



2D K-MEANS CLUSTERING

GPU LAB

Supervise by:
Mr. Gasim Mammadov

Submitted by:
Saiprasad Salkar (3507403)

Table of Contents

Table of Contents.....	1
ABSTRACT	2
PROBLEM STATEMENT	2
IMPLEMENTATION.....	2
RESULT	5
PERFORMANCE RESULT	6
WORK CONTRIBUTION	6
CONCLUSION.....	6
REFERENCES	6

ABSTRACT

K-means clustering is a method of vector quantization, originally from signal processing, that aims to partition n observations into k clusters in which each observation belongs to the cluster with the nearest mean (cluster centres or cluster centroid), serving as a prototype of the cluster. Here in this project, we are implementing this algorithm on OpenCL platform which includes parallelizing the different parts of algorithm to achieve the optimum runtime.

PROBLEM STATEMENT

Implement 2D k-means clustering. Write an OpenCL program to cluster a set of points using K-means. Consider $K=n$ clusters. Consider Euclidean distance as the distance measure. Randomly initialize clusters based on the $K=n$ clusters and iterate to get the optimum clustering for the input data points.

IMPLEMENTATION

Clustering is one of the most frequently utilized forms of unsupervised learning. In this project, we'll explore one of the most common forms of clustering i.e., K-means. To implement this algorithm, we need the data samples as input.



dataset.csv

The Algorithm for K-means clustering is implemented in the following steps:

1. The dataset is extracted and stored into the dataframe or feature_matrix.

```
////////////////////////////////////  
/* Get Data from CSV for GPU */  
////////////////////////////////////  
float* csv_to_float_matrix_GPU()  
{  
    std::ifstream csv_file;  
    csv_file.open("C:/University/Course_Subjects/Semester_3/Lab/Project_K_Means_Clustering/data/dataset.csv", std::ios::in);  
    char str[210];  
    int start = 0, end, row_num = 0, col_num, first = 1;  
    std::string row_values, substr, delim;  
    int nsamples = N_Samples, dims = Dimension;  
    float* feature_matrix = new float[nsamples * dims];  
    while (csv_file.getline(str, 210))  
    {  
        if (first)  
        {  
            first = 0;  
            continue;  
        }  
        row_values = str;  
        delim = ",";  
        start = 0;  
        end = row_values.find(delim);  
        col_num = 0;  
        while (end != std::string::npos)  
        {  
            substr = row_values.substr(start, end - start);  
            feature_matrix[(row_num * dims) + col_num] = stof(substr);  
            col_num++;  
            start = end + delim.length();  
            end = row_values.find(delim, start);  
        }  
        feature_matrix[(row_num * dims) + col_num] = stof(row_values.substr(start, end));  
        row_num++;  
        col_num++;  
    }  
    return feature_matrix;  
}
```

2. Once the feature matrix is available then we calculate the maximum value for each column. Example: If we have a N-Dimension data sample then there will be N maximum values.

```
__kernel void max_value_calc(__global float *dataframe, __global float *max_value)
{
    size_t j = get_global_id(0);
    float max = 0.0;

    for (int i = 0; i < samples; i++)
    {
        if (max < dataframe[(i*dim) + j])
        {
            max = dataframe[(i*dim) + j];
        }
    }
    max_value[j] = max;
}
```

3. We initialize clusters as per the input, if the number of clusters is 5 then we initialize 5 random clusters, Initialization of the clusters depend upon the maximum value of the columns like below:

```
__kernel void clusters_initialization(__global float *cluster_init, __global float *max_value)
{
    int j;
    size_t i = get_global_id(0);
    int rand = i+1;

    for(j = 0; j < dim; j++)
    {
        cluster_init[(i*dim) + j] = (max_value[j] * (rand^7)) / ((rand+1)^13);
    }
}
```

4. Once the initialization of cluster is complete then we calculate the Euclidean distance between the data samples and the initialized cluster points.

```
__kernel void calculate_distance(__global float *dataframe, __global float *clusters, __global float *distances)
{
    int i = get_global_id(0);
    printf("j=%d\n", i);
    int nclusters = get_global_size(0);
    int j, k;
    int cols = 16;
    int nsamples = 10000;
    for(j = 0; j < nsamples; j++)
    {
        float distance = 0;
        for(k = 1; k < cols; k++)
        {
            distance = distance + ((dataframe[(j*cols) + k] - clusters[(i*(cols-1)) + k - 1]) *
                                   (dataframe[(j*cols) + k] - clusters[(i*(cols-1)) + k - 1]));
        }
        distances[(j*nclusters) + i] = distance/cols;
    }
}
```

5. Thereafter we assign the samples to the cluster based on the Euclidean distance. This is done by calculating the minimum Euclidean distance between the samples and the n-clusters.

```
__kernel void assign_cluster(__global float *dataframe, __global float *distances,
                            __global float *clusters, __global float *cluster_number, __global int *changed)
{
    size_t id = get_global_id(0);
    int i, j, min, cluster, k;
    int count = 0;
    size_t num_clusters = get_global_size(0);
    float new_cluster[dim];


    for(i = 0; i < dim; i++)
    {
        new_cluster[i] = 0;
    }

    for(i = 0; i < samples; i++)
    {
        min = distances[(i*num_clusters)];
        cluster = 0;
        for(j = 0; j < num_clusters; j++)
        {
            if(min > distances[(i*num_clusters) + j])
            {
                min = distances[(i*num_clusters) + j];
                cluster = j;
            }
        }
    }
}
```


6. Then we iterate this cluster assignment to get the optimum clustering.

```
int flag = 0;
for(i = 0; i < dim; i++)
{
    if(count != 0)
    {
        new_cluster[i] /= count;
    }
    else
    {
        new_cluster[i] = 100;
    }
    if(new_cluster[i] != clusters[(id*(dim)) + i])
    {
        flag = 1;
    }
    clusters[(id*(dim)) + i] = new_cluster[i];
}
changed[id] = flag;
}
```

7. At last, we write all our results to a file which depicts the sample vs cluster group.



Kmeans_CPU_result.c
sv

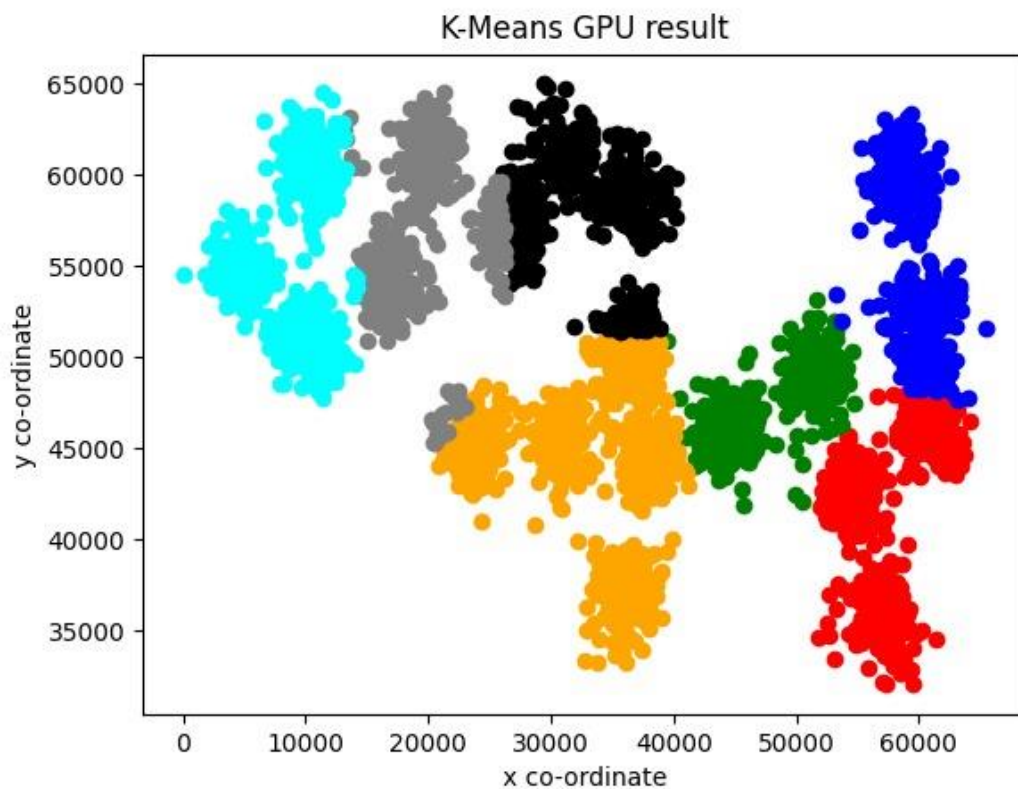
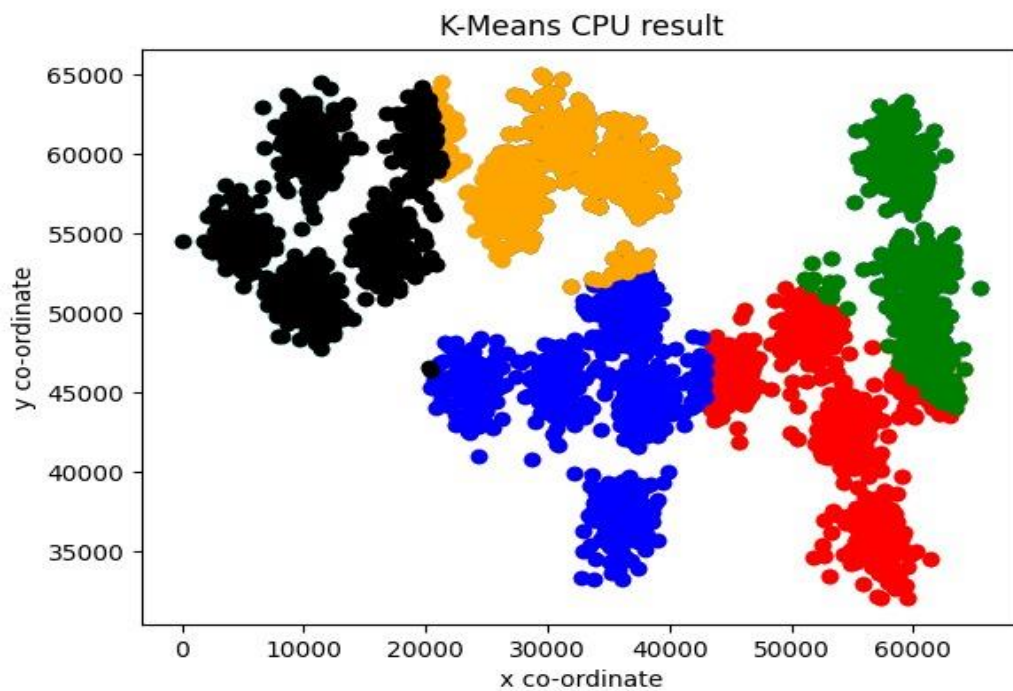


Kmeans_GPU_result.c
sv

RESULT

After running the K-means clustering algorithm we hereby find the cluster depicted in the below picture:

The picture has been plotted in python. And the results are for CPU as well as for GPU.



PERFORMANCE RESULT

Time performance on CPU and GPU for the K-means clustering algorithm:

```
Microsoft Visual Studio Debug Console
Number of iterations for assignment of cluster CPU:40
CPU_TIME:0.063
Using platform 'Intel(R) OpenCL HD Graphics' from 'Intel(R) Corporation'
Using device 1 / 1
Running on Intel(R) UHD Graphics
Number of iterations for assignment of cluster GPU:33
GPU Time: 0.006810s
```

WORK CONTRIBUTION

Independently, all the team members have understood and implemented the algorithm for K-means clustering. Taking reference from all our individual algorithms, the final algorithm has been integrated to achieve the optimum runtime on GPU.

CONCLUSION

We successfully completed K-means Clustering on GPU and the tried minimizing the execution time on GPU (speedup) and from the performance result we can see that the GPU time is 0.006810s

REFERENCES

1. <https://towardsdatascience.com/an-introduction-to-clustering-algorithms-in-python-123438574097>
2. https://en.wikipedia.org/wiki/K-means_clustering
3. <https://www.youtube.com/watch?v=4b5d3muPQmA>
4. https://www.youtube.com/watch?v=EIItUEPCIZM&t=471s&ab_channel=codebasics
5. <http://cs.joensuu.fi/sipu/datasets/>