Parallel Computing

DBSCAN Algorithm
A CUDA implementation

Giacomo Magistrato

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 DBSCAN 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 point for the neighbors classification.
  - Epsilon radius.
  - Minimum points to define a core point.

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

- The source code of this project can be retrieved at the following location:
- https://github.com/Mayo98/DBSCAN-Algorithm-Parallel

Algorithm Overview

- The **DBSCAN** (Density-Based Spatial Clustering of Applications with Noise) clustering is an unsupervised machine learning algorithm designed to identify clusters of densely distributed points and separate noise (outliers).
- It handle data with **arbitrary shapes.**
- It is **insensitive to the number of clusters**.

Key Parameters:

- **ε** (eps) the maximum radius to consider points as neighbors.
- **minPts**: the minimum number of points required to define a cluster.

During the execution DBSCAN classify points as:

- **Core** point: a point with at least minPts points within the ε radius.
- **Border** point: a point that does not meet the density of minPts but is close to a core point.
- **Noise** point: a point that is neither core nor border (considered as noise).

Algorithm Overview

Once the points classification is done, DBSCAN perform the **cluster expansion.** From each core point clusters are created by including all points directly or indirectly reachable within the **ε** radius.

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**.

After the cluster expansion phase a **centroid** is defined for each cluster.

Algorithm Overview

Initialization

- Setting the parameters **ε** and **minPts** we define the accuracy of the clustering.
- Data are read and all the coordinates are stored in 2 arrays in order to implement SoA
- In order to make the execution faster all the points are also stored in a KD-Tree that will be used for the neighbors research.

UNIVERSITÀ DEGLI STUDI FIRENZE

Sequential Algorithm Overview

Neighbors Research

- The algorithm starts by calling the function *checkNearPoints()* which iterates over each point and for each of these generates an **adjacency list** containing the neighbors within a radius **ε.**
- For each point the research is made through the KD-Tree.

Cluster Expansion

- Once the neighbor research is done, a **DFS** is performed starting from the first core-point encountered in the dataset. This will create clusters by adding every directly or indirectly reachable points from the starting one.

Centroids definition

- This step consists into computing 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.

Parallel Algorithm Overview

Neighbors Research

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

Performed in 2 different kernel function call:

- *CountNeighbors():* each thread iterates through the points following idx and calculating the distance from all the other points. If it finds neighbors, it update the *neighborsCounter* and the *totalEdges* of the dataset.
- Once the cont is ended a *neighborStartIdx* array is computed using a Prefix Sum from the Cub library. This will help to find the exact idx where the neighbors for a point starts in the Adjacent list.
- Using the *totalEdges* value, the *adjacientList* is allocated in the host*;*
- *makeGraphStep2():* same behavior as CountNeighbors() but this time each thread fill the adjacientList with the neighbors found for each point.

Parallel Algorithm Overview(2)

Cluster Expansion

- Once the neighbor research is done, a parallel **BFS** is performed starting from the first core-point encountered in the dataset. This will create clusters by adding every directly or indirectly reachable points from the starting one.
- A BFS exploration is performed because DFS is best suited for sequential execution only.

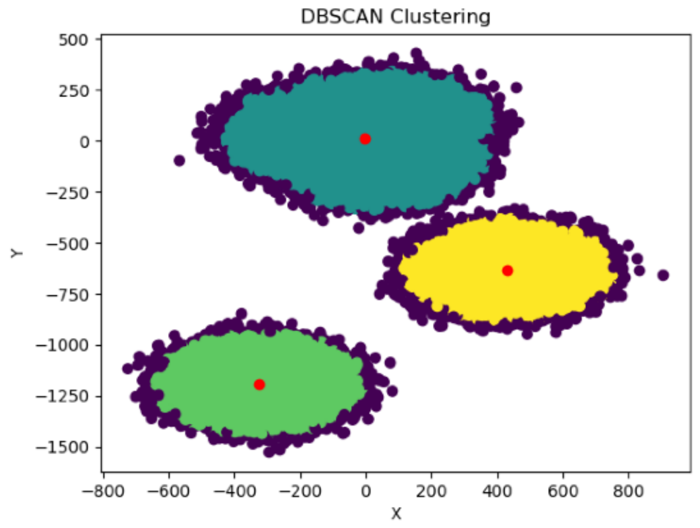
Centroid definition

- 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 centroids are computed, 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). Different size of the datasets but same domains. Except for the smallest datasets.
- Testing the execution time using different numbers of Thread per Blocks for the parallel version.
- Testing the execution time using different values for the key parameters **ε** and **minPts**.

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 ε and minPts

- Speedup is expected to be grater proportionally to the value of **ε**. Increasing it there will many more neighbors for each points and consequently many more points to analyze and cluster. While the executions time will be inversely proportional to the value of minPts.

UNIVERSITÀ DEGLI STUDI FIRENZE

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**.

Speedups with eps=10 - minPts= 25

	8	16	128	512
1000	0.06	0.06	0.06	0.06
100K	2.91	3.29	3.86	4.20
600K	5.00	8.43	13.77	13.79
800K	5.74	8.24	12.48	12.73
1M	5.02	8.53	13.99	14.22

Average execution time with eps=10 - minPts= 25

Size \ N threads	Sequential	8	16	128	512
1000	30ms	520ms	472ms	466ms	467ms
100K	7.65s	2.62s	2.32s	1.98s	1.83s
600K	123.90s	24.74s	14.68s	9.28s	8.98
800K	201.25s	35.05s	24.40s	16.12s	15.80s
1M	280.92s	55.96s	32.92s	20.07s	19.75s

Speedups with eps=10 - minPts= 250

	8	16	128	512
1000	0.06	0.06	0.07	0.07
100000	2.80	2.03	1.94	1.79
600K	4.95	8.35	13.62	13.74
800K	5.67	8.22	12.44	12.73
1M	5.01	8.53	13.88	13.99

Average execution time with eps=10 - minPts= 250

Size \ N threads	Sequential	8	16	128	512
1000	50ms	632ms	600ms	580ms	594ms
100K	7.44s	2.66s	3.68s	4.05s	4.14s
600K	123.04s	24.56s	14.62s	9.03s	8.97s
800K	198.27s	34.95s	24.02s	15.96s	15.52s
1M	277.68s	55.40s	32.59s	20.01s	19.84s

UNIVERSITÀ DEGLI STUDI FIRENZE

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**.

Speedups with eps=15 - minPts= 15

	8	16	128	512
1000	0.14	0.13	0.12	0.13
100K	4.47	5.80	5.86	5.92
600K	7.28	12.06	18.86	19.03
800K	8.67	10.53	24.91	24.92
1M	11.11	18.84	30.23	30.83

Average execution time with eps=15 - minPts= 15

Size \ N threads	Sequential	8	16	128	512
1000	73ms	529ms	558ms	570ms	552ms
100K	12.43s	2.84s	2.17s	2.12s	2.09s
600K	187.26s	25.71s	15.51s	9.92s	9.84s
800K	489.78s	46.47s	36.29s	19.87s	19.66s
1M	768.86s	69.19s	40.80s	25.39s	24.93s

Speedups with eps=5 - minPts= 250

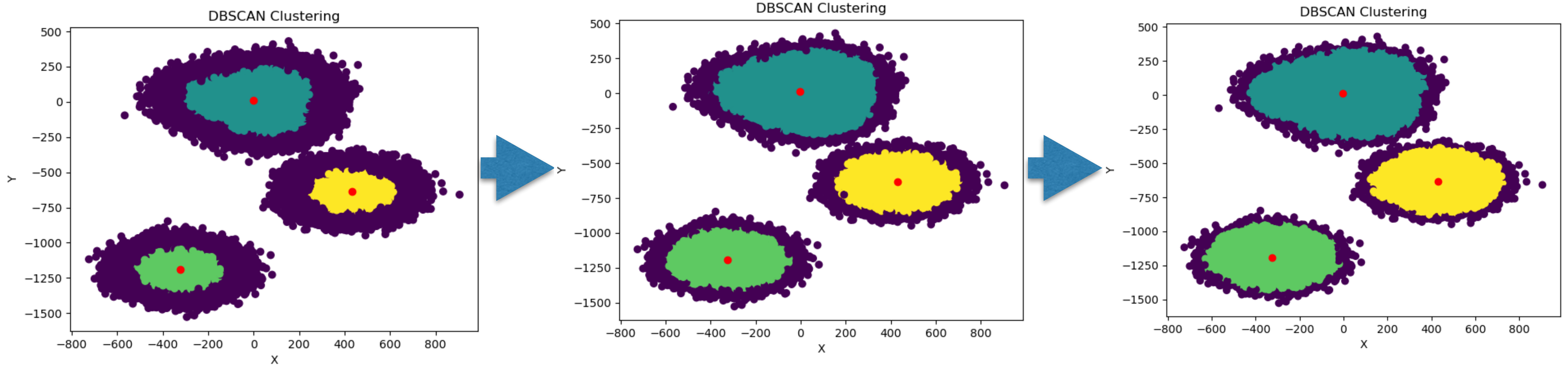
	8	16	128	512
1000	0.04	0.03	0.03	0.02
100K	1.02	1.35	0.84	0.67
600K	1.12	1.98	3.56	3.67
800K	1.20	2.00	4.01	4.06
1M	1.22	2.21	4.07	4.10

Average execution time with eps=5 - minPts= 250

Size \ N threads	Sequential	8	16	128	512
1000	40ms	450ms	521ms	602ms	594ms
100K	2.31s	2.26	1.71	2.74	3.49s
600K	26.29s	23.54s	13.26s	7.41s	7.24s
800K	60.35s	40.67s	30.10s	16.08s	15.89s
1M	78.17s	63.72s	35.30s	19.19s	19.05s

Clustering Result

- In the images below are shown some results of the clustering for the dataset with 1 million points.
- By varying the key parameters it is possible to see the differences in results



Eps = 10, minPts 250

Eps = 10, minPts 25

Eps = 15 minPts = 15

- Different parameters has different number of computations

Performance Evaluation

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

- 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 values of **ε** and **minPts**. With **ε** = 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 0.02 and occurs in the case where **ε** = 5, minPts= 250 and the dataset size is the smallest.

- The best speedup is 30.83 and occurs in the case where **ε** = 15, mintPts = 15 and the dataset size is the greatest.