UNIVERSITÀ
DEGLI STUDI
FIRENZE

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

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

19.01.2021

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.
- Each class or cluster, is represented by a centroid, which describes the characteristics the data must have or approximate such to make part 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 application:
  - items are 2D points generated randomly using a *uniform real distribution*
  - *k* centroids representing the center of mass of a cluster are chosen randomly among the generated points
  - *Euclidean distance* is used as a metric

- The algorithm consists of an iterative procedure:
  1. *Initialization:* generate *N* 2D points dataset and *K* centroids.
  2. *Assignment:* assign each point to a centroid.
  3. *Update:* update each centroid's characteristics.
  4. *Check convergence:* check the stop criterion.
  5. Repeat steps *2 - 4* until the stop criterion is reached.

UNIVERSITÀ DEGLI STUDI FIRENZE

Implementation

UNIVERSITÀ DEGLI STUDI FIRENZE

Data structures (Sequential & OpenMP)

- data items of this application are 2D points 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){}

};
```

- data points and clusters are stored in a *std::vector<Point>* which is a sequence container very efficient accessing it's elements.

Dataset generation

```
void generate_random_dataset(int nr_points) {
    std::random_device rd;
    std::mt19937 generator(rd());
    std::uniform_real_distribution<double> distribution(0.0, 1.0);

    const std::string rand_dataset_name =
            "rand_dataset_" + std::to_string(nr_points) +
            ".txt";

    const std::string rand_dataset_path = "../datasets/" + rand_dataset_name;
    if (!(std::ifstream(rand_dataset_path))) {
        std::ofstream rand_dataset_file(rand_dataset_path);
        for (auto i = 0; i < nr_points; i++) {
            for (auto j = 0; j < 2; j++) {
                rand_dataset_file << distribution(generator) << " ";
            }
            rand_dataset_file << "\n";
        }
        rand_dataset_file.close();

        std::cout << "Random Dataset Generated: " + rand_dataset_name + "\n";
    }
}
```

UNIVERSITÀ DEGLI STUDI FIRENZE

Load Dataset

```
void load_dataset(std::vector<Point> &dataset, std::ifstream &dataset_file) {
    std::cout << "Loading dataset in progress ...\n";
    std::string file_line, word;

    while (getline(dataset_file, file_line)) {

        std::istringstream string_stream(file_line);
        std::vector<std::string> row;

        while(getline(string_stream, word, ' ')){
            row.push_back(word);
        }

        dataset.emplace_back(stod(row[0]), stod(row[1]));

    }
}
```

UNIVERSITÀ DEGLI STUDI FIRENZE

Centroids generation

```
std::vector<Point> generate_initial_centroids(const std::vector<Point> &dataset,
      const long num_clusters) {
    std::vector<Point> initial_centroids(num_clusters);
    std::vector<int> random_vector(dataset.size());
    std::iota(random_vector.begin(), random_vector.end(), 0);
    std::shuffle(random_vector.begin(), random_vector.end(), std::mt19937(std::
        random_device()()));
    for (auto i = 0; i < num_clusters; i++) {
        initial_centroids[i] = dataset[random_vector[i]];
        initial_centroids[i].cluster = i;
    }
    return initial_centroids;
}
```

UNIVERSITÀ DEGLI STUDI FIRENZE

Sequential implementation

```
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};

}
```

UNIVERSITÀ DEGLI STUDI FIRENZE

OpenMP implementation

```
tuple<std::vector<Point>, vector<Point>> k_means_OPENMP(vector<Point> points, vector<
    Point> centroids, int max_epochs, int nr_threads){

    vector<Point> centroids_details (centroids.size());
    bool iterate;
    int iteration = 0;

    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);

    return {points, centroids};

}
```

assign_clusters()

```
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;

    }
}
```

UNIVERSITÀ DEGLI STUDI FIRENZE

update_centroids()

```
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;

}
```

UNIVERSITÀ DEGLI STUDI FIRENZE

CUDA implementation Data structure

- for the CUDA version, data has been organized in a *Structure of Array* as to have *coalesced access to memory*.
- instead of *std::vector<Point>* it has been used *dynamically allocated arrays* for storing points and centroids, with sizes of:
  - dataset array

```
const auto num_bytes_dataset = num_p * num_attributes * sizeof(double);
```

  - centroids array

```
const auto num_bytes_centroids = num_c * num_attributes * sizeof(double);
```

- there have been allocated mainly 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));
```

- and inside the *host* main function

```
double *centroids_details;

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

- dataset and centroids generation is common to the entire application, therefore we use:

```
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_p, host_dataset);
transform_into_array(initial_centroids, num_c, host_centroids);
```

- afterwards we transfer the arrays to *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));
```

Computation

```
__host__ std::tuple<double *, double *>
kmeans_cuda(double *device_dataset, const short num_centroids,
            double *device_centroids, const int num_points, const short point_size,
                  const int max_epochs)
{...
do{
        host_iterate = false;
        CUDA_CHECK_RETURN(cudaMemcpy(device_iterate, &host_iterate, sizeof(bool),
              cudaMemcpyHostToDevice));

        iteration ++;

        CUDA_CHECK_RETURN(cudaMemset(centroids_details, 0, num_centroids * point_size
               * sizeof(double)));

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

        cudaDeviceSynchronize();

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

        cudaDeviceSynchronize();

        CUDA_CHECK_RETURN(cudaMemcpy(&host_iterate, device_iterate, sizeof(bool),
              cudaMemcpyDeviceToHost));

    } while(host_iterate && iteration < max_epochs);
```

assign_clusters()

```
__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);
    }
}
```

update_centroids()

```
__global__ void update_centroids(double* device_centroids, double* centroids_details,
      bool* iterate) {

    int unsigned id_centroid = blockDim.x * blockIdx.x + threadIdx.x;

    double centroid_x_new, centroid_y_new, centroid_x_old, centroid_y_old = 0;

    if (id_centroid < NUM_CENTROIDS) {

        centroid_x_new = centroids_details[id_centroid + (NUM_CENTROIDS * 0)] /
             centroids_details[id_centroid + (NUM_CENTROIDS * 2)];
        centroid_y_new = centroids_details[id_centroid + (NUM_CENTROIDS * 1)] /
             centroids_details[id_centroid + (NUM_CENTROIDS * 2)];

        centroid_x_old = device_centroids[id_centroid + (NUM_CENTROIDS * 0)];
        centroid_y_old = device_centroids[id_centroid + (NUM_CENTROIDS * 1)];

        if(!check_tollerance(centroid_x_old, centroid_x_new, centroid_y_old,
             centroid_y_new)) {
            *iterate = true;
            device_centroids[id_centroid + (NUM_CENTROIDS * 0)] = centroid_x_new;
            device_centroids[id_centroid + (NUM_CENTROIDS * 1)] = centroid_y_new;
        }
    }
}
```

Test & Results

Test details

- Tests have been run on a:
  - Intel Core i7-860 2.8 GHz, quad-core
  - GeForce GTX 980

```
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
    //load dataset
    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
      }
  }
}
```

Results - Sequential vs. OpenMP

- 1.000 points

# clusters	Sequential time	2 thread time	2 thread speedup	4 thread time	4 thread speedup	8 thread time	8 thread 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**

- 100.000 points

# clusters	Sequential time	2 thread time	2 thread speedup	4 thread time	4 thread speedup	8 thread time	8 thread 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**

- 10.000.000 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**

Results - 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	-	-

1.000	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	-

10.000	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

100.000	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

1.000.000	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

10.000.000	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