UNIVERSITÀ DEGLI STUDI FIRENZE

K-Means Implementatoin

Parallel Computing 9CFU

Claudio Pagnini

Introduction

- K-Means Clustering is an unsupervised machine learning algorithm used for **data clustering** and **pattern recognition**.
- The goal of K-Means is to partition data points into **K clusters**, where each cluster is represented by its **centroid**.
- It is one of the most widely used clustering algorithms due to its **simplicity** and **efficiency**.

Introduction

Given a set of observation in a 2D dimensional and K random centroids

Assignment Step: For each data point $\mathbf{x}_i$, find the nearest centroid $\mathbf{c}_j$ based on the distance metric (Euclidean distance in our case):

$$j = \arg\min_k \|\mathbf{x}_i - \mathbf{c}_k\|^2$$

Assign $\mathbf{x}_i$ to cluster j.

Introduction

Update Step: After the assignment step, update the centroids $\mathbf{c}_j$ of each cluster by taking the mean of all data points assigned to that cluster:

$$\mathbf{c}_j = \frac{1}{n_j} \sum_{\mathbf{x}_i \text{ assigned to cluster } j} \mathbf{x}_i$$

where n_j is the number of data points assigned to cluster j.
The K-Means algorithm iteratively repeats these two steps until **convergence**, where the centroids stop changing significantly or a maximum number of iterations is reached

UNIVERSITÀ DEGLI STUDI FIRENZE

Proposed approach

In this work we propose 3 different implementations of the algorithm:

1. A sequential version with Python
2. A parallel version with OpenMP using AoS and SoA architectures
3. A parallel version with Cuda

UNIVERSITÀ DEGLI STUDI FIRENZE

Proposed approach

”AoS” and ”SoA” are two different ways of organizing data in memory, particularly when dealing with collections of related elements

AoS (Array of Structures) groups related data into an array of structures, while **SoA** (Structure of Arrays) organizes data into separate arrays based on attributes. AoS is intuitive and easy to use, while SoA can offer performance benefits in certain scenarios, especially for large datasets and optimized memory access.

UNIVERSITÀ DEGLI STUDI FIRENZE

Sequential Implementation

Alg 1. Assignment

```
def assign_centroid(points, centroids, points_assg):
    for i in range(counter):
        distance = sys.maxsize
        for j in range(num_clusters):
            dist = distance_2d(points[i, 0], points[i, 1],
            centroids[j, 0], centroids[j, 1])
            if dist < distance:
                distance = dist
                points_assg[i] = j
```

Alg 2. Centroid Update

```
def centroid_update(points, points_assgn):
    centroids_sum = np.zeros((num_clusters, 2))
    cluster_size = np.zeros(num_clusters)
    for i in range(counter):
        clust_id = points_assgn[i]
        clust_id = int(clust_id)
        cluster_size[clust_id] = cluster_size[clust_id] + 1
        centroids_sum[clust_id, 0] += points[i, 0]
        centroids_sum[clust_id, 1] += points[i, 1]
    cluster_size = np.vstack((cluster_size, cluster_size))
    return centroids_sum / cluster_size.T
```

OpenMP

The OpenMP API allow to transform the sequential version into a parallel one with several directives

Assignment AoS

```
UPDATE(points, clusters)
#pragma omp for schedule(static)
        for each point in points
           distance<-INF
           index<-0
           for each cluster in clusters
                temp_dist<-Euc_Distance(point, cluster)
                if temp_dist<distance
                    distance<-temp_dist
                    index<-cluster
                point.setClusterId(index)
                cluster[index].addPoint(point)
```

Cluster.h

```
void add_point(Point point)
    {
#pragma omp atomic
        candidate_coord_x += point.get_coord_x();
#pragma omp atomic
        candidate_coord_y += point.get_coord_y();
#pragma omp atomic
        size++;
    }
```

OpenMP

Assignment SoA

```
    int num_points = points[0].size;
    int num_clusters = clusters[0].size;

#pragma omp parallel for
    for (int i = 0; i < num_points; i++)
    {
        int closest_cluster = 0;
        double min_distance = distance(points[0], clusters[0], i, 0);

        for (int j = 1; j < num_clusters; j++)
        {
            double dist = distance(points[0], clusters[0], i, j);
            if (dist < min_distance)
            {
                min_distance = dist;
                closest_cluster = j;
            }
        }
        points[0].cluster_id[i] = closest_cluster;
    }
    // Reassign points to clusters
    for (int i = 0; i < num_clusters; i++)
    {
        clusters[0].point_count[i] = 0;
        clusters[0].coord_x[i] = 0.0;
        clusters[0].coord_y[i] = 0.0;
    }
```

OpenMP

Assignment SoA

```
for (int i = 0; i < num_points; i++)
{
    int cluster_id = points[0].cluster_id[i];
    clusters[0].point_count[cluster_id]++;
    clusters[0].coord_x[cluster_id] += points[0].x_coord[i];
    clusters[0].coord_y[cluster_id] += points[0].y_coord[i];
}
```

This code calculates the closest cluster for each point and updates the cluster assignment for each point. Then, it reassign points to clusters based on their assignments. In this case, you don't need a critical or atomic directive because you are not performing concurrent writes to shared variables. Each point is assigned to a cluster independently based on its distance calculation, and there are no shared variables that multiple threads write to simultaneously.

Cuda

CUDA is a parallel computing platform that leverages the massive parallelism of GPUs to accelerate computations.

By using CUDA to program GPUs, developers can take advantage of their parallel processing power, leading to significant speedups over traditional CPUs

This implementation of k-means with cuda points on the 2dimensional arrays which can be splitted in 2 distinct one dimensional arrays, for points and for cluster:

$$\text{points}$$
$$p_x = [x_1 ... x_N] and p_y = [y_1 ... y_N]$$
$$\text{clusters}$$
$$c_x = [x_1 ... x_K] and c_y = [y_1 ... y_K]$$

Cuda

Cuda Main

```
Cluster_Assignment(p_x,p_y,c_x,c_y)
    c_sum_x = 0
    c_sum_y = 0
    Sum_Update(p_x,p_y,c_x,c_y,
                          c_sum_x,c_sum_y)
    Repeat for K cluster:
       c_x=c_sum_x/c_size
        c_y=c_sum_y/c_size
```

Assignment

```
idx = blockIdx.x*blockDim.x + threadIdx.x
if (idx >= N) return
float min_dist = INFINITY
int closest_centroid = 0
for(int c = 0; c<K; ++c)
    dist = Euclidean_Distance(p_x[idx],
             d_y[idx], c_x[c], c_c[c])
    if(dist < min_dist)
        min_dist = dist
        closest_centroid=c
d_Cluster_Membership[idx]=closest_centroid
```

Cuda

Update

```
idx = blockIdx.x * blockDim.x + threadIdx.x
if (idx >= N) return
clust_id = d_Cluster_Membership[idx]
Add(c_sum_x[clust_id], p_x[idx])
Add(c_sum_y[clust_id], p_y[idx])
Add(c_size[clust_id], 1)
```

UNIVERSITÀ DEGLI STUDI FIRENZE

Experimental Results

The metric used to compare the performances of the sequential algorithm with the OpenMP and Cuda implementation is the speedup, computed as:

$$S = \frac{t_S}{t_P}$$

where t_S and t_P are respectively the execution time of the sequential and the parallel implementation
The tests have been executed on a Laptdop that do not allow logical cores with:

- OS: Windows 11 Pro
- CPU: AMD Ryzen 5 4500U CPU @2.38GHz
- GPU: GEForce GTX 1650 Mobile / 4GB with CUDA 11.1

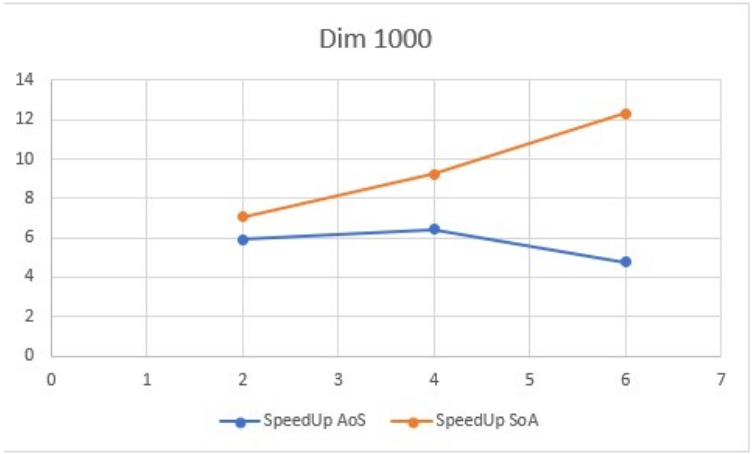
UNIVERSITÀ DEGLI STUDI FIRENZE

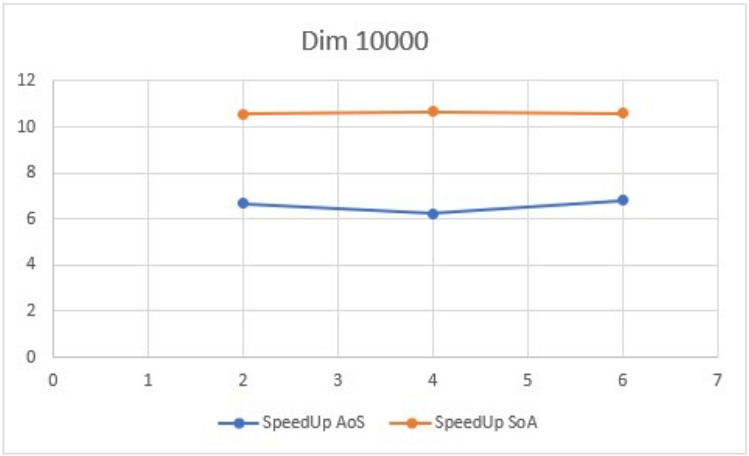
Experimental Results

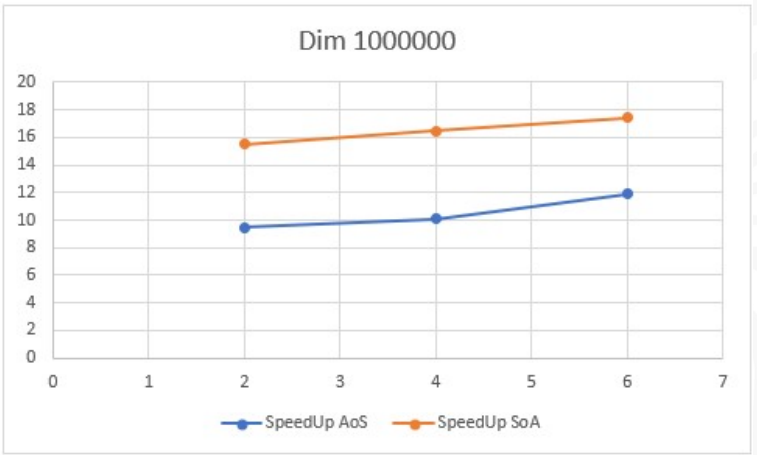
The datasets used to evaluate the different implementations have been generated with the make_blob() function of sklearn.datasets . They are gaussian distributions with 5 centers and standard deviation equal to 0.5 and are composed by respectively 100, 1000, 10000, 100000 and 10000002D points.

Performance Evaluation - OpenMp

To evaluate the performances of the OpenMP implementation, it has been executed on each dataset with an increasing number of threads.







UNIVERSITÀ DEGLI STUDI FIRENZE

Performance Evaluation - Cuda

It has been tested CUDA implementation by varying block dimension with fixed 10000 points dataset.

Block Dim	CUDA
32	0.0073 s
64	0.0068
128	0.00548s
256	0.0093 s
512	0.011 s
1024	0.013 s

UNIVERSITÀ DEGLI STUDI FIRENZE

Performance Evaluation - Cuda

It has been tested CUDA implementation by varying dimension with fixed block dimension to 128.

Dim	Sequential	CUDA	SpeedUp
1000	0.148 s	0.00375 s	39.47
10000	1.46 s	0.008813 s	165.72
100000	14.18 s	0.07724 s	183.68
1000000	97.15 s	0.5131 s	189.38

UNIVERSITÀ DEGLI STUDI FIRENZE

Conclusion

As a final result, a global comparison has been conducted and therefore only the best results for each dataset dimension and each implementation have been considered. In table 3 we can see that the CUDA algorithm abundantly outperforms both the sequential and the OpenMP ones, at the expense of a more complicated implementation. However OpenMP lets us to achieve a noticable speedup with just several directive.

Dim	Sequential	OpenMP	OpenMP Speedup	CUDA	CUDA Speedup
1000	0.148 s	0.012 s	12.33	0.00375 s	39.47
10000	1.46 s	0.137 s	10.65	0.0088 s	165.72
100000	14.18 s	1.25 s	11.34	0.077 s	183.68
1000000	97.15 s	5.58 s	17.41	0.51 s	189.38