UNIVERSITÀ
DEGLI STUDI
FIRENZE

Parallel Computing

K-Means Algorithm
A CUDA implementation

Giacomo Magistrato

UNIVERSITÀ DEGLI STUDI FIRENZE

Introduction

- **Objective**: The aim of this project is to outline the differences, in terms of performances, between a Sequential and a Parallel implementation of the K-Means Algorithm

- The Parallel implementation makes use of **CUDA** in order to implement parallelism. This can potentially lead to a much faster execution compered to the sequential implementation of the instructions.

- In order to evaluate performances, both implementation uses the same:
  - Dataset.
  - Initial centroids.
  - Stop criterion (the number of iteration).

- The dataset used for testing the project, was created using a Python script.

- The source code of this project can be retrieved at the folowing location:
- https://github.com/Mayo98/KMeansParallel

Algorithm Overview

The K-Means clustering is an unsupervised machine learning algorithm that is used to partition a given dataset into multiple groups (clusters), based on the similarities and correlations between the contained records.

By being unsupervised, the algorithm is capable of detecting complex patterns and trends autonomously, with no prior guidance.

The objective is to minimize the intra-cluster differences between the points and maximize the inter-cluster ones instead.

The differences between the points (2D points in this case) are evaluated with the **Euclidean distance**.

At each iteration for each point the distance from the centroids is calculated and the point is assigned to the cluster of closest centroid. Then for every cluster the position of the centroid is updated.

The algorithm ends once reached the fixed number of iteration.

Algorithm Overview

Initialization

- The algorithm starts by calling the function *index_Generator()* which select K points from the dataset as initial centroids.

- These will be used in every execution of the algorithm.

- Setting the parameter K we define how many cluster we want to reach.

- Setting the parameter *iter* we define the number of iteration of the algorithm.

- Data are read and all the coordinates are stored in 2 arrays in order to implement SoA

- An array of integer is used to identify the membership of the points to a cluster. It's initialized with all -1 values. First selected index defines first clusters.

Algorithm Overview

Sequential Version

Assignment

- The algorithm iterates all the points and through the function *getNearestClusterId()* calculates distances from cluster centroids following the Euclidean Distance metric.
- Then assigns the points to the closest cluster and updates the size of the corresponding cluster.

Update

- This step consists into computing the new centroids positions by calculating the average coordinates of the points contained inside the related clusters. The centroids then occupy those positions to better approxymate the cluster's characteristics. The new positions will be used during the next iteration to compute the distances again.

Algorithm Overview

Parallel Version

Assignment

- The Assignment is made using a CUDA Kernel function called *ClusterAssignment().*

- Using *idx = blockIdx.x*blockDim.x + threadIdx.x* is identified the global thread index within the CUDA execution grid.

- Each Thread iterates through the points following idx and calculate the distances from each centroid, assigning points to the closest centroid.

Update

- In this phase is used another CUDA Kernel function called *clusterPointsSum()*.

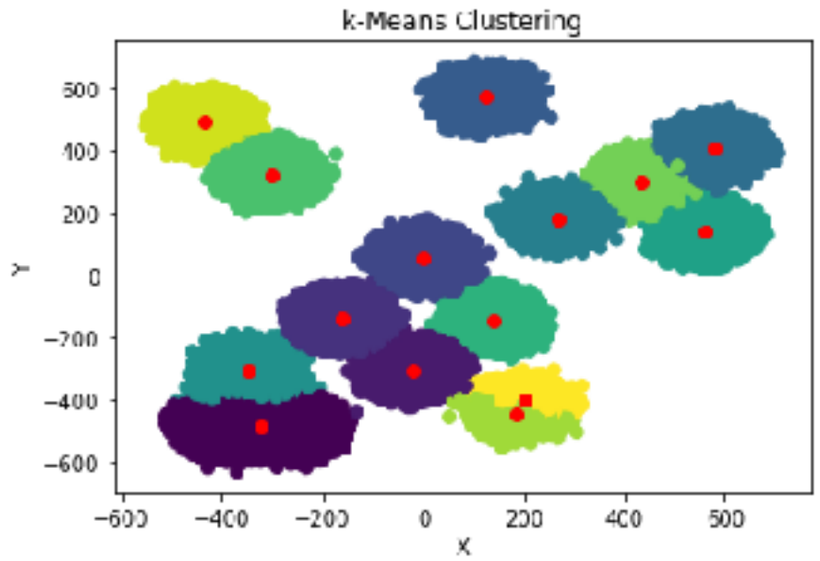
- Atomically updates the sum of the x and y coordinates of the cluster, using *AtomicAdd.*

- Then computer the new centroid by dividing each sum for the cluster size.

Algorithm Overview

Printing phase

- This part belong to both implementation.
- Once max iteration is reached, the points with the respective clusters they belong to are saved in a file.
- Centroids are stored in a separated file.
- Then using a Python script, using these files as the input, is it possible to plot the dataset clustered.



Testing Machine

Specification

The unifi server was used to compile the project and run the tests via ssh

- Ubuntu 22.04.4 LTS (GNU/Linux 6.5.0-41-generic x86_64)
- Intel(R) Xeon(R) Silver 4314 CPU @ 2.40GHz, 16 cores
- Memory: 64GB
- NVIDIA Corporation GA106 [RTX A2000 12GB]

Testing

The testing is based on three strategies:

- Testing the execution time using different sizes of the data set (number of points)

- Testing the execution time using different numbers of Thread per Blocks for the parallel version.

- Testing the execution time using different numbers of centroids.

In order to evaluate the performance and compare the execution times of the sequential version(T_{seq}) and the parallel one(T_{par}), is used the Speedup function:

$$SpeedUp = \frac{Tseq}{Tpar}$$

What do I expect

Changing size of dataset

- Speedup is expected to be lower to 1 when the dataset size is small (e.g. <=1000 points) because of the overhead introduced by the parallelization for the allocation and synchronization of the threads, that makes sequential version faster. While speedup could be grater than 1 as soon as the number of points of the dataset grow up.

Changing numbers of threads

- Speedup is expected to be grater proportionally to the number of threads and the text file size.

Changing the number of centroids

- Speedup is expected to be grater proportionally to the number of centroids due to the number of possible assignments that a point can have.

Results

In the table below are shown the speedups for the executions of the algorithm. On **rows** are diplayed the **number of points** of the dataset, on the **columns** the number of **TPB**.

K-Means Speedups with K = 15

	8	16	32	128
1000	1.10	0.90	0.88	0.62
100000	6.4	7.91	10.65	13.21
750000	7.5	11.92	17.70	18.30
1 M	7.60	12.20	18.01	18.51

K-Means Speedups with K = 5

	8	16	32	128
1000	0.88	0.71	0.70	0.55
100000	5.04	7.20	8.6	7.11
750000	5.45	7.86	10.02	10.5
1 M	5.56	7.92	10.05	11.16

Average execution time with K = 15

Size \ N threads	Sequential	8	16	32	128
7500000	77.60s	10.34s	6.53s	4.38s	4.22s
1 M	104.25	13.71s	8.54s	5,79s	5.63s

Average execution time with K = 5

Size \ N threads	Sequential	8	16	32	128
7500000	28.02s	5.14s	3.56s	2.80s	2.52s
1 M	36.96	4.67s	6.64s	3.67s	3.31s

Results(2)

In the table below are shown the speedups for the executions of the algorithm. On **rows** are diplayed the **number of points** of the dataset, on the **columns** the number of **TPB**.

K-Means Speedups with K = 50

	8	16	32	128
1000	1.12	0.93	0.77	0.75
100000	7,6	13,71	18.10	21.03
750000	7.7	14.69	19,5	25,9
1 M	7.68	14,56	24,40	26.30

Average execution time with K = 50

Size \ N threads	Sequential	8	16	32	128
7500000	252,53s	32,60s	17,19s	12,91s	9,53s
1 M	336,79s	43,83s	23,13s	13,8s	12,71s

Performance Evaluation

- As can be seen from the results, running the algorithm in parallel with small datasets leads to terrible speedups, regardless of the number of centroids.

- As expected as the size of the dataset increases, the speedup also increases.

- We can see a big difference in terms of speedup by changing the number of centroids. With K = 15 the number of operations grows and parallelism works very well in these cases.

Due to the observations made above we can see that:

- The worst speedup is 0.55 and occurs in the case where K = 5 and the dataset size is the smallest.

- The best speedup is 26.30 and occurs in the case where K = 50 and the dataset size is the greatest.