PC-2020/21 K-Means

Anonymous CVPR submission

Paper ID ****

Abstract

In this mid-term project we have implemented K-Means clustering and then we have applied it to image in order to achieve a color quantization as visual result. We have realized three different implementations, one sequential in Java and two parallel in Java and CUDA C/C++, and then compared the execution time to obtain and consider the speed up.

Future Distribution Permission

1. Introduction

K-Means, described in Section 2, is one of the simplest and most used algorithm for clustering points, although its performance, especially for its most basic version, is not so high. In general, the points must belong to an Euclidean space, like $\mathbb{R}^n$, (or a convex subset thereof) because the algorithm is based on the concepts of Euclidean distance and "average" between points; but is possible to generalize it for non-Euclidean space like string's space, where we have a notion of distance, but there's not an "average" between points.

In this project we have implemented K-Means considering the Euclidean case and for convenience we restrict the domain to $\mathbb{R}^n$ and its hyperrectangular subsets (which are the most common case).

As application of this algorithm we have choosen the images: applying a clusterization method to an image (to be precise to the pixels of an image) we obtain a color quantization of the image, which can be a first result for image segmentation. We describe this aspect in Section 3.

In the following sections we present our implementation solutions: the first of them is a sequential solution developed in Java, the second is a parallel version that use Java high level framework for concurrency and the third is also a parallel version but implemented in CUDA C/C++. In the lasts sections we report the result of the performance tests, discussing the speed up and proposing our personal considerations about the efficiency of the solutions.

2. K-Means Clustering

K-Means clustering is a family of clustering methods based on the a priori knowledge of the number of clusters into which the given points must be classified. Some algorithms to deduce the optimal value of k exist, but this is outside the main topic of this project, so we assume to have k as input parameter (and we evaluate the main results also as value of k varies).
In this project we will consider the standard version of K-Means, also known as naïve K-Means [1]; the idea behind it is quite simple:

0. select k points $m_1^{(0)}, \dots, m_k^{(0)}$ from the dataset that we will consider as **centroids** of step-0 cluster;

1. assign each point of the dataset to the cluster identified with the nearest centroid;

$$S_i^{(t)} = \left\{ x_p \middle| \|x_p - m_i^{(t)}\| \leq \|x_p - m_j^{(t)}\| \right\} \quad (1)$$

2. update the clusters' centroids with the average of its points;

$$m_i^{(t+1)} = \frac{1}{\left|S_i^{(t)}\right|} \sum_{x_j \in S_i^{(t)}} x_j \tag{2}$$

3. repeat from step 1 as long as the clusters "change enough" respect to the previous iteration.

This algorithm is guaranteed to converge (if we are using the Euclidean distance), though, as we can see, there are various aspects open to free choice, let's discuss them.

2.1. Clusters Initialization

About how to choose the initial centroids there are mainly two approaches: we can take as centroids k points in order to maximise their relative distance or we can take as centroids the point nearest to the centroids of a raw clusterization (perhaps based on a sample of the dataset). We have adopted the first approach as in [2] and we report the algorithm:

```
Pick the first point at random;
**while** there are fewer than $k$ points **do**
    Add the point whose minimum distance from the selected points is as large as possible;
**end while**
```

2.2. Stop Condition(s)

Also for the stopping rule is possible to define different conditions. Ideally we would like to stop when none of the data points change cluster from one iteration to the next one (that is we achieve the minimum sum of distances of every point from its cluster's centroids), since is guaranteed the convergence. As relaxation of this condition we could take as stopping rule the "convergence" of the diameter of the clusters (intended as the maximum distance between two points of a single cluster) or the position of centroids; obviously considering as "convergence" theirs change under a certain threshold.

In this project we assume as stop condition the latter case, taking as threshold the value TOLERANCE.

3. Image Color Quantization

As already mentioned, with K-Means is possible to get a color quantization of an image. This process aims to reduce the number of colors in an image, grouping the pixels in clusters and then assuming all pixels in a cluster of the same color, in particular the average color of the cluster.

To do that pixels should be considered as points in the space of colors (thus loosing momentarily the information relative to the spatial coordinate inside the image) and applying clusterization algorithm to them. This process can be applied to pixels from RGB color model, but also in gray scale model or others; in the first case the domain of the points is a subset of $\mathbb{R}^3$ ($[0, 255]^3$), in the second a subset of $\mathbb{R}$ ($[0, 1]$). In this project we will consider the RGB case without loss of generality. Moreover, as usual in practical use, the RGB values assume only integer values, so it isn't anymore an Euclidean space (*e.g.* the strings case mentioned above); however during the computation we abandon the discrete representation in favor of the continuous one; only at the end we return to the finite representation.

4. Solution Implementation

In our opinion the most suitable language/framework for implementing K-Means is CUDA, since most of calculation is independent and so the algorithm doesn't require much synchronization; and in addition the typical dimension of datasets is really huge (*e.g.* for an image in 4K resolution and 16:10 ratio, the pixels amount to 10.5M). Then we have thinked that for the second parallel version and the sequential one it would have been interesting compare them in the same language and so we have choosen Java since it support natively an high level framework for concurrency.

We have divided the algorithm in 3 subroutine: the calculation of the initial centroids, then the assignment of each point to a cluster and finally the update of centroids. All this operation require an iteration over all the points in dataset. The main idea that has guided the parallelization of the code is to parallelize this subroutines since they are, at least this is what we assumed theoretically, the hotspot in term of computation time (however we will discuss about this in the next Section presenting the results of profiling of the sequential version). A fourth minor subroutine is that of checking the stop condition, but this is an iteration over the k centroids and typically k is a quite low value; there is the possibility to parallelize also this operation, but we expect a relatively small improvement (if not a deterioration for too small value of k due to parallelization overhead).

5. Sequential Solution in Java

6. Parallel Solution in Java

7. Parallel Solution in CUDA

8. Tests and Results

References

[1] K-means clustering. Wikipedia.

[2] J. Leskovec, A. Rajaraman, and J. Ullman. *Mining Massive Datasets*. Cambridge University Press, 2011.

9. Additional Notes

9.1. Technical Specification

All tests were runned on/with:

- Intel Core i7-6500U (2.5 GHz up to 3.1 GHz, 2 cores, 4 threads, 4 MB of cache);
- NVIDIA GeForce 920MX (Compute Capability 5.0, Maxwell architecture, 2 GB of memory);
- Ubuntu 20.04;
- Java SE 11;
- CUDA 11;

10. Appendix