Lloyd's K-means Clustering algorithm

1st Rita Gomes
Informatics Department
Master in Informatics Engineering
University of Minho
Braga, Portugal
PG50723

2nd Pedro Araújo
Informatics Department
Master in Informatics Engineering
University of Minho
Braga, Portugal
PG50684

***Abstract*—This report consists of presenting an optimized version of a simple k-means algorithm based on Lloyd's algorithm, developed by the group in C programming language.**

***Index Terms*—algortihm, clusters, perfomance, metrics, optimization**

I. Introduction

The aim of this project is to evaluate the learning of code optimization techniques, including code analysis techniques/tools through the use of C programming language and the Search application as programming interface. For this, the Lloyd's K-means Clustering algorithm will be studied and the main objective is to implement it in the most efficient and appropriate way possible. Initially, a first version of the algorithm and the appropriate optimizations are implemented according to the impacts they have on the algorithm in studying. Finally, through the analysis of some metrics such as the average runtime execution, the final results of the experiment allow evaluating the performance of the algorithm. This article is divided into 4 sections. Section 2 explains the Lloyd's K-means Clustering algorithm and its implementation. Section 3 provides which metrics were considered and why, and the results obtained for each version of the algorithm. Finally, section 4 concludes the article and provides some considerations for future work.

II. Lloyd's K-means Clustering algorithm

To process the learning data, the K-means algorithm starts with a first group of randomly selected centroids, which are used as the beginning points for every cluster, and then performs iterative (repetitive) calculations to optimize the positions of the centroid. It halts creating and optimizing clusters when either:

- The centroids have stabilized — there is no change in their values because the clustering has been successful.
- The points don't change your associated cluster, because the centroid of cluster associated it's the closer one.

In other words, given a number k, separate all data in a given partition into k separate clusters, each with a center that acts as a representative, and there are also N points generated randomly that will be assign to the clusters. There are iterations that reset these centers then reassign each point to the closest center calculating the euclidean distances from the point to all centroid clusters. Then the next iteration repeats until the centers do not move. The algorithm is as follows:

1. Assign each data point to the cluster Ci corresponding to the closest cluster representative xi ($1 \leq i \leq k$)

2. After the assignments of all n data points, compute new cluster representatives according to the center of gravity of each cluster.

The Lloyd's algorithm often converges to a local minimum of the squared error distortion rather than the global minimum.

We used C as the programming language to implement this algorithm using one structure for the points (which was used both for Points and Clusters), where the variable ***idCsize*** was used for differentiation, so it can be either the id of the associated cluster for the Points array or the size of each cluster for the Clusters array.

```
typedef struct points
{
    int idC_size;
    float x;
    float y;
}Points;
```

Fig. 1. Struct for points

III. Exploration and evaluation of the solution

In order to obtain a better performance, the group decided to use strategies such as spatial and temporal locality. The following figure is related to the temporal location, where the "point" is accessed several times, so it is probably stored in the registry, which causes faster access to the data.

```
int associaPontos(Points *cluster, Points *pontos){
    int flag = 0;
    int clusterAntigo,clusterIdMin,i,j;
    float distMin;
    float distancias[K]; // vetor que irá conter as distancias eucli
    Points ponto,centroidMin;
    for(i = 0; i < N; i++){
            ponto = pontos[i];
            clusterIdMin = ponto.idC_size;
            clusterAntigo = clusterIdMin;
            centroidMin = cluster[clusterAntigo];
            distMin = calculaDist(ponto,centroidMin);
            for(j = 0; j < K; j++){
               distancias[j] = calculaDist(ponto,cluster[j]);
               distancias[j+1] = calculaDist(ponto,cluster[j+1]);
               j++;
            }
```

Fig. 2. Temporal locality

In the following image we have the use of spatial locality, since both the ***accumulateX*** and ***accumulateY*** are accessed in sequential positions in memory, which causes fewer misses in the cache.

Identify applicable funding agency here. If none, delete this.

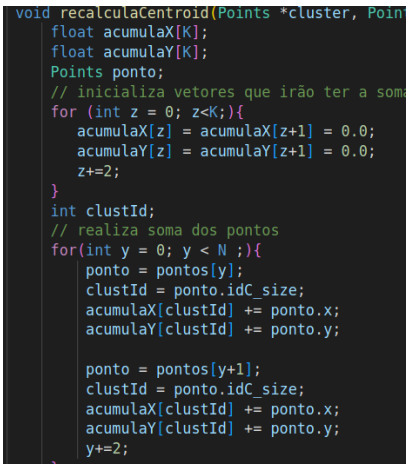

Fig. 3. Spatial locality

In order to compare the different types of optimization, the group decided to create the following table in order to make it easier to compare the results obtained in each one. The data was obtained by the perf tool.

Metrics	-O0	-O2	Loop unrolling && -O3 && vetorization	Loop unrolling && -O2	Loop unrolling && vetorization
Average Runtime	1min5.145s	29.1s	9.5s	28.07s	12.5s
Clock cycles	233.745.156.724	88.295.827.208	40.725.178.770	97.241.958.118	39.658.343.494
Instructions	285.940.139.906	66.865.344.392	39.553.696.185	65.939.115.471	63.859.409.989
Average CPI	0.8	1.3	1.0	1.5	0.6
L1-D Misses	153.392.938	143.051.315	140.245.357	143.735.845	139.841.602

Fig. 4. Results obtained with the different optimizations

Firstly, we did some tests around gcc -O flags (-00,-O2,-O3). The group realized that each degree of optimization produces a reduction of generated instructions, but it has the trade-off to increase the number of cycles. This increase its explained by the fact that compiler generates complex instructions in order to achieve performance, and with that complexity comes data dependencies such as Read After Write (RAW) that makes the CPU wait longer.

In order to optimize the original code, the group tried to use a well-known method, Loop-Unrolling. This technique basically removes or reduces iterations by anticipating next iterations into just one. It increases program efficiency because it decreases the number of instructions with the reduction of ***if*** statements and allows to execute them in parallel.

To perform this optimization, we did unrolling (with magnitude of 2) to almost all loops in our code, mainly in the functions that contained more iterations, such as: ***associaPontos*** (this function associates the points to another cluster if the distance is inferior) and ***recalculaCentroid*** (this function calculates the mean of all points associated to each cluster, and changes the centroid).

Another optimization technique the group attempted to do was to vectorize the code. Vectorization is a special case of Single Instructions Multiple Data (SIMD) that parallelize a single instruction stream capable of operating on multiple data elements, which processes each instruction very fast. This can be observed on the table above (Fig. 4), where the number of cycles was reduced in 1/3 of the process without vectorization. To vectorize our code, we tried as much as possible to avoid using data dependencies and if/else statements in inner loops.

One change that we did in our code was in function ***associaPontos***, where inside the innerloop there was the calculation of the distance between the point and the cluster centroid and also an ***if*** statement to change the associated cluster if the new cluster distance was smaller. We decided to change this function to first calculate all distances using loop unrolling, and only after that do the associated cluster changes, thus removing ***if*** statement of that loop and making it more vectorizable.

Accordingly to the obtained results, the group is able to conclude that the algorithm has its best perfomance when optimization is applied with loop unrolling, vetorization and flag -O3.

The output obtained by the group is shown in the following image:

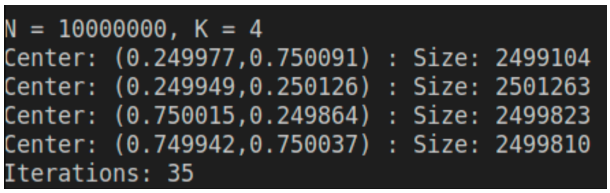

Fig. 5. Output

Although this output is not exactly the same as the expected output, it is very close, as only a few clusters were not considered and the number of iterations was lower.

IV. Conclusion and Future Work

With the accomplishment of this practical work, the group concludes that it was possible to consolidate knowledge taught during the classes about optimizations and analyze their impacts on performance. It was also a great way for us to explore perf tool for performance analysing.

The group believes they have accomplished with objective of project, however there is room for improvement in the future, such as implementing a more parallelized algorithm making the code even more vectorized and doing more testing with different metrics.

References

[1] Education Ecosystem, “Understanding K-means Clustering in Machine Learning”, Towards Data Science, Sep 2018.