K-Means Clustering Algorithm

Adriano Di Dio
E-mail address
adriano.didio@stud.unifi.it

Abstract

Purpose of this paper is to describe how the K-Means Clustering Algorithm works and how it can be implemented using a programming language.
In particular, it will be shown how to implement the main algorithm in C and how it can be sped up using multiple threads through the CUDA framework.

Future Distribution Permission

1. Introduction

K-Means Clustering is an algorithm which is used to partition a set of points into k clusters in which each point belongs to the closest cluster.
A point is said to be in a cluster when the distance from the center of the cluster,called centroid, is the smallest among all selected clusters.
The main algorithm can be divided into 4 main steps:

- Initialization Phase
- Assignment Step.
- Update Step.
- Convergence Check Step.

Details about each one will be given in the next chapters, in particular in chapter 2 a brief explanation of the algorithm is given and a sample implementation using pseudo-code is shown, then in chapter 3 and 4 the sequential and parallel version is described, and finally, in the last chapter, the performance will be evaluated by showing the differences between the sequential and parallel version.

2. Algorithm

K-Means algorithm, as stated in the introduction, is made of four steps that alternates until the last one, that represents the exit condition, tells the algorithm to stop since there are no updates to the centroids position.
In order to describe it we need to define a few key elements.
First we need to define the data-set as a group of d-dimensional points:

$$(x_1, x_2, x_3, \cdots, x_\mathrm{n}) \tag{1}$$

Then we need to define our centroids as a set made of k elements ($k \leq n$) (where k is the number of selected clusters):

$$S = \{S_1, S_2, S_3, \cdots, S_\mathrm{k}\} \tag{2}$$

the goal is to partition the data-set into k clusters to minimize the within-cluster distance between each point and the centroids.
The first step is the Initialization Phase, in this phase, the algorithm has to find some random points that will be used as centroids to initialize

it.
In this implementation the initial centroids are chosen from the starting k points contained in the data-set, this enable the algorithm to be predictable so that the differences between sequential and parallel version won't depends from the centroid's starting position,as long as the data-set is kept the same.
Then the clusters are calculated using the chosen centroid's position, by calculating the distance between each point and centroid, that are then evaluated in order to assign it to the nearest centroid:

$$S_i^{(t)} = \left\{ x_p : \left\| x_p - m_i^{(t)} \right\|^2 \leq \left\| x_p - m_j^{(t)} \right\|^2 \ \forall j, 1 \leq j \leq k \right\} \tag{3}$$

since we now have created a new cluster made of n-points, the centroids position needs to be updated by calculating the new mean value based on the position of all the points within that cluster:

$$m_i^{(t+1)} = \frac{1}{\left|S_i^{(t)}\right|} \sum_{x_j \in S_i^{(t)}} x_j \tag{4}$$

The algorithm, repeats these two steps until the updated centroids position is not so much different from the old one, in fact, if the distance between the new cluster position and the old one is smaller than a tolerance value, then the algorithm is said to have converged and the iterations can stops.
Below, a pseudo-code implementation of the algorithm:

2.1. Pseudo-Code

An example implementation can be seen below:

```
1:  procedure KMeansAlgorithm(Dataset[],n,k)
2:      for $i \leftarrow 1$ to $k$ do
3:          Centroid[i] = Dataset[i]
4:      while true do
5:          $SetClusters \leftarrow 0$
6:          for $i \leftarrow 1$ to $n$ do
7:              $Min \leftarrow$ -INF
8:              for $j \leftarrow 1$ to $k$ do
9:                  $Dist \leftarrow Dist(Dataset_i, Centroid_j)$
10:                 if $Dist < Min$ then
11:                     $Min \leftarrow Dist$
12:                     $Cluster_i \leftarrow Centroid_j$
13:         for $i \leftarrow 1$ to $k$ do
14:             $ClusterSize \leftarrow 0$
15:             $Sum \leftarrow 0$
16:             for $j \leftarrow 1$ to $n$ do
17:                 if $Cluster_i \neq i$ then
18:                     continue
19:                 $Sum \leftarrow Sum + Dataset_j$
20:                 $ClusterSize \leftarrow ClusterSize + 1$
21:             $Sum \leftarrow Sum/ClusterSize$
22:             $Delta \leftarrow VSub(Sum, Centroids_i)$
23:             if $Delta > Threshold$ then
24:                 continue
25:             $SetClusters \leftarrow SetCluster + 1$
26:         if $SetClusters == k$ then
27:             exit
```

Algorithm 1: KMeansAlgorithm Pseudo-Code

3. Implementation

3.1. Tools

Tools are used to generate the data-set and to plot the result of the algorithm.
They works only in 2D, are able to load/save only using the csv[1] file format and works only using python 3.

3.1.1 BlobGenerator

This python script is used to generate a bi-dimensional data-set that is saved in a csv file.

3.1.2 Usage

The usage is simple and requires just to call the script in a console:

```
$ python3 PyBlobGenerator.py
```

[1]Comma-Separated-Value

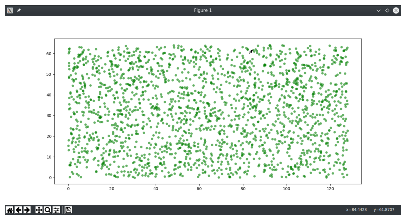


Figure 1: PyBlobGenerator

This, by default will generate a csv file named 'data_blob.csv' containing 2048 random points.
It is possible to specify the output file name:

```
$ python3
 PyBlobGenerator.py -o FileName.csv
```

and it can also generate an higher/lower number of points by using the parameters width and height.
These two parameters defines the region in which the point must be generated.
Finally it is possible to view the points in a plot by running the command:
$ python3 PyBlobGenerator.py -s
Full usage example:
$ python3 PyBlobGenerator.py -s -o OutFile.csv -w 128 -he 128
In this example the script will generate a random distribution of points within a 128x128 region,will save the output file in OutFile.csv and before saving will show a window containing the generated plot.

3.1.3 PlotGenerator

This python script is used to show the result of the K-Means Algorithm by loading up to n, where n is the number of steps to view, csv files from the 'In' folder.
The main program,in facts, outputs the result as a pair of file containing the data-set and the centroids.

3.1.4 Usage

It only takes one parameters which is the number of steps to show and will display a single windows containing up to n plots stacked horizontally,each displaying a specific step of the algorithm:

```
$ python3  PyPlotGenerator.py 1
```

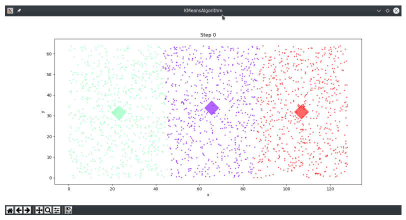


Figure 2: PyPlotGenerator

3.2. Main Program

The algorithm is implemented in C and, as it can be seen from the Tools, it uses a fixed data-set that is loaded from a csv file.
This is done in order to be able to compare the sequential and parallel version, and see the difference in terms of execution times,without having to worry about the possible differences that can happens due to a different choice of Centroids/-Points.
The program is able to read the csv file and infer the vector dimension, called 'Stride' in the source code,in order to determine if the algorithm has to run on 2D,3D etc...
The program takes two arguments: the path to the dataset file in csv format and the number of centroids.
E.G:

```
$ ./KMeans  Dataset.csv 10
```

3.2.1 Sequential Version

The sequential version is made using a single function that contains an infinite loop that repeats all the required step until the exit condition is hit.
The following structure have been declared to define a point and a centroid:

```c
typedef struct Point_s {
    float *Position;
    int CentroidIndex;
    int Stride;
} Point_t;

typedef struct Centroid_s {
    float *Position;
    int    Stride;
} Centroid_t;
```

Listing 1: Data Structure Definition

The code can be divided in different parts:

First we need to initialize the algorithm by choosing the centroids:

```c
    Centroids = malloc(sizeof(Centroid_t) *
        ↪ NumCentroids );
    for( i = 0; i < NumCentroids; i++ ) {
        Centroids[i].Position = malloc(Stride *
            ↪ sizeof(float));
        for( j = 0; j < Stride; j++ ) {
            Centroids[i].Position[j] = Dataset->
                ↪ Points[i].Position[j];
        }
        Centroids[i].Stride = Stride;
    }
```

Listing 2: Centroids Initialization

as it can be seen from the snippet above, the centroids list is initialized as an array containing K points, where each point has dimension equals to the Stride.

After initialization, the main loop is started and first step is executed:

```c
        for(i = 0; i < Dataset->NumPoints; i++) {
            Min = 99999;
            for(j = 0; j < NumCentroids; j++) {
                Distance = PointDistanceSquared(
                    ↪ Dataset->Points[i].
                    ↪ Position,Centroids[j].
                    ↪ Position,Stride);
                if( Distance < Min ) {
                    Dataset->Points[i].
                        ↪ CentroidIndex = j;
                    Min = Distance;
                }
            }
        }
```

Listing 3: Minimum Distance Calculation

in this step, the distance between each point of the data-set and the centroids list is evaluated using a simple function:

```c
float PointDistanceSquared(float *PointA,float *
    ↪ PointB,int Stride)
{
    float Sum;
    int i;
    Sum = 0.f;
    for( i = 0; i < Stride; i++ ) {
        Sum += (PointB[i] - PointA[i]) *
            (PointB[i] - PointA[i]);
    }
    return Sum;
}
```

Listing 4: Distance Calculation

that returns the distance, this is used to find the closest centroid for that point in order to build a new cluster.
After building the cluster list, the mean position of the cluster is calculated:

```c
        NumClustersSet = 0;
        for( int i = 0; i < NumCentroids; i++ ) {
            memset(Sum,0,Stride * sizeof(float));
            ClusterSize = 0;
            for( j = 0; j < Dataset->NumPoints; j
                ↪ ++ ) {
                if( Dataset->Points[j].
                    ↪ CentroidIndex != i ) {
                    continue;
                }
                for( k = 0; k < Stride; k++ ) {
                    Sum[k] += Dataset->Points[j].
                        ↪ Position[k];
                }
                ClusterSize++;
            }
            if( ClusterSize == 0.f ) {
                continue;
            }
            for( k = 0; k < Stride; k++ ) {
                Sum[k] /= ClusterSize;
            }
```

Listing 5: Distance Calculation

as it can be seen by the above code, we have to make sure that the cluster size is not zero, this is a special case that could occur when an higher number of clusters is selected.
Before updating the centroids position, the difference between this new position and the old one is calculated in order to see if we have reached the stop condition (centroids position not changing):

```c
            PointSubtract(Sum,Centroids[i].
                ↪ Position,Delta,Stride);
            for( k = 0; k < Stride; k++ ) {
                if( fabsf(Delta[k]) >
                    ↪ KMEANS_ALGORITHM_TOLERANCE
                    ↪ ) {
```

```
                break;
            }
        }
        if( k == Stride ) {
            NumClustersSet++;
        }
```

Listing 6: Distance Calculation

'kmeans_algorithm_tolerance' is a variable that define the max allowed difference between the two positions (the old centroid and the new calculated mean).
Finally the centroid position is updated:

```
            memcpy(Centroids[i].Position,Sum,
                ↪ Stride * sizeof(float));
```

Listing 7: Distance Calculation

when all centroids' positions have been updated the function check if all the centroids have not changed during last loop:

```
        if( NumClustersSet == NumCentroids ) {
            break;
        }
```

Listing 8: Distance Calculation

If the number of cluster that were not updated is equal to the number of centroids, then the algorithm has finished and the function proceed to save the data-set and the centroids into a csv file.

3.2.2 Parallel Version

Looking at the sequential version, we can see that many operations could benefit from a parallel implementation, since the work can be split-up in multiple threads.
The parallel version has been implemented using two frameworks:Cuda and OpenMP.
The differences between the two implementation are related to where the code is executed, while Cuda uses an NVidia GPU to schedule and run the threads, the OpenMP framework uses the CPU.

3.2.2.1 CUDA

The CUDA framework offers a simple API that can be used to launch thread organized in block and grids.
All the CUDA functions are checked using a macro that calls a function to look for any error when calling one of the Cuda function.
The parallel version is made of several function, each one having a specific task that is done by launching a kernel and waiting for his completion. The first step is the initialization, here, we need to initialize the data-set by copying it into the GPU:

```
float *CudaInitPointList(PointArrayList_t *
    ↪ PointList)
{
    float *DevicePointList;
    int    PointListSize;

    PointListSize = PointList->NumPoints *
        ↪ PointList->Stride * sizeof(float);
    CUDA_CHECK_RETURN(cudaMalloc((void **)&
        ↪ DevicePointList, PointListSize));
    CUDA_CHECK_RETURN(
        cudaMemcpy(DevicePointList,PointList->
            ↪ Points,
                PointListSize,
                    ↪ cudaMemcpyHostToDevice)
                    ↪ );
    return DevicePointList;
}
```

Listing 9: CUDA Data-Set Initialization

using built-in CUDA functions cudaMalloc and cudaMemcpy to respectively allocate the memory into the GPU and to transfer the data from host to device.
Then we need to initialize the centroids:

```
float *CudaInitCentroids(int NumCentroids,float *
    ↪ DevicePointList,int NumPoints,int Stride)
{
    float *DeviceCentroidOutputList;
    int    CentroidListSize;
    dim3   BlockSize;
    dim3   GridSize;

    CentroidListSize = NumCentroids * Stride *
        ↪ sizeof(float);

    CUDA_CHECK_RETURN(cudaMalloc((void**)&
        ↪ DeviceCentroidOutputList,
        ↪ CentroidListSize));

    BlockSize = dim3(64, Stride, 1);
    GridSize = dim3((NumCentroids + BlockSize.x -
        ↪  1) / BlockSize.x,1,1);

    CentroidsInitKernel<<<GridSize,BlockSize>>>
            (DeviceCentroidOutputList,
                ↪ NumCentroids,DevicePointList,
             NumPoints,Stride);

    return DeviceCentroidOutputList;
}
```

Listing 10: CUDA Centroids Initialization

this function launch a kernel with the given BlockSize and GridSize that sets the Centroids data using multiple threads:

```
__global__ void CentroidsInitKernel(float *
    ↪ Centroids,int NumCentroids, float *Points,
    ↪ int NumPoints,int Stride)
{
    int ThreadIndexX;
    int ThreadIndexY;
    ThreadIndexX = blockIdx.x * blockDim.x +
        ↪ threadIdx.x;
    ThreadIndexY = blockIdx.y * blockDim.y +
        ↪ threadIdx.y;
    if( ThreadIndexX < NumCentroids &&
        ↪ ThreadIndexY < Stride) {
        Centroids[ThreadIndexX * Stride +
            ↪ ThreadIndexY] = Points[
            ↪ ThreadIndexX * Stride +
            ↪ ThreadIndexY];
    }
}
```

Listing 11: CUDA Centroids Initialization

each centroid is set independently from each other by using an unique ThreadIndexX and ThreadIndexY to access the array.
Other variables, needed by the algorithm, are simply allocated on the device and a call to 'cudaMemset' is made to initialize it.
As an example, the snippet below show the initialization of the 'ClusterCounter' variable:

```
float *CudaInitClusterCounter(int NumCentroids)
{
    float *DeviceClusterCounter;
    int  ClusterCounterSize;

    ClusterCounterSize = NumCentroids * sizeof(
        ↪ float);

    CUDA_CHECK_RETURN(cudaMalloc((void **)&
        ↪ DeviceClusterCounter,
        ↪ ClusterCounterSize));

    CUDA_CHECK_RETURN(cudaMemset((void *)
        ↪ DeviceClusterCounter,0,
        ↪ ClusterCounterSize));

    return DeviceClusterCounter;
}
```

Listing 12: CUDA Example Initialization

Full code of the initialization phase is below:

```
    DevicePointList = CudaInitPointList(PointList
        ↪ );
    DeviceCentroidList = CudaInitCentroids(
        ↪ NumCentroids,DevicePointList,PointList
        ↪ ->NumPoints,Stride);
    OldCentroidListSize = NumCentroids * Stride *
        ↪  sizeof(float);
    CUDA_CHECK_RETURN(cudaMalloc((void **)&
        ↪ DeviceOldCentroidList,
        ↪ OldCentroidListSize));
    DeviceClusterCounter = CudaInitClusterCounter
        ↪ (NumCentroids);
    DeviceClusterList = CudaInitAssignments(
        ↪ PointList);
    DeviceDistanceList = CudaInitDistances(
        ↪ NumCentroids,PointList->NumPoints);
    SumSize = sizeof(int);
    CUDA_CHECK_RETURN(cudaMalloc((void **)&
        ↪ DeviceSum,SumSize));
```

Listing 13: CUDA Main Initialization Code

Then, as seen in the sequential version, the main loop is declared as a simple infinite while loop and the algorithm begins:

```
    while( 1 ) {
        CudaComputeDistances(DeviceDistanceList,
            ↪ DeviceCentroidList,NumCentroids,
            ↪ DevicePointList,PointList->
            ↪ NumPoints,Stride);
        CudaBuildClusterList(DeviceClusterList,
            ↪ DeviceDistanceList,PointList->
            ↪ NumPoints,NumCentroids,Stride);
        cudaMemcpy(DeviceOldCentroidList,
            ↪ DeviceCentroidList,
            ↪ OldCentroidListSize,
            ↪ cudaMemcpyDeviceToDevice);
        CudaUpdateCentroidList(DeviceCentroidList
            ↪ ,NumCentroids,DevicePointList,
            ↪ PointList->NumPoints,
            ↪ DeviceClusterList,
                            DeviceClusterCounter,
                                ↪ Stride);
        Sum = CudaCompareCentroidsList(DeviceSum,
            ↪ DeviceCentroidList,
            ↪ DeviceOldCentroidList,NumCentroids
            ↪ ,Stride);
        if( Sum == 1 ) {
            break;
        }
        Step++;
    }
```

Listing 14: CUDA Main Loop

The first step, called ‘CudaComputeDistances’ launches a kernel to calculate the distances between points and centroids:

```
void CudaComputeDistances(float *
    ↪ DeviceDistanceList,float *
    ↪ DeviceCentroidList,int NumCentroids,float
    ↪ *DevicePointList,
                                  int NumPoints,int
                                      ↪ Stride)
{
    dim3   BlockSize;
    dim3   GridSize;
    BlockSize = dim3(32,32,1);
    GridSize = dim3((NumPoints + BlockSize.x - 1)
        ↪  / BlockSize.x ,(NumCentroids +
        ↪ BlockSize.y - 1) / BlockSize.y ,1);
    ClusterComputeDistanceSquaredKernel<<<
        ↪ GridSize,BlockSize>>>
        (DeviceDistanceList,DeviceCentroidList,
            ↪ NumCentroids,DevicePointList,
            ↪ NumPoints,Stride);
}
```

Listing 15: CUDA Distance Calculator Kernel Launcher

the actual kernel is below:

```
__global__ void
    ↪ ClusterComputeDistanceSquaredKernel(float
    ↪ *Distances,
    float *Centroids,int NumCentroids,float *
        ↪ Points,int NumPoints,int Stride)
{
    int ThreadIndexX;
    int ThreadIndexY;
    int CentroidIndex;
    int DatasetIndex;
    int i;
    float LocalDistance;

    ThreadIndexX = blockIdx.x * blockDim.x +
        ↪ threadIdx.x;
    ThreadIndexY = blockIdx.y * blockDim.y +
        ↪ threadIdx.y;
    if( ThreadIndexX < NumPoints && ThreadIndexY
        ↪ < NumCentroids ) {
    LocalDistance = 0.f;
        for( i = 0; i < Stride; i++ ) {
            CentroidIndex = ThreadIndexY * Stride
                ↪  + i;
            DatasetIndex = ThreadIndexX  * Stride
                ↪  + i;
            LocalDistance += (Centroids[
                ↪ CentroidIndex] - Points[
                ↪ DatasetIndex]) *
                        (Centroids[CentroidIndex]
                            ↪  - Points[
                            ↪ DatasetIndex]);
        }
        Distances[ThreadIndexX * NumCentroids +
            ↪ ThreadIndexY] = LocalDistance;
    }
}
```

Listing 16: CUDA Distance Calculator Kernel

in the snippet above, the distance is calculated between a centroid and a point, the sum is first done locally by iterating over the vectors’ components and then stored in a shared array to be used later when calculating the minimum distance.
Then, the clusters are built by evaluating the calculated distances through another kernel:

```
__global__ void BuildClusterListKernel(int *
    ↪ Clusters,float *Distances,int NumPoints,
    ↪ int NumCentroids,int Stride)
{
    int ThreadIndexX = blockIdx.x * blockDim.x +
        ↪ threadIdx.x;
    int DistanceArrayBaseIndex = ThreadIndexX *
        ↪ NumCentroids;
    float Min;
    int MinIndex;
    int i;

    if( ThreadIndexX < NumPoints ) {
        Min = INFINITY;
        MinIndex = 0;
        for( i = 0; i < NumCentroids; i++ ) {
            if( Distances[DistanceArrayBaseIndex
                ↪ + i] < Min ) {
                Min = Distances[
                    ↪ DistanceArrayBaseIndex + i
                    ↪ ];
                MinIndex = i;
            }
        }
        Clusters[ThreadIndexX] = MinIndex;
    }
}
```

Listing 17: CUDA Cluster Builder Kernel

each point selects the nearest centroid and saves it into a shared array containing the built cluster list.
Now, we need to update the centroids position, by using the newly built clusters, but before doing so, we need to save the old centroid list in order to be able to evaluate if the centroids position have changed or we have reached the stop condition:

```
        cudaMemcpy(DeviceOldCentroidList,
            ↪ DeviceCentroidList,
            ↪ OldCentroidListSize,
            ↪ cudaMemcpyDeviceToDevice);
```

Listing 18: CUDA Centroid List Saving

after saving it,we can calculate the new position by first clearing out the centroid list array:

```
    CUDA_CHECK_RETURN(cudaMemset((void *)
        ↪ DeviceCentroidList,0,NumCentroids *
        ↪ Stride * sizeof(float)));
    CUDA_CHECK_RETURN(cudaMemset((void *)
        ↪ DeviceClusterCounter,0,NumCentroids *
        ↪ sizeof(float)));
```

Listing 19: CUDA Centroid Position Update

then, we first start by calculating the sum of all points belonging to each cluster:

```
__global__ void SumPointsInClustersKernel(float *
    ↪ Centroids,int NumCentroids,float *
    ↪ ClusterCounter,int *Clusters,float *Points
    ↪ ,int NumPoints,int Stride)
{
    int ThreadIndexX;
    int ThreadIndexY;
    int CentroidIndex;
    ThreadIndexX = blockIdx.x * blockDim.x +
        ↪ threadIdx.x;
    ThreadIndexY = blockIdx.y * blockDim.y +
        ↪ threadIdx.y;
    if( ThreadIndexX < NumPoints && ThreadIndexY
        ↪ < Stride ) {
        CentroidIndex = Clusters[ThreadIndexX];
        atomicAdd(&(Centroids[CentroidIndex *
            ↪ Stride + ThreadIndexY]),Points[
            ↪ ThreadIndexX * Stride +
            ↪ ThreadIndexY]);
        atomicAdd(&(ClusterCounter[CentroidIndex
            ↪ ]), 1.f);
    }
}
```

Listing 20: CUDA Mean Sum Calculation

in this function we need to use an atomic instruction since multiple threads can access the array for updating it in parallel, without it, a data race condition could occur.
Finally the actual mean is calculated by launching another kernel:

```
__global__ void MeanPointsInClustersKernel(float
    ↪ *Centroids,int NumCentroids,float *
    ↪ ClusterCounter,int Stride)
{
    int ThreadIndexX;
    int ThreadIndexY;
    int NumClusters;

    ThreadIndexX = blockIdx.x * blockDim.x +
        ↪ threadIdx.x;
    ThreadIndexY = blockIdx.y * blockDim.y +
        ↪ threadIdx.y;
    NumClusters = ClusterCounter[ThreadIndexX];
    if( ThreadIndexX < NumCentroids &&
        ↪ NumClusters != 0) {
        Centroids[ThreadIndexX * Stride +
            ↪ ThreadIndexY] /= ( ClusterCounter[
            ↪ ThreadIndexX] / Stride );
    }
}
```

Listing 21: CUDA Mean Calculation

where the actual mean is calculated, in this function we need to check if the number of cluster is not zero, due to the possibility of a cluster being empty.
Finally before starting a new iteration of the loop the exit condition is evaluated by launching a kernel that has to check if the old centroids list is similar to the new centroids list.

```
__global__ void CompareCentroidsKernel(int *Sum,
    ↪ float *Centroids,float *OldCentroids,int
    ↪ NumCentroids,int Stride)
{
    int ThreadIndexX;
    int ThreadIndexY;
    float Delta;
    int Value;
    ThreadIndexX = blockIdx.x * blockDim.x +
        ↪ threadIdx.x;
    ThreadIndexY = blockIdx.y * blockDim.y +
        ↪ threadIdx.y;

    if( ThreadIndexX < NumCentroids ) {
        Delta = fabsf(Centroids[ThreadIndexX *
            ↪ Stride + ThreadIndexY] -
            ↪ OldCentroids[ThreadIndexX * Stride
            ↪  + ThreadIndexY]);
        Value = Delta <
            ↪ KMEANS_ALGORITHM_TOLERANCE ? 1 :
            ↪ 0;
        atomicAdd(Sum,Value);
    }
}
```

Listing 22: CUDA Exit Condition Calculator Kernel

this kernel evaluates all the centroids elements in the old and new list by calculating the difference between the elements.
If the difference is lower than the tolerance value, then a value of 1 is added to the Sum array otherwise 0.
When the kernel completes, the sum variable is copied from Device to host:

```
    return Sum == (NumCentroids * Stride) ? 1 :
        ↪ 0;
```

Listing 23: CUDA Exit Condition Sum Copy

and returned, full function can be seen below:

```
int CudaCompareCentroidsList(int *DeviceSum,float *DeviceCentroidList,float * DeviceOldCentroidList,int NumCentroids,int Stride)
{
    dim3   BlockSize;
    dim3   GridSize;
    int    Sum;

    BlockSize = dim3(256,Stride,1);
    GridSize = dim3((NumCentroids + BlockSize.x - 1) / BlockSize.x,1,1);

    CUDA_CHECK_RETURN(cudaMemset((void *)DeviceSum,0,sizeof(int)));
    CompareCentroidsKernel<<<GridSize,BlockSize>>>(DeviceSum,DeviceCentroidList,DeviceOldCentroidList,NumCentroids,Stride);

    CUDA_CHECK_RETURN(cudaMemcpy(&Sum,DeviceSum,sizeof(int),cudaMemcpyDeviceToHost));

    return Sum == (NumCentroids * Stride) ? 1 : 0;
}
```

Listing 24: CUDA Exit Condition Kernel Launcher

as it can be seen above, the function returns 0 if the centroids positions have changed during last iteration or 1 if not.
In the main loop, after calling it, we check to see if we can stop by comparing the returned value to 1:

```
        Sum = CudaCompareCentroidsList(DeviceSum,DeviceCentroidList,DeviceOldCentroidList,NumCentroids,Stride);
        if( Sum == 1 ) {
            break;
        }
```

Listing 25: CUDA Exit Condition

if it is true then the program stops the main loop,saves the result in a csv file and exit.

3.2.2.2 OpenMP

The OpenMP implementation is written starting from the sequential version, by adding some OpenMP clauses to instruct the library to use multiple threads.
The first step is to initialize the centroids, as seen in the previous implementations, we take the first k points from the dataset, where k is the number of clusters selected:

```
    #pragma omp parallel for private(j) num_threads(MaxThreadNumber) schedule(static, (NumCentroids*Stride)/MaxThreadNumber)
    for( i = 0; i < NumCentroids; i++ ) {
        for( j = 0; j < Stride; j++ ) {
            Centroids[i * Stride + j] = Dataset->Points[i * Stride +j];
        }
    }
```

Listing 26: OpenMP Centroid Initialization

as we can see from the above code snippet, OpenMP uses a different approach, rather than having a library that exposes some functions, OpenMP uses pre-processor directives to instruct the compiler to use multiple threads, there are some function declared as normal C function like the one required in our code to get the maximum number of hardware threads using the function:

```
    MaxThreadNumber = omp_get_max_threads();
```

Listing 27: OpenMP Get Max Number of Threads Function

this number will be used throughout the implementation to optimize the thread usage.
In this specific case, we run multiple threads using a static scheduler, where each threads gets a chunk based on the number of centroids and the stride (the maximum dimension of the vector) and divides it by the maximum number of hardware threads.
After initialization, we need to calculate the distances between each point and each centroid:

```
        #pragma omp parallel for private(j,k) num_threads(MaxThreadNumber) \
        schedule(static, (Dataset->NumPoints*NumCentroids)/MaxThreadNumber)
        for( i = 0; i < Dataset->NumPoints; i++ ) {
            for( j = 0; j < NumCentroids; j++ ) {
                float LocalDistance = 0.f;
                for( k = 0; k < Stride; k++ ) {
                    LocalDistance += (Centroids[j * Stride + k]
                        - Dataset->Points[i * Stride + k]) *
                        (Centroids[j * Stride + k ] - Dataset->Points[i * Stride + k]);
                }
                Distances[i * NumCentroids + j] =
                    ↪  LocalDistance;
            }
        }
```

Listing 28: OpenMP Distance Calculation

as we can see from the above code, we need to run three nested for loops to perform this calculation, in this instance, OpenMP is instructed to use a static scheduler as seen in the previous code, which causes OpenMP to divide the work among threads into chunk of size equal to the product of the maximum allowed iteration of the three loops,to calculate all the required distances.
After calculating all the distances we need, for each point in the dataset, to find the closest centroid:

```
        #pragma omp parallel for schedule(guided)
            ↪  \
            shared(Distances,Clusters,
                ↪ ClusterCounter) private(j)
        for( i = 0; i < Dataset->NumPoints; i++ )
            ↪  {
            float Min = INFINITY;
            int Index = -1;
            for( j = 0; j < NumCentroids; j++ ) {
                float LocalDistance = Distances[i
                    ↪  * NumCentroids + j];
                if( LocalDistance < Min ) {
                    Min = LocalDistance;
                    Index = j;
                }
            }
            #pragma omp atomic write
            Clusters[i] = Index;
            #pragma omp atomic
            ClusterCounter[Index]++;
        }
```

Listing 29: OpenMP Nearest Centroid Calculation

in the above code, we can see that the scheduler has changed to a new type called ‘guided’, this scheduler divides the work into multiple chunks of decreasing size until all the calculations have been performed, after each iteration of the centroid’s array, the closest one is found and saved into a shared array along with the number of points that belongs to that specific cluster.
It is important to note that, since the cluster’s array is shared we need to make sure that each thread exclusively access it by using the directive ‘atomic’ that makes sure that only one thread at time can access the shared resources.

After creating the new clusters, we need to update the centroids position:

```
        memset(ClusterMeans,0,ClusterMeansSize);
        #pragma omp parallel for firstprivate(
            ↪ ClusterCounter) shared(
            ↪ ClusterMeans)
        for( int i = 0; i < Dataset->NumPoints *
            ↪ Stride; i++ ) {
            int PointIndex = i / Stride;
            int StrideIndex = i % Stride;
            int CentroidIndex = Clusters[
                ↪ PointIndex];
            int LocalAddValue = Dataset->Points[
                ↪ PointIndex * Stride +
                ↪ StrideIndex];
            #pragma omp atomic
            ClusterMeans[CentroidIndex * Stride +
                ↪  StrideIndex] += LocalAddValue
                ↪ ;
        }
```

Listing 30: OpenMP Mean Calculation I

as we can see from the above code, we starts by calculating the sum of all positions of the points belonging to a specific cluster, and, since the ‘ClusterMean’ array is shared we need to make sure that there are no data-races by adding the atomic instruction.
After performing the sum of all points we need to divide it by the number of points contained in each cluster to obtain the mean value:

```
        #pragma omp parallel for firstprivate(
            ↪ ClusterCounter) shared(
            ↪ ClusterMeans)
        for( int i = 0; i < NumCentroids * Stride
            ↪ ; i++ ) {
            int CentroidIndex = i / Stride;
            int StrideIndex = i % Stride;
            int NumClusters = ClusterCounter[
                ↪ CentroidIndex];
            if( NumClusters == 0 ) {
                continue;
            }
            #pragma omp atomic
            ClusterMeans[CentroidIndex * Stride +
                ↪  StrideIndex] /= (float)
                ↪ NumClusters;
        }
```

Listing 31: OpenMP Mean Calculation II

the code is similar to the previous one, only difference is the final operation being a division rather than a sum.
Finally, we need to see if we have reached our exit condition by comparing the new centroid’s

position with the old one:

```
        Sum = 0.f;
        #pragma omp parallel for shared(
            ↪ ClusterMeans,Centroids) reduction
            ↪ (+: Sum)
        for( i = 0; i < NumCentroids * Stride; i
            ↪ ++ ) {
            float Delta;
            float Value;
            int CentroidIndex = i / Stride;
            int StrideIndex = i % Stride;
            Delta = fabsf(ClusterMeans[
                ↪ CentroidIndex * Stride +
                ↪ StrideIndex] - Centroids[
                ↪ CentroidIndex * Stride +
                ↪ StrideIndex]);
            Value = Delta <
                ↪ KMEANS_ALGORITHM_TOLERANCE ?
                ↪ 1.f : 0.f;
            Sum = Sum + Value;
        }
```

Listing 32: OpenMP Centroid's Position Comparison

the above snippet, uses a different directive to divide the work among the threads called a reduction.
The algorithm is simple, for each centroid, the difference between the old value and the new is compared to a tolerance value, if the difference is less than the tolerance value, a value of one is added to the sum variable otherwise zero, the exit condition occurs, when after iterating over all centroids, the sum variable is equal to the Number of centroids times the stride:

```
        if( Sum == NumCentroids * Stride ) {
            break;
        }
```

Listing 33: OpenMP Exit Condition

as we can see from the above snippet, if it is the case then the two arrays are said to be similar and the program can terminate his execution, otherwise a new iteration is done until this condition is hit.

4. Metrics

In order to see what advantages the parallel implementation has brought we need to measure the execution time of both implementations by using the same data-set as well as the same number of clusters.
Run-Times were collected using an 'Intel(R) Core(TM) i5-4200M' for the c and OpenMP version while the CUDA version used an 'NVidia GeForce GT 720M' that has a compute capability of 2.1.
The execution time is measured using the same function:

```
int StartSeconds = 0;
int SysMilliseconds()
{
    struct timeval tp;
    int CTime;
    gettimeofday(&tp, NULL);
    if ( !StartSeconds ){
        StartSeconds = tp.tv_sec;
        return tp.tv_usec/1000;
    }
    CTime = (tp.tv_sec - StartSeconds)*1000 + tp.
        ↪ tv_usec / 1000;
    return CTime;
}
```

Listing 34: Time Function

that measure the elapsed time in milliseconds, using the function 'gettimeofday()' and the results are available in the following table:

Points	Clusters	C	OpenMP	Cuda
2048	256	232 ms	89 ms	136 ms
2048	500	582 ms	258 ms	240 ms
32768	10	292 ms	287 ms	72 ms
32768	250	48046 ms	6153 ms	11781 ms
50000	250	47392 ms	9867 ms	13741 ms
50000	512	103881 ms	24345 ms	33461 ms

Table 1: Execution Times

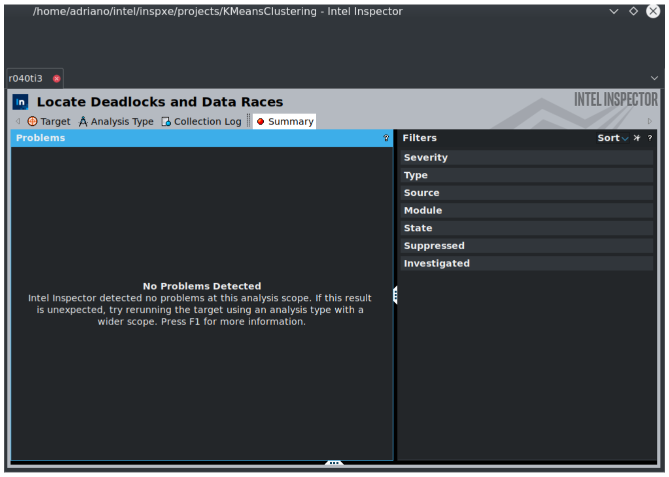
Speed-Up is calculated in the next table:

Points	Clusters	OpenMP	Cuda
2048	256	2.60	1.70
2048	500	2.25	2.42
32768	10	1.01	4.05
32768	250	7.80	4.07
50000	250	4.80	3.44
50000	512	4.26	3.10

Table 2: Speed-Up

as we can see from the table above, both par-

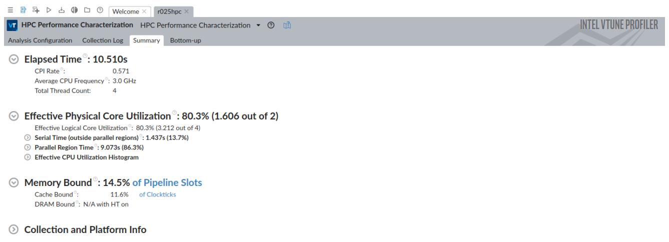
allel version performs better than the sequential one even in simple cases like the first.
We can also see that, with an higher number of Points/Clusters the parallel version is much faster than the sequential one. All the three versions have been tested using valgrind to show any memory leaks, and none was reported.
The OpenMP version was debugged using 'Intel Inspector' which reported no data races:



No Problems Detected
Intel Inspector detected no problems at this analysis scope. If this result is unexpected, try rerunning the target using an analysis type with a wider scope. Press F1 for more information.

Figure 3: Intel Inspector

as it can be seen in the above image.
OpenMP code was also profiled using Intel's Vtune Profiler that showed optimal core usage when trying to run the last test case (50000 points and 512 clusters):



Elapsed Time: 10.510s
CPI Rate: 0.571
Average CPU Frequency: 3.0 GHz
Total Thread Count: 4

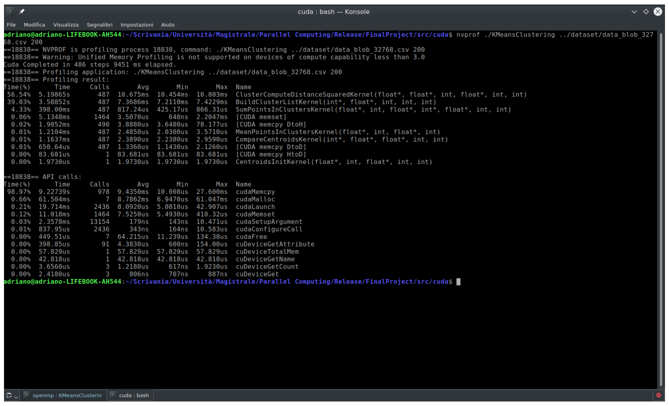
Effective Physical Core Utilization: 80.3% (1.606 out of 2)
Effective Logical Core Utilization: 80.3% (3.212 out of 4)
Serial Time (outside parallel regions): 1.437s (13.7%)
Parallel Region Time: 9.073s (86.3%)
Effective CPU Utilization Histogram

Memory Bound: 14.5% of Pipeline Slots
Cache Bound: 11.6% of Clockticks
DRAM Bound: N/A with HT on

Collection and Platform Info

Figure 4: Intel Vtune Profiler

Finally, the CUDA version was profiled using NVidia's profiler called nvprof:



```
adriano@adriano-LIFEBOOK-AH544:~/Scrivania/Università/Magistrale/Parallel Computing/Release/FinalProject/src/cuda$ nvprof ./KMeansClustering ../dataset/data_blob_327
68.csv 200
==18838== NVPROF is profiling process 18838, command: ./KMeansClustering ../dataset/data_blob_32768.csv 200
==18838== Warning: Unified Memory Profiling is not supported on devices of compute capability less than 3.0
Cuda Completed in 486 steps 9451 ms elapsed.
==18838== Profiling application: ./KMeansClustering ../dataset/data_blob_32768.csv 200
==18838== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 56.54%  5.19865s       487  10.675ms  10.454ms  10.803ms  ClusterComputeDistanceSquaredKernel(float*, float*, int, float*, int, int)
 39.03%  3.58852s       487  7.3686ms  7.2110ms  7.4229ms  BuildClusterListKernel(int*, float*, int, int, int)
  4.33%  398.00ms       487  817.24us  425.17us  866.31us  SumPointsInClustersKernel(float*, int, float*, int*, float*, int, int)
  0.06%  5.1348ms      1464  3.5070us     640ns  2.2047ms  [CUDA memset]
  0.02%  1.9052ms       490  3.8880us  3.6480us  78.177us  [CUDA memcpy DtoH]
  0.01%  1.2104ms       487  2.4850us  2.0300us  3.5710us  MeanPointsInClustersKernel(float*, int, float*, int)
  0.01%  1.1637ms       487  2.3890us  2.2380us  2.9590us  CompareCentroidsKernel(int*, float*, float*, int, int)
  0.01%  650.64us       487  1.3360us  1.1430us  2.1260us  [CUDA memcpy DtoD]
  0.00%  83.681us         1  83.681us  83.681us  83.681us  [CUDA memcpy HtoD]
  0.00%  1.9730us         1  1.9730us  1.9730us  1.9730us  CentroidsInitKernel(float*, int, float*, int, int)

==18838== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 98.97%  9.22739s       978  9.4350ms  10.008us  27.600ms  cudaMemcpy
  0.66%  61.504ms         7  8.7862ms  6.9470us  61.047ms  cudaMalloc
  0.21%  19.714ms      2436  8.0920us  5.0810us  42.907us  cudaLaunch
  0.12%  11.018ms      1464  7.5250us  5.4930us  410.32us  cudaMemset
  0.03%  2.3578ms     13154     179ns     143ns  10.471us  cudaSetupArgument
  0.01%  837.95us      2436     343ns     164ns  10.583us  cudaConfigureCall
  0.00%  449.51us         7  64.215us  11.239us  134.38us  cudaFree
  0.00%  398.85us        91  4.3830us     600ns  154.00us  cuDeviceGetAttribute
  0.00%  57.829us         1  57.829us  57.829us  57.829us  cuDeviceTotalMem
  0.00%  42.818us         1  42.818us  42.818us  42.818us  cuDeviceGetName
  0.00%  3.6560us         3  1.2180us     617ns  1.9230us  cuDeviceGetCount
  0.00%  2.4180us         3     806ns     707ns     887ns  cuDeviceGet
adriano@adriano-LIFEBOOK-AH544:~/Scrivania/Università/Magistrale/Parallel Computing/Release/FinalProject/src/cuda$ 
```

Figure 5: NVidia NVProf

that, as it can be seen from the above image, highlights the top kernels that required most of the time.
In particular the kernels 'ClusterComputeDistanceSquaredKernel' and 'BuildClusterListKernel' are slower than the others.
These two function are slower since they need to calculate and then find the nearest centroid for each point in the dataset.
This means that the time required to perform these two calculations will grow linearly as the number of points or centroids increases.