K-means Algorithm

Claudio Pagnini
claudio.pagnini@stud.unifi.it

Abstract

K-Means is a parametric clustering technique with an embarassingly parallel structure and for this reason it's suitable for parallel computing. In this work, an OpenMP implementation will be presented and the execution times of each version will be compared. A particular focus will be given to the speedup obtained with the parallel versions for datasets of increasing dimension and to the comparison between two style of structure design: AoS vs SoA.

1 Future Distribution Permission

The author of this report give permission for this document to be distributed to Unifi-affiliated students taking future courses.

2 Introduction

K-means clustering is a very popular unsupervised machine learning algorithm. In the partition-based clustering algorithm, K-means algorithm has many advantages such as simple mathematical ideas, fast convergence, and easy implementation [1]. The basic idea of having a cluster is to collect of datas which are aggregated together because of certain similarities. In order to achive this, a target number K has to be defined, which refers to the number of centroids we need in the dataset. A centroid is the location that represents the center of the cluster.

2.1 K-Means Algorithm

The K-Means Algorithm is an iterative cluster algorithm. It uses the euclidean distance as metric in order to split out N points finding K groups with the same features: each point of that particular group is too far from the others. Given an initial set of observations in a ddimensional Euclidean space:

$$x_1, x_2, \ldots, x_N$$

Given a set of k random centroids:

$$c_1^{(1)}, c_2^{(1)}, \ldots, c_K^{(1)}$$

the algorithm proceeds by alternating between two steps:

Assignment step: Assign each observation to the cluster with the nearest centroid, the one with the least squared Euclidean distance.

$$C_i^{(t)} = \left\{ x_p : \left\| x_p - c_i^{(t)} \right\|^2 \leq \left\| x_p - c_j^{(t)} \right\|^2, \forall j : 1 \leq j \leq K \right\}$$

Update Step: Recalculate centroids for observations assigned to each cluster.

$$c_i^{(t+1)} = \frac{1}{\left| C_i^{(t)} \right|} \sum_{x_j \in C_i^{(t)}} x_j$$

The algorithm converges when assignments and centroids no longer change.

2.2 AoS vs SoA design structure

In the context of computer science and software engineering, "AoS" and "SoA" stand for "Array of Structures" and "Structure of Arrays," respectively. They 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.

3 Proposed Approach

In this paper 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

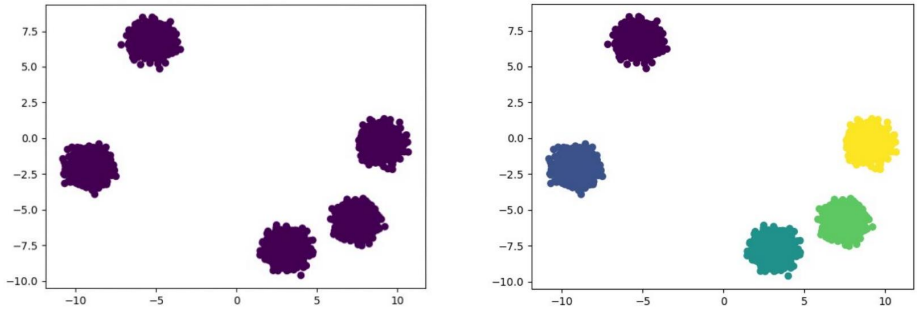

Figure 1. Example of a dataset with 10000 points respectively not clustered and clustered

It's been choosen to build implementations of K-means that work in 2-dimensional Euclidean spaces and the centroids are initialized by taking K points at random from the N observations.

3.1 Python

The implementation with Python just follow the principle of K-means

K-means **Main**

```
for i in range(iterations):
    assign_centroid(points, centroids, points_assg)
    centroids = centroid_update(points, points_assg)
```

The iterations constant inside the range represents the number of iterations of the algorithm. The assign_centroid (showed in the **Alg. 1**) function manage the first step of a K-means iteration in order to assign each points to the cluster with the nearest centroid.

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
```

The **Alg. 2** shows the last step of K-Means: cluster centroids update.

Alg 2. 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
```

3.2 OpenMp

The OpenMP API allow to transform the sequential version into a parallel one with several directives. It takes the same sequential implementation with minimal changes called #pragma omp parallel directive shown in **Alg 3 - Alg 3.1** . As same as the sequential solution, there is a main which calls in order **Alg 3 - Alg 3.1** and then **Alg 4 - Alg 4.1**. Unlike the sequential implementation, every points assigned to each cluster may risk to be executed by one ore more threads at time. So, for this reason, we need to call that section "critical".

Alg 3. Assignment AoS

```
            double min_distance;
            int cluster_id;
            int num_points = points.size();
            int clust_size = clusters.size();

#pragma omp parallel default(shared) private(min_distance, cluster_id)

    {
#pragma omp for schedule(static)
        for (int i = 0; i < num_points; i++)
        {

            Point &p = points[i];
            Cluster temp_Cluster = clusters[0];
            min_distance = distance(p, temp_Cluster);
            cluster_id = 0;

            for (int j = 1; j < clust_size; j++)
            {
                double now_distance = distance(p, clusters[j]);
                if (now_distance < min_distance)
                {
                    min_distance = now_distance;
                    cluster_id = clusters[j].get_cluster_Id();
                }
            }

            p.set_id_c(cluster_id);

        clusters[cluster_id].add_point(p);
        }
    }
```

Alg 3.1 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;
    }

    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];
    }
```

Alg 4. Update AoS

```
void update_centroids(vector<Cluster> &clusters){
      for(int i=0;i<clusters.size();i++){
              clusters[i].update_coords();
              clusters[i].free_point();
     }

}
```

Alg 4.1 Update SoA

```
void update_centroids(vector<Cluster> &clusters){
    int num_clusters = clusters[0].size;

    for (int i = 0; i < num_clusters; i++)
    {
        if (clusters[0].point_count[i] > 0)
        {
            clusters[0].coord_x[i] /=
                clusters[0].point_count[i];
            clusters[0].coord_y[i] /=
                clusters[0].point_count[i];
        }
    }

}
```

In the SoA version, the code calculate 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.

3.3 Cuda

CUDA is a parallel computing platform that leverages the massive parallelism of GPUs to accelerate computations. GPUs are designed to perform thousands of computations simultaneously, making them ideal for applications that require intensive computation, such as scientific simulations, image and video processing, and machine learning. 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:

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

The **Alg 5.** shows the pseudocode of the main part of the algorithm

Alg 5. 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
```

The **Alg 6.** shows the pseudocode of the assignment

Alg 6. 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
```

The **Alg 7.** shows the pseudocode of the update

Alg 7. 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)
```

4 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 datasets used to evaluate the different implementations have been generated with the make_blob() function of sklearn.datasets [1]. 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.

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

In order to obtain statistically significant data, each iteration has been performed in experimental triplicate. The statistical mean of the three results was then calculated and reported as final results.

4.1 OpenMP

To evaluate the performances of the OpenMP implementation, it has been executed on each dataset with an increasing number of threads. The results for the $1000, 10000, 100000$ and 1000000 dataset are shown respectively in Table 1, Table 2, Table 4, Table 6 and Table 5.

Dim	1000			
Sequential	0.148 s			
Thread	Time Aos	Speedup AoS	Time SoA	Speedup SoA
2	0.025 s	5.92	0.021 s	7.05
4	**0.023** s	6.43	0.016 s	9.25
6	0.031 s	4.77	**0.012** s	12.33

Table 1. Speedup for 1000 points varying the number of threads with OpenMP (best result in bold)

Dim	10000			
Sequential	1.46 s			
Thread	Time Aos	Speedup AoS	Time SoA	Speedup SoA
2	0.219 s	$6,67$	0.138 s	10,57
4	0.234 s	$6,24$	**0.137** s	10,65
6	**0.215** s	$6,8$	0.138 s	10,58

Table 2. Speedup for 10000 points varying the number of threads with OpenMP (best result in bold)

Dim	100000			
Sequential	14.18 s			
Thread	Time Aos	Speedup AoS	Time SoA	Speedup SoA
2	2.24 s	6, 33	1.33 s	10,67
4	2.13 s	6, 65	1.55 s	9,14
6	**2.13** s	6, 65	**1.25** s	11,34

Table 3. Speedup for 100000 points varying the number of threads with OpenMP (best result in bold)

Dim	1000000			
Sequential	97.15 s			
Thread	Time Aos	Speedup AoS	Time SoA	Speedup SoA
2	10.26 s	9.46	6.27 s	15.49
4	9.63 s	10.09	5.89 s	16.49
6	**8.15** s	11.92	**5.58** s	17.41

Table 4. Speedup for 1000000 points varying the number of threads with OpenMP (best result in bold)

For the 1000 points dataset (table 1), it's curious to see that the speedup decreases for more than 4 threads. This shows how, for such a low number of points, the overhead of the threads outweighs the gain from using them. As expected, this phenomenon disappears in the datasets with more points and the use of more threads lowers the execution time (and consequently increases the speedup). The results show how OpenMP lets us reach a speedup equal to at least 15 at the expense of several pragma directive, so OpenMP proves to have an excellent speedup / development cost ratio.

4.2 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 s
128	**0.00548***s*
256	0.0093 s
512	0.011 s
1024	0.013 s

Table 5. Execution time for 10000 points dataset while changing the block dimension (best result in bold)

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

Table 5. Execution time for 1000, 10000,100000 and 10000000 points dataset with fixed block dimension at 128.

4.3 Comparison

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

Table 6. Global comparison between sequential, OpenMP and CUDA best results varying dataset dimension

5 Conclusions

In this work, the K-means clustering algorithm was presented and it was shown how its embarassingly parallel structure makes it suitable for parallel computing. A parallel implementation with OpenMP was developed by adding just a directive and it allows to obtain a speedup equal to more than 15 . Then, a CUDA implementation was presented and, with its 200 speedup, it showed how the use of GPUs makes K-means applicable to datasets intractable with a CPU.

6 References

[1] Ahmed, M.; Seraj, R.; Islam, S.M.S. The k-means Algorithm: A Comprehensive Survey and Performance Evaluation. Electronics 2020, 9, 1295.

[2] Robert Strzodka,Chapter 31 - Abstraction for AoS and SoA Layout in C++,Editor(s): Wen-mei W. Hwu,In Applications of GPU Computing Series,GPU Computing Gems Jade Edition,Morgan Kaufmann,2012.

[3] Using OpenMP : portable shared memory parallel programming / Barbara Chapman, Gabriele Jost, Ruud van der Pas.

[4] D. S. B. Naik, S. D. Kumar and S. V. Ramakrishna, "Parallel processing of enhanced K-means using OpenMP," 2013 IEEE International Conference on Computational Intelligence and Computing Research, Enathi, 2013, pp. 1-4, doi: 10.1109/ICCIC.2013.6724291.

[5] Hooda, Hanu, and Rainu Nandal. "Implementation of K-means clustering algorithm in CUDA." Int J Enhanc Res Manag Comput Appl 3 (2014): 829-833.

[6] C. Zeller, "Cuda c/c++ basics,"' NVIDIA Coporation, 2011.