K-Means clustering: parallel implementation in CUDA

Nicola Carkaxhija
Dept. of Information Engineering, University of Florence
Via di S. Marta 3, 50139 – Florence, Italy
Nicola.carkaxhija@stud.unifi.it

Abstract

Cluster analysis plays a critical role in a wide variety of applications, but it has to face the continuously increasing volume of data; parallel computing is one of the most promising solution to overcome the computational challenge and process data in a reasonable amount of time. The subject of the experiment is k-Means, a popular clustering algorithm, whose sequential version has been parallelized using the CUDA framework to enable heterogeneous computing on NVIDIA Graphics Processing Units (GPUs). The measure used to compare the performance improvement is speedup.

1. Introduction

Given a set of points, clustering is the task of partitioning them on the base of an appropriate distance measure, so that similar points belong to the same group and different ones end in separate clusters.

One of the most popular clustering methods is k-means, an algorithm that iteratively assigns d-dimensional real vectors (the points) to the nearest over a set of k clusters - value usually known in advance - that are identified through their centroids.

The algorithm is composed of four phases:

1. the initialization, during which k points are randomly selected as centroids;

2. the assignment of each point to the nearest cluster;

3. the update of the cluster centers;

4. reiteration of the previous two steps until a convergence criterion is met.

The convergence to an optimal global solution is not guaranteed, and many aspects could be analyzed more deeply: heuristics may be adopted for the initialization and stopping criterion, but their discussion lies outside of the purpose of the experiment, that is improving the performance in terms of execution time, so that a simple implementation is used.

However, as the initialization is random, the algorithm does not yield the same result at each run; then the sequential and parallel versions are executed on the same instance of points, to remove any dependence from data and make the comparison effectively meaningful.

The improvements of parallelization are evaluated computing the speed-up. Furthermore, the analysis has been carried out by varying the dataset size and the number of clusters.

Algorithm k-Means Clustering

Input:
- $\mathcal{X} = \{x_1, ..., x_n\}$
- k (number of clusters)

Initialise:
Cluster centroids $\mathcal{C} = \{c_1, ..., c_k\} \in \mathbb{R}^m$ randomly

For every $x_i \in 1, ..., n$, set

$$c_j := \arg\min_{j \in 1,...,k} ||x_i - c_j||$$

For every $c_j \in 1, ..., k$, set

$$c_j := \frac{\sum_{i=1}^{n} 1\{c_i = j\} x_i}{\sum_{i=1}^{n} 1\{c_i = j\}}$$

Repeat steps 3 and 4 until centroids convergence

Output:
- $\mathcal{C} = \{c_1, ..., c_k\} \in \mathbb{R}^m$ (centroids)
- $\mathcal{Y} = \{y_1, ..., y_n\}$

Figure 1: pseudo-code of a sequential basic version of k-means.

2. Data dependence analysis

From the pseudo-code in Figure 1, it can be observed that k-means, by its very nature, is intrinsically well suited for parallelization since most of the computations are independent from each other.

In particular, the effort will be addressed on the assignment and update steps, whose incidence on the execution time is the most significative because of the nested loops, whilst the other operations have a marginal contribution.

During the assignment phase the outer loop iterates over the points, carrying out for each of them a sequential search through the inner loop, to determine the nearest cluster, store its index in an "assignment" array and increment the respective entry of a counter array that keeps track of the new cardinality of the clusters. Since each research is independent on the others, they can be carried out simultaneously by different threads. However, in order to avoid inconsistencies due to concurrent accesses, the counter entries are increased through an atomic addition.

Similarly, since centroids are independent the update step also may be performed concurrently instead of sequentially, one centroid at a time; the same observation holds at a component level: given a centroid, every entry also can be computed independently by the others. However, the update must take place through atomic additions to avoid simultaneous accesses to the same location of the centroids array.

3. Parallelization

The structure of the parallel version is faithful to the sequential one, it underwent just slight modifications. Beyond these, the parallel implementation required preliminary and conclusive additional operations of memory allocation and freeing respectively, since data must be moved from the host to the device memory and back to enable GPU computation.

Some part like initialization have been left unchanged since no benefit is visible because of the marginal role played.

So that, also the stopping criterion is the same for both the versions. Many attempts have been carried out to parallelize it, but no evident improvement emerged. Furthermore, there is a more relevant reason to adopt a simpler solution, but this choice is motivated in the following section.

Hereinafter the most relevant code in the original and counterpart version:

```
// compute nearest cluster
for(int i = 0; i < N; i++){
    float minDistance = FLT_MAX;
    short int minIndex = -1;
    for(int j = 0; j < K; j++){
        float distance = 0.0;
        for (int l = 0; l < P; l++){
            float diff = points[i * P + l] \
                    - centroids[j * P + l];
            distance += pow(diff, 2);
        }
        if(distance < minDistance){
            minDistance = distance;
            minIndex = j;
        }
    }
    assignments[i] = minIndex;
    counter[minIndex]++;
}
```

```
// compute nearest cluster
short int index = blockIdx.x * blockDim.x + threadIdx.x;
if (index < n) {
    float minDistance = FLT_MAX;
    short int minIndex = -1;
    for (int i = 0; i < k; i++) {
        float distance = 0.0;
        for (int j = 0; j < p; j++){
            float diff = points[index * p + j] \
            - centroids[i * p + j];
            distance += pow(diff, 2);
        }
        bool compare = (minDistance <= distance);
        minDistance = compare * minDistance \
                + (1 - compare) * distance;
        minIndex = compare * minIndex + (1 - compare) * i;
    }
    assignments[index] = minIndex;
    atomicAdd(&(counter[minIndex]), 1);
}
```

Figure 2: code responsible for the nearest cluster search, before and after parallelization respectively.

```
// compute means
for(int i = 0; i < N; i++){
    int clusterId = assignments[i];
    for(int j = 0; j < P; j++)
        centroids[clusterId * P + j] += \
            points[i * P + j];
}

for(int i = 0; i < K; i++)
    for(int j = 0; j < P; j++)
        centroids[i * P + j] /= counter[i];
```

```
// compute mean
short int index = blockIdx.x * blockDim.x + threadIdx.x;
if (index < n) {
    short int clusterIndex = devAssignments[index];
    for(int i = 0; i < p; i++)
        atomicAdd(&(devCentroids[clusterIndex * p + i]), \
          points[index * p + i] / counter[clusterIndex]);
}
```

Figure 3: code responsible for the centroids update, before and after parallelization respectively.

4. Experimental results

The sequential and parallel algorithms have been compared on synthetic data randomly generated according to a uniform distribution. This phase is not taken into account in the execution time: it would be meaningless since in a realistic scenario the dataset is given.

As stopping criterion, it has been preferred a maximum number of iterations instead of convergency check, to get around the drawback of evaluating the centroids variation by setting a proper threshold, that is critical when it comes to run k-means on massive inputs in a reasonable amount of time. Indeed, it may be noticed that to the purpose of speed-up calculation it is not necessary the algorithms to converge, but it is sufficient a comparison of the execution times deployed by the two versions to parity of iterations, as provided that is a number enough high to make memory operations (between host and device) negligible compared to the total execution time.

In the following, the speed-up semi-logarithm charts obtained on a NVIDIA Tesla K80 graphic card.

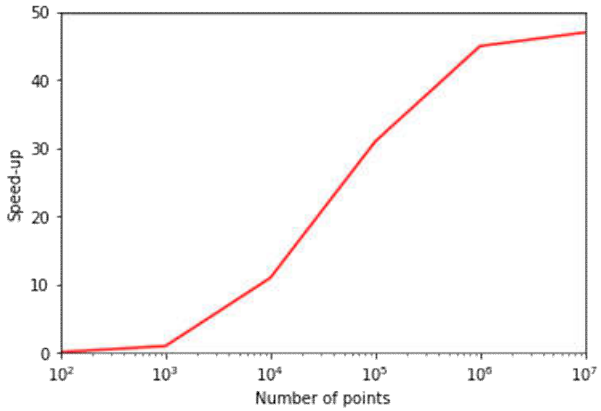

Figure 4: speed-up by varying the number of points, with 256 clusters and block size 1024.

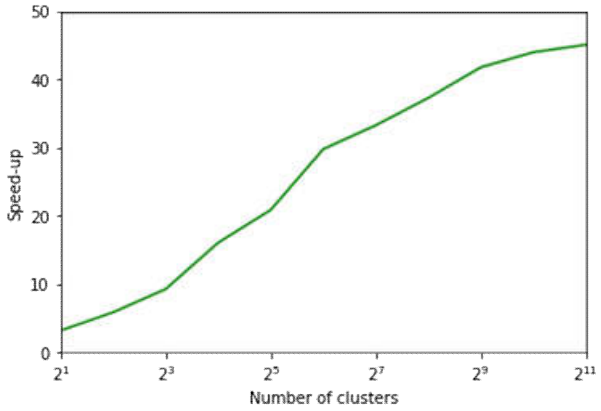

Figure 5: speed-up by varying the number of clusters, with 100 000 points and block size 1024.

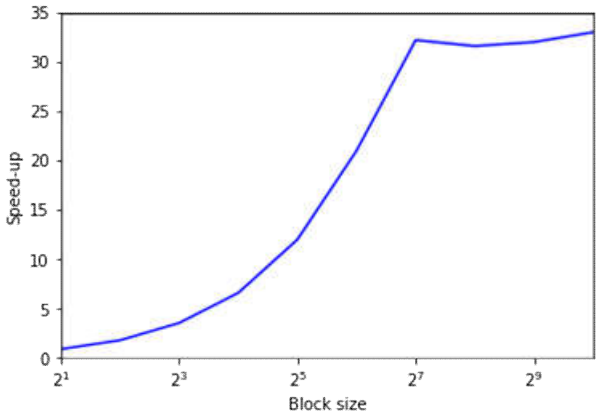

Figure 6: speed-up by varying the block size, with 100 000 points and 256 clusters.

For each of the previous plots, the trend of the results is the same regardless the choice of the fixed parameters, so that any consideration made on them states as general.

5. Conclusions

Since data are handled independently during the main operations, the algorithm greatly benefits of parallelization.

As shown in Figures 4, 5 and 6, exploiting GPU resources led to evident improvements regardless the block size, the number of points and the number of points, except for particularly small instances of the last one, that don't allow to tear down the load of overheading operations.

The optimal block sizes are those between 128 and 1024, that is the maximum possible on the before-mentioned card. Fixed this value, the maximum speed-up is around 47, and it is reached when the number of points or clusters reaches a order of magnitude of millions and thousands respectively.

References

[1] Wikipedia: k-means clustering

https://en.wikipedia.org/wiki/K-means_clustering