Sequential and Parallel implementation of k-means algorithm in C++, OpenMP and CUDA

Mihail Teodor Gurzu
7060245
mihail.gurzu@stud.unifi.it

Abstract

This report describes the implementation of the K-means algorithm using C++ for the sequential version and OpenMP, CUDA for the parallel versions. Are presented the ressults of a series of tests on the three implementations showing the differences between them in terms of computation time and speedup at vary of the number of points and centroids.

Future Distribution Permission

1. Introduction

K-means algorithm is an *unsupervised* machine learning algorithm which aims to partition or classify a set of data items into a certain class or cluster, based on data properties and characteristics. Each class or cluster, is represented by a *centroid*, which describes the characteristics the data must have or approximate such to make pat of the class represented by the centroid, which can be a real element from the dataset but not necessarily. The algorithm uses a *metric* to calculate the *"distance"* between each data item and the centroids, assigning the data item to the *"nearest"* centroid.

In this implementation, the algorithm is used for classifying a set of *2D points* to a certain number of *k* clusters. The metric used for assigning the data points to a certain cluster is the *Euclidean distance*, such that every centroid represents the corresponding clusters center of mass. The algorithm follows an iterative procedure:

1. *Initialization:* generate *N* 2D points dataset and *K* centroids randomly chosen from the generated dataset.
2. *Assignment:* for each point in the dataset calculate the *distance* from each centroid. Assign the point to the *minimum distant* centroid. As said, in this application *Euclidean distance* is used as metric.
3. *Update:* update each centroid's characteristics. In this application, the update phase is represented by each clusters center of mass.
4. *Check convergence:* check the stop criterion which in this application is reached when the after an assignment phase all clusters center of mass doesn't change or the maximum iterations number is reached, whichever happens first.
5. Repeat steps *2 - 4* until the stop criterion is reached.

2. Implementation

Data structures

As already stated, the application processes 2D points, assigning them to *K* clusters. Each point is represented as a *struct*:

```
struct Point {
  double x, y;
  int cluster;

  Point() : x(0.0), y(0.0), cluster(0){}
  Point(double x, double y, int c = 0) : x(
      x), y(y), cluster(c){}
}
```

The points and centroids are stored in a *std::vector<Point>* which is a sequence container representing arrays, very efficient accessing it's elements.

These data structures are common to the *C++* sequential version and the *OpenMP* parallel version since the identical code structure of these two implementations. Regarding the *CUDA* parallel version, the data structures used are different and will be discussed in the corresponding section.

Follows some general utility functions which are common to all implemented versions of the application.

Dataset generation

The application processes a certain dataset of size *N*, where N is indicated by the user, which if not available is generated by the application using a *uniform real distribution* with values range from 0 to 1:

```
void generate_random_dataset(int nr_points)
```

this function checks if the specified size dataset is already available and if not so generates *nr_points* 2D elements and stores them in a *rand_dataset_(nr_points).txt* file in a certain directory.

Load Dataset

The application loads a specified dataset by the user using the following function:

```
void load_dataset(std::vector<Point> &
    dataset, std::ifstream &dataset_file)
```

This function takes in input a *Point* vector, which will store the dataset to be processed and the path to file containing the raw dataset.

Point is a struct:

the *load_dataset()* function will fulfill the *vector<Point>* with a *struct* for each generated point using the second constructor listed above. Initially all points belong to cluster *0* which is a default cluster.

Centroids generation

After having loaded the dataset and stored the Points in a vector, follows the centroid generation, selecting *K* centroids randomly from the Points vector:

```
std::vector<Point>
    generate_initial_centroids(const std::
    vector<Point> &dataset, const long
    num_clusters)
```

As we can see, the function takes in input the Points vector and the number of the clusters (defined by the user) and returns a *std::vector<Point>* containing the *num_clusters* centroids randomly chosen from the Points vector.

2.1. Sequential implementation

The sequential version of the application uses the data structures described in section 2.

The sequential version takes in input the vectors containing the points and the centroids, and the *max_epochs* variable which represents the maximum number of iterations:

```
tuple<std::vector<Point>, vector<Point>>
    k_means_SEQ(vector<Point> points, vector
    <Point> centroids, int max_epochs)
```

the function returns the vector of points, each one having it's *cluster* attribute set accordingly to the cluster's id of belonging, and the centroids vector containing the centroids final coordinates.

The sequential version is composed essentially of two methods:

- *assign_clusters()* method, which assigns each Point to a cluster. This method iterates over each point and assigns the point to the *'nearest'* centroid as described in the introduction of this report in section 1.

  ```
  void assign_clusters(vector<Point>*
      points, vector<Point>* centroids,
      vector<Point>* centroids_details)
  ```

  this method takes in input the points and centroids vectors, and also the *centroids_details* vector which will store the data useful to update the centroids later. In particular, the vector contains for each centroid, the sum of x and y coordinates of each point and the number of total points assigned to the corresponding cluster.

- *update_centroids()* method, which updates the centroids coordinates after each assignment phase:

```
bool update_centroids(vector<Point>*
    centroids, vector<Point>*
    centroids_details)
```

this method takes in input the centroids and centroids_details vectors, and after recalculating and updating the new coordinates of each centroid, it returns a *boolean* variable indicating:

  - *true*: the centroids coordinates have changed so another iteration is necessary.
  - *false*: the centroids coordinates have not changed meaning that the algorithm converged to a solution.

this variable is used in the *Check convergence* phase as described in section 1.

Here is how these two methods work together:

```
tuple<std::vector<Point>, vector<Point>>
   k_means_SEQ(vector<Point> points, vector
   <Point> centroids, int max_epochs)
{
  vector<Point> centroids_details (
     centroids.size());
  bool iterate;
  int iteration = 0;
  do
  {
    iteration ++;

    assign_clusters(&points, &centroids, &
        centroids_details);

    iterate = update_centroids(&centroids,
        &centroids_details);

  }
  while(iterate && iteration < max_epochs);
  return {points, centroids};
}
```

2.2. OpenMP implementation

OpenMP is a implicit threading framework consisting of a set of compiler directives, library routines, and environment variables that influence run-time behavior. It's major characteristic is that the programmer doesn't have to change the structure of a sequential *C++* code in order to transform it in a parallel version. This framework is best at parallelizing *for* loops computing a *SPMD* (single program multiple data), once is guaranteed:

- *Sequential consistency:* the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

- *Bernstein's Conditions:* statements or operations can be interchanged without modifying the program results.

That being said, parallelizing the sequential version of this application using OpenMP consists in a set of directives to the compiler. Each of the methods described in the sequential implementation section has been parallelized by adding these directives:

assign_clusters()

This method has to iterate on each point and calculate the distances between every centroid, updating after processing each point the *centroids_details* vector. Since each Point will be processed by a single thread and since the distance calculations needs only reading the actual coordinates of the centroids we can use the *#pragma omp for* directive which tells the compiler that the loop can be parallelized with the available threads.

Several threads will modify the *centroids_details* vector, since many points will be assigned to a certain centroid, hence we need to synchronize the access to this vector. This can be achieved by using the *#pragma omp atomic* directive which ensures that a specific storage location is accessed atomically, avoiding this way the possibility of race conditions.

```
void assign_clusters_OPENMP(vector<Point>*
    points, vector<Point>* centroids, vector
    <Point>* centroids_details)
{
#pragma omp for
  for(int i = 0; i < (*points).size(); i++)
  {
    int bestCluster = 0;
    auto minDist = __DBL_MAX__;
    for(int j = 0; j < (*centroids).size();
        j++)
    {
      double dist = distance((*points)[i],
          (*centroids)[j]);
      if (dist <= minDist)
      {
        minDist = dist;
        bestCluster = j;
      }
    }
    (*points)[i].cluster = bestCluster;
#pragma omp atomic
    (*centroids_details)[bestCluster].x +=
        (*points)[i].x;
#pragma omp atomic
    (*centroids_details)[bestCluster].y +=
        (*points)[i].y;
#pragma omp atomic
    (*centroids_details)[bestCluster].
        cluster += 1;
  }
}
```

update_centroids()

This method iterates on each centroid and updates it's coordinates. Finally it resets the *centroids_details* vector to be ready for the next iteration. Since each index of *centroids* and *centroids_details* is processed by a single thread there's no need for synchronization.

It's good to say that parallelizing this method has not such a big impact as parallelizing the *assign_clusters()* method. Specially when the number of clusters is very low, we risk that the overhead implied by the thread management will overcome the benefits from parallelizing the code.

However, the tests of the application have been made computing even with a 1000 clusters, so it has been parallelized this method too.

```
bool update_centroids_OPENMP(vector<Point>*
     centroids, vector<Point>*
    centroids_details)
{
  double centroid_x_new, centroid_y_new,
      centroid_x_old, centroid_y_old = 0;
  bool iterate = false;
#pragma omp for
  for(int i = 0; i < centroids->size(); i
      ++)
  {
    centroid_x_new = (*centroids_details)[i
        ].x / (*centroids_details)[i].
        cluster;
    centroid_y_new = (*centroids_details)[i
        ].y / (*centroids_details)[i].
        cluster;

    centroid_x_old = (*centroids)[i].x;
    centroid_y_old = (*centroids)[i].y;

    if(centroid_x_old != centroid_x_new ||
        centroid_y_old != centroid_y_new)
    {
      iterate = true;
      (*centroids)[i].x = centroid_x_new;
      (*centroids)[i].y = centroid_y_new;
    }

    (*centroids_details)[i].x = 0;
    (*centroids_details)[i].y = 0;
    (*centroids_details)[i].cluster = 0;
  }
return iterate;
}
```

Finally, the main function has been parallelized by adding the following directive:

```
...
do{
  iteration++;

#pragma omp parallel num_threads(nr_threads
    ) default(none) shared(points, centroids
    , centroids_details, iterate)
    {
      assign_clusters_OPENMP(&points, &
          centroids, &centroids_details);

      iterate = update_centroids_OPENMP(&
          centroids, &centroids_details);
    }
} while(iterate && iteration < max_epochs);
     centroids_details, iterate)
...
```

2.3. CUDA implementation

CUDA (Compute Unified Device Architecture) is a parallel computing platform and application programming interface (API) model created by Nvidia. It allows software developers and software engineers to use a CUDA-enabled graphics processing unit (GPU) for general purpose processing – an approach termed GPGPU (General-Purpose computing on Graphics Processing Units). The CUDA platform is a software layer that gives direct access to the GPU's virtual instruction set and parallel computational elements, for the execution of compute kernels [wikipedia].

For the CUDA implementation of the application, the logic remains mostly the same. Using CUDA we have to manage two memories:

- **CPU:** *host memory*
- **GPU:** *device memory*

In order to operate at best on the GPU, the data structures used in this version is quite different:

Data structures

For the CUDA version the data has been organized in a *structure of array* manner in order to have *coalesced access to the memory*. That means that instead of vectors it has been used *dynamically allocated arrays* for storing the points dataset and the centroids with sizes of:

- dataset array:

```
const auto num_bytes_dataset =
    num_points * num_dimensions * sizeof
    (double);
```

- centroids array:

```
const auto num_bytes_centroids =
    num_clusters * num_dimensions *
    sizeof(double);
```

where *num_dimensions* is equal to three, since we have to store the same attributes as in the struct *Point* listed in section 2.

There have been allocated in total four arrays:

- for the *host*

```
auto *host_dataset = (double *) malloc(
    num_bytes_dataset);

auto *host_centroids = (double *) malloc
    (num_bytes_centroids);
```

- for the *device*

```
CUDA_CHECK_RETURN(cudaMalloc((void **) &
    device_dataset, num_bytes_dataset));

CUDA_CHECK_RETURN(cudaMalloc((void **) &
    device_centroids,
    num_bytes_centroids));
```

Since the Point dataset and cetroids generation is common to the entire application, therefore we have to transform the generated *Array of Structure* in a *Structure of Array* using the following method:

```
void transform_into_array(const std::vector
    <Point> &data, const int num_rows,
    double *array)
{
  for (auto i = 0; i < num_rows; i++)
  {
    array[i + (num_rows * 0)] = data[i].x;
    array[i + (num_rows * 1)] = data[i].y;
    array[i + (num_rows * 2)] = data[i].
        cluster;
  }
}
```

by calling:

```
transform_into_array(dataset, num_points,
    host_dataset);

transform_into_array(initial_centroids,
    num_clusters, host_centroids);
```

hence the access to the attributes of a point *i*, will be done as follows:

- for the *x* coordinate:

```
host_dataset[i + (num_points * 0)]
```

- for the *y* coordinate:

```
host_dataset[i + (num_points * 1)]
```

- for the *cluster* attribute:

```
host_dataset[i + (num_points * 2)]
```

this applies to the centroids array too, by using *num_centroids* instead of *num_points* and of course it's the same for the *device* arrays.

Afterwards, we transfer the *Structure of Arrays* to the *device memory*:

```
CUDA_CHECK_RETURN(cudaMemcpy(device_dataset
    , host_dataset, num_bytes_dataset,
    cudaMemcpyHostToDevice));

CUDA_CHECK_RETURN(cudaMemcpy(
    device_centroids, host_centroids,
    num_bytes_centroids,
    cudaMemcpyHostToDevice));
```

finally, inside the *host* main function we allocate the *centroids_details* array:

```
double *centroids_details;

CUDA_CHECK_RETURN(cudaMalloc((void **) &
    centroids_details, num_centroids *
    num_dimensions * sizeof(double)));
```

Computation

There have been implemented two *kernels*:

- *assign_clusters()*

```
assign_clusters<<<(num_points +
    THREAD_PER_BLOCK - 1) / THREAD_PER_BLOCK
    , THREAD_PER_BLOCK>>>(device_dataset,
    device_centroids, centroids_details);
```

- *update_centroids()*

```
update_centroids<<<1, num_centroids>>>(
    device_centroids, centroids_details,
    device_iterate);
```

where *THREAD_PER_BLOCK* is equal to *1024* and *device_iterate* is a *boolean* variable used for the *check convergence* step.

Between these two kernel calls we have a synchronization point *cudaDeviceSynchronize()* since we need to wait for the computation of the first kernel to terminate before proceeding with the second one.

As already stated, the logic of computation is the same as in the other versions of the application, the only changing thing being the data access. To have an idea of how it works, follows the *assign_clusters()* kernel code:

```
__global__ void assign_clusters(double*
    device_dataset, double* device_centroids
    , double* centroids_details)
{
  long id_punto = threadIdx.x + blockIdx.x
     * blockDim.x;

  if (id_punto < NUM_POINTS)
  {
    double punto_x, punto_y, centroid_x,
        centroid_y = 0;

    punto_x = device_dataset[id_punto +
              (NUM_POINTS * 0)];
    punto_y = device_dataset[id_punto +
              (NUM_POINTS * 1)];

    long best_centroid_id = 0;
    double distMIN = INFINITY;

    for (int i = 0; i < NUM_CENTROIDS; i++)
    {
      centroid_x = device_centroids[i +
                  (NUM_CENTROIDS * 0)];
      centroid_y = device_centroids[i +
                  NUM_CENTROIDS * 1)];

      auto dist = distance(punto_x,
          centroid_x, punto_y, centroid_y);
      if (dist < distMIN)
      {
        best_centroid_id = i;
        distMIN = dist;
      }
    }

    device_dataset[id_punto + (NUM_POINTS *
        2)] = best_centroid_id;
    doubleAtomicAdd(&centroids_details[
        best_centroid_id + (NUM_CENTROIDS *
        0)], punto_x);
    doubleAtomicAdd(&centroids_details[
        best_centroid_id + (NUM_CENTROIDS *
        1)], punto_y);
    doubleAtomicAdd(&centroids_details[
        best_centroid_id + (NUM_CENTROIDS *
        2)], 1);
  }
}
```

3. Tests

Tests have been run on a:

- CPU details
- GPU details

All versions of the application have been tested on datasets of random generated points as described in section 2, using different numbers of clusters and different number of treads for *OpenMP*:

```
int main()
{
  int num_points[] = {100, 1000, 10000,
      100000, 1000000, 10000000};
  int num_clusters[] = {10, 50, 100, 1000};
  int num_threads[] = {2, 4, 8};
  int max_epochs = 20;

  for (auto p : num_points)
  {
    // generate p Points dataset if not
        available

    std::vector<Point> dataset(p);

    //load dataset into the vector

    for (auto c : num_clusters)
    {
      if (c < p)
      {
        //generate centroids

        // compute SEQUENTIAL

        // prepare Structure of Arrays
        // prepare host and device memory

        //compute CUDA

        for (auto t : num_threads)
        {
          // compute OpenMP

        }
      }
    }
  }
}
```

3.1. Results

Results represent an average of 10 iterations where the computation time doesn't exceed *200 seconds* in which case the result represents a single iteration. The timings include only the computation time of each version and does not include dataset generation, centroids generation and transformation in *Structure of Arrays* methods. Each version of the application has been run on the same dataset and same generated centroids.

The results will be presented in two different sections for each dataset used:

- *sequential vs. OpenMP*
- *sequential vs. CUDA*

3.2. Sequential vs. OpenMP

100 points

#	Sequential	2 thread		4 thread		8 thread	
clusters	time	time	speedup	time	speedup	time	speedup
10	1.57415e-3	4.50105e-4	3.49	**2.20115e-4**	**7.15**	2.7304e-4	5.76
50	2.23364e-3	1.16821e-3	1.91	6.55396e-4	3.40	**2.81844e-4**	**7.92**

1000 points

#	Sequential	2 thread		4 thread		8 thread	
clusters	time	time	speedup	time	speedup	time	speedup
10	2.38316e-2	1.28891e-2	1.84	**6.93089e-3**	**3.43**	7.19663e-3	3.31
50	1.02973e-1	5.62904e-2	1.82	2.93544e-2	3.50	**1.98667e-2**	**5.18**
100	1.9067e-1	1.08647e-1	1.75	4.70549e-2	4.05	**2.27784e-2**	**8.37**

10000 points

#	Sequential	2 thread		4 thread		8 thread	
clusters	time	time	speedup	time	speedup	time	speedup
10	2.31849e-1	1.27039e-1	1.82	7.13549e-2	3.24	**4.94646e-2**	**4.68**
50	1.06555	5.76748e-1	1.84	3.18103e-1	3.34	**2.16165e-1**	**4.92**
100	2.1262	1.14159	1.86	6.25769e-1	3.39	**4.23186e-1**	**5.02**
1000	16.5422	10.093	1.63	5.71863	2.89	**4.96944**	**3.32**

100000 points

#	Sequential	2 thread		4 thread		8 thread	
clusters	time	time	speedup	time	speedup	time	speedup
10	2.47633	1.38707	1.78	7.61601e-1	3.25	**5.88133e-1**	**4.21**
50	11.6454	6.74652	1.72	3.7049	3.14	**2.9305**	**3.97**
100	23.9038	13.0483	1.83	7.09962	3.36	**5.04194**	**4.74**
1000	212.32	113.696	1.86	68.7125	3.08	**51.9228**	**4.08**

100000 points

#	Sequential	2 thread		4 thread		8 thread	
clusters	time	time	speedup	time	speedup	time	speedup
10	24.4441	13.549	1.80	7.99844	3.05	**5.66374**	**4.31**
50	112.231	59.7821	1.87	36.3999	3.08	**27.3157**	**4.10**
100	222.941	123.332	1.80	71.8734	3.10	**49.2834**	**4.52**
1000	2213.09	1204.71	1.83	651.162	3.39	**465.924**	**4.74**

10000000 points

# **clusters**	Sequential time	2 thread time	2 thread speedup	4 thread time	4 thread speedup	8 thread time	8 thread speedup
10	247.419	138.407	1.78	73.7417	3.35	**53.426**	**4.63**
50	1099.42	586.803	1.87	318.664	3.45	**217.305**	**5.05**
100	2241.82	1262.62	1.77	701.285	3.19	**486.829**	**4.60**
1000	21894.6	11946.8	1.83	6547.24	3.34	**4608.83**	**4.75**

3.3. Sequential vs. CUDA

100	10 clusters	50 clusters	100 clusters	1000 clusters
Sequential	1.57415e-3	2.23364e-3	-	-
CUDA	2.95279e-3	1.90538e-3	-	-
Speedup	-	1.17	-	-

1000	10 clusters	50 clusters	100 clusters	1000 clusters
Sequential	2.38316e-2	1.02973e-1	1.9067e-1	-
CUDA	1.76233e-2	1.19106e-2	1.79624e-2	-
Speedup	1.35	8.64	10.61	-

10000	10 clusters	50 clusters	100 clusters	1000 clusters
Sequential	2.31849e-1	1.06555	2.1262	16.5422
CUDA	1.19511e-1	3.13272e-2	3.35137e-2	1.17289e-1
Speedup	1.93	34.01	63.44	141.03

100000	10 clusters	50 clusters	100 clusters	1000 clusters
Sequential	2.47633	11.6454	23.9038	212.32
CUDA	2.03975	1.91504e-1	1.63803e-1	9.88928e-1
Speedup	1.21	60.81	145.93	214.698

1000000	10 clusters	50 clusters	100 clusters	1000 clusters
Sequential	24.4441	112.231	222.941	2213.09
CUDA	18.9062	2.03021	1.56286	8.76065
Speedup	1.29	55.28	142.65	252.61

10000000	10 clusters	50 clusters	100 clusters	1000 clusters
Sequential	247.419	1099.42	2241.82	21894.6
CUDA	171.032	19.1936	15.1799	90.301
Speedup	1.44	57.28	147.684	242.462