CUDA-DClust+: Revisiting Early GPU-Accelerated DBSCAN Clustering Designs

Madhav Poudel
School of Informatics, Computing, and Cyber Systems
Northern Arizona University
Flagstaff, AZ, U.S.A.
mp2525@nau.edu

Michael Gowanlock
School of Informatics, Computing, and Cyber Systems
Northern Arizona University
Flagstaff, AZ, U.S.A.
michael.gowanlock@nau.edu

Abstract—**Density-based clustering algorithms are widely used unsupervised data mining techniques to find the clusters of points in dense regions that are separated by low-density regions. This algorithm is inherently sequential and has limitations in its parallel implementation. There have been several parallel algorithms presented in the literature for multi-core CPUs and many-core GPUs. One such algorithm for the GPU is CUDA-DCLUST. In this paper, we propose a new GPU-accelerated DBSCAN algorithm with several optimizations. In comparison to prior work, our algorithm, CUDA-DCLUST+: (i) computes the indexing structure on the GPU, (ii) uses kernel fusion to combine the index search and cluster expansion kernels, which reduces communication and synchronization overhead with the host, and (iii) seed list management control is primarily given to the GPU rather than the CPU, which further decreases CPU-GPU communication overhead. We compare our algorithm to three state-of-the-art parallel algorithms in the literature on six real-world datasets. We find that our algorithm achieves a speedup of up to ∼23× over the fastest GPU algorithm.**

Index Terms—**Clustering, DBSCAN, GPGPU, Graphics Processing Unit, Outlier Detection, Machine Learning**

I. INTRODUCTION

Clustering is a popular method in data processing and analysis. It is used in many areas such as machine learning, data mining, pattern recognition, image analysis, and bioinformatics [1], [2]. Density-based spatial clustering of applications with noise (DBSCAN) [3] is a density-based clustering algorithm. It assigns points that are close together to a cluster, while assigning points an outlier label when they lie in low-density regions [4]. Unlike other clustering algorithms, DBSCAN is robust to noise and the algorithm does not require specifying the number of clusters in advance.

The algorithm performs range queries in a radius around all points in the dataset; therefore, the time complexity of the algorithm increases with dataset size. Consequently, scaling DBSCAN to large data volumes remains a major challenge [5]. To address this challenge, the GPU can be leveraged to drastically improve the performance of the algorithm.

There have been several attempts to parallelize DBSCAN in the literature [6], [7], [8], [9], [10]. This paper examines a pioneering DBSCAN algorithm, CUDA-DCLUST [9], that was shown to outperform a parallel multi-core CPU algorithm [11]. Since that pioneering work, there have been additional GPU-accelerated DBSCAN algorithms, and in particular, a recent summary comparing GPU DBSCAN algorithms by Mustafa et al. [11], showed that G-DBSCAN [8] outperformed CUDA-DCLUST. Despite G-DBSCAN outperforming CUDA-DCLUST, this paper examines the initial algorithm designs in CUDA-DCLUST, that targeted GPU architecture. We leverage the main ideas in CUDA-DCLUST, and show that in contrast to the results shown by Mustafa et al. [11], our algorithm, CUDA-DCLUST+, outperforms G-DBSCAN. In summary, this paper makes the following contributions.

1) Unlike prior work, we construct the index in parallel on the GPU instead of the CPU. This reduces CPU/GPU communication overhead. 2) We use kernel fusion to combine the DBSCAN cluster expansion and index search kernels. This decreases the CPU/GPU communication and synchronization overhead. 3) We minimize communication between CPU and GPU by computing most of the seed list management on the GPU rather than the CPU. 4) In prior work, a size limitation was imposed on the seed list management array used for cluster expansion. To prevent inaccurate results, extra non-negligible work was required. To address this problem, we propose a new collision detection and merging method.

Mustafa et al. [11] showed that G-DBSCAN outperforms CUDA-DCLUST in their DBSCAN comparison paper. Using our optimizations, we show that CUDA-DCLUST+ achieves up to ∼23× speedup compared with G-DBSCAN, demonstrating that the original CUDA-DCLUST design works well when optimized for newer GPU architectures.

The paper is organized as follows: Section II presents background information. Section III presents our algorithm and associated optimizations. Section IV presents the performance evaluation. Finally, Section V concludes the paper.

II. BACKGROUND AND RELATED WORK

A. Overview of the DBSCAN algorithm

DBSCAN is a popular density-based clustering algorithm. The algorithm does not take as input the number of clusters, such as k-means [12], but rather uses a density threshold to generate clusters, and it detects outliers (or noise) in a dataset. The algorithm assumes that clusters are located in high-density regions, which are separated by low density regions. The key idea is to find the neighborhood of a point with at least a minimum number of points within a given range [3]. Points with a sufficient density are chained together to form clusters. We describe key DBSCAN concepts below. ***Dimension***: The dimension, d, is the number of coordinates that define a point. ***Point*:** Each point in the dataset is defined by d-dimensional coordinates where a point $p_i \in D$ contains the coordinates $p_i = (p_i^1, p_i^2, \ldots, p_i^d)$. ***Dataset*:** A dataset D is a set of points to be clustered; it contains points $p_i \in D$, where $i = 1, \ldots, |D|$. ***dist(p, q)*:** The function *dist(p, q)* is the Euclidean distance between points p and q, which is consistent with prior work [9], [11]. ϵ**:** The search radius around a point. Points p and q are considered neighbors if $dist(p, q) \leq \epsilon$. ***minPts*:** The minimum number of points within an ϵ-neighborhood that are needed to form a cluster.

Based on ϵ and *minPts*, point p can be classified into three categories defined as follows: ***Core Point*:** A point p that has at least *minPts* points within a distance ϵ. ***Border Point*:** A point p which has fewer than *minPts* points within a distance ϵ but is in the neighborhood of a core point. ***Noise Point*:** A point p which is neither a core point nor a border point.

DBSCAN takes a point and searches its ϵ-neighborhood. The ϵ-neighborhood of a point p consists of all the points in the dataset, D, whose distance from point p are less than or equal to ϵ i.e. $dist(p, q) \leq \epsilon$. DBSCAN relies on two relationships: direct density reachablity and density connectedness. Given two points $p \in D, q \in D$, point q is direct density reachable from p, if p is a core point and q is in the ϵ-neighborhood of p. Similarly, points p and q are connected, if p and q are density reachable from a point x, or a chain of points that are density connected, i.e., $p, x_1, \ldots, x_n, q$, where $x_1, \ldots, x_n$ form a density connected chain between the points. The major steps involved in the DBSCAN algorithm are given as follows [3]. 1) Find all neighboring points within ϵ and identify the core points with more than *minPts* neighbors. 2) For each core point, if it is not already assigned to a cluster, create a new cluster. 3) Find the density connected points and assign them to the same cluster as the core point. 4) Iterate through the remaining unvisited points in the dataset. The points that do not belong to any of the clusters are considered noise points.

B. Past Optimizations of the DBSCAN algorithm

ϵ-neighborhood searches may account for more than 95% of the computation time [13]. Since the GPU can perform many distance calculations in parallel, it is a good architecture for computing ϵ-neighborhood searches. We limit our literature review to GPU-accelerated DBSCAN algorithms [6], [7], [8], [9] because related work shows that GPUs outperform parallel multi-core CPU DBSCAN algorithms.

We describe GPU algorithms as follows. CUDA-DClust [9] uses a tree-based directory indexing structure to support the ϵ-neighborhood search by limiting the number of points to search when expanding the clusters. Another algorithm is G-DBSCAN [8] that uses the graph structure to index data and perform breadth-first searches on the GPU to compute the clusters. Thapa et al. [7] leverages the GPU by replacing the ϵ-neighborhood search in the DBSCAN algorithm with a GPU function executing many GPU threads. Points are assigned to different GPU threads to find the ϵ-neighborhood of each point in the dataset through a brute force approach that does not use an index. Welton et al. [6] proposes an algorithm that effectively processes dense regions of data using data partition and merging techniques to avoid the scalability limits on large datasets. It combines MRNet [14], a tree-based distribution network for distributed-memory computing, with hybrid CPU-GPU nodes, and a "dense box" algorithm [6], [15], [16] that reduces the number of distance calculations. Gowanlock et al. [17] purposed a hybrid CPU/GPU DBSCAN approach that exploits the memory bandwidth on the GPU for fast index searches and employs a batching scheme for CPU/GPU data transfers to obviate the limitations of GPU memory. Similarly, Gowanlock [15] proposed another algorithm that clusters on the billion-point scale using multi-core CPUs and many-core GPUs. In addition, it uses grid-based indexing to perform ϵ-neighborhood searches and uses the dense box algorithm to avoid distance calculations.

The algorithms by Welton et al. [6], and Gowanlock [15] are designed for large datasets and are unsuitable for comparison to our algorithm (e.g., for small to medium sized datasets, the algorithm by Gowanlock [15] has non-negligible overhead, as it expects large datasets to be processed). In addition, Welton et al. [6], and Gowanlock et al. [15] use the dense box algorithm to avoid the distance calculations. The dense box algorithm does not improve performance in low density regions and is only useful in very high density regions when *minPts* is very small. The algorithm by Gowanlock et al. [17] focuses on DBSCAN clustering under the conditions where a single dataset should be clustered with multiple ϵ and *minPts* parameters. As such, the focus of optimization is on pipelining the execution of DBSCAN instances, rather than clustering a dataset with a single set of parameters. Thus, it is unsuitable for comparison with this work.

To summarize, in our experimental evaluation, we compare our algorithm to the following algorithms: CUDA-DClust, G-DBSCAN, and CPU-DBSCAN. CPU-DBSCAN is a parallel CPU algorithm modified from Thapa et al. [7] referred to as "Multi-Threaded CPU DBSCAN" in Mustafa et al. [11]. We selected these state-of-the-art algorithms as they have been presented in the algorithm comparison paper of Mustafa et al. [11]. In that paper [11], they report that G-DBSCAN outperforms CUDA-DClust and CPU-DBSCAN. In this paper, we show that by optimizing CUDA-DClust, we are able to outperforms G-DBSCAN and CPU-DBSCAN.

III. CUDA-DClust+ Algorithm

We present our algorithm, CUDA-DClust+, that leverages many of the algorithm designs in CUDA-DClust [9]. For clarity, the algorithm uses the notation as shown in Table I.

The algorithm takes as input the dataset D, ϵ, and *minPts*, and outputs a list of points and their corresponding cluster or whether it has been assigned a noise label. The algorithm constructs an index on the dataset D based on the number of partitions r at each level of the tree. Then, it performs

TABLE I
NOTATION USED IN THE ALGORITHM.

Symbol	Description
ϵ	Radius of neighborhood around a point.
minPts	Minimum number of points required to form a cluster.
r	Number of partition size used in index structure.
$\|D\|$	Number of data point in a dataset.
f	Fraction of noise points in a dataset with a given set of ϵ and *minPts* parameters.

the cluster expansion on the GPU and merges the clusters on the CPU. To better understand the algorithm designs, we first outline the concepts used in CUDA-DCLUST algorithm. Then, we explain the CUDA-DCLUST+ components and optimizations.

A. The CUDA-DClust Algorithm

The CUDA-DCLUST [9] algorithm uses the concept of chains for parallelization. Each CUDA block is assigned a point for expansion and threads within the block compute the ϵ-neighborhood search in parallel. The cluster formed from the expansion is called a *chain*, where a chain is a subset of a larger cluster. The threads in a block incrementally grow a chain. While the chains grow, there is a possibility that two of the chains contain the same point. This is called a collision, and a *collision matrix* is used to record these conflicts. Later, the collided chains are merged to form a single cluster. The algorithm uses a *seed list* to keep track of points that are to be expanded and uses an atomic operation to assign the points to the cluster to avoid race conditions. These concepts of CUDA-DCLUST are explained in detail as follows.

1) Chains and the Collision Matrix: A chain is a collection of data points belonging to a shared density-based cluster processed by CUDA blocks during the execution of the cluster expansion kernel. The threads assigned to each of the blocks are utilized for the ϵ-neighborhood searches. Collisions that occur between points within chains are recorded on the GPU in the collision matrix and are merged on the CPU.

2) Seed List: The seed list is a data structure that keeps track of points to be expanded. The seed list is assigned to each block in the algorithm before the execution of the cluster expansion kernel. The seed lists are updated when points are processed to find more members of the cluster. The size of the seed list is a constant, where the maximum size may be reached during the execution of the program. In this case, a point may be discarded and not correctly assigned to its cluster. To address this problem, CUDA-DCLUST uses a refill seed list operation, which is a data structure that keeps track of additional points that are needed to refill the seed list for further chain/cluster expansions.

3) Cluster Expansion Kernel: The cluster expansion kernel determines the members (points) assigned to a chain and expands it to include new members. Once the seed list is assigned with new points from the CPU, each CUDA block expands the last point from the seed list and decreases the seed list size. Threads from each block work together to find neighbors in parallel. The points with a distance $\leq \epsilon$ of the point being expanded are marked as neighbors and are added to the seed list by the threads of the respective block. If there are fewer than *minPts* neighbors, they are stored in a data structure denoted as the *Quarantine*. The quarantined points are marked as candidates of a chain if there are $\geq$ *minPts* neighbors. Otherwise, expanded points are assigned to the set of noise points. The potential collisions between the chains during expansion are recorded in the collision matrix.

4) Index Structure: All ϵ-neighborhood searches can be performed using a brute-force approach, yielding a complexity of $|D|^2$. Alternatively, an index can be used to reduce the complexity of ϵ-neighborhood searches by pruning the search. Böhm et al. [9] used an index that partitions the dataset into levels where each level indexes on a different dimension. The indexed version of their algorithm is called CUDA-DCLUST*. For clarity, in this paper, we refer to CUDA-DCLUST as the indexed version of the algorithm (CUDA-DCLUST*). We do not compare our work to the brute force version, as it is significantly slower than using an index.

We will show that the original CUDA-DCLUST algorithm has significant potential for improvement. We summarize our optimizations to the CUDA-DCLUST algorithm below.

1) CUDA-DCLUST constructs the index using the CPU. In contrast, CUDA-DCLUST+ performs the construction of the index on the GPU in parallel. Since index construction takes non-negligible time, computing it in parallel on the GPU improves performance.
2) CUDA-DCLUST uses a constant seed list size to keep track of points in the cluster expansion kernel. The constant seed list size can inadvertently discard points from the cluster. To mitigate this issue, it refills the seed list to minimize discarding points [9]. In contrast, CUDA-DCLUST+ uses a correction merge routine that merges the discarded points to their clusters. This improves performance and slightly improves the accuracy of the clustering results.
3) CUDA-DCLUST uses a large seed list to maintain accurate clustering results. In contrast, a large seed list size is not necessary for CUDA-DCLUST+, as discarded points are merged by our correction merge routine.
4) In CUDA-DCLUST+, the overhead of communication between CPU and GPU is minimized by managing the seed list within the cluster expansion kernel until the seed lists are empty. Limiting CPU/GPU synchronization and other overheads improves performance.

B. Index Construction

The index is a tree where each level l of the tree indexes a different dimension of D. The total number of levels of the tree is thus the data dimensionality (d) of the dataset. Figure 1 shows the index structure with $r = 3$ partitions, and $d = 2$ dimensions/levels. Level 1 partitions the data points in the first dimension, and level 2 partitions the data points in the second dimension. Each node represents a bin, and the collection of all bins represents the index structure denoted by I. Each node stores the dimension, and range of the data points. The

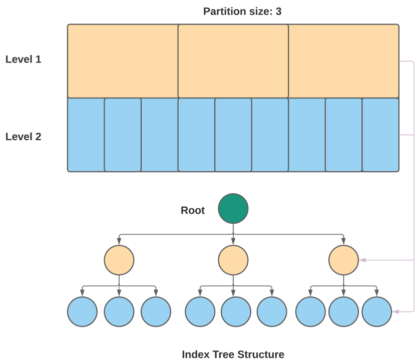

Fig. 1. Index structure with $r = 3$ partitions, and $d = 2$ dimensions.

location of the data points in the dataset are stored as values in key/value pairs where the key denotes bin ID. While the nodes contain minimal information, the use of key/value pairs allow us to search the index using binary searches (using the Thrust library [18]) to determine which candidate points may be within the ϵ-neighborhood of a given query point. We detail the search of the index in Section III-D.

The index is constructed as follows. We begin by computing the minimum and maximum coordinates of the points in each dimension denoted as $min_l = min(p_i^l \in D)$ and $max_l = max(p_i^l \in D)$, where $l = 1, \ldots, d$. The minimum and maximum point coordinates are used to determine whether we can partition the dataset at each level into r partitions, where we select an initial number of partitions, r', which is a parameter that we choose experimentally. We then compute the minimum width across all dimensions as follows: $w_{min} = min(max_l - min_l)$. We check if $(w_{min}/r') \geq \epsilon$. This constraint ensures that the width of each bin is at least ϵ units wide to limit the index search to three partitions at each level of the tree. Otherwise, the complexity of the index search would increase significantly, thus degrading performance. In summary, we select r as follows using r' as an initially selected number of partitions.

$$r = \begin{cases} \frac{w_{min}}{r'}, & \text{if } \frac{w_{min}}{r'} \geq \epsilon \\ \lfloor \frac{w_{min}}{\epsilon} \rfloor, & \text{otherwise} \end{cases}$$

Consequently, the bin width at all levels of the tree, b, is simply $b = \frac{w_{min}}{r}$.

For a single ϵ-neighborhood search, the maximum number of bins that need to be searched is 3^d.

To minimize the memory consumption of the index, we constrain the number of partitions, $r \leq 500$, and we will show experimentally that a good value for $r = 10 - 500$ across several real-world datasets[1].

Algorithm 1 Main routine of CUDA-DCLUST+.

```
1: procedure MAIN($D$, $\epsilon$, $minPts$, $r$)
2:     indexKernel($D$, $r$, $I$)
3:     fillSeedList($D$, sl)
4:     while true do
5:         expansionKernel($D$, cluster, $I$, sl, colMat, corMerge)
6:         mergeClusters (cluster, sl, colMat, corMerge)
7:         completed = fillSeedList($D$, sl)
8:         if completed = true then
9:             break
   return
```

C. Algorithm Overview

We present an overview of CUDA-DCLUST+ in Algorithm 1. The main function starts by taking the parameters D, ϵ, *minPts*, and r. First, GPU memory is allocated for the index (I). The *indexKernel* inserts all data points into I and generates bins (line 2). The seed list is initialized with one data point per CUDA block for expansion (line 2). The process of cluster expansion and cluster merging is carried out by coordination of the GPU and CPU in several iterations until all the data points are processed (lines 4–9). *expansionKernel* performs cluster expansion and seed list management until the seed lists (*sl*) are empty (line 5). The *mergeClusters* function merges the collisions recorded in collision matrix (*colMat*) and performs the correction merge (*corMerge*) to complete the clustering process for the iteration in the CPU (line 6). The *fillSeedList* function fills the new points in the seed list for the next instance of cluster expansion (line 7). It returns a completion status by checking the number of remaining unprocessed points (lines 8–9). If the completion status is true then the algorithm terminates. The pseudocode of *indexKernel*, *mergeClusters*, and *fillSeedList* functions are omitted due to space constraints.

D. Cluster expansion and Index-Supported Range Queries

The expansion and range query kernel expands the cluster and performs ϵ-neighborhood searches using the index. It uses a seed list for the management of points to be expanded. Each of the CUDA blocks expands one point at one instance of kernel execution. The cluster that is formed at each kernel invocation is called a chain. The collision between the chains is recorded in the collision matrix. The chains are merged on the CPU to form intermediate clusters (which may be finalized if no other points are to be assigned to the cluster). The original CUDA-DCLUST algorithm uses a seed list for the management of points to be expanded. However, it is limited by its constant size. It uses a data structure called "refill seed list" to cluster the discarded points caused by seed list overflow. We have removed the refill seed list functionality and instead perform an additional cluster merge (called *corMerge* in Algorithm 1) that addresses the seed list size limitation. It merges the collision between the chain and the intermediate cluster that were formed in the previous iteration of the loop.

The pseudocode of the *expansionKernel* is given in Algorithm 2. The *expansionKernel* starts by taking the initial seed points from the CPU for each of the CUDA blocks for

[1] In contrast to selecting the number of partitions at each level, another approach would be to create ϵ-length bins in each dimension. However, one potential drawback of this approach is that for small values of ϵ, the index may require a significantly large number of bins, which may exceed global memory capacity. For this reason, we parameterize the number of partitions.

expansion. Each of the blocks is identified with the chain ID (*cID*) which is equal to the CUDA block ID (line 2). The collision matrix (*colMat*) and correction merge (*corMerge*) are assigned with the *unProc* state (line 3). The size of the seed list (*slsz*) for each of the blocks is checked (line 4). If the seed list (*sl*) contains points, then the expansion of the last point from the seed list is performed in each block. If the seed lists are empty, then the control is transferred to the CPU which adds a single seed point to each of the blocks and starts the next iteration of expansion. The Neighbor count (*nbCnt*) is initialized to zero. The point for each of the blocks is extracted from the seed lists and the size of the seed list is decremented (lines 5–7). The *indexResults* data structure is used to store the bin identifier (*bID*) list. It is assigned with *unProc* (line 8). The expanded point is searched in the index structure to get the range of data points and perform the ϵ-neighborhood search (line 9). Due to space constraints, we omit the pseudocode of the index search in the algorithm, denoted as *searchIndex*.

The data points that fall within the bins are iterated in parallel by the CUDA block to find whether they should be assigned to the chain (lines 10–21). The distance between the candidate point and the expanded point is calculated. If the distance of candidate points in the bins are within ϵ of the expanded point (line 16), then the point is added as a neighbor. The first *minPts* neighbors are stored in the *Quarantine* (lines 18–19). If *nbCnt* exceeds *minPts* (lines 20–21) then we call the *processPoint* function (line 27) and mark the neighbor points as members of the chain (line 28). In addition, they are added to the seed list for further expansion (lines 29–30). If the neighbor points already belong to one of the chains, then the collision between two chains is stored in the collision matrix (lines 31–33). Similarly, if the neighbor point already belongs to an intermediate cluster, then the collision between the chain and the intermediate cluster is recorded in the *corMerge* (lines 34–38). The point marked as noise in the past and belonging to the current chain is marked as a member of the chain (lines 39–40).

After the *processPoint* function returns, the data points in *Quarantine* are marked as members of the chain, if the number of neighbors exceed *minPts* (lines 22–24). Otherwise, the expanded point is marked as *noise* (lines 25–26). Furthermore, the record of the collision matrix and correction merge are processed on the CPU after the seed lists are emptied (Algorithm 1 line 6). After merging the clusters, the seed list is filled with new points and the next iteration is executed.

E. Merging Clusters and Filling the Seed List

The cluster merging process is coordinated between the CPU and GPU. The records of the collision matrix and correction merge are merged on the CPU. After merging, the data points are assigned to the intermediate clusters in parallel using the Thrust library [18]. Then, for each of the CUDA blocks, a data point is filled in the seed list for further expansion on the CPU. Furthermore, unprocessed points are tracked to determine whether the DBSCAN algorithm has completed processing all points in the dataset.

Algorithm 2 DBSCAN Cluster Expansion.

```
 1: procedure EXPANSIONKERNEL($D$, cluster, $I$, sl, colMat, corMerge)
 2:     if threadIdx.x = 0 then cID ← blockID
 3:     initUnprocess(colMat, corMerge)
 4:     for slsz[cID] $\neq$ 0 do
 5:         if threadIdx.x = 0 then
 6:             slsz[cID] ← slsz[cID] - 1; nbCnt ← 0;
 7:             pID ← sl[cID][slsz - 1]; p ← $D$[pID];
 8:         initUnprocess(indexResults)
 9:         searchIndex(p, cID, $D$, indexResults, $I$)
10:         for k ∈ $3^d$ do
11:             bID ← indexResults[cID][k]
12:             if bID = unProc then
13:                 break
14:             for x ∈ $I$[bID] do
15:                 candP ← $D$[x];
16:                 if dist(p[i], candP[i]) $\leq \epsilon^2$ then
17:                     nbCnt ← nbCnt + 1
18:                     if nbCnt < minPts then
19:                         Quarantine[nbCnt] ← x
20:                     else
21:                         processPoint(x)
22:         if nbCnt ≥ minPts then
23:             for i ∈ minPts do
24:                 processPoint(Quarantine[i])
25:         else
26:             cluster[pID] ← noise
    return
27: procedure PROCESSPOINT($id$)
28:     oldCID ← CAS(cluster[id], unProc, cID)
29:     if oldCID = unProc then
30:         slsz ← atomicAdd(slsz, 1); sl[cID][slsz] ← = id
31:     if oldCID<gridDim.x and oldCID≠cID and oldCID≠noise then
32:         if oldCID < cID then
33:             colMat[oldCID][cID] ← 1
34:     if oldCID ≥ gridDim.x then
35:         for i ∈ corMergeSize do
36:             changedCID ← CAS(corMerge[cID][i], unProc, oldCID)
37:             if changedCID = unProc or  changedCID = oldCID then
38:                 break
39:     if oldCID = noise then
40:         CAS(cluster[id], noise, cID)
    return
```

IV. Experimental Evaluation

A. Datasets

We use six real-world datasets that were used in other works for examining DBSCAN performance [11], [17]. The datasets are summarized below.

NGSIM: Next Generation Simulation (NGSIM) vehicle trajectory dataset [19] consists of vehicle trajectory data on three U.S. highways. The dataset contains over 11,800,000 points with several attributes, where we use the local road coordinates. We selected up to 10^6 data points for the experiments.

Spatial and *Spatial3D*: This 3D-Road Network [20] dataset consists of road network data of North Jutland in Denmark. It consists of over 400,000 data points with several attributes. Regarding *Spatial*, we used two-dimensional longitude and latitude information. In the *Spatial3D*, we used the additional altitude information along with longitude and latitude. We selected up to 400,000 data points for the experiments.

Porto: This dataset is from a Taxi Service Trajectory Prediction Challenge dataset [21] that consists of trajectories of 442 taxis in the city of Porto, Portugal. It contains over 81,000,000 location points. It has different attributes like taxi ID, trip ID, timestamp, day type, and polyline. We extracted the GPS coordinates from the polyline attribute for our experiment. We selected up to 10^7 data points for the experiments.

Iono and *Iono3D*: These are the real-world space weather datasets of the ionosphere [22] in two and three dimensions, respectively. The 2 dimensional datasets contain longitude and latitude and the 3 dimensional dataset adds the total electron content value. We use up to 1,864,620 data points.

In our experiments, we sampled the data points to make a consistent comparison with the experimental methodology by Mustafa et al. [11]. One reason the data points were sampled in prior work was that G-DBSCAN has a large memory footprint and cannot be executed on large datasets. For the interested reader, our datasets are available online [2].

B. Experimental Methodology

Our platform consists of 2x Intel Xeon E5-2620 v4 2.1 GHz CPUs with 16 total physical cores and 128 GiB of main memory. The platform is equipped with an NVIDIA Quadro GP100 GPU and runs CUDA 11.3 [23]. The host code is written in C/C++. All source code including the reference implementations are compiled with the O3 optimization flag, and experiments are averaged over three trials. The implementations are made publicly available and configured as follows[3].

CUDA-DClust+: In all GPU kernels, we use a CUDA block size of 256 and execute 256 blocks. We use a seed list size of 128 and the correction merge array is of 512 elements. We chose a smaller seed list size than CUDA-DClust in Böhm et al. [9] to limit the GPU memory footprint. The data is stored as 64-bit floats.

CUDA-DClust: For this algorithm, we reproduce the major ideas of Böhm et al. [9], as the source code of the algorithm is not publicly available. The index is identical to that in CUDA-DClust+. We used a CUDA block size of 256 and execute 256 blocks. We configure the seed list to be of size 1024 elements as selected by Böhm et al. [9] to closely reproduce their experiments. The data is stored as 64-bit floats.

G-DBSCAN: The algorithm uses 256 CUDA blocks per kernel invocation, where the number of threads per block are computed by the algorithm at run time. Data are stored as 32-bit floats.

CPU-DBSCAN: This is a modified version of the code by Thapa et al. [7] that replaces the GPU kernel function with a CPU function that uses multi-threading. We use 16 threads, which are the number of physical cores on our platform. Recall that this algorithm does not use an indexing structure, and is brute force (Section II-B). The data is stored as 32-bit floats.

Note that the use of 64-bit floats is more expensive than 32-bit floats. Since CUDA-DClust+ and CUDA-DClust use 64-bit floats, it puts the algorithm at a disadvantage compared to G-DBSCAN and CPU-DBSCAN algorithms. Thus, the performance gains of CUDA-DClust+ over G-DBSCAN and CPU-DBSCAN are a lower bound.

TABLE II
DEFAULT PARAMETERS USED IN EXPERIMENT FOR 6 DATASETS.

Parameters	*NGSIM*	*Spatial*	*Iono*	*Porto*	*Spatial3D*	*Iono3D*
minPts	8	8	4	8	2	4
ϵ	1.25	0.008	1.5	0.008	0.08	1.5
r	100	80	80	100	10	40
$\|D\|$	400K	400K	400K	160K	400K	400K
f	0.060	0.002	0.001	0.003	0.013	0.001

C. Experimental Parameters

DBSCAN takes as input *minPts* and ϵ. We select a practical range *minPts* and ϵ, such that we do not obtain too low or high degrees of noise. Our parameters yield a fraction $f \approx 0.001 - 0.40$ noise points across all experiments.

The number of the partitions (r) are used in CUDA-DClust and CUDA-DClust+ as these require r to be selected for the indexing procedure. While we will show that r can be selected in a large range to achieve good performance, we select a default value for each dataset. The default parameters are shown in Table II for the six datasets.

D. Evaluation on Two Dimensional Datasets

In the following experiments, we vary ϵ, *minPts*, $|D|$, and r. When a parameter is not reported in the figure, we use the default value as shown in Table II.

1) Performance Impact of the Number of Partitions Parameter (r): Figure 2(a)–(d) plots the execution time vs. r for various values of $|D|$ in CUDA-DClust+ (the performance impact will be the same for CUDA-DClust, as they share the same index). The bins in the index are constructed by partitioning the data space into r partitions at each level of the tree. Therefore, a good value of r is dependent on the data distribution of a given dataset. On the *NGSIM* dataset, the best r is 40 when $|D| = 50,000$, 60 when $|D| = 100,000-200,000$, and 100 when $|D| = 400,000 - 800,000$. A similar pattern can be observed in *Spatial*, *Iono*, and *Porto* datasets. We can infer that the best value of r is increasing with increasing D. In *Iono* at $|D| = 400,000$, the best value is $r = 80$. Observe that increasing from $r = 80$ to $r = 100$ yields a longer execution time due to more expensive index searches. Overall, we find that the number of partitions r can be selected in a large range and obtain good performance. This is, minor changes to r do not severely degrade performance, as the algorithm is not largely sensitive to this parameter.

Note that in Figure 2(d) on *Porto*, when $r \geq 60$, the bin width, $b < \epsilon$ thus, we cannot show these data points.

Figure 2(e)–(h) plots the execution time vs. r for various values of ϵ. On the *Spatial* dataset, the best value is $r = 100$ when $\epsilon = 0.002 - 0.004$, $r = 80$ when $\epsilon = 0.006 - 0.008$, and $r = 60$ when $\epsilon = 0.01$. A similar pattern can be observed on the *NGSIM*, *Iono*, and *Porto* datasets. We infer that a larger r value is best for the smaller values of ϵ and vice versa. A smaller ϵ value requires more partitions to avoid

[2] https://rcdata.nau.edu/gowanlock_lab/datasets/CUDA_DCLUST_datasets/CUDA_DCLUST_HiPC2021.zip.

[3] https://github.com/l3lackcurtains/fast-cuda-gpu-dbscan.

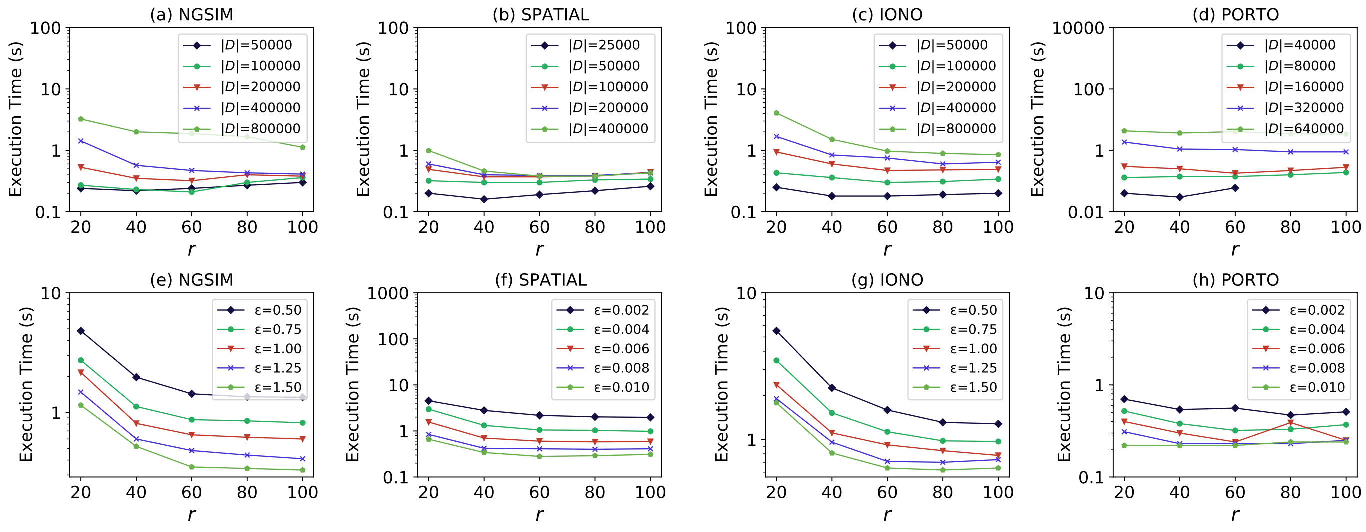

Fig. 2. The execution time vs. the number of partitions in CUDA-DCLUST+ (r) on the two-dimensional datasets. (a)–(d) Performance impact as a function of the ϵ parameter. (e)–(h) Performance impact as a function of the $|D|$ parameter.

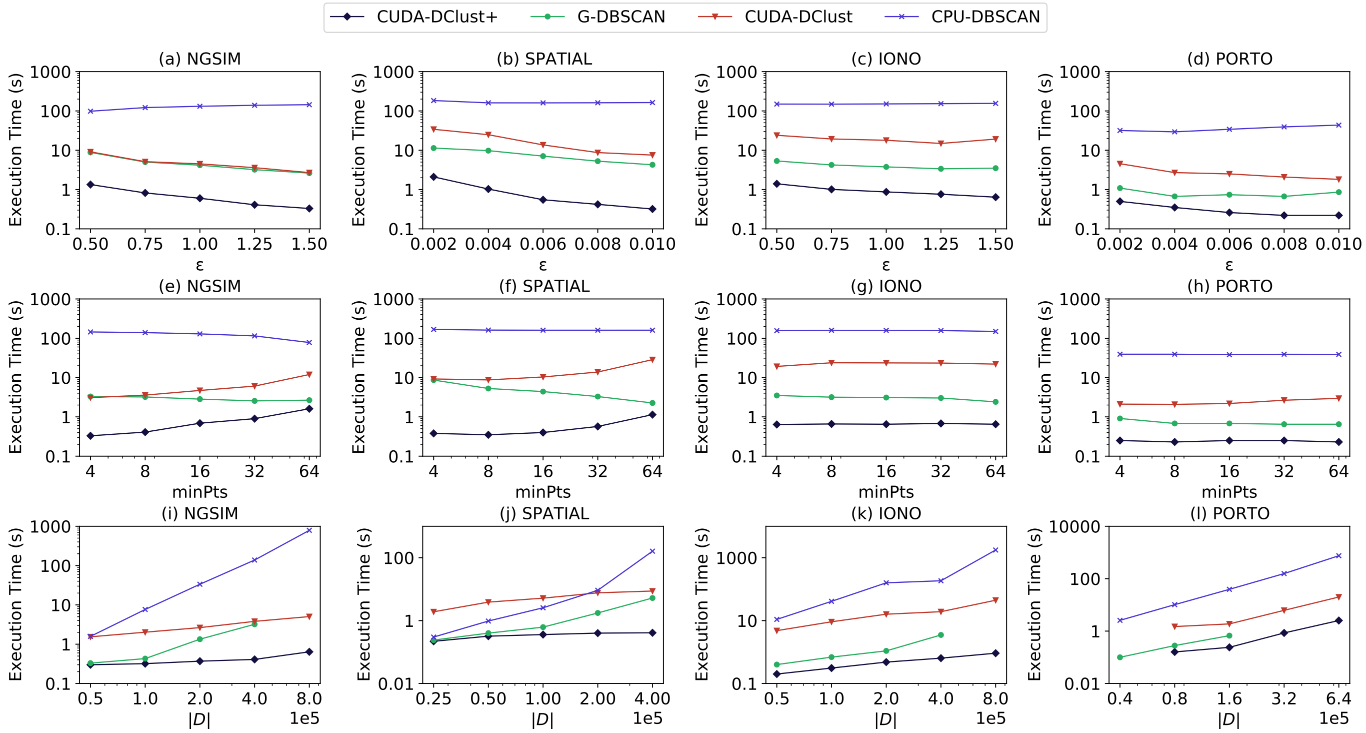

Fig. 3. Performance comparison of CUDA-DCLUST+, CUDA-DCLUST, G-DBSCAN and CPU-DBSCAN on two dimensional datasets. (a)–(d) Execution time as a function of the ϵ parameter; (e)–(h) Execution time as a function of *minPts* parameter; and, (i)–(l) Execution time as a function of the $|D|$ parameter. The fraction of noise points across the columns for each of the datasets *NGSIM*, *Spatial*, *Iono* and *Porto* are $f = 0.025 - 0.404$, $f = 0.001 - 0.184$, $f = 0.001 - 0.003$, and $f = 0.001 - 0.017$, respectively.

wasteful distance calculations because a smaller value of r creates larger sized bins. However, a larger r increases the execution time to search the index. Thus, there is a trade-off between index search and distance calculation overhead. Despite this trade-off, a good value partition number is $r = 60 - 100$ across all 2-dimensional datasets and values of ϵ.

Recall from Section III-B that an alternative index design would be to create ϵ-length bins. However, if we use ϵ-length bins, then the search time would increase for small values of ϵ because we would have a large number of partitions, r. Thus, this experiment demonstrates why we do not use ϵ-length bins.

2) Impact of the ϵ Parameter: Figure 3(a)–(d) plots the execution time vs. ϵ on the four datasets. We observe that CPU-DBSCAN performs the worst of all algorithms. This is expected as the algorithm does not use an index, and so it has a complexity of $|D|^2$.

On *NGSIM*, G-DBSCAN and CUDA-DClust have nearly identical performance. On the *Spatial*, *Iono*, and *Porto* datasets, G-DBSCAN performs better than CUDA-DClust by a significant degree. We find that CUDA-DClust+ yields superior performance compared with all other algorithms. G-DBSCAN remains the second-best performing GPU algorithm on all datasets. CUDA-DClust+ achieves the greatest speedup of up to 13.34× over G-DBSCAN on the *Spatial* dataset at $\epsilon = 0.01$. The execution time of CUDA-DClust+, and CUDA-DClust decreases with increasing ϵ across all datasets. Smaller values of ϵ require a larger number of partitions, r, for effective index searches which is evident from Figure 2(a)–(d). Since we are using a constant r for this experiment, this impact can be observed on CUDA-DClust and CUDA-DClust+. In addition, smaller ϵ values form a large number of small clusters, which increases the communication and data transfer between CPU and GPU in CUDA-DClust, and CUDA-DClust+.

3) Impact of the minPts Parameter: Figure 3(e)–(h) plots the execution time vs. *minPts* on the four datasets. We observe that CPU-DBSCAN performs the worst of all algorithms. In all datasets, the G-DBSCAN performs better than the CUDA-DClust. On the *NGSIM* dataset at *minPts* $= 4 - 8$, G-DBSCAN and CUDA-DClust have nearly equal execution time. Similarly, On *Spatial* dataset at *minPts* $= 4$, G-DBSCAN and CUDA-DClust have nearly equal execution times. The CUDA-DClust+ in all experiments performs better than all other algorithms. CUDA-DClust+ achieves its greatest speedup of up to 22.74× over G-DBSCAN on the *Spatial* dataset at *minPts* $= 4$. The execution time of G-DBSCAN on all datasets tends to decrease with increasing *minPts*. In contrast, the execution time of CUDA-DClust and CUDA-DClust+ is increasing with increasing *minPts*. At a larger values of *minPts*, DBSCAN finds fewer clusters and many noise points. Since G-DBSCAN creates a small adjacency list in the graph which is fast to traverse, the execution time decreases with increasing *minPts*. In contrast, CUDA-DClust and CUDA-DClust+ need to frequently fill the seed list from the CPU, and process fewer seed points in the expansion kernel, which degrades performance.

4) Scalability with Dataset Size ($|D|$): Figure 3(i)–(l) plots the execution time vs. $|D|$ on the four datasets to observe how the algorithms scale with dataset size. Scalability of CPU-DBSCAN is worse than the other algorithms with increasing $|D|$. At $|D| = 25,000$ on the *Spatial* dataset, CPU-DBSCAN performs equally to G-DBSCAN, CUDA-DClust performs worse than G-DBSCAN and CUDA-DClust+ performs slightly better than G-DBSCAN. Similarly, CUDA-DClust performs worse than CPU-DBSCAN until $|D| = 200,000$ on *Spatial*. Furthermore, on the *NGSIM* and *Porto* datasets, G-DBSCAN has similar execution times compared to CUDA-DClust+ for smaller dataset sizes. We find that G-DBSCAN performs similarly to CUDA-DClust+ on small dataset sizes, $|D|$.

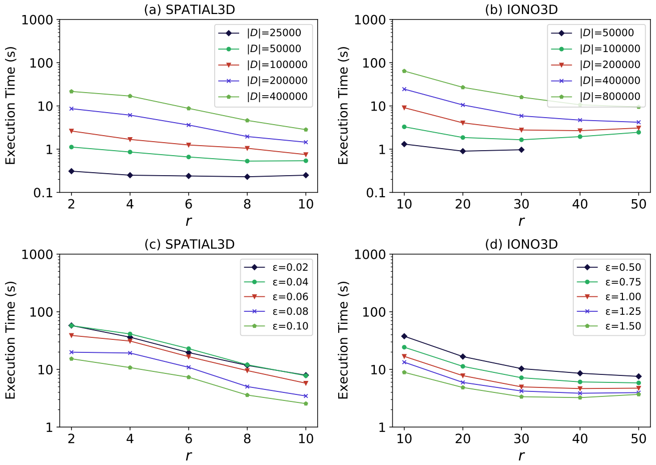

Fig. 4. The execution time vs. the number of partitions in CUDA-DClust+ (r) on the three-dimensional datasets. (a)–(b) Performance impact as a function of the ϵ parameter. (c)–(d) Performance impact as a function of the $|D|$ parameter.

There are several missing data points regarding the execution of G-DBSCAN because the algorithm exceeds global memory capacity of the GPU. This occurs on the *Iono* dataset at $|D| = 800,000$ and on the *Porto* dataset at $|D| \geq 320,000$.

On the *Porto* dataset at $|D| = 400,000$, the execution time for CUDA-DClust and CUDA-DClust+ are not shown because we use the value of $r = 100$ in all plots; this value of r breaks the constraint that the bins need to be at least ϵ units in length. In practice, our code will execute with a smaller value of r (see Section III-B)[4].

Across all datasets, the speedup of CUDA-DClust+ increases gradually with increasing $|D|$. CUDA-DClust+ achieves the greatest speedup of up to 12.85× at $|D| = 400,000$ on the *Spatial* dataset over G-DBSCAN.

E. Comparing CUDA-DClust+ *vs.* CUDA-DClust *on Three Dimensional Datasets*

We compare CUDA-DClust+ and CUDA-DClust on two three-dimensional datasets. G-DBSCAN and CPU-DBSCAN are excluded because they are not designed for more than two dimensional datasets [11].

In the following experiments, we vary ϵ, *minPts*, $|D|$, and r. When a parameter is not reported in the figure, we use the default value as shown in Table II.

1) Performance Impact of the Number of Partitions Parameter (r): Figure 4(a)–(b) plots the CUDA-DClust+ execution time vs. r for various values of D. On *Spatial*, $r = 8$ yields the best performance when $|D| = 25,000 - 50,000$, whereas $r = 10$ yields the best performance when $|D| \geq 100,000$. A similar pattern can be observed on the *Iono3D* dataset.

[4] For reference, at $p = 60$, the execution time of CUDA-DClust+ and CUDA-DClust are 0.04 seconds and 0.62 seconds, respectively.

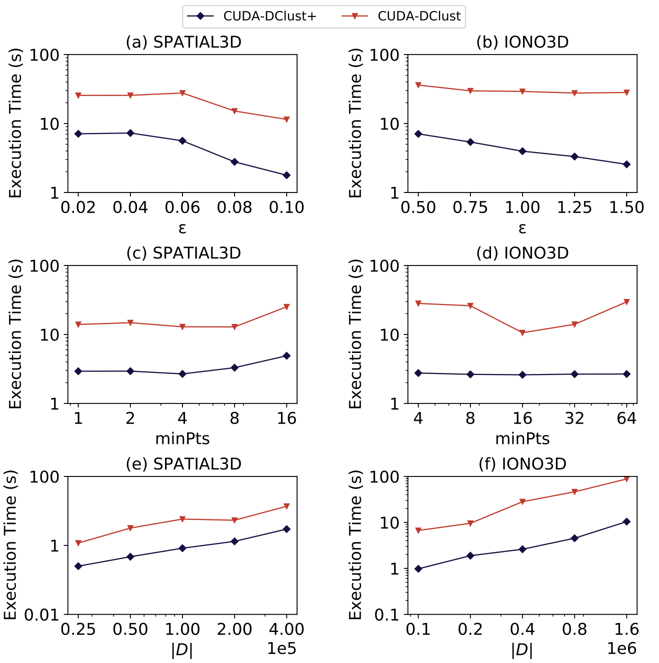

Fig. 5. Performance comparison of CUDA-DCLUST+, and CUDA-DCLUST on three dimensional datasets. (a)–(b) Execution time as a function of the ϵ parameter; (c)–(d) Execution time as a function of the *minPts* parameter; and, (e)–(f) Execution time as a function of the $|D|$ parameter. The fraction of noise points across the columns for each of the datasets *Spatial3D*, and *Iono3D* are $f = 0.006 - 0.380$, and $f = 0.001 - 0.110$, respectively.

TABLE III
TIME DISTRIBUTION PARAMETERS. WE COMPARE TWO VALUES OF r FOR EACH DATASET. r_g REFERS TO A VALUE OF r THAT YIELDED GOOD PERFORMANCE ACROSS EACH DATASET; r_c IS A CONSTANT VALUE OF r.

Parameters	*NGSIM*	*Spatial*	*Iono*	*Porto*
minPts	8	8	4	8
ϵ	1.25	0.008	1.5	0.008
$\|D\|$	1M	400K	1.86M	10M
f	0.050	0.002	0.001	0.001
r_g	150	80	100	500
r_c	70	70	70	70

TABLE IV
PERCENTAGE OF THE TOTAL EXECUTION TIME FOR CUDA-DCLUST+ IN FOUR TWO-DIMENSIONAL DATASETS FOR TWO VALUES OF r (r_g AND r_c).

Algorithm Component	*NGSIM*	*Spatial*	*Iono*	*Porto*
r_g				
indexKernel	32.39%	18.18%	4.89%	14.14%
mergeClusters	14.08%	22.72%	9.78%	0.32%
fillSeedList	1.40%	2.27%	0.54%	0.12%
expansionKernel	35.23%	50.00%	41.32%	69.74%
CPU/GPU communication and data transfer	16.90%	6.83%	43.47%	15.68%
Total time (s)	0.71	0.41	1.84	15.62
r_c				
indexKernel	9.30%	14.28%	2.31%	0.06%
mergeClusters	11.62%	20.40%	8.33%	0.06%
fillSeedList	1.16%	2.04%	0.48%	0.01%
expansionKernel	65.13%	57.16%	50.00%	97.12%
CPU/GPU communication and data transfer	12.79%	6.12%	38.88%	2.75%
Total time (s)	0.86	0.49	2.16	105.09

On these datasets, we infer that as $|D|$ increases, the best performance is obtained by increasing r.

Figure 4(c)–(d) plots the CUDA-DCLUST+ execution time vs. r for various values of ϵ. On the *Spatial3D* dataset, $r = 10$ yields best performance for all values of ϵ. On the *Iono3D* dataset, $r = 50$ yields the best performance when $\epsilon = 0.5 - 0.75$, and $r = 40$ when $\epsilon \geq 1.00$. We observe that the best value of r typically decreases with increasing ϵ.

2) Impact of the ϵ Parameter: Figure 5(a)–(b) plots the execution time vs. ϵ. CUDA-DCLUST+ performs better than CUDA-DCLUST on both datasets. The execution time of CUDA-DCLUST+ and CUDA-DCLUST decreases with the increasing ϵ. This is because, as was shown in Section IV-D2 as the fraction of noise points, f, decreases with increasing ϵ, there are fewer bins, which improves the efficiency of the index search. In summary, we find that CUDA-DCLUST+ achieves the greatest speedup of up to 10.18× compared to the CUDA-DCLUST on *Iono3D*.

3) Impact of the minPts Parameter: Figure 5(c)–(d) plots the execution time vs. *minPts*. CUDA-DCLUST+ performs better than the CUDA-DCLUST on both of the datasets. The execution time of the CUDA-DCLUST+ and CUDA-DCLUST increases with the increasing *minPts* on both datasets. This is because CUDA-DCLUST+ and CUDA-DCLUST compute fewer seed lists and fill the seed list on the CPU, which degrades performance. The CUDA-DCLUST+ achieves a greatest speedup over CUDA-DCLUST of up to 11.12× on *Iono3D*.

4) Scalability with Dataset Size ($|D|$): Figure 5(e)–(f) plots the scalability of $|D|$ on the two datasets. We can observe that the CUDA-DCLUST+ performs better than the CUDA-DCLUST in both of the datasets in every value of $|D|$. The speedup of CUDA-DCLUST+ is increasing within the increase of $|D|$. The CUDA-DCLUST+ achieves the greatest speedup over CUDA-DCLUST of up to 10.00× on *Iono3D*.

F. Algorithm Time Distribution

We present the execution time of the CUDA-DCLUST+ algorithm components (described in Section III-C and in Algorithm 1). We examine the datasets as shown in Table III. We compare two values for the number of partitions (r): r_g is a value of r that performs well for the given experimental scenario, and r_c is a constant value of r across the datasets.

Table IV shows the time distribution of CUDA-DCLUST+. Regarding r_g, we observe that *fillSeedList* is the least time consuming component across all of the datasets. *fillSeedList* executes sequentially on the CPU and has a negligible impact on the overall execution time. The execution time for CPU/GPU communication and data transfer is higher than *mergeClusters* in *NGSIM*, *Iono*, and *Porto* datasets. However, it is lower than *mergeClusters* on the *Spatial* dataset. In this case, *Spatial* requires a significant number of merges to calculate the clusters. *mergeClusters* has a smaller impact on algorithm performance than CPU/GPU communication.

We find that indexing the dataset (*indexKernel*), requires the greatest fraction of time. This is surprising, as index construction was previously found to be negligible [9].

We compare to the constant value of r, ($r_c = 70$) in Table IV, and find that across all datasets, index construction time is a much lower fraction of the total time compared to the results using r_g. However, increased index construction time improves the pruning of the algorithm, where we perform fewer distance calculations with r_g. Therefore, while we spend more time constructing the index, the total execution time with r_g is lower than r_c.

V. Discussion & Conclusion

Discussion of CUDA-DClust: In this paper, we revisited the design of CUDA-DClust [9] and improved the performance of the algorithm. In particular, compared to CUDA-DClust, our algorithm CUDA-DClust+ parallelized index construction and data point insertion, prioritized using the GPU for seed list management, efficiently merged collisions, and introduced a correction merge routine. CUDA-DClust+ is compared with the CUDA-DClust, G-DBSCAN, and CPU-DBSCAN algorithms on six real-world datasets.

Böhm et al. [9] find that CPU index construction requires negligible time; however, we find that it is non-negligible on the GPU in certain instances. If we use a constant number of partitions for all experimental scenarios, then index construction may be negligible (Table IV, *Iono* and *Porto* with $r_c = 70$); however, as was shown, this comes at the expensive of overall algorithm execution time. Böhm et al. [9] may have found index construction to be negligible compared to the other algorithm components because those components were not as optimized as we have shown here. Thus, indexing in their work only required a negligible fraction of the total time.

Discussion of Mustafa et al. [11]: This experimental comparison paper of different DBSCAN algorithms showed that G-DBSCAN outperforms CUDA-DClust. Likewise, our implementation of CUDA-DClust performed worse than G-DBSCAN. However, we find that our optimized CUDA-DClust+ outperforms G-DBSCAN achieving a speedup of up to $\sim 23\times$. Therefore, the design of the original algorithm by Böhm et al. [9] is competitive with newer state-of-the-art algorithms when further optimized.

Future work includes investigating the dense box algorithm [6], [15], [16] that can be used to further reduce the number of distance calculations in high density regions.

Acknowledgment

We thank Eleazar Leal for providing the G-DBSCAN and CPU-DBSCAN source code used in our experiments. This material is based upon work supported by the National Science Foundation under Grant No. 2042155.

References

[1] T. S. Madhulatha, "An overview on clustering methods," *IOSR Journal of Engineering*, vol. 02, no. 04, pp. 719–725, 2012.

[2] J. Oyelade, I. Isewon, F. Oladipupo, O. Aromolaran, E. Uwoghiren, F. Ameh, M. Achas, and E. Adebiyi, "Clustering algorithms: their application to gene expression data," *Bioinformatics and Biology Insights*, vol. 10, p. BBI.S38316, 2016.

[3] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise," ser. KDD'96. AAAI Press, 1996, p. 226–231.

[4] K. Bindra and A. Mishra, "A detailed study of clustering algorithms," in *6th Intl. Conf. on Reliability, Infocom Technologies and Optimization*. IEEE, 2017, pp. 371–376.

[5] P. S. Bradley, U. Fayyad, and C. Reina, "Scaling Clustering Algorithms to Large Databases," in *Proc. of the Fourth Intl. Conf. on Knowledge Discovery and Data Mining*, 1998, p. 9–15.

[6] B. Welton, E. Samanas, and B. P. Miller, "Mr. Scan: Extreme scale density-based clustering using a tree-based network of GPGPU nodes," in *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 1–11.

[7] R. J. Thapa, C. Trefftz, and G. Wolffe, "Memory-efficient implementation of a graphics processor-based cluster detection algorithm for large spatial databases," in *2010 IEEE International Conference on Electro/Information Technology*, 2010, pp. 1–5.

[8] G. Andrade, G. Ramos, D. Madeira, R. Sachetto, R. Ferreira, and L. Rocha, "G-DBSCAN: A GPU Accelerated Algorithm for Density-based Clustering," *Procedia Computer Science*, vol. 18, pp. 369–378, 2013.

[9] C. Böhm, R. Noll, C. Plant, and B. Wackersreuther, "Density-based clustering using graphics processors," in *Proc. of the 18th ACM Conf. on Information and Knowledge Management*, 2009, pp. 661–670.

[10] M. M. A. Patwary, D. Palsetia, A. Agrawal, W.-k. Liao, F. Manne, and A. Choudhary, "A new scalable parallel DBSCAN algorithm using the disjoint-set data structure," in *Intl. Conf. for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2012, pp. 1–11.

[11] H. Mustafa, E. Leal, and L. Gruenwald, "An Experimental Comparison of GPU Techniques for DBSCAN Clustering," in *2019 IEEE Intl. Conf. on Big Data*. IEEE, 2019, pp. 3701–3710.

[12] Y. Li and H. Wu, "A clustering method based on K-means algorithm," *Physics Procedia*, vol. 25, pp. 1104–1109, 2012.

[13] D. Arlia and M. Coppola, "Experiments in parallel clustering with DBSCAN," in *European Conf. on Parallel Processing*. Springer, 2001, pp. 326–331.

[14] P. C. Roth, D. C. Arnold, and B. P. Miller, "MRNet: A software-based multicast/reduction network for scalable tools," in *Proc. of the 2003 ACM/IEEE Conf. on Supercomputing*. IEEE, 2003, pp. 21–21.

[15] M. Gowanlock, "Hybrid CPU/GPU clustering in shared memory on the billion point scale," in *Proc. of the ACM Intl. Conf. on Supercomputing*. ACM, Jun. 2019, pp. 35–45.

[16] A. Prokopenko, D. Lebrun-Grandie, and D. Arndt, "Fast tree-based algorithms for DBSCAN on GPUs," *arXiv preprint arXiv:2103.05162*, 2021.

[17] M. Gowanlock, C. M. Rude, D. M. Blair, J. D. Li, and V. Pankratius, "A hybrid approach for optimizing parallel clustering throughput using the GPU," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 4, pp. 766–777, 2018.

[18] "Thrust - parallel algorithms library," accessed: 2021-07-12. [Online]. Available: https://thrust.github.io/

[19] "Next generation simulation (NGSIM) vehicle trajectories and supporting data," accessed: 2021-07-12. [Online]. Available: https://data.transportation.gov/Automobiles/Next-Generation-Simulation-NGSIM-Vehicle-Trajector/8ect-6jqj

[20] C. Guo, Y. Ma, B. Yang, C. S. Jensen, and M. Kaul, "Ecomark: evaluating models of vehicular environmental impact," in *Proc. of the 20th Intl. Conf. on Advances in Geographic Information Systems*, 2012, pp. 269–278.

[21] L. Moreira-Matias, J. Gama, M. Ferreira, J. Mendes-Moreira, and L. Damas, "Predicting taxi–passenger demand using streaming data," *IEEE Transactions on Intelligent Transportation Systems*, vol. 14, no. 3, pp. 1393–1402, 2013.

[22] V. Pankratius, A. Coster, J. Vierinen, P. Erickson, and B. Rideout, *GPS Data Processing for Scientific Studies of the Earth's Atmosphere and Near-Space Environment*, 01 2017.

[23] "CUDA toolkit documentation," accessed: 2021-07-12. [Online]. Available: https://docs.nvidia.com/cuda/