PC-2024/25 Parallel Computing Laboratory

Giacomo Magistrato
E-mail address
giacomo.magistrato@edu.unifi.

Abstract

The objective of this report is to analyze and evaluate the advantages of implementing parallelism through the GPUs utilization, in relation to the K-Means clustering algorithm. The report aims at highlighting how parallel programming can enhance the efficiency of the algorithm, and under which circumstances, with the eventually encountered limitations. More specifically, the benefits of parallelism have been outlined by implementing the algorithm in C++ and creating a parallel version of the program, in which part of the code is executed by a GPU. The source code will be released publicly.

1. Introduction

Density-Based Spatial Clustering of Applications with Noise (DBSCAN) is a powerful and widely used clustering algorithm that groups data points based on density rather than predefined cluster shapes or the number of clusters. Unlike partitioning methods such as k-means, which require the number of clusters to be specified in advance, DBSCAN autonomously identifies clusters of arbitrary shapes while effectively handling noise and outliers.

DBSCAN operates based on two key parameters: ϵ (epsilon), which defines the radius of a neighborhood around a data point, and MinPts, which specifies the minimum number of points required to form a dense region. The algorithm classifies points into three categories: core points, which have at least MinPts neighbors within a radius of ; border points, which fall within the neighborhood of a core point but do not meet the MinPts requirement themselves; and noise points, which are neither core nor border points and are treated as outliers.

One of the main advantages of DBSCAN is its ability to discover clusters of varying shapes and sizes, making it particularly effective for spatial data, image segmentation, and anomaly detection. Additionally, it does not assume a spherical cluster structure, unlike k-means, which is sensitive to initial centroids and struggles with non-convex clusters. Another key strength of DBSCAN is its robustness to noise, as it naturally separates outliers from dense clusters.

However, DBSCAN also has certain limitations. The selection of optimal and MinPts values can be challenging, as these parameters significantly impact clustering results. Moreover, DBSCAN struggles with datasets where clusters have varying densities, as a single value may not be appropriate for all clusters. Additionally, its performance can degrade with high-dimensional data due to the curse of dimensionality, where distance metrics become less meaningful.

Despite these challenges, DBSCAN remains a fundamental algorithm in unsupervised learning and is widely used in applications such as geospatial analysis, social network analysis, and customer segmentation. Its ability to detect meaningful structures in data without requiring prior knowledge of the number of clusters makes it a versatile and valuable tool in data mining and machine learning. In this projects a sequential and a parallel version of this algorithm are proposed in

order to evaluate the differences in terms of performances.

2. The DBSCAN Algorithm

The algorithm consists of a sequence of steps in which the data points are assigned to a cluster depending on the proximity to the other points. The given dataset allows to work directly with no encoding necessity, and the preprocessing logic of the data has not been implemented. The implemented clustering logic comprises the following steps:

- **Initialization**: Data are read from file and stored in arrays in order to implement SoA. Key parameters, ϵ and minPts, are defined. 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. This regard only the sequential implementation.

- **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 based onto a specific distance metric. In this case, the **Euclidean distance** between two n-dimensional points:

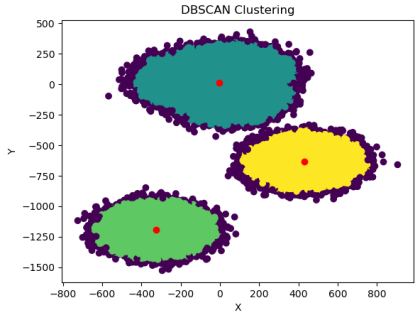
  $d(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$

  $\mathbf{x} = (\mathbf{x}_1, x_2, \ldots, x_n)$
  $\mathbf{y} = (y_1, y_2, \ldots, y_n)$
  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.

- **Result:** After the fixed iteration all the results are stored in external files. Using a Python script result can be plot as following:



3. GPU implementation

The parallelized algorithm consists of three main parts, for which three CUDA kernel functions have been accordingly defined. The parallel version of the project then relies on the kernels, that are exe- cuted on the GPU, to achieve better performances.

3.1. Neighbors Research

For the first part of the algorithm, two different kernel function are definet in order to identify the neighborhood of each point.
CountNeighbors()
Each thread iterates through the points following idx and calculating the distance from the other points. If it finds neighbors, it update the neighborsCounter array 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. The "global " keyword tells the compiler that the kernel is launched by the host, but executed on the device. All the pointers to

the device allocated memory are passed here as references in order to allow each thread to work with them. The kernel is typically launched with a call in which the total number of blocks and threads per block is specified just before the invocation. Below is presented the code for this section:

```
__global__ void count(...) {
    int idx = threadIdx.x + blockIdx.x *
        blockDim.x;
    while (idx < numPoints) {
        float epsSquared = eps * eps;
        unsigned int num = 0;
        for (int i = 0; i < numPoints; i++)
             {
            if (idx == i) continue;
            float distSquared = ((d_xval[i]
                 - d_xval[idx]) * (d_xval[i]
                 - d_xval[idx])) +((d_yval[i
                ] - d_yval[idx]) * (d_yval[i
                ] - d_yval[idx]));
            if (distSquared <= epsSquared)
                {
                num++;
            }
        }
        d_nodeDegs[idx] = num;
        atomicAdd(d_totalEdges, num);
        idx += blockDim.x * gridDim.x;
    }
```

After the first kernel call, the execution come back to the host where the Adjacient list is allocated using the value of *totalEdges*. Using the prefix sum of the Cub library the *neighborStarIndices* array is computed.

```
auto *h_adjList = (unsigned int *) malloc(*
    h_totalEdges * sizeof(unsigned int));
    CUDA_CHECK(cudaMalloc(&d_adjList, *
        h_totalEdges * sizeof(unsigned int))
        );
    void* d_temp_storage = nullptr;
    size_t temp_storage_bytes = 0;
    cub::DeviceScan::ExclusiveSum(
        d_temp_storage, temp_storage_bytes,
        d_nodeDegs, d_neighborStartIndices,
        total_points);

    cudaMalloc(&d_temp_storage,
        temp_storage_bytes);

    cub::DeviceScan::ExclusiveSum(
        d_temp_storage, temp_storage_bytes,
        d_nodeDegs, d_neighborStartIndices,
        total_points);

    cudaFree(d_temp_storage);
```

MakeGraphStep2 same behavior as CountNeighbors() but this time each thread fill the adjacientList with the neighbors found for each point.

```
__global__ void makeGraphStep2KernelA(...)
    {
    unsigned int idx = threadIdx.x +
        blockIdx.x * blockDim.x;

    while (idx < numPoints) {
        int startIdx =
            d_neighborStartIndices[idx];
        int countNeighbors = 0;

        for (unsigned int i = 0; i <
            numPoints; i++) {
            if (idx == i) continue;
            float distSquared = (d_xval[i]
                - d_xval[idx]) * (d_xval[i]
                - d_xval[idx]) +(d_yval[i] -
                 d_yval[idx]) * (d_yval[i]-
                d_yval[idx]);

            if (distSquared <= eps * eps) {
                unsigned int curr =
                    startIdx +
                    countNeighbors;
                adjList[curr] = i;
                countNeighbors++;
            }
        }
        idx += blockDim.x * gridDim.x;
```

3.2. Cluster Expansion

Once the neighbor research is done, a parallel BFS is performed starting from the first corepoint 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.

```
__global__ void BFS(...) {
    unsigned int idx = threadIdx.x +
        blockIdx.x * blockDim.x;
    unsigned int numFrontier = 0;
    while (idx < total_points) {
        if (d_frontier[idx] == 1) {
            atomicExch(&d_frontier[idx], 0)
                ;
```

```
            atomicSub(d_frontierCounter, 1)
                ;
            atomicExch(&d_visited[idx], 1);
            if (d_clusterVal[idx] ==
                NOT_CLASSIFIED ||
                d_clusterVal[idx] == NOISE)
                {
                atomicExch(&d_clusterVal[
                    idx], clusterId);
            }
            // Stop BFS se idx non    Core.
            if (d_nodeDegs[idx] >= minPts)
                {
                unsigned int idx_start =
                    d_neighborStartIndices[
                    idx];
                for (unsigned int i = 0; i
                    < d_nodeDegs[idx]; i++)
                    {
                    unsigned int v =
                        d_adjList[idx_start
                        + i];
                    if (d_visited[v] == 0)
                        {
                        atomicExch(&
                            d_frontier[v],
                            1);
                        atomicAdd(
                            d_frontierCounter
                            , 1);
                    }}}}
        idx += blockDim.x * gridDim.x;
```

3.3. 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 in order to avoid concurrency. Then computer the new centroid by dividing each sum for the cluster size.

```
__global__ void clusterPointsSum(...){
int idx = blockIdx.x * blockDim.x +
    threadIdx.x;
    while(idx<total_points) {
        int clusterId = d_clusterVal[idx];
        if(clusterId >= 0) {
            atomicAdd(&(d_clusterSumX[
                clusterId]), d_xval[idx]);
            atomicAdd(&(d_clusterSumY[
                clusterId]), d_yval[idx]);
            atomicAdd(&(d_clusterSize[
                clusterId]), 1);
        }
        idx += blockDim.x * gridDim.x;
```

4. Performance Analysis

The testing phase has been carried out by executing both the sequential and the parallel algorithm implementations multiple times. Both versions share the same dataset and the same point where the clustering starts. The performed tests consist of multiple execu-tions with different combinations of data points numbers, epsilon and minPts values.

4.1. Speedup

In order to evaluate the benefits in terms of performances between the two versions of the program, the speedup metric has been calculated. The speedup is the Sequential Execution Time divided by the Parallel Execution Time

4.2. Considerations

By fixing the number of data points and looking at the elapsed times of the two versions, it is possible to notice that the time required by the CPU grows much faster than the time required by the GPU, but in both cases the increment is strictly dependent on the values of ϵ and MinPts. It's worth noticing that the bigger the input data size is, the more is beneficial to achieve parallelism thanks on the GPU.

4.3. Results

In order to evaluate the overall performances, each one of the following combinations of clusters and data points sizes has been tested 10 times. The tests have been structured with all the combinations in between the following sets:
ϵ = 5, 10, 15
Datapoints = 1000, 100000, 600000, 800000, 1000000
TPB = 8,16, 128, 512
minPts = 5,15,25,250 Speedups are show below:

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	1.56	2.00	4.01	4.06
1M	1.22	2.21	4.07	4.10

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

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

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

5. Conclusion

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 and we can see a big difference in terms of speedup by changing the ϵ values. With $\epsilon = 15$ and minPts = 15 the number of operation grows and parallelism works very well in this cases.

Due to the observations made above we can see that the worst speedup is 0.06 and occurs in the case where $\epsilon = 5$ and the dataset size is the smallest while the best speedup is 39.85 and occurs in the case where $\epsilon = 15$, min pts = 15 and the dataset size is the greatest.

The source code of the project and its related re- porting are publicly available on GitHub: https://github.com/Mayo98/DBSCAN-Algorithm-Parallel