# Scalable Scientific Computing
# CMDA 4634

# CUDA G-DBSCAN
# Fall 2023

Bobby Alvarez

December 22, 2023

# Contents


**I Final projects** **5**

**1 CUDA G-DBSCAN** **7**

1.1 CUDA G-DBSCAN Algorithm . . . . . 7

1.1.1 G-DBSCAN Algorithm . . . . . 8

1.1.2 CUDA Implementation . . . . . 10

1.1.3 Testing and Verification . . . . . 14

1.1.4 Future Work and Improvements . . . . . 15

1.2 Bibliography . . . . . 17

# Part I

# Final projects

# Lecture 1

# CUDA G-DBSCAN

Bobby Alvarez

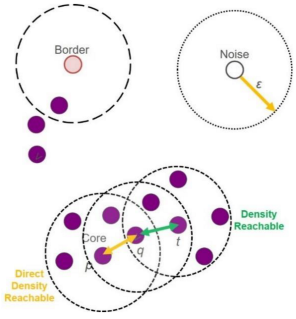
## Contents


**1.1 CUDA G-DBSCAN Algorithm . . . . . 7**
1.1.1 G-DBSCAN Algorithm . . . . . 8
1.1.2 CUDA Implementation . . . . . 10
1.1.3 Testing and Verification . . . . . 14
1.1.4 Future Work and Improvements . . . . . 15
**1.2 Bibliography . . . . . 17**


## 1.1 CUDA G-DBSCAN Algorithm

### DBSCAN Preliminaries

The foundations of the CUDA G-DBSCAN project are based on the clustering algorithm known as DBSCAN or Density-based spatial clustering of applications with noise. This algorithm creates clusters of points that are located within a distance of each other with a certain density based on two parameters, $\epsilon$ and minPoints. $\epsilon$ is the radius around a point, and minPoints is the minimum number of points, all within $\epsilon$ euclidean distance of each other needed to create a density cluster. In its essence, the DBSCAN algorithm can be broken down into these steps:




1. Find all neighbor points within $\epsilon$ euclidean distance of each point, assign the point as CORE

point if it has more than minPoints neighbors.

2. Find each connected neighbor that is a CORE point for each point and add it to a cluster.

3. If a point is not a core point, assign the point to a cluster that has a core point $\epsilon$ distance away, this is called a BORDER point. Otherwise assign the point as a NOISE point.

This algorithm has a time complexity of $O(n^2)$ in the worst-case scenario and $O(n \log n)$ average-case time complexity.

### 1.1.1 G-DBSCAN Algorithm

Now we propose an algorithm given in [**5**] that makes the DBSCAN algorithm easily implemented in parallel code. In general, the way the G-DBSCAN algorithm works is: Pick a CORE point v in the data set. Set the clusters of the unvisited neighbors of v as v's cluster. Choose a new CORE point v that has not been visited and create a new cluster. Repeat until all points have been visited. We can further break this algorithm down into four different elements, starting with building a graph that connects all points with other points in its neighborhood. We then have a function that identifies the cluster for each point, in this function we have a call to a breadth first search function that calls BFS GPU kernel that visits and marks each CORE neighbor of neighbors for each unvisited point in the data set. From the marked points, we can then identify if the point type and cluster. Here we layout the pseudocode for each element of the algorithm:

**Algorithm 1** Graph Construction

```
1: procedure MakeGraph(minPoints, ϵ, data, Graph)
2:     for c_i ∈ data do
3:         for c_j ∈ data do
4:             if distance(c_i, c_j) ≤ ϵ then
5:                 InsertEdge(c_i, c_j, Graph)
6:             end if
7:         end for
8:         data = data − c_i
9:         if c_i.numEdges > minPoints then
10:            c_i.type = CORE
11:        else
12:            c_i.type = NOISE
13:        end if
14:    end for
15: end procedure
```

For algorithm 1, for each point combination of points in the data set, we must make an edge between each point (excluding self-loops) if the distance between the edges are less than $\epsilon$. If an edge has as many or more neighbors as $minPoints$, then we set it's type to a CORE point. Otherwise it is a NOISE point.

**Algorithm 2** Cluster Identification

```
1: procedure IdentifyCluster(Graph)
2:     cluster ← 0
3:     for v ∈ Graph do
4:         if v.visited ≠ 1 and v.type ≠ CORE then
5:             v.visited ← 1
6:             v.cluster ← cluster
7:             BreadthFirstSearch(v, Graph, cluster)
8:             cluster ← cluster + 1
9:         end if
10:    end for
11: end procedure
```

For Algorithm 2, we visit a point in the data set given it is unvisited and it is a CORE point. We then set the point as visited and assign it's cluster as the current cluster. Next we call the BFS function and then increment the cluster. Repeat steps for each point in the data set.

**Algorithm 3** CPU Breadth First Search

```
1: procedure BreadthFirstSearch(u, Graph, cluster)
2:     Declare Xa[1...V]
3:     Declare Fa[1...V]
4:     Initialize Fa and Xa with 0
5:     Fa[u] = 1
6:     while Fa ≠ ∅ do
7:         for v ∈ V in parallel do
8:             Invoke BreadthFirstSearchKernel(Graph, Fa, Xa) on the grid
9:         end for
10:    end while
11:    Return Xa to host
12:    for c ∈ V do
13:        if Xa[v] = 1 then
14:            v.cluster = cluster
15:            v.visited = 1
16:            if NOT v.type = CORE then
17:                v.type = BORDER
18:            end if
19:        end if
20:    end for
21: end procedure
```

For Algorithm 3, we create a boolean array $Xa$ used to mark points that have been visited for a BFS call on a specific point $u$. We also have a boolean frontier array $Fa$ used to mark points that we want to visit in the future. Initialize all values in $Fa$ and $Xa$ to 0, except for $Fa[u] = 1$, the first value we want to visit. While there are any points in $Fa$ that we need to visit, we want to invoke our GPU BFS Kernel. After that we return $Xa$ to the host array. For every point in v in the data set, if point v was visited (for point u's BFS call), we will set the point's state as visited and set it's cluster as the current cluster. If point v is not a CORE point then mark as a BORDER point. Note that $Xa$ is local to one specific point $u$ called from the loop in $IdentifyCluster(Graph)$. $Xa[v]$ is transferred to

host and the visited state for point $v$ is stored in the point data type.

**Algorithm 4** GPU Breadth First Search Kernel

```
1: procedure BreadthFirstSearchKernel(Graph, Fa, Xa)
2:     tid ← getThreadID
3:     if Fa[tid] then
4:         Fa[tid] ← 0
5:         Xa[tid] ← 1
6:         for neighbors vid of tid do
7:             if NOT Xa[nid] then
8:                 Fa[nid] ← 1
9:             end if
10:        end for
11:    end if
12: end procedure
```

For Algorithm 4, with each thread index if a point $tid$ is marked to be visited in our frontier array $Fa$, then as visited by removing it from $Fa$ and adding it to $Xa$. For each neighbor $nid$ of $tid$, if it has not been visited in $Xa$, then mark it in $Fa$ to be visited in the future.

### 1.1.2 CUDA Implementation

For my implementation of the G-DBSCAN algorithm I used CUDA to parallelize the graph making function as well as the GPU BFS Kernel. I will go over relevant information for each piece of the implementation in this section. /

#### Algorithm 1 Implementation

For the MakeGraph algorithm, you must split it into three steps in order to make it parallel. First you have to count the number of neighbors within an $\epsilon$ for each point in the data set and store it in an array in device memory. You can use a thread for each data point.

```
__global__ void makeGraph1(Point *points, int *numNeighbors, int numPoints, float eps) {

    int tid = threadIdx.x + blockIdx.x * blockDim.x;

    if (tid < numPoints) {
        for (int i = 0; i < numPoints; ++i) {
            if (i == tid) {
                continue;  // Skip the point itself
            }
            if (distance(points[tid], points[i]) <= eps )
                numNeighbors[tid]++;
        }
    }
}
```

Now in order to store our graph we are using an adjacency matrix, so we can use the prefix sum to keep track of where in the adjacency list we store neighbors for a given point when paired with the numNeighbors array.

```
int* makeGraph(Point *c_points, float eps, int minPts, int *c_numNeighbors, int* c_startPos, int
    numPoints) {

    int T = 64;
    int B = (numPoints + T - 1)/ T;

    int *c_adjList;

    int *h_numNeighbors = (int*)malloc(numPoints*sizeof(int));
    int *h_startPos = (int*)malloc(numPoints*sizeof(int));

    makeGraph1 <<<T,  B>>> (c_points, c_numNeighbors, numPoints, eps);
    cudaDeviceSynchronize();

    cudaMemcpy(h_numNeighbors, c_numNeighbors, numPoints * sizeof(int), cudaMemcpyDeviceToHost);

    h_startPos[0] = 0;
    for (int i = 1; i < numPoints; i++) {
        h_startPos[i] = h_startPos[i - 1] + h_numNeighbors[i - 1];
    }
    int adjCount = h_startPos[numPoints - 1]  + h_numNeighbors[numPoints - 1];

    cudaMalloc(&c_adjList, adjCount *sizeof(int));

    cudaMemcpy(c_startPos, h_startPos, numPoints * sizeof(int), cudaMemcpyHostToDevice);

    makeGraph2 <<<T, B>>> (c_points, c_adjList, c_startPos, numPoints, eps, minPts);
    cudaDeviceSynchronize();

    return c_adjList;
}
```

After that, we again call a second makeGraph function that will store each neighbor of each thread on the adjacency list in parallel. It will also check if a point is a CORE point or NOISE.

```
__global__ void makeGraph2(Point *points, int *adjList, int *startPos, int numPoints, float eps,
    int minPts) {

    int tid = threadIdx.x + blockIdx.x * blockDim.x;

    if (tid < numPoints) {
        int start = startPos[tid];

        int numNeighbors = 0;
        for (int i = 0; i < numPoints; ++i) {
            if (i == tid) {
                continue;
            }

            if (distance(points[tid], points[i]) <= eps) {
                adjList[start + numNeighbors] = i;
                numNeighbors++;
            }
        }
        if (numNeighbors + 1 > minPts)
            points[tid].type = CORE;
        else
            points[tid].type = NOISE;
    }
}
```

### Algorithm 2 Implementation

The pseudocode and the implementation for algorithm 2 are very similar and no modification is

needed. Note that some optimizations were made so we only have to Malloc memory outside of the loop.

```
void IdentifyClusters(Point *h_points, int *startPos, int *adjList, int *numNeighbors, int
     numPoints) {
    bool *c_Xa, *c_Fa;

    cudaMalloc(&c_Xa, numPoints*sizeof(bool));
    cudaMalloc(&c_Fa, numPoints*sizeof(bool));

    bool *h_Xa = (bool*)malloc(numPoints*sizeof(bool));
    bool *h_Fa = (bool*)malloc(numPoints*sizeof(bool));

    int clusterId = 0;
    for (int i = 0; i < numPoints; ++i) {
        if (h_points[i].visited == 0 && h_points[i].type == CORE) {
            h_points[i].visited = 1;
            h_points[i].clusterId = clusterId;
            CPU_BFS(h_points, startPos, adjList, numNeighbors, h_Xa, c_Xa, h_Fa, c_Fa, i,
                clusterId, numPoints);
            clusterId++;
        }
    }

    cudaFree(c_Xa);
    cudaFree(c_Fa);
    free(h_Xa);
    free(h_Fa);
}
```

### Algorithm 3 Implementation

Or third algorithm makes the call to the GPU BFS Kernel and then classifies each point based on if it was marked in $Xa$. I tried to use a reduction kernel for the while loop, but it did not end up making the performance better. In fact scaling performance was worse after that. I believe that to be due to the amount if data that is moved between device and host when adding another kernel. This is similar to the case of the label nodes function, it is not worth it to parallelize due to the movement of data.

```
void CPU_BFS(Point *h_points, int *c_startPos, int *c_adjList, int *c_numNeighbors, bool *h_Xa,
     bool *c_Xa, bool *h_Fa, bool *c_Fa, int v, int clust, int numPoints) {

    memset(h_Fa, 0, numPoints * sizeof(bool));

    cudaMemset(c_Xa, 0, numPoints*sizeof(bool));

    // Put node v in the frontier
    h_Fa[v] = 1;

    int T = 128;
    int B = (numPoints + T - 1)/ T;

    while (std::any_of(h_Fa, h_Fa + numPoints, thrust::identity<bool>())) {
        cudaMemcpy(c_Fa, h_Fa, numPoints*sizeof(bool), cudaMemcpyHostToDevice);
        GPU_BFS_Kernel <<<T, B>>> (c_startPos, c_adjList, c_numNeighbors, c_Fa, c_Xa, numPoints);
        cudaMemcpy(h_Fa, c_Fa, numPoints*sizeof(bool), cudaMemcpyDeviceToHost);
    }

    cudaMemcpy(h_Xa, c_Xa, numPoints*sizeof(bool), cudaMemcpyDeviceToHost);

    // Label nodes
    for (int n = 0; n < numPoints; ++n) {
        if (h_Xa[n] == 1) {
            h_points[n].clusterId = clust;
            h_points[n].visited = 1;
```

```
            if (h_points[n].type != CORE)
                h_points[n].type = BORDER;
        }
    }
}
```

### Algorithm 4 Implementation

Here is the implementation of the GPU BFS Kernel. Essentially for each data point we have a thread that goes through the neighbors of the the point *tid* and checks to see if they need to be visited.

```
__global__ void GPU_BFS_Kernel(int *startPos, int *adjList, int *numNeighbors, bool *Fa, bool *Xa,
      int numPoints) {

    int tid = threadIdx.x + blockIdx.x * blockDim.x;

    if (tid < numPoints) {
        if (Fa[tid] == 1) {
            Fa[tid] = 0;
            Xa[tid] = 1;

            int startId = startPos[tid];

            for (int i = 0; i < numNeighbors[tid]; ++i) {
                int nid = adjList[startId + i];
                if(Xa[nid] == 0)
                    Fa[nid] = 1;
            }
        }
    }
}
```

### Getting It Running

In order to run this code, I would recommend using at CUDA Toolkit Version $\geq$ 12.0.0. I tried to run the code on the Pascal cluster, but it did not work. I would guess that this was due to some versioning discrepancy. To run the code you need to use .txt files for your points with pair of coordinates $x$ and $y$. Separate $x$ and $y$ with a space, and each points is on a new line. To compile and run, this is all you need:

```
 nvcc -o GDBSCAN GDBSCAN.cu
./GDBSCAN <eps> <minPts> <filename>
```

Additionally, as my project proposal required, I created python code that can take different store names an use the Overpass API from OSM to find the coordinates of businesses with matching names. With this you can you the output file from that python script and do cluster analysis on it using a customized CUDA code that specifically takes the outputs from the python script. You can then run it in the python cluster visualizer that I made. To compile the GDBSCAN_OSM code use:

```
 nvcc -o GDBSCAN GDBSCAN_OSM.cu
./GDBSCAN <eps> <minPts>
```

### 1.1.3 Testing and Verification

For this section we will discuss performance tests comparing different sized data sets, as well as verifying that our data is correct. One thing to note about DBSCAN, it is not entirely deterministic, so results may vary due to difference in order of point traversal. In order to test my code I make a python program to plot each point with it's associated cluster. That code is in the GitLab repository.

**Table 1.1:** DBSCAN Parameters and Elapsed Time

| N | eps | minPoints | Elapsed Time (ms) |
|---|---|---|---|
| 312 | 2 | 5 | 3.063 |
| 10000 | 5.9 | 4 | 31.818 |
| 50775 | 9 | 5 | 254.419 |

Note that the parameters number of points, and points themselves all contribute highly to the elapsed time. These tests were done on a Windows Desktop running WSL2 running Ubuntu 20.04.6 with NVIDIA CUDA Toolkit Version 12.2.140 and g++ Version 9.4.0. For hardware we have an NVIDIA RTX 3080 10GB GPU, an AMD Ryzen 9 5900X CPU, and 32GB of RAM.

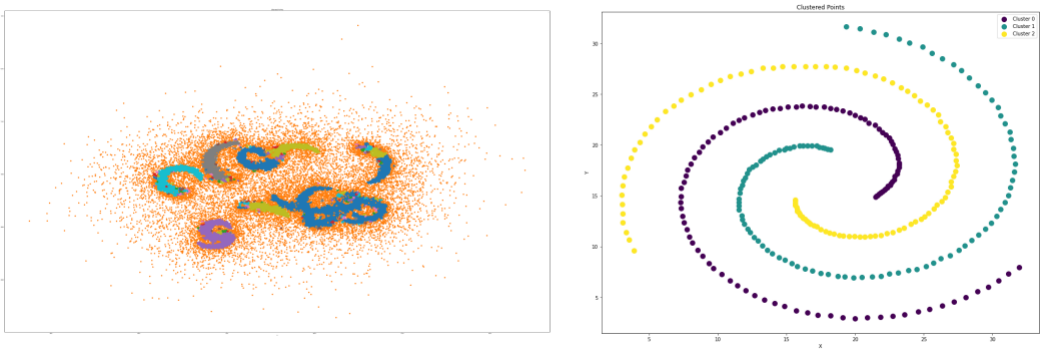
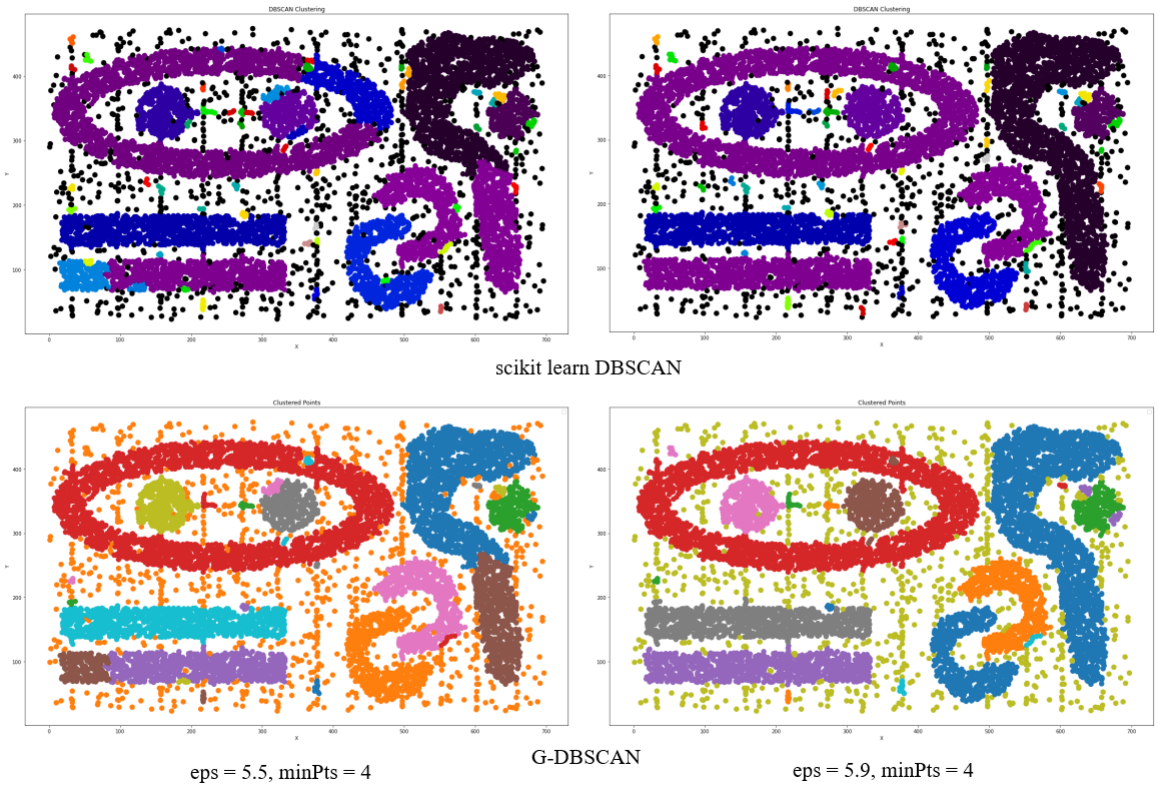



**Figure 1.1:** 50K Points (Left) and 312 Points (Right)

#### Correctness

While there is no true way of testing for correctness in DBSCAN, for different methods the clusters should still be extremely similar, but probably not exactly perfect. To show this we can do a quick visual check to make sure our clusters look close to each other across methods.




While we can see that there is a bit of a difference between the scikit learn DBSCAN implementation and G-DBSCAN, I did cross check the results with another paper and our implementation appears to be closer to the paper's results than sklearn was. This is probably due to a difference in technique as the sklearn package is most likely some parallel other parallel implementation of DBSCAN.

### 1.1.4 Future Work and Improvements

There are a multitude of alternate DBSCAN clustering algorithms optimized for GPU and parallel programming. One of the fastest methods that may be worth looking into is FDBSCAN, which uses disjointed set data structures along with bounding volume hierarchies in order to compute the neighborhoods for these points. There are even further optimizations that make use of ray-tracing hardware to do the BVH traversal as fast as possible. These would surely be interesting and challenging new topics.

## 1.2 Bibliography


[1] Andrey Prokopenko, Damien Lebrun-Grandié, Daniel Arndt, *Fast tree-based algorithms for DBSCAN on GPUs*, *CoRR*, **abs/2103.05162**, 2021, https://arxiv.org/abs/2103.05162.

[2] Hamza Mustafa, Eleazar Leal, Le Gruenwald, *An Experimental Comparison of GPU Techniques for DBSCAN Clustering*, *2019 IEEE International Conference on Big Data (Big Data)*, 2019, pp. 3701-3710, 10.1109/BigData47090.2019.9006169.

[3] G. Karypis, Eui-Hong Han, V. Kumar, *Chameleon: hierarchical clustering using dynamic modeling*, *Computer*, 1999, **32**(8), pp. 68-75, 10.1109/2.781637.

[4] Pasi Fränti, Sami Sieranoja, *K-means properties on six clustering benchmark datasets*, *Applied Intelligence*, 2018, **48**(12), pp. 4743-4759, http://cs.uef.fi/sipu/datasets/.

[5] Guilherme Andrade, Gabriel Ramos, Daniel Madeira, Rafael Sachetto, Renato Ferreira, Leonardo Rocha, *G-DBSCAN: A GPU Accelerated Algorithm for Density-based Clustering*, *Procedia Computer Science*, 2013, **18**, pp. 369-378, 2013 International Conference on Computational Science, 1877-0509, https://doi.org/10.1016/j.procs.2013.05.200, https://www.sciencedirect.com/science/article/pii/S1877050913003438.

[6] Ryan Davidson, *DBSCAN Algorithm from Scratch in Python*, *Medium*, 2020, https://scrunts23.medium.com/dbscan-algorithm-from-scratch-in-python-475b82e0571c.