdx.y + threadIdx.y;
    const int size = DIMENSION * BLOCKSIZE_16;
    __shared__ float cluster_center_share[BLOCKSIZE_16][size];
    __shared__ float obj_data_share[BLOCKSIZE_16][size];
    if(row < obj_num){
        float *obj_data = &dev_obj_data[dimension * blockDim.y * blockIdx.y];
        float *center_data = &dev_center_data[dimension * blockDim.x *
        blockIdx.x];
```

```
        for (int xidx = threadIdx.x; xidx < dimension; xidx += BLOCKSIZE_16) {
            int idx = dimension * threadIdx.y + xidx;
            obj_data_share[threadIdx.y][xidx] = obj_data[idx];
            cluster_center_share[threadIdx.y][xidx] = center_data[idx];
        }
        __syncthreads();
    }
    if(col < cluster_num && row < obj_num){
        dev_distance[row * cluster_num + col] =
distance(obj_data_share[threadIdx.y], cluster_center_share[threadIdx.x],
dimension);
    }
}
```

Step 3 update the cluster index(in class).

At first, for each point, I use 1D structure and a for loop to find the least distance index, then I tried to use the add reduction thought, use 2D structure and each thread compares two value then update, to find the least distance index in the block, but when I test some big data, the result turns to be incorrect, and I use the 1D loop to compare the time.

Step 4 Update the new centers(in class).

With cluster index, I use 1D structure to add reduction to calculate the addition of each dimension, use atomic add to calculate the count of each cluster. Then calculate the new centers by division.

Time comparation(ms)

Number	CPU	GPU	SHARE_M	iterator
100	0.059	0.316	0.354	5
1000	1.421	0.644	0.636	14
10000	22.475	3.317	3.111	22
100000	8928	1314	1221	90

Dimension=3, k=3

Number	CPU	GPU	SHARE_M	iterator
100	0.169	0.399	0.320	5
1000	6.765	1.781	1.360	28
10000	188.77	49.55	17.35	80
100000	1183	167.9	100.11	50

Dimension=10, k=3

Number	CPU	GPU	SHARE_M	iterator
100	0.238	0.383	0.376	8
1000	4.196	0.774	0.684	15
10000	99.4	11.50	4.78	37
100000	2210	131.2	91.30	83

Dimension=3, k=10

Number	CPU	GPU	SHARE_M	iterator
100	3.378	0.249	0.238	4
1000	197.05	2.33	1.32	11
10000	11313	110	81	68
100000	840s	9.7s	7s	523

Dimension=30, k=100

Conclusion

If the data set is small, The CPU calculation time will be less than the GPU. If k=3, d=3 and n =100000, because I use 16*16 block size, most threads do not calculate, it is only 7 times faster.

If the k and d are larger enough than 16, for example k =100, d = 30, it will only take 7s by using shared memory, and the CPU needs 840s, there may be 120 times faster calculated by GPU.

Refrrence

https://www.cs.stevens.edu/~mordohai/classes/cs559_f16/cs559f16_Week13.pdf