Parallel Implementations of K-means Clustering using OpenMP and CUDA

Leonardo Di Iorio
E-mail address
leonardo.diiorio@stud.unifi.it

Abstract

In the following paper the k-means clustering algorithm will be analysed. Three different implementation will be shown: a sequential implementation of the algorithm, a parallel implementation on a Intel i7-10750H CPU using the OpenMP framework and a parallel implementation using CUDA on a GPU (NVDIA GeForce RTX 2070 Super). The goal of this paper is to show the improvements in terms of performance that can be obtained through different tecniques of code parallelization.

Future Distribution Permission

1. Introduction

Clustering is a technique for grouping objects in an unsupervised way. Each grouping, or cluster, represents a membership class. Having no examples from which to learn, clustering exploits similarities among the data it has to analyze, similarities that can be of various kinds but that essentially define a distance between points in the dataset.
K-means is one of the most widely used and best performing clustering algorithms. It's based on centroids. A centroid is a point belonging to the feature space that averages the distances between all the data belonging to the cluster associated with it. It represents a kind of barycenter of the cluster and in general is not one of the points in the dataset. The algorithm is iterative and articulated in the three following steps:

1. **Inizialization:** Given a dataset of size n, the number K of clusters into which it is to be partitioned is chosen. Then K points in the dataset are randomly choosen to represent the initial centroids. The only condition is that they are not coincident in fact, usually, to avoid trouble in the algorithm convergence we make sure that they are far enough apart;
2. **Point assignment:** At this stage, the algorithm associates each point in the dataset with the nearest centroid. To do this it calculates the Euclidean distance between each point and the centroids, then assigns the point to the minimum distance centroid;
3. **Centroids update:** As a consequence of the point assignment step, it is likely that the composition of the clusters has changed. In this step we then calculate the new centroid of each cluster by computing the average of all points belonging to the cluster under consideration.

2. Implementation

In this section we will present a description of the three different implementations of the algorithm that have been made.

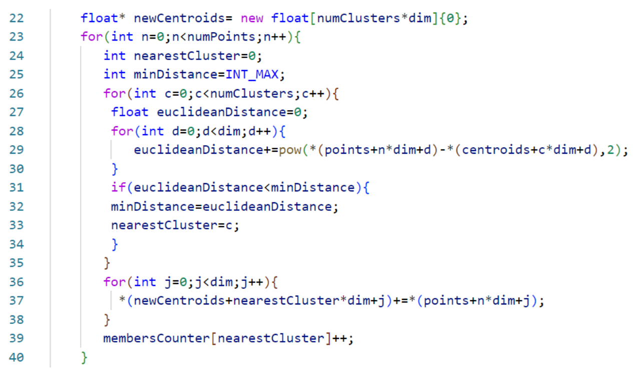
2.1. Sequential

The sequential version of the code is implemented in C++. Despite this, the code is not developed with an object-oriented programming approach but is instead implemented in order to be cache-friendly. In fact, this is a key step to avoid slowdowns in execution and to be able to achieve code that is as similar as possible between the sequential and parallel versions.

So in view of the above, a single class was implemented, called $Kmeans$. It is characterized by only three methods:

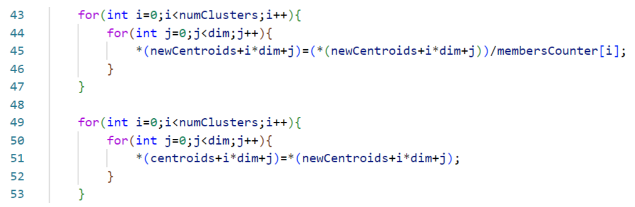
- The constructor;
- The `kmeansIteration` method, within which the operations related to a single iteration of the algorithm are implemented;
- The `computeKmeans` method, which is responsible for orchestrating the iterative execution of the algorithm;

Points and centroids are represented as linearized arrays of float values, where each value corresponds to the coordinate of a specific point or centroid (depending on the array under consideration). The j-th coordinate of the i-th point in the array and placed at position $i*dim+j$ where dim is the number of dimensions of the point. Thus, the arguments of the constructor are as follows: The constructor arguments are then as follows: the number of point sizes, the number of points, the number of clusters, the array of points, the array of centroids, and the array of counters. The latter array has a number of components equal to the number of clusters and keeps track in each component of the number of points belonging to the corresponding cluster. The centroids are initialized with the first K points of the dataset. Let's now show in detail the implementation related to the point assignment step and the centroids update step. The code for the point assignment step is shown in the figure below:



```
float* newCentroids= new float[numClusters*dim]{0};
for(int n=0;n<numPoints;n++){
   int nearestCluster=0;
   int minDistance=INT_MAX;
   for(int c=0;c<numClusters;c++){
    float euclideanDistance=0;
    for(int d=0;d<dim;d++){
       euclideanDistance+=pow(*(points+n*dim+d)-*(centroids+c*dim+d),2);
    }
    if(euclideanDistance<minDistance){
    minDistance=euclideanDistance;
    nearestCluster=c;
    }
   }
   for(int j=0;j<dim;j++){
     *(newCentroids+nearestCluster*dim+j)+=*(points+n*dim+j);
   }
   membersCounter[nearestCluster]++;
}
```

As visible in the code, a temporary array is initialized that will contain the sum of the points assigned to the same cluster. Then, for each point the Euclidean distance from each centroid is calculated by accessing the arrays of points and centroids in the manner described above. The nearest centroid is then kept track of. When the algorithm has finished checking the distance of the point from each centroid, the coordinates of the point under consideration are then added in the component of the temporary array corresponding to the nearest centroid and the relative cluster counter is incremented by one. The procedure is performed for each point in the dataset. The next step is related to the update of centroids, we show the related code in the figure below:



```
for(int i=0;i<numClusters;i++){
    for(int j=0;j<dim;j++){
        *(newCentroids+i*dim+j)=(*(newCentroids+i*dim+j))/membersCounter[i];
    }
}

for(int i=0;i<numClusters;i++){
    for(int j=0;j<dim;j++){
        *(centroids+i*dim+j)=*(newCentroids+i*dim+j);
    }
}
```

As shown in the code, the new centroid associated with each cluster is calculated by computing the average among all points belonging to that cluster. Finally, the updated values of the centroids contained in the temporary array are copied into the centroid array.

2.2. OpenMP

OpenMP is API (application program interface) for writing parallel shared-memory applications. The parts of the implemented code that need to be parallelized in order to achieve performance improvement are the external for loops. Both those related to the point assignment step and those related to the udpate step of the centroids. As for the management of critical sections, in order to avoid race condition it is necessary to introduce a specific OpenMP directive before the expressions to the lines of the code above. To parallelize the loops for the following directive has been used:

```
#pragma omp parallel for
```

To handle the critical expressions mentioned above the following OpenMP directive has been used:

```
#pragma omp atomic
```

2.3. CUDA

NVIDIA provides a programming interface for parallel computing known as CUDA (Compute Unified Device Architecture) which allows direct programming of the NVIDIA hardware. Using NVIDIA devices to execute massively parallel algorithms yields a many times speedup over sequential implementations on conventional CPUs. As already seen in the sequential version, an object-oriented programming approach was also not used here in order to achieve cache friendly code and better performance. A CUDA program consists of kernel functions, executed on the GPU, and host functions, executed on the CPU. Parallelization was performed with respect to the n points in the dataset with a single kernel function. Thus, compared to the parallel version obtained with OpenMP, the external fors were replaced by ifs (which determine whether or not the centroid is updated), based on the thread index obtained as follows:

```
int t = blockIdx.x * blockDim.x + threadIdx.x;
```

The call to the kernel function is shown below:

```
kmeansIterationKernel << <ceil(numPoints / (float)blDim), blDim >> >
```

The main functions used in the code are highlighted:

- The following functions were used to allocate and release memory on the GPU, respectively: `cudaMalloc` and `cudaFree`;
- The `cudaMemcpy` function was used to pass the arrays from host to device;
- In order to avoid problems during execution, the atomic expression `atomicAdd` was used to increment the counter of points belonging to a cluster in the assignment phase;
- In order to ensure the correct output with respect to updating centroids, the `__threadfence` function was inserted before the section of code related to centroids updating;

3. Performance

This section will show the results in terms of performance improvement obtained with the different parallelization tools.

3.1. Experimental setup

As anticipated in previous sections, an Intel i7-10750H CPU and a GPU NVDIA GeForce RTX 2070 Super were used. The stop criterion for all versions of the algorithms is a maximum number of iterations. So when each algorithm reaches that number of global iterations it stops. This was intended to ensure the same workload for all algorithms and thus as fair a comparison as possible on performance. The points constituting the datasets are randomly generated by a uniform distribution and are characterized by three dimensions. For the time measurements we have used `system_clock` from the `<chrono>` header.

3.2. Results of the comparison

Results were visualized by means of two main types of experiments. In the first case, the number of clusters,K, was kept stable by increasing the number of points, n. In the second case, the number of points was fixed and we varied the number of clusters. We report in figure 1 the comparison of the different technologies with variable number of points valued with respect to absolute time. Then, we show in figure 2 the result of the same comparison showing this time the speedup obtained by the two parallel implementations. The speedup is the speed increase when going from a sequential version to a parallel one. In other words, it's the ratio of the sequential time to the parallel time. As visible from the first graph both parallel versions outperform the sequential version from the outset. In particular, the CUDA version performs better than the OpenMP version. In the speedup graph this information is even more evident, in fact we can see a significantly higher

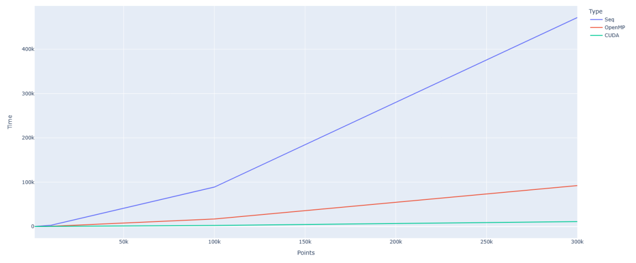


Figure 1. Comparison between different impelementations. On x axis the number of points and on y axis the time taken to complete the execution. $K = \sqrt{n}$

.

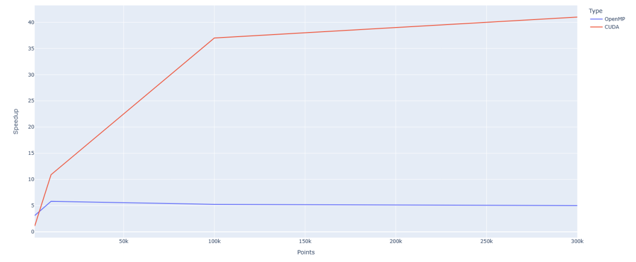


Figure 2. Comparison between different impelementations. On x axis the number of points and on y axis the speedup.

speedup in the implementation with CUDA. Similarly, we now show the results for the performance of the three implementations as the number of clusters changes, while keeping the number of points fixed. In this case $n = 2^{16}$.

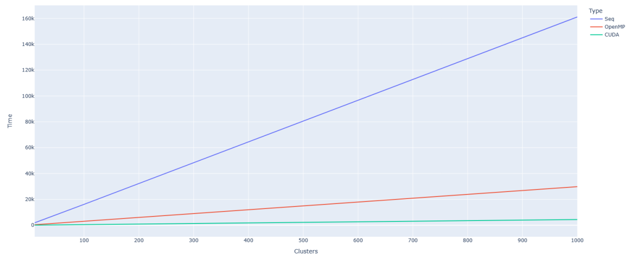


Figure 3. Comparison between different impelementations. On x axis the number of clusters and on y axis the time.

In figure 3 the results are shown with respect to the time taken to complete the execution. Again we can see how the two parallel versions outperform the sequential one, with better performance offered by the CUDA version. Figure 4 shows the comparison between the two parallel versions in terms of speedup.

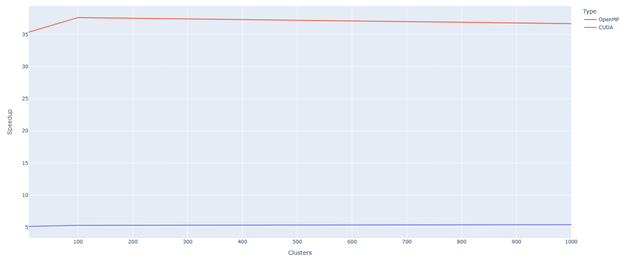


Figure 4. Comparison between different impelementations. On x axis the number of clusters and on y axis the speedup.

Finally, we show in figure 5 how the execution time of the OpenMP version varies with the number of threads:

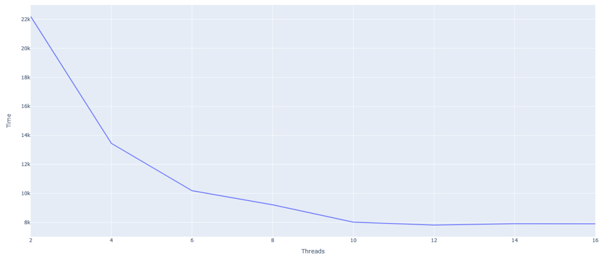


Figure 5. On x axis the number of thread and on y axis the time.

4. Conclusions

This discussion looked at the comparison of three different implementations of the K-means algorithm: sequential, OpenMP and CUDA. As can be guessed, the configuration that offers better performance turns out to be the one in CUDA. Note, however, that very often such an implementation requires a higher workload than a similar OpenMP implementation. In many contexts, given the speedup achieved with the OpenMP version anyway, it may be convient to consider this approach.