Parallel Implementations of *k*-means Clustering using OpenMP and CUDA

Marco Benelli
E-mail address
`marco.benelli@stud.unifi.it`

Luca Bindini
E-mail address
`luca.bindini@stud.unifi.it`

Abstract

In this paper we will analyze the k*-means clustering algorithm with three different implementations.*

*The first one is a sequential implementation, the second one is a parallel implementation on a CPU (*Intel i5-6200U *Quad Core) using the* OpenMP *framework and the last one is a parallel implementation on a GPU (*Nvidia GTX 980*) using* CUDA.

The main purpose of this paper is to evaluate the performance of the algorithm especially in terms of speedup of the parallel versions compared to the sequential version.

Future Distribution Permission

1. Introduction

The *k*-means algorithm is an unsupervised learning algorithm that finds a fixed number of clusters in a dataset.

Clusters represent groups that divide objects according to whether or not there is some similarity between them. When using the *k*-means algorithm, a centroid is defined for each cluster, which is a point (artificial or real) representative of the cluster itself.

The algorithm is iterative and based on the following steps:

1. **Initialization:** This is done by choosing the dimension of the dataset (n) and k initial centroids arranged randomly. By choosing the number of centroids, the clusters which the dataset will be composed of and therefore the groupings to be made and displayed are chosen.
2. **Points assignment:** In this phase, the algorithm analyzes each of the data points and assigns them to the nearest centroid. The Euclidean distance between each data points and each centroid is calculated. Each data point will then be assigned to the centroid whose distance is minimum.
3. **Centroids update:** After step 2 it is likely that new clusters have formed, as the previous ones will be assigned (or removed depending on whether they have passed to another cluster) new data points. Consequently, the average position of the centroids is recalculated. The new value of a centroid will be the average of all data points that have been assigned to the new cluster.

2. Sequential version

The code for the sequential version has been written in C++. However, it doesn't use many of the object-oriented programming concepts. This was done because we wanted to avoid arrays of structures (*AoS*) and preferred structures of arrays (*SoA*), with the reasoning being that *SoA*s are more cache-friendly. Writing cache-friendly code is important when dealing with parallel programs and not so much when talking about sequential ones, but we wanted the sequential and parallel versions to be as close as possible so as to evaluate the speedup in the fairest way possible.

Having said that, the only class we coded is named `KMeans` and its public methods are a constructor, a destructor and an `execute` method.

A `KMeans` object is constructed by passing it the array of points, the number of points (`n`), the number of dimensions of each point and the number of clusters (k). The points are represented as a linearized array of real numbers where each value represents the coordinate of a certain point. In particular, the i-th coordinate of the j-th point is the kept in position i*n+j of the array. We chose i*n+j and not j*d+i (where d is the dimension of the points) to have coalesced access to the memory (which is important mostly for the *CUDA* version). Inside the constructor we also allocate space for the array centrs, which keeps track of the positions of the centroids, which are represented the same way as the points. The centroids are initialized to take the positions of the first k points.

We have implemented two different versions of the algorithms. The first one is the more intuitive one, as it follows more closely the description of the *k*-means algorithm given in the introduction. The second one is probably less intuitive, but more easily parallelizable. We'll now look at the details of both these implementation.

2.1. Naive implementation

In the naive implementation, the execute methods calls two private methods, assign and update. In this case, the class KMeans, also has the object field ids, an array of n integers that keeps track of which point is assigned to which cluster.

Here follows the body of assign:

```
for (int p = 0; p < n; p++) {
    float minDist = INFINITY;
    float dist;
    for (int c=0; c<K; c++) {
        dist = pointDist(p, c);
        if (dist < minDist) {
            minDist = dist;
            ids[p] = c;
        }
    }
}
```

And this is the body of update (the only things missing are the resetting of each the centroids and the final division by numPoints, to get the average instead of just the sum):

```
for (int c=0; c<k; c++) {
    int numPoints = 0;
    for (int p=0; p<n; p++) {
        if (ids[p] == c) {
            for (int i=0;i<d;i++)
                centrs[k*i+c] +=
                  points[n*i+p];
            numPoints++;
        }
    }
}
```

2.2. Better implementation

In this other version, what the execute method does is calling a private method that performs a single iteration of the k-means algorithm in a loop until the stopping condition is satisfied. Below you can see a portion of the body of the private method mentioned. The only things missing from this code are the resetting of newCentrs and clusterSize (some object fields of the KMeans class specific to this implementation) before the shown section, and the assignment to centrs of the new positions.

```
for (int p=0; p<n; p++) {
    int minId = 0;
    float minDist = INFINITY;
    float dist;
    for (int c=0; c<k; c++) {
        dist = pointDist(p, c);
        if (dist < minDist) {
            minDist = dist;
            minId = c;
        }
    }
    clusterSize[minId]++;
    for (int i=0; i<d; i++)
        newCentrs[i*k+minId]
          += points[i*n+p];
}
```

3. OpenMP version

OpenMP is an API used to specify high-level parallelism in C++ programs. For some programs it can be really easy to make them parallel with just a few directives.

In our case the parts we should parallelize are the outer loops (both the outer loops in the naive implementation as well as the outer loop of the better implementation). To parallelize a loop in *OpenMP*, we can add the following line at its beginning:

```
#pragma omp parallel for
```

We should also add `default(none)`, just to make sure that we specify all the variables declared outside the loop as either `shared` or `private`.

One thing we should be careful with is how we handle the increments at lines 12 and 14 of the better implementation, since we don't want to run into a race condition. What we need to do is add a line like the following before each of the critical expressions.

```
#pragma omp atomic
```

For the naive implementation, there is no need for atomics, since this version is more easy to parallelize.

4. CUDA version

CUDA is a parallel computing platform and programming model developed by *NVIDIA* for general computing on its own GPUs. *CUDA* enables developers to create compute-intensive applications by using the power of GPUs for the parallelizable portion of code.

The code for the this version was written using the *NVIDIA Nsight* development environment and compiled with the *NVIDIA CUDA* Compiler (*NVCC*). A *CUDA* program consists of kernel functions, executed on the GPU, and host functions, executed on the CPU. Just like the sequential version, an object-oriented approach was not used for the same reasons of better performance and cache-friendly approach. In a first naive implementation the parallelization took place on the number of points n in the points assignment phase and on the number of clusters k in the centroids update phase using two different kernel functions: `assign` and `update`. Compared to the sequential and *OpenMP* version the code is very similar with the difference that the external `for` loops used for parallelization have been replaced with `if`s that exploit the thread index obtained as follows:

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

Here are the calls to the two kernel functions of the naive version:

```
assign<<<ceil(N/(float)bDim),
         bDim>>>

update<<<1, K>>>
```

The `ceil` function was used to approximate `gridDim`, ie the number of blocks, to the superior integer.

A better implementation was made using a single kernel parallelized only on the number of points n, managing to obtain better performance, as can be seen in the next section.

Below is the call to the better version kernel:

```
kMeansKer<<<ceil(n/(float)bDim),
            bDim>>>
```

The *CUDA* functions `cudaMalloc` and `cudaFree` were used to allocate and release memory on the GPU. For the passage of points between host and device and vice versa, the `cudaMemcpy` function was used with the different parameters `cudaMemcpyHostToDevice` or `cudaMemcpyDeviceToHost`.

5. Performance

Now that we've looked at all the different implementations, using many diverse technologies, let's look at how they all perform.

5.1. The setup

The sequential and *OpenMP* versions were run on an *Intel Core i5-6200U*, while the *CUDA* ones on an *NVIDIA GeForce GTX 980*.

The datasets of points used were randomly generated using a uniform distribution with 2 dimensions.

As we've already said, the only stopping criterion used was the iteration count, meaning that, in this case, the execution would seize after a set amount of iterations. This was done to make sure that all the executions would do the same amount of work, so that the random generation of points would not play a role in the performance evaluation. In our case, we chose to set the number of iterations to 64.

For the time measurements, `system_clock` from the `<chrono>` header was used, a C++11 addition.

5.2. Comparison of the different technologies

Since the k-means clustering algorithm has two main parameters, namely n (the number of points) and k (the number of clusters), we cannot show a plot in which both those parameters vary independently. To get around this issue, we decided to plot two distinct types of graph. The first one is the one in figure 1, in which we plotted the time with respect to n. While the second one is the one from figure 2, where n remained constant and we only varied k.

In the first graph you can see that the *OpenMP* version beats the sequential version from the very beginning, while the *CUDA* one only becomes viable when we start having a lot of points. This is mainly due to a more substantial overhead for the *CUDA* version.

Another important thing to measure is the speedup, which is the speed increase between when going from a sequential version to a parallel one. More precisely, it's the ratio of the sequential time to the parallel time. Figures 3 and 4 show the same information from figures 1 and 2, but looking at the speedups instead of the absolute times.

In the other plots, for the *CUDA* version we used the best `blockDim` that would take the least amount of time. We can take a look at how much time other `blockDim` values might take, as shown in the graph in figure 5. Here the best `blockDim` is 256.

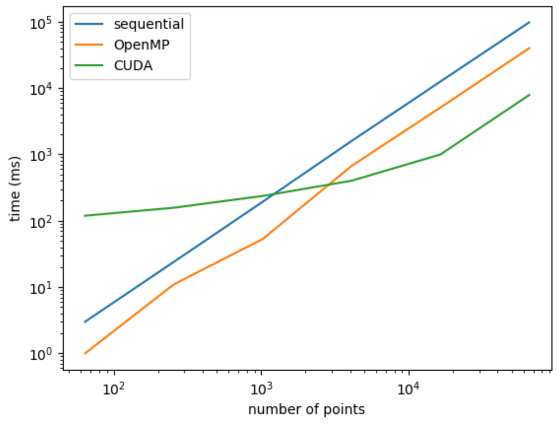

Figure 1. The times taken by the different implementations. On the x axis the number of points n and on the y axis the time in milliseconds. In this case k takes the form $k = \sqrt{n}$, meaning that we have k clusters with k points each (on average).

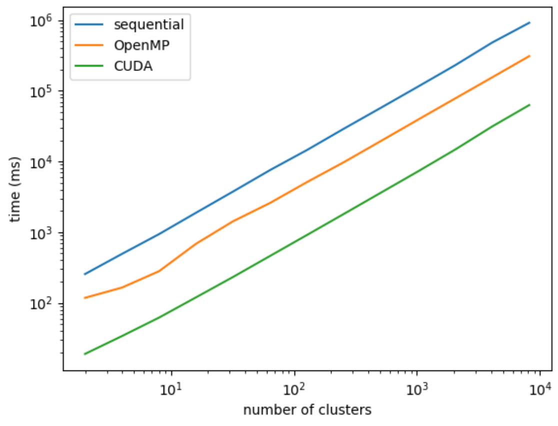

Figure 2. The times taken by the different implementations. On the x axis the number of clusters k and on the y axis the time in milliseconds. In this case $n = 2^{14}$.

5.3. Naive implementations' performance loss

The results in the previous subsection use the better implementations for both *OpenMP* and *CUDA*. We can take a look at the difference between the naive and the better implementations. The graph in figure 6 shows all of the different

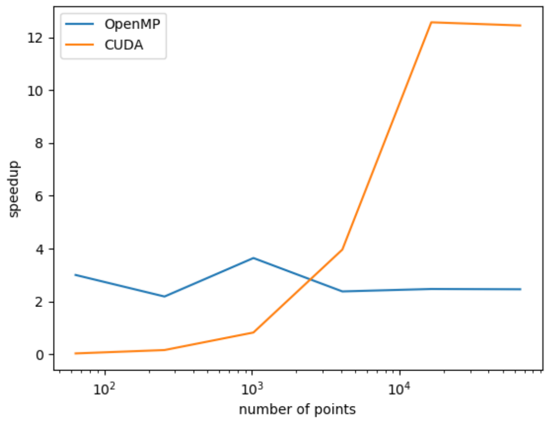

Figure 3. The speedups of the two parallel implementations. On the x axis the number of points n and on the y axis the speedup. As before, $k = \sqrt{n}$.

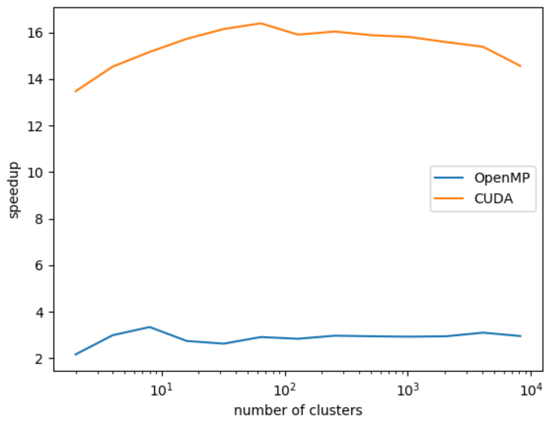

Figure 4. The speedups of the two parallel implementations. On the x axis the number of clusters n and on the y axis the speedup. As before, $n = 2^{16}$.

versions and how they compare to each other.

As we can see from the plots, when using *OpenMP* the naive implementation is just as good as the better one (at least with the values used in this case). However, when dealing with *CUDA*, we should make sure to parallelize as much as possible, meaning that we should create as many threads as points and not just one thread per cluster.

6. Conclusions

We have seen the k-means algorithm implemented in various ways and using different technologies. There is no doubt that the *CUDA* version is the best one when we are using enough points to make it worth the effort. However, it's important to note that this implementations is the easiest to mess up, as we have seen from the naive approach. The *OpenMP* implementation, on the other hand, is more robust to a less efficient parallelization and is also always better than the sequential version, even when dealing with few points or clusters.

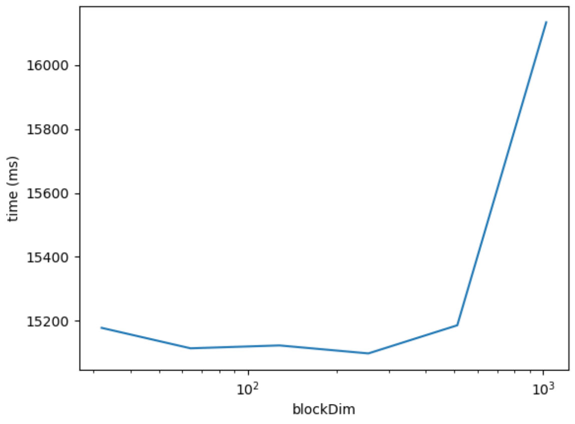

Figure 5. The time taken by the *CUDA* implementation with different *blockDim*. In this case $n = 2^{16}$ and $k = \sqrt{n}$.

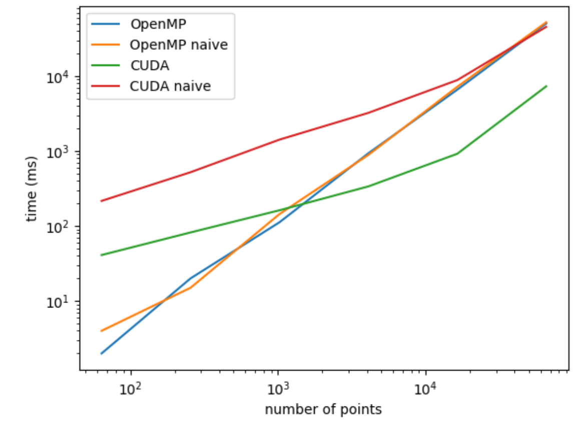

Figure 6. The times taken by the different implementations. On the x axis the number of points n and on the y axis the time in milliseconds. In this case k takes the form $k = \sqrt{n}$.