# K-Means Algorithm Optimization

Analysis and solutions for optimizing computationally heavy algorithms

1st Vicente Moreira
*Escola de Engenharia*
*Universidade do Minho*
Braga, Portugal
pg50799@alunos.uminho.pt

2nd Joana Alves
*Escola de Engenharia*
*Universidade do Minho*
Braga, Portugal
pg50457@alunos.uminho.pt

***Abstract*—This document will present the various steps of analysis, optimization and evaluation taken with the objective of improving the performance of the well-known clustering algorithm *K-Means*.**

***Index Terms*—K-Means, Optimization, Analysis, Algorithm**

## I. Introduction

### A. Context

Since the general invention of computation, these machines had an exponential growth in their computational capacity, as the famous "**Moore's Law**" says, for every two years, the numbers of transistors duplicates. Although, this growth is only observed in the processing field, leaving other fields like memory/disk access speed and even battery capacity further behind every year.

For this reason, optimization of programs and applications is still relevant nowadays, since some small mistakes in the development of a critical function can reduce its performance dramatically.

### B. The Problem and Strategy Adopted

In this project, our team will develop a simple clustering algorithm known as *K-Means*. This algorithm will be written in the C programming language and will be based on the **Lloyd algorithm**.

After its implementation, our team will evaluate the developed code, with the objective of finding potential improvements on code efficiency and reduce processing time. These changes will be documented, implemented and tested, creating various versions of our algorithm.

## II. Tests and Metrics To Be Evaluated

In this section we will briefly explain how the many tests were conducted, along with what metrics were used to compare the various versions developed.

This algorithm will be executed in a remote machine, with the intent of controlling various external factors and provide a predictable set of outcomes. We are also going to execute 5 tests for each version, using 10 million points and 4 clusters, and calculate the median of all metrics, with the intent of reducing possible variations.

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

These will be the metrics evaluated in each test:

- **Execution Time (Texec)** - This is our main metric as it defines in real time how long the user will wait for the execution of the algorithm.
- **Clocks per Instruction (CPI)** - This metric will help in defining what kind of instructions are used and its super scalability.
- **Number of Instructions (#I)** - This metric will help compare the size of the developed code between the different versions.
- **Level 1 Cache Misses (LD1 Misses)** - This metric will help evaluate the efficiency of memory access.

## III. Initial Approach

### A. Data Structure

For our team's first approach, we considered using C structures like "Point" and "Cluster", to ease code writing and interpretation. But since structures are saved as pointers, and the structures might not be organized in memory, not only we would have to make two memory accesses to retrieve it's data, we also couldn't take advantage of concepts like **spacial locality**, so this idea was abandoned.

Our team ended up using two constants N and K, representing the number of points and clusters being used for the algorithm, respectively. For the data in the points and clusters, we used two arrays for each structure, one that defines the X coordinates and the other the Y coordinates, for a total of 4 arrays, 2 of N size and 2 of K size, where the index identifies which point or cluster is being read/written. We also used a N sized array of integers to define which clusters the points belong to, where the index identifies the point and the data in the cell determines the cluster.

### B. Algorithm Phases

In order to follow **Lloyd's algorithm** and create a clear division of phases, we decided to implement 4 functions:

- **Allocation** - This function will allocate all the necessary memory needed for the execution of the algorithm.
- **Initialization** - In this function, the variables allocated will be populated with random coordinates.
- **Cluster Assignment** - This phase is responsible to assign which clusters the points belong to. This attribution is

made by finding the smallest distance between the point and all the centroids using the Euclidean distance.

- **Recalculation of Centroids** - In this phase, the coordinates of the centroids are recalculated using the median of all the points the cluster was attributed.

These last two function shall be executed in a loop until, in the Cluster Assignment phase, no point changed its cluster.

## IV. Analysis and Improvements

### A. Analysis Method and Initial Findings

For each version of the algorithm, we used the commands ***perf stat*** and ***perf record*** to obtain our metrics and analysis of which functions and sections of code were requiring more processing time, so we could focus our efforts on improving the efficiency of those sections.

Before starting our tests, the team suspected that the most critical section would be the **Cluster Attribution** phase, where for each 10 million points, we would have to calculate the distance for all 4 clusters, for a total of 40 million calculations per iteration. Another suspected critical section was the **Recalculation of Centroids** phase, where every cluster would have to check the arrays of all 10 million points, and if they belonged to its cluster, sum the values of the coordinates.

### B. Versions Developed and Comparisons

In this section we will list all the versions of the algorithm developed, explaining their differences along with why we decided to make those changes in the first place. We will also present a table with the final results: (Check Fig 2 for a complete table of tests)

- **Version 1 - Base:** This version is the base version and represents the clustering algorithm in its simpler form. This version was compiled with the *-O0* flag.
- **Version 2 - Compiler Optimization:** This version is equal to the last one, only difference being the compilation process, using the compiler optimizations. This version was compiled with the *-O2* flag.
- **Version 3 - Reduced Function Call:** In this version, we removed the call to the function *distanciaEuclidiana*, opting instead to implement it directly on our code. This change aims to reduce the overheads created by function calling procedures, since this section is executed N * K times per iteration. This version was compiled with the *-O2* flag.
- **Version 4 - Reduced Array Iteration:** This version has a major difference in its logic on the **Recalculation of Centroids** phase. On the base version, for each cluster K, we iterate over the array of N points in order to recalculate the centroid, which meant iterating over the same N array K times. In this version, we updated the logic so we only have to iterate over the N array once, by storing the necessary results of the K clusters in arrays, completing the recalculating process once the iteration is over. This version was compiled with the *-O2* flag.
- **Version 5 - Non-Use of *math.h* Library** This version removes the usage of the ***sqrtf*** and ***powf*** functions present in the *math.h* library, used in the calculation of the Euclidean distance between points and centroids. The square root calculation was also removed entirely since the distance was only needed for comparison purposes, so withdrawing this step won't affect the final values. This version was compiled with the *-O2* flag.
- **Version 6 - Vectorization** In this version, we added support for vectorization by aligning all the arrays and compiling with the flags *-ftree-vectorize -msse4* and *-O2*.

| Version | Relative Gain | TExec (s) | CPI | #I | LD1 Cache Misses |
|---|---|---|---|---|---|
| V1 | - | 89,178 | 0,740 | 3,80E+11 | 4,12E+08 |
| V2 | 5,187 | 17,191 | 0,84 | 6,45E+10 | 4,06E+08 |
| V3 | 1,354 | 12,6918 | 0,76 | 5,17E+10 | 4,01E+08 |
| V4 | 1,568 | 8,0962 | 0,54 | 4,67E+10 | 1,64E+08 |
| V5 | 1,685 | 4,8036 | 0,4 | 3,27E+10 | 1,71E+08 |
| V6 | 1,041 | 4,6134 | 0,4 | 3,27E+10 | 1,71E+08 |

Fig. 1. Final results table

### C. Discussion of Results

As we can see in the table above, there was a large performance gain between **V1** and **V2** since the compiler optimizations reduced the number of instructions executed to 1/5 of it's non optimized version. This happens most likely due to the reduced number of instructions executed inside the inner loops in functions like **Cluster Assignment** and **Recalculation of Centroids**, since any instructions removed here will reduce the overall instruction count by N * K per Iteration. (EX: 10M * 4 * 32 = 1280 Millions).

For the same reason, we can see these gains in **V3**, where the instructions reduced were related to function calling procedures inside the inner loop of **Cluster Assignment**.

On **V4**, our performance gain came from eliminating the N * K loop inside the **Recalculation of Centroids** function. Theoretically, this should improve the function performance by K times, although this gain is not reflected on the results since this function, even though it's rated second on processing time, it doesn't occupy a large percentage of the algorithm.

On **V5**, for similar reasons as **V3**, we saw a good performance gain, along with the bonus of not needing to calculate the square root on every distance.

At last, on **V6**, we didn't see much performance gains as expected. After analysing the generated assembly code we could confirm the use of vectorized registers. We suspect that either the overheads brought by vectorization or our code lacking a good support for this technique made our improvements have little to no impact.

## V. Conclusion

Our team is satisfied with the results achieved on this project, having improved the original algorithm execution time from 89s to 4.6s, representing a 19x improvement. We are also motivated by the knowledge acquired in its development and the possibility of exploring this optimization further, since we should increase performance by correctly applying vectorization.

## VI. Attachments

| Version | NºTest | TExec (s) | CPI | #I | LD1 Cache Misses |
|---|---|---|---|---|---|
| V1 | 1 | 82,574 | 0,7 | 3,80E+11 | 4,08E+08 |
| | 2 | 83,311 | 0,7 | 3,80E+11 | 4,10E+08 |
| | 3 | 81,96 | 0,7 | 3,80E+11 | 4,10E+08 |
| | 4 | 91,273 | 0,7 | 3,80E+11 | 4,12E+08 |
| | 5 | 106,771 | 0,9 | 3,80E+11 | 4,21E+08 |
| V2 | 1 | 18,496 | 0,9 | 6,45E+10 | 4,06E+08 |
| | 2 | 16,128 | 0,8 | 6,45E+10 | 4,06E+08 |
| | 3 | 14,538 | 0,7 | 6,45E+10 | 4,07E+08 |
| | 4 | 17,886 | 0,9 | 6,45E+10 | 4,04E+08 |
| | 5 | 18,907 | 0,9 | 6,45E+10 | 4,07E+08 |
| V3 | 1 | 11,637 | 0,7 | 5,17E+10 | 4,00E+08 |
| | 2 | 11,813 | 0,7 | 5,17E+10 | 4,00E+08 |
| | 3 | 14,144 | 0,8 | 5,17E+10 | 4,02E+08 |
| | 4 | 15,17 | 0,9 | 5,17E+10 | 4,02E+08 |
| | 5 | 10,695 | 0,7 | 5,17E+10 | 4,00E+08 |
| V4 | 1 | 6,911 | 0,5 | 4,67E+10 | 1,63E+08 |
| | 2 | 7,366 | 0,5 | 4,67E+10 | 1,63E+08 |
| | 3 | 9,429 | 0,6 | 4,67E+10 | 1,67E+08 |
| | 4 | 9,505 | 0,6 | 4,67E+10 | 1,66E+08 |
| | 5 | 7,27 | 0,5 | 4,67E+10 | 1,63E+08 |
| V5 | 1 | 5,104 | 0,4 | 3,27E+10 | 1,71E+08 |
| | 2 | 4,607 | 0,4 | 3,27E+10 | 1,70E+08 |
| | 3 | 5,011 | 0,4 | 3,27E+10 | 1,71E+08 |
| | 4 | 4,639 | 0,4 | 3,27E+10 | 1,71E+08 |
| | 5 | 4,657 | 0,4 | 3,27E+10 | 1,71E+08 |
| V6 | 1 | 4,684 | 0,4 | 3,27E+10 | 1,71E+08 |
| | 2 | 4,535 | 0,4 | 3,27E+10 | 1,70E+08 |
| | 3 | 4,531 | 0,4 | 3,27E+10 | 1,71E+08 |
| | 4 | 4,685 | 0,4 | 3,27E+10 | 1,71E+08 |
| | 5 | 4,632 | 0,4 | 3,27E+10 | 1,71E+08 |

Fig. 2. Result Table Completes