Fast tree-based algorithms for DBSCAN for low-dimensional data on GPUs

Andrey Prokopenko
Oak Ridge National Laboratory
Oak Ridge, Tennessee, USA
prokopenkoav@ornl.gov

Damien Lebrun-Grandié
Oak Ridge National Laboratory
Oak Ridge, Tennessee, USA
lebrungrandt@ornl.gov

Daniel Arndt
Oak Ridge National Laboratory
Oak Ridge, Tennessee, USA
arndtd@ornl.gov

ABSTRACT

DBSCAN is a well-known density-based clustering algorithm to discover arbitrary shape clusters. While conceptually simple in serial, the algorithm is challenging to efficiently parallelize on many-core GPU architectures. Common pitfalls, such as asynchronous range query calls, result in high thread execution divergence in many implementations. In this paper, we propose a new framework for GPU-accelerated DBSCAN, and describe two tree-based algorithms within that framework. Both algorithms fuse the search for neighbors with updating cluster information, but differ in their treatment of dense regions of the data. We show that the time taken to compute clusters is at most twice that of determination of the neighbors. We compare the proposed algorithms with existing CPU and GPU implementations, and demonstrate their competitiveness and performance using a fast traversal structure (bounding volume hierarchy) for low dimensional data. We also show that the memory usage can be reduced by processing object neighbors dynamically without storing them.

CCS CONCEPTS

• **Computing methodologies** → *Parallel algorithms.*

KEYWORDS

DBSCAN, bounding volume hierarchy, parallel algorithm, GPU

ACM Reference Format:
Andrey Prokopenko, Damien Lebrun-Grandié, and Daniel Arndt. 2023. Fast tree-based algorithms for DBSCAN for low-dimensional data on GPUs. In *52nd International Conference on Parallel Processing (ICPP 2023), August 07–10, 2023, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3605573.3605594

1 INTRODUCTION

Clustering is a data mining technique that splits a set of objects into disjoint classes (*clusters*), each containing similar objects. Dbscan (Density-Based Spatial Clustering of Applications with Noise) [11] is a density-based clustering algorithm. It is useful when the number of clusters or their shape is not known *a priori*. It is used in a diverse set of applications such as bioinformatics, noise filtering and outlier detection, cosmology, image segmentation, and others.

The Dbscan algorithm requires the identification of close neighbors for each data point. Its breadth-first search nature makes parallelization a challenge. Major progress occurred in the last two decades, starting from the master-slave [3, 43] and MapReduce [9, 19] approaches, and transitioning to using shared memory [17, 23, 31, 32, 40] and GPU [2, 6, 14, 15, 25, 30, 34, 38, 41, 42] implementations, and even approximate algorithms [8, 12, 26, 33]. Using the Union-Find technique for cluster labeling, introduced in [32], was a particularly important breakthrough as it fundamentally changed the nature of the algorithm, breaking with its breadth-first search origins.

In this work, we first introduce a general parallel algorithm with sufficient degree of parallelism for thousands of cores available on GPUs. All components of the algorithm are executed on a GPU.

We then propose two concrete implementations. We prioritize using an indexing structure with a fast batched neighborhood search to maintain algorithm performance. Specifically, we use a bounding volume hierarchy (BVH), a structure predominantly used in computer graphics for ray tracing [27]. We combine it with a synchronization-free union-find technique introduced in [21]. Our approach allows processing the found neighboring points on-the-fly, reducing the overall memory consumption of the limited GPU memory. We introduce several traversal optimization techniques and reduce the number of distance calculations used by the algorithm in dense regions. We show significant performance improvements over available multi-threaded CPU and GPU Dbscan implementations. Since the local Dbscan implementation is an inherent component of a full distributed algorithm, the proposed algorithm can be easily plugged into most distributed frameworks to improve the overall performance.

This paper focuses on the low-dimensional (e.g., spatial) data for two reasons. First, this work was motivated by scientific simulations, such as cosmology. The data in these simulation is commonly low-dimensional (e.g., 3D), and the main challenge lies in its size, reaching 500 million data points for a single GPU (with a full simulation requiring hundreds or thousands of GPUs). Given that the data is often analyzed *in-situ*, it is imperative for the underlying algorithm to be fast. Second, an implementation of a tree-based indexing structure for high dimensions on an accelerator such as GPU is a challenging task in itself, as the "curse of dimensionality" creates challenges for the popular data structures used for low-dimensional data [5].

Our key contributions are:

ICPP 2023, August 07–10, 2023, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-0843-5/23/08…$15.00
https://doi.org/10.1145/3605573.3605594

- We reformulate the DBSCAN algorithm to expose more parallelism required for an efficient GPU implementation.
- We use BVH as the search index, selected for its high efficiency on GPUs.
- We develop a new way to reduce the number of calculations in the dense data regions through including dense cells into a hybrid BVH hierarchy together with sparse data, combining the benefits of both search index and grid-based methods.
- We provide the first performance portable algorithm and implementation for the DBSCAN, and provide a comprehensive set of experiments on three architectures (AMD EPYC 7763 CPU, Nvidia A100 GPU, AMD MI250X GPU).

The remainder of the paper is organized as follows. Section 2 introduces the DBSCAN algorithm and related work. Section 3 describes a general framework for a GPU DBSCAN implementation allowing for fine-grained parallelism. In Section 4, we describe two tree-based algorithms within that framework. Finally, we demonstrate the algorithm performance and performance portability in Section 5 and derive our conclusions and future work in Section 6.

2 BACKGROUND

2.1 DBSCAN algorithm

We briefly outline the DBSCAN algorithm in this Section, referring the readers to [11] for more details.

Let X be a set of n points to be clustered. For a point to be in a cluster, the density in its neighborhood has to exceed some threshold, i.e., its neighborhood has to contain at least a minimum number of points. This is formalized using two user-provided parameters: $minPts \in \mathbb{N}^+$ and $\varepsilon \in \mathbb{R}^+$.

An *ε-neighborhood* of a point x is defined as $N_\varepsilon(x) = \{y \in X \mid dist(x,y) \leq \varepsilon\}$, with $dist(\cdot,\cdot)$ being a distance metric for the set X (e.g., Euclidean). The *minPts* parameter defines the minimum number of points for a point to be considered inside a cluster, and a point x is called a *core point* if $|N_\varepsilon(x)| \geq minPts$. A point y is *directly density-reachable* from a point x if x is a core point and $y \in N_\varepsilon(x)$. A point y is *density-reachable* from a point x if there is a chain of points $x_1, \ldots, x_n$, $x_1 = x$, $x_n = y$, such that x_{i+1} is directly density-reachable from x_i. Points x and y are called *density-connected* if there exists a point z in X such that both x and y are density-reachable from z. Finally, a point x is called a *border point* if it is density-reachable from a core point, but is not a core point itself. The points that are not core or border points are called *noise* and are considered to be outliers not belonging to any cluster. Any cluster then consists of a combination of core points (at least one) and border points (possibly, none). Note, that as a border point may be density-reachable from multiple core points, it could potentially belong to multiple clusters. Implementations of the algorithm may differ in their handling of such border points, but typically assign them to a single cluster.

The special case of $minPts = 2$ (sometimes called Friends-of-Friends in the cosmology literature) is equivalent to finding strongly connected components in the adjacency graph $G = (V, E)$, where $V = X$ and two vertices x and y have an (undirected) edge between them if $dist(x,y) \leq \varepsilon$. In this case, there are no border points, and a point either belongs to a cluster as a core point, or is in the noise.

Algorithm 1 DBSCAN algorithm

```
1:  procedure DBSCAN(X, minPts, ε)
2:      for each unvisited point x ∈ X do
3:          mark x as visited
4:          N ← GetNeighbors(x, ε)
5:          if |N| < minPts then
6:              mark x as noise
7:          else
8:              C ← {x}
9:              for all y ∈ N do
10:                 N ← N \ y
11:                 if y is not visited then
12:                     mark y as visited
13:                     N̄ ← GetNeighbors(y, ε)
14:                     if |N̄| ≥ minPts then
15:                         N ← N ∪ N̄
16:                 if y is not a member of any cluster then
17:                     C ← C ∪ {y}
```

The pseudocode for the DBSCAN algorithm is shown in the Algorithm 1. The algorithm starts at an arbitrary point $x \in X$, computing its ε-neighborhood N (line 4). If x is not a core point, i.e. $|N| < minPts$, x is tentatively marked as noise (line 6), and another point is chosen. Otherwise, the algorithm constructs a new cluster C by incrementally adding points that are density-reachable from x in a breadth-first search manner (lines 8-17), including the points that may have been previously marked as noise. Border points are assigned to the first encountered cluster that they are density-reachable from. The algorithm has a computational complexity of $O(n^2)$, or $O(n \log n)$ if a spatial indexing structure (e.g., k-d tree [4] or R-tree [16]) is used.

DBSCAN* proposed in [7] simplified the algorithm by removing the notion of border points completely, thereby improving consistency with the statistical interpretation of clustering. While not addressed in this work, the algorithms proposed in this paper can be easily adapted for DBSCAN*, with several further optimizations possible.

2.2 Related work

Many papers detail parallelization techniques in distributed [17, 19, 20, 31, 32, 41–43] and shared memory [23, 31, 32, 40] contexts. Here, we focus on the works addressing the algorithm parallelization using GPUs.

[6] proposed two algorithms. CUDA-DClust creates sub-clusters (chains) of points density-reachable from each other. Multiple chains are created simultaneously in parallel on a GPU. The algorithm keeps track of chain collisions through a collision matrix, which is resolved on the CPU in the final stage. CUDA-DClust* is an extension of CUDA-DClust that uses an indexing technique (based on a constant number of directory level partitions) for the computation of $N_\varepsilon(x)$. Two slight modifications of CUDA-DClust, reducing the number of memory transfers between a CPU and a GPU, and identifying core points prior to cluster generation, were proposed in Mr. Scan [42]. [38] offloads the $N_\varepsilon(x)$ computation to the GPU by assigning points in X to different threads, which check the distance to x in parallel. G-DBSCAN [2] constructs the adjacency graph using an all-to-all computation on the GPU, and

then executes a parallel breadth-first search with level synchronization. An extension of CUDA-DClust is realized in CudaSCAN [25], which trims the amount of required distance evaluations by partitioning a data set into subregions and performing local clustering within the sub-regions in parallel. A special case of DBSCAN with $minPts = 2$ was studied in [35], where an implicit graph structure combined with a disjoint-set algorithm was used to find strongly connected components utilizing a cell partitioning of the domain as an indexing structure. [14] utilizes a hybrid CPU-GPU approach in which the neighbors of each point are first identified on the GPU, then the neighbor list is transferred to the host, where the clustering is performed. In [29], the authors compared existing GPU implementations (the algorithm in [38], CUDA-DClust* [6] and G-DBSCAN [2]), and found G-DBSCAN to be the fastest but requiring significantly more memory (166× of CUDA-DClust) due to storing the adjacency graph. [13] extended the work [14], addressing the limitations of the GPU memory by using a batched mode to incrementally compute $N_\varepsilon(x)$, and explored avoiding distance calculations in the dense regions by superimposing a regular grid over the domain, with a special treatment of the cells containing at least *minPts* points, called *dense cells*. CUDA-DClust+ of [34] further improved CUDA-DClust by reducing the amount of CPU-GPU communications and moving more kernels to GPU. A new approach to implement DBSCAN using Nvidia RTX (ray-tracing hardware) was proposed in [30], improving performance for low-density datasets.

This work shares similarities with several of the mentioned algorithms. Similar to [2], our algorithm operates on the adjacency graph. However, in this work, the graph is implicit and is never fully formed, resolving many of the memory constraints of the algorithm identified in [29]. Compared to [35], which can be seen as a precursor, this work implements the full DBSCAN algorithm, uses a synchronization-free non-iterative union-find algorithm, and uses and optimizes a tree-based different indexing structure. Like in this work, [14] identified batched neighbor search as a key to performance; however, that approach produced a full adjacency graph and relied on CPU for the clustering itself. We follow the ideas introduced in [13, 35, 36, 42], and utilize an auxiliary regular grid to reduce the number of distance calculations. Compared to the mentioned works, however, the cells of the grid become primitives used in the construction of the tree, both reducing the size of the tree, and allowing for an easier merge of dense cells. Finally, compared to most of the works mentioned, the algorithm only uses the GPU with no support from a CPU, requiring no data transfer between host and device memories during the execution.

3 PARALLEL DBSCAN FRAMEWORK FOR GPUS

3.1 Disjoint-set based DBSCAN

The main obstacle to the parallelization of the DBSCAN algorithm in the original form (Algorithm 1) is its breadth-first manner of encountering new points, and the linear time required to update the existing neighbor set N. The algorithm proposed in [32] breaks with its breadth-first nature, and serves as the foundation for this work. Instead of maintaining an explicit list of indices, the authors used the UNION-FIND [37] approach to maintain a disjoint-set data structure. The approach relies on two main operations: UNION and FIND. FIND(x) determines the representative of a set that a point x belongs to, while UNION(x, y) combines the sets that x and y belong to.

The UNION-FIND algorithm is typically implemented using trees. For any point x, its representative, returned by FIND(x), is the root of the tree containing x. The UNION(x, y) operation merges two trees (containing x and y) by pointing the parent pointer of one tree root (e.g., FIND(x)) to the other (FIND(y)). If x and y belong to the same set, then FIND(x) and FIND(y) return the same index, and no merging is required. The procedure starts with creating a forest of singleton non-overlapping trees, each corresponding to a set consisting of a single data point. The method proceeds by progressively combining pairs of sets through merging corresponding trees.

From an implementation perspective, the trees in the UNION-FIND algorithm are stored using a flat array, which we will refer to as *labels*. A parent of a node in a tree is then the value of the label corresponding to that node. The FIND operation follows the values of labels until encountering an index that is the same as its label, which indicates that it is the root of that tree. Two trees are merged by changing the label of the root of one of the trees to that of the other.

Algorithm 2 Disjoint-set DBSCAN algorithm

```
1:  procedure DSDBSCAN(X, minPts, ε)
2:      for each point x ∈ X do
3:          N ← GetNeighbors(x, ε)
4:          if |N| ≥ minPts then
5:              mark x as core point
6:              for each y ∈ N do
7:                  if y is marked as a core point then
8:                      Union(x, y)
9:                  else if y is not a member of any cluster then
10:                     mark y as a member of a cluster
11:                     Union(x, y)
```

Algorithm 2 reproduces the disjoint-set DBSCAN (DSDBSCAN) algorithm as proposed in [32] (Algorithm 2), shown here for completeness. Each point now only computes its own neighborhood (Line 3). If it is a core point, its neighbors are assigned to the same cluster (Lines 8 and 11).

In the original paper, a thread or an MPI rank executed the algorithm sequentially for a subset of data constructed by partitioning, and merged the results in parallel to obtain the final clusters. For GPUs, however, more available parallelism is desired to improve the efficiency. In the next Section, we reformulate the algorithm to allow that.

3.2 Parallel disjoin-set based DBSCAN

While the amount of the parallelism in Algorithm 2 may be sufficient for shared- or distributed-memory implementations, it is insufficient for GPU implementations with thousands or tens of thousands threads. Therefore, our goals were to reformulate the algorithm to accommodate such a high number of threads, and to to reduce thread execution divergence (executing different code) and data divergence (reading or writing disparate locations in memory) in the algorithm.

Algorithm 2 consists of two distinct kernels: the neighbor search, and the disjoint-set structure update. It is clear that the former is more computationally demanding than the latter. Without taking

appropriate care, calling `GetNeighbors` asynchronously by different threads will result in high execution and data divergence. This is especially true when an index structure, such as *k*-d tree or R-* tree, is used. Thus, the neighbor searches are executed simultaneously for all points in a batched mode.

We next address the limited amount of available GPU memory. Storing all the neighbors found on Line 3 for all threads executed at the same time may not be possible, given that the number of such neighbors may be a significant fraction of the overall dataset size. This can be addressed by observing that the neighbor list is being used in two different contexts. For assessing whether a point is a core point on Line 4, the only information required is the number of neighbors, but not the neighbors themselves. In the loop on Line 6, the neighbors are assigned to the same cluster as part of the Union-Find algorithm. The key observation here is that the neighbors may be processed independently and in any order. In other words, it is possible to process them as they are determined and execute the Union operation on-the-fly for each neighbor, discarding the found neighbor after that.

Given these findings, we split the algorithm into two phases. In the first phase, called *preprocessing*, the algorithm determines the core points. We note that while it is possible to do this by computing the exact number of neighbors $|N_\varepsilon(x)|$, it is not necessary. If the neighbors of a point are discovered incrementally (whether through a tree traversal, or otherwise), it is sufficient to encounter just *minPts* neighbors to determine a core point (unless executing a sweep over multiple values of *minPts*).

The second phase, called *main*, proceeds with the knowledge of core points, and executes $\textsc{Union}(x, y)$ for each pair of close neighbors as they are being discovered. This general formulation leaves a lot of room for optimizations. For example, many of the distance calculations may be eliminated. We examine this in more detail in Section 4.

The two-phase approach results in dramatic reduction of the consumed memory and in better avoidance of thread and data divergence. The memory consumption does not depend on the values of ε and *minPts* and is linear with respect to the number of points in a dataset (assuming the used search index obeys this, too). This makes it possible to execute the algorithm for much larger datasets. As was observed in earlier works, algorithms that store full neighbor lists (e.g., G-DBSCAN) tend to run out of memory even for smaller datasets, particularly in situations where $|N_\varepsilon(x)| \gg minPts$ for a significant fraction of points.

An additional advantage of the two-phase approach is that it exposes edge-level parallelism in addition to the vertex-level parallelism. One could consider using multiple threads collaborating on a single point, with each thread assigned one of the outgoing edges in the adjacency graph. Such an approach would require implementing a search index (tree or otherwise) with multiple threads collaborating on a single search query.

The pseudocode for the parallel disjoint-set Dbscan (PDSDbscan) algorithm is shown in Algorithm 3. The preprocessing phase is executed on Lines 3-4. The check on Line 2 allows the preprocessing phase to be skipped in the special case when $minPts = 2$. In this case, any pair of points found within distance ε in the main phase is guaranteed to consist of core points. The Union-Find algorithm is performed on Lines 8 and 11.

Algorithm 3 Parallel disjoint-set Dbscan algorithm

```
1:  procedure PDSDbscan(X, minPts, ε)
2:      if minPts > 2 then
3:          for each point x ∈ X in parallel do
4:              determine whether x is a core point
5:      for each pair of points x, y such that dist(x, y) ≤ eps in parallel
    do
6:          if x is a core point then
7:              if y is a core point then
8:                  Union(x, y)
9:              else if y is not yet a member of any cluster then
10:                 critical section:
11:                     mark y as a member of a cluster
12:                     Union(x, y)
```

The operations on Lines 11 and 12 must be executed in a single critical section. If a thread is marking y as a member of its own cluster, no other thread is allowed to execute Union with y. Otherwise, it may lead to the "bridging" effect, where a border point within distance ε of two separate clusters may result in merging those clusters together. In practice, it is possible to use the labels array for both clustering information, and as an indicator for whether a border point is a member of a cluster. In this approach, the check on Line 9 compares the label of point y with y. If they are identical, the label is assigned the representative of x. It allows us to replace the critical section with a single atomic compare-and-swap operation.

In summary, the proposed approach allows execution of the full Dbscan algorithm on a GPU fully in parallel. No data transfers between a CPU and a GPU are necessary as long as both the data and the chosen search index fit into the GPU memory.

4 TREE-BASED ALGORITHMS

4.1 FDBSCAN

FDbscan ("fused" DBSCAN) fuses tree traversal with the Union-Find algorithm. It uses a bounding volume hierarchy (BVH), a structure commonly used in computer graphics for ray tracing, for the search index. While any tree can be used, BVH has been shown to be very efficient for low-dimensional data on GPUs. Linear BVH (LBVH) (e.g., [22]), are well suited for GPUs, with low data and thread divergence during both construction and traversal.

The parallelization is done over all points of a dataset, with each thread assigned a single point. The neighbor search is executed in bulk (i.e., with all threads launching at the same time). The threads are sorted using space-filling curve to reduce data and execution divergence during the traversal. Each thread executes a stack-less top-down traversal. In the preprocessing phase, we use the recommendation from the previous Section, terminating the traversal of a thread once a *minPts* neighbors are encountered. In the main phase, the algorithm executes Union operation when a new neighbor is found, without storing said neighbor.

We use an additional optimization in the main phase. In Algorithm 3, the algorithm can be seen as operating on the edges of the adjacency graph. As the results of $\textsc{Union}(x, y)$ and $\textsc{Union}(y, x)$ are identical from a cluster membership perspective, it is sufficient to process each edge only once. To facilitate this, we introduced a new hierarchy traversal algorithm. Given a thread corresponding to a

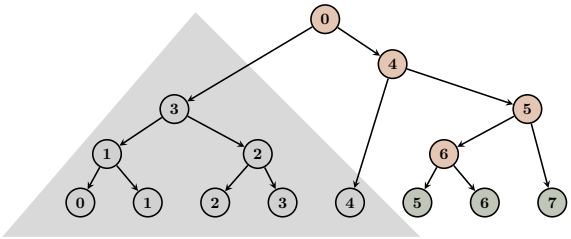

Figure 1: An example of the tree traversal mask for a thread corresponding to a point with index 4.

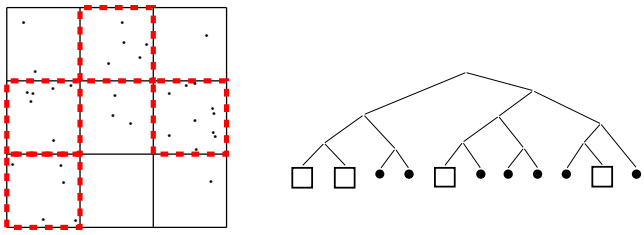

Figure 2: Left: regular grid with grid size $\varepsilon/\sqrt{d}$ superimposed over the dataset. The dense cells for *minPts* = 5 are shown in red. Right: BVH constructed from a mixed set of objects.

point with index i, a subtree corresponding to the leaf nodes with indices less than i is hidden from the thread. This way, the thread avoids entering the subtrees with lower leaf indices, guaranteeing that all the found neighbors would have indices $j > i$, thus guaranteeing that each pair of neighboring points is processed exactly once. Figure 1 demonstrates the tree mask for a thread corresponding to index 4. The thread would stay in the right subtree of the root, skipping the left subtree entirely. The advantages of such an approach include fewer memory accesses used during the traversal, reduced number of distance computations, and reduced number of Union-Find operations.

4.2 FDBSCAN-DenseBox

A given combination of *minPts* and ε often results in the number of neighbors within an ε-neighborhood of a point significantly exceeding the value of *minPts*. In this case, many of the distance computations may be avoided. In this Section, we propose an alternative approach to FDbscan which takes advantage of this fact.

Eliminating extra distance computations has been studied in [13, 42]. The methods operate by superimposing a uniform Cartesian grid and processing cells with at least *minPts* points more efficiently. We integrated these ideas into a tree-based search index, which we call FDbscan-DenseBox.

The procedure starts with computing the bounds of the data set and imposing a regular grid over the computational domain. The grid cell length is set to be $\varepsilon/\sqrt{d}$, with d being the data dimension. This choice guarantees that the diameter of each cell does not exceed ε. Next, we calculate a cell index for all points in the dataset, and determine the number of points in each cell. The cells with at least *minPts* are called *dense*. Figure 2 demonstrates a grid superimposed over a set of points, with dense cells for *minPts* = 5 marked in red. It is clear that all the points in the dense cells are core points, and belong to the same cluster. Thus, the distance calculations among the points in the same dense cell can be eliminated.

The number of dense cells and the number of points inside them depend heavily on the dataset data distribution and the parameters ε and *minPts*. If the value of ε is small compared to the domain size, the number of grid cells in each dimension may be in thousands or more, resulting in billions of grid cells. The data is then spread across a relatively small population of non-empty cells. Searching for nearby cells in this situation becomes non-trivial. While it is possible to do a series of binary searches over a list of cells to produce a list of neighboring non-empty cells, in this work we use an alternative approach.

To accommodate dense boxes, we modify the BVH construction algorithm of FDbscan. In FDbscan-DenseBox, the hierarchy is constructed out of a mix of points outside of dense cells and the boxes of the dense cells. This is possible to do as the BVH only requires bounding volumes for a set of objects. Thus, such mixing does not impose any additional constraints. The use of this approach with other trees, such as k-d tree, would pose more challenges.

Given the knowledge that all points in dense cells are core points, only the points outside of dense cells have to be examined to identify the remaining core points in the preprocessing phase. For every such point, the algorithm finds all nearby objects within distance ε using the BVH. If the found object is an isolated point, the neighbor count is incremented by one. If it is a box (corresponding to a dense cell), a linear search over all points in that cell is performed, incrementing the count each time a point is within distance ε. Similar to FDbscan, the neighbors are only counted until reaching the *minPts* threshold, after which the procedure terminates.

At the beginning of the main phase, the Union operation is executed for all points within the same dense cell. Then, the neighborhood search is performed for all points in the dataset. During the search, once an object within distance ε is found for an individual point, one of two cases may happen. In the first case, the found object is a dense box. In this case, it is sufficient to determine whether a single point of that dense box is within distance ε. A thread checks the distances to all points in that dense cell linearly, until either a point within ε is found, in which case Union() is called, or all points are exhausted. In the second case, the found object is another point (outside of any dense cell). As the newly found point is within ε, the usual resolution depending on the core status of both points is executed.

One drawback of FDbscan-DenseBox, compared to FDbscan, is its use of arithmetic operations (e.g., summation) when dealing with the cell computations. These calculations may suffer from a loss of precision in the situations where the value of ε is tiny compared to the coordinates of the data points, potentially resulting in erroneous results. This should be detected and guarded against in an implementation. Alternatively, this could be addressed by using a higher precision floating point numbers, or through hashing techniques. FDbscan, on the other hand, only uses min and max operations on the user data and does not have this limitation.

4.3 Union-Find

We chose the algorithm proposed in [21] as our Union-Find approach, being synchronization-free on GPUs. Like most efficient

Table 1: Datasets and the default parameters

Name	d	n	Source	Description	Default parameters		
					ε	*minPts*	Samples
2D-NGSIM	2	∼12M	[1]	GPS loc	1.0	10	100K
2D-Porto	2	∼81M	[28]	GPS loc	0.005	10	100K
2D-SS-simden	2	10M	[12]	Generated	1000	10	100K
2D-SS-varden	2	10M	[12]	Generated	1000	10	100K
3D-Hacc	3	∼37M	[18]	Cosmology	0.042	10	1M
3D-SS-simden	3	10M	[12]	Generated	1000	10	1M
3D-SS-varden	3	10M	[12]	Generated	1000	10	1M
5D-SS-simden	5	10M	[12]	Generated	1000	10	1M
5D-SS-varden	5	10M	[12]	Generated	1000	10	1M
7D-SS-simden	7	10M	[12]	Generated	1000	10	1M
7D-SS-varden	7	10M	[12]	Generated	1000	10	1M
7D-Household	7	∼2M	[10]	Power	2.0	10	1M

implementations, it uses pointer jumping, a technique to shorten paths of the trees (associated with disjoint sets) during the Find operation. Specifically, the work uses "intermediate pointer jumping", which compresses the path of all elements encountered on a way to the tree root by making every element skip over the next element, halving the path length in each traversal. Because the path compression does not guarantee that all paths are fully compressed at the end of the main phase (i.e., that the label of each point in the same cluster is identical at the end of the main phase), an extra finalization phase is introduced to make each point directly to the representative.

5 EXPERIMENTAL RESULTS

In our implementation, we used ArborX [24], an open-source library for the tree-based implementations using Kokkos library [39] for a device-independent programming model. Kokkos offers parallel execution patterns (parallel loops, reductions, scans) to abstract from a specific hardware. Kokkos also provides abstractions for execution and memory resources. The Kokkos library[1] provides C++ abstractions and supports hardware through backends, including Nvidia GPUs (Cuda), AMD GPUs (HIP), and serial hosts (Serial).

The implemented algorithms are available in the main ArborX repository[2].

The ArborX library provides several features suitable for our implementation. It allows for an early traversal termination, which is used in the preprocessing phases of both FDbscan and FDbscan-DenseBox. The callback functionality of the library allows execution of a user-provided code on a positive match, which is used both in preprocessing for the neighbor count and in the main phase for the Union-Find kernels.

Testing environment. The numerical studies presented in the paper were performed using AMD EPYC 7763 (64 cores[3]), Nvidia A100 (40GB) and a single GCD (Graphics Compute Die) of AMD MI250X[4]. The chips are based on TSMC's N7+, N7 and N6 technology, respectively, and can be considered to belong to the same generation.

[1]https://github.com/kokkos/kokkos
[2]https://github.com/arborx/ArborX
[3]Run as 56 cores, with 8 cores dedicated to OS processes
[4]Currently, HIP (Heterogeneous-computing Interface for Portability) – the programming interface provided by AMD – only allows the use of each GCD as an independent GPU.

We used Clang 14.0.0 compiler for AMD EPYC 7763, NVCC 11.5 for Nvidia A100, and ROCm 5.4.3 for AMD MI250X.

Datasets. As mentioned in Section 1, in this work we focus on the low-dimensional data. For our experiments, we used a combination of artificial and real-world datasets listed in Table 1 to comprehensively evaluate our algorithm and meet our study goals. The GPS locations (*2D-NGSIM* and *2D-Porto*), cosmology (*3D-HACC*) and electric power consumption (*7D-Household*) datasets replicate real-world conditions. The datasets generated with [12] allow us to explore more structure and dimensionalities. *SS-simden* and *SS-varden* refer to the datasets with similar-density and variable-density clusters, respectively.

5.1 Parallel algorithms comparison

In this Section, we compare the performance of FDbscan and FDbscan-DenseBox algorithms with several other implementations: G-Dbscan [2] (only available for 2D datasets), Pdsdbscan-S [31] and Tepp [40]. We did not include the results for CUDA-DClust [6] as it was many orders of magnitude slower. Unfortunately, we were also not able to compare to the recent CUDA-DClust+ [34] code[5], as it consistently produced wrong results and did not match the performance reported in [34]; the problem seems to be related to `thrust::equal_range` routines and is being investigated by the original authors at the time of this publication.

We study the behavior of the algorithms for each dataset varying one of the three parameters, ε, *minPts*, and the number of drawn random samples, while keeping the other two fixed at the default values shown in Table 1. The default number of samples for the 2D datasets was chosen to be lower to accomodate G-Dbscan's memory consumption.

In this Section, the G-Dbscan, FDbscan and FDbscan-DenseBox experiments were performed on Nvidia A100.

Impact of ε. Figure 3 demonstrates the impact of the parameter ε on the execution times while keeping *minPts* and problem size fixed at the default values. Increasing ε increases the size of each neighborhood $N_\varepsilon(x)$, thus increasing the cluster sizes. The range of ε for each problem was chosen in such a way that the number of clusters qualitatively changes from many small clusters to a few large ones.

We first observe that for the 2D cases where G-Dbscan was able to run, it is an obvious outlier in terms of performance. Generally, Pdsdbscan-S is the second slowest, running significantly slower than Tepp and FDbscan, particularly for larger values of ε. Tepp is competitive with FDbscan in some situations, particularly for large values of ε and *2D-Porto*, where the densities of the data points are high and FDbscan performs a lot of unnecessary computations. However, FDbscan-DenseBox outperforms Tepp in almost all situations, except for the *2D-Porto* and the largest values of ε for *2D-SS-simden*, *2D-SS-varden*. FDbscan outperforms FDbscan-DenseBox for lower values of ε in most situations, which corresponds to situations with lower density values, and thus few (if any) dense cells. The rule of thumb is to use FDbscan for very low sparsity situations, and FDbscan-DenseBox otherwise.

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

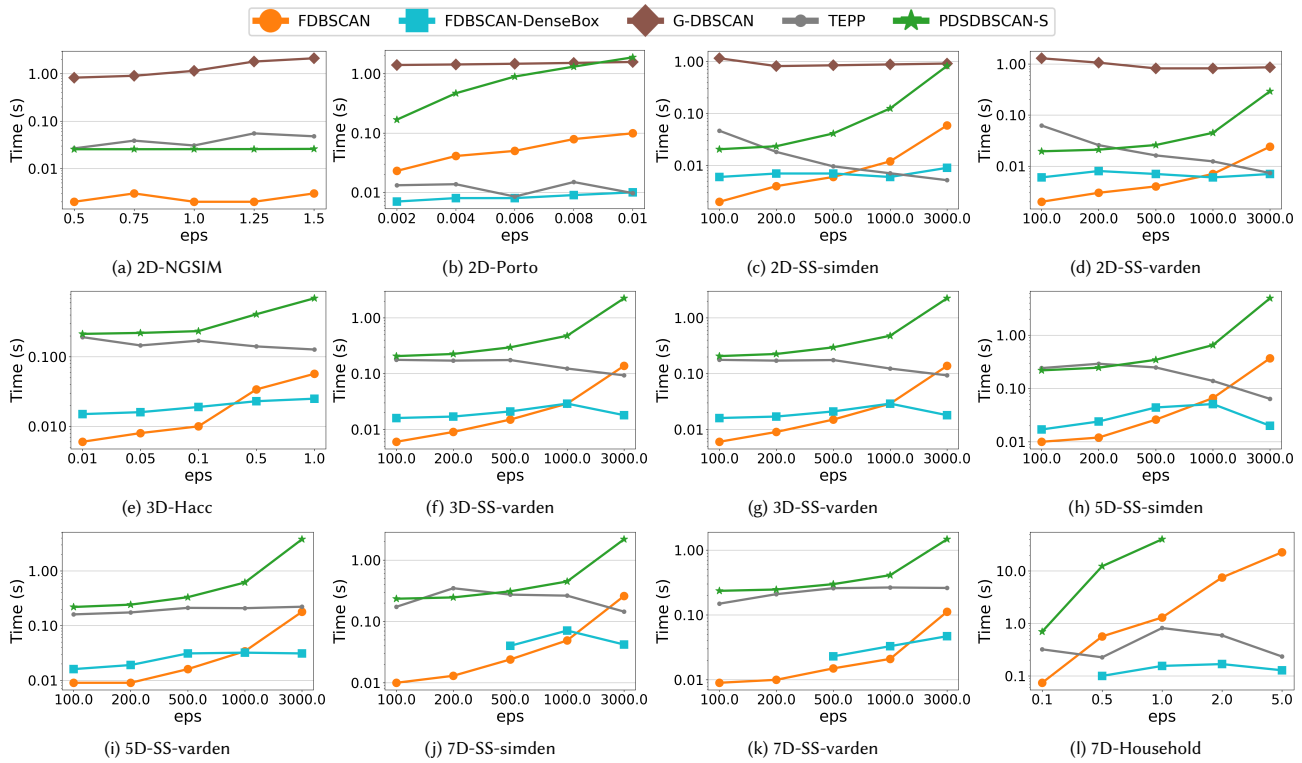

Figure 3: Impact of the ε parameter on the execution time.

It is important to note the few missing data points in the plots. First, we see that FDBSCAN-DENSEBOX is completely missing in *2D-NGSIM*, and in several lower ε values for *7D-SS-simden* and *7D-SS-varden*. As we mentioned at the end of Section 4.2, a combination of the domain size and the values of ε may lead to FDBSCAN-DENSEBOX losing precision and potentially leading to the wrong results. This is exactly what is happening here, and our implementation of FDBSCAN-DENSEBOX aborted the computation. We also observe missing data for PDSDBSCAN-S for *7D-Household*, where it ran out of memory.

Another interesting observation is the expected dependence of FDBSCAN on the ε parameter: larger values of ε result in the longer runtimes, as it increases the size of $N_\varepsilon(x)$ neighborhoods, and FDBSCAN has no mechanisms to avoid additional computations. On the other hand, the time for FDBSCAN-DENSEBOX is relatively stable for the full range of ε.

Impact of minPts. Figure 4 shows the effect of varying the *minPts* parameter while keeping ε and the problem size fixed at the default values.

We observed that in most situations the algorithms exhibit little change in the behavior, except for FDBSCAN-DENSEBOX which trends slower for larger *minPts* values as the number of the dense cells decreases. For many datasets, FDBSCAN performs faster than FDBSCAN-DENSEBOX due to the chosen fixed value of ε. The growth in FDBSCAN results is explained by the longer preprocessing phase, as the early termination only happens once *minPts* neighbors are found; the main phase is almost unaffected by the *minPts* parameter. The preprocessing phase of FDBSCAN-DENSEBOX is affected in a similar way, but in addition, the main phase also takes longer due to larger mixed hierarchy sizes due to lower number of dense cells. This is particularly noticeable in *2D-Porto* and *7D-Household*. We see that either FDBSCAN or FDBSCAN-DENSEBOX are still universally the fastest algorithms, often by a large margin.

Impact of the number of points in the dataset. For our final comparison, we varied the size of the problem by increasing the number of drawn samples for each dataset while keeping the values of ε and *minPts* fixed. We chose random sampling as we could not rely on the organization points in the datasets. However, this results in the problems becoming denser with increasing size, affecting the performance in addition to the increases in size.

Figure 5 presents the results, shown in log-log scale. TEPP and FDBSCAN-DENSEBOX scale similarly and slower than PDSDBSCAN-S and FDBSCAN-DENSEBOX. Between FDBSCAN and FDBSCAN-DENSEBOX, for almost all datasets there is a point at which the FDBSCAN-DENSEBOX becomes faster due to reaching sufficient density. Both G-DBSCAN and PDSDBSCAN-S are clear outliers in terms of performance.

In addition to missing FDBSCAN-DENSEBOX data points for the *2D-NGSIM* due to the loss of precision, we also note G-DBSCAN running out of memory at very modest problem sizes. This is expected as G-DBSCAN stores the full adjacency matrix data, so that even 40GB A100 memory is not sufficient for storage.

Summary. FDBSCAN and FDBSCAN-DENSEBOX clearly prove to be very competitive algorithms, often outperforming other existing algorithms by an order of magnitude, with FDBSCAN-DENSEBOX

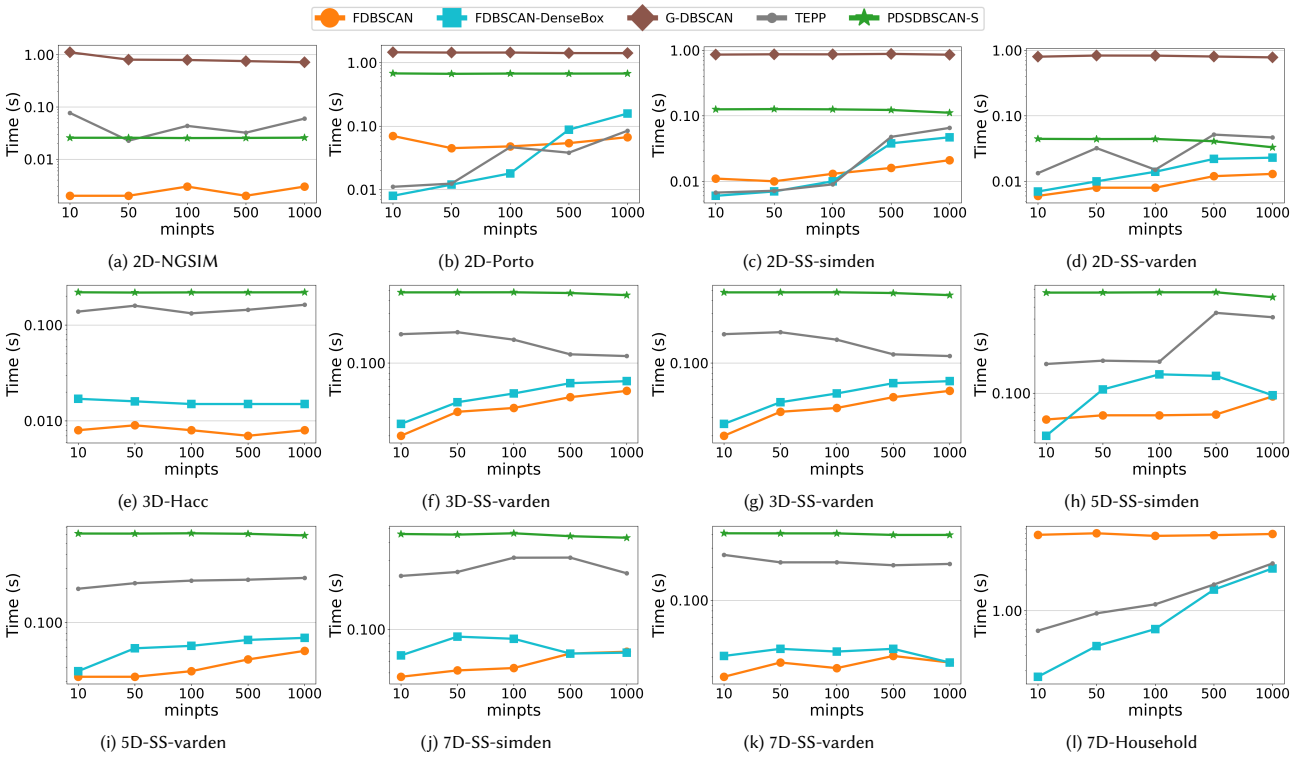

Figure 4: Impact of the *minPts* parameter on the execution time.

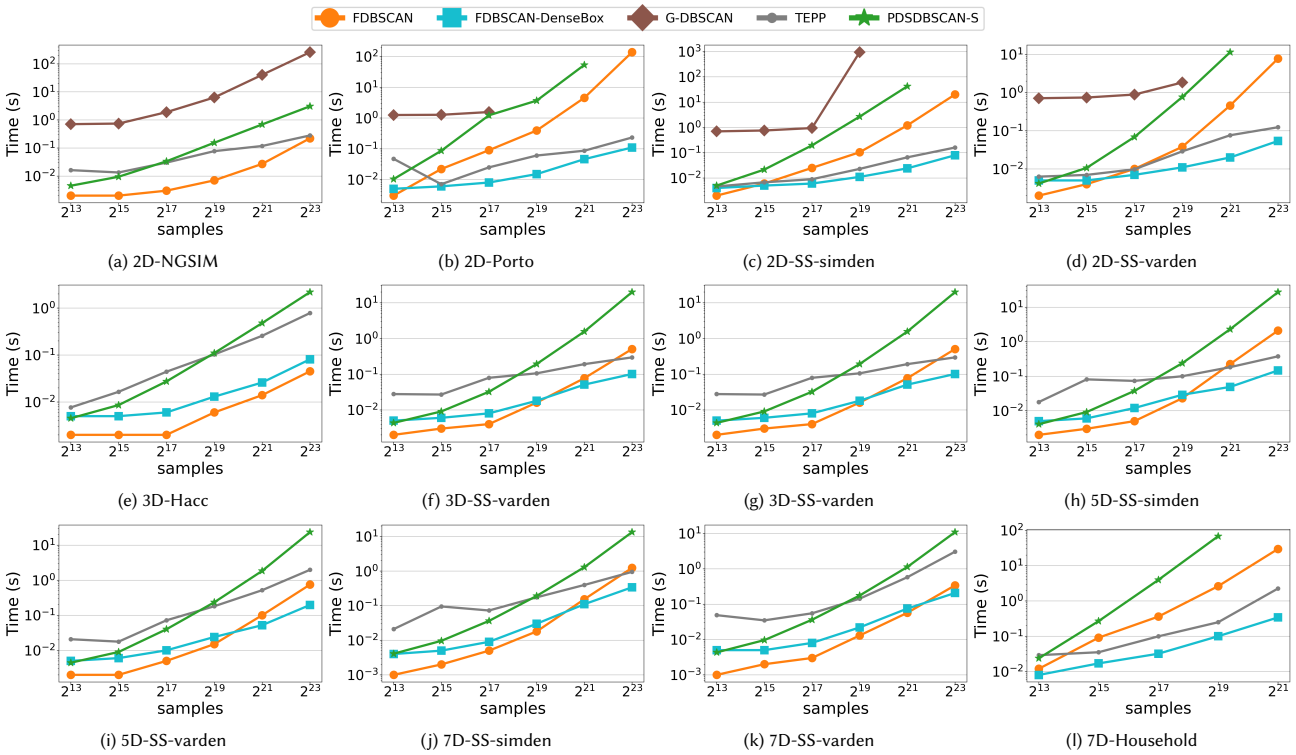

Figure 5: Impact of the number of samples drawn from a dataset on the execution time.

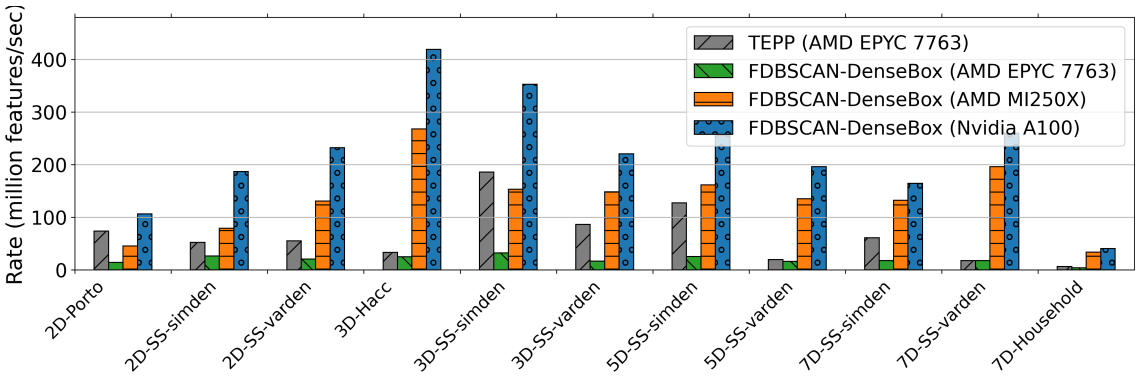

Figure 6: Rate comparison across different hardware architectures.

being the typically the much faster of the two. Both algorithms proposed in this paper do not suffer from the significant memory limitations. The closest competitor to the algorithms is the Tepp multi-threaded implementation.

5.2 Performance portability

In this Section, we discuss the performance portability of the implemented algorithms through the use of the Kokkos library [39]. Figure 6 shows the performance of the FDbscan-DenseBox algorithm on different hardware: AMD EPYC 7763 (through OpenMP backend), AMD MI250X (through HIP backend), and Nvidia A100 (through CUDA backend). Tepp baseline is provided for AMD EPYC 7763. The results are presented as the rate, million features (product of the number of points and dimension) per second.

We see that AMD MI250X is 1.2-2.3× slower than Nvidia A100, which is explained by using a single GCD. The OpenMP implementation is 1.0-5.7× slower than Tepp, and is expected given that the algorithm is designed for GPU architectures.

Similar performance portability results hold for the FDbscan algorithm.

6 CONCLUSIONS AND FUTURE WORK

We presented a general parallel approach for Dbscan on GPUs, and introduced two algorithms based on a bounding volume hierarchy tree implementation. These algorithms were evaluated against the other existing CPU and GPU algorithms, demonstrating their excellent performance. The algorithms were shown to be performance portable and able to run on a variety of hardware architectures, including multi-threaded CPUs and GPUs. We showed that a special treatment of dense areas by using an auxiliary Cartesian grid is advantageous in many situations.

Algorithmically, we see a number of research directions to pursue. Similar to [13], we envision using a heuristic to automatically switch between FDbscan and FDbscan-DenseBox for a given problem. An introduction of a batched mode is of interest for applications where the data and the index do not fit in the GPU memory. Other directions of research include combining the proposed approach with distributed computations, lowering memory requirements of the used search index, and incorporating other Dbscan variants such as Dbscan*.

ACKNOWLEDGMENTS

The authors are grateful to Dr. Eleazar Leal for providing the source code for the algorithms used in [29] paper for comparison. This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

REFERENCES

[1] 2018. Next Generation Simulation (NGSIM) Vehicle Trajectories and Supporting Data. Available online: https://catalog.data.gov/dataset/next-generation-simulation-ngsim-vehicle-trajectories-and-supporting-data. Accessed: 2021-03-06.

[2] Guilherme Andrade, Gabriel Ramos, Daniel Madeira, Rafael Sachetto, Renato Ferreira, and Leonardo Rocha. 2013. G-DBSCAN: A GPU Accelerated Algorithm for Density-based Clustering. *Procedia Computer Science* 18 (Jan. 2013), 369–378. https://doi.org/10.1016/j.procs.2013.05.200

[3] Domenica Arlia and Massimo Coppola. 2001. Experiments in Parallel Clustering with DBSCAN. In *Euro-Par 2001 Parallel Processing*, Rizos Sakellariou, John Gurd, Len Freeman, and John Keane (Eds.). Springer, Berlin, Heidelberg, 326–331. https://doi.org/10.1007/3-540-44681-8_46

[4] J. L. Bentley. 1975. Multidimensional Binary Search Trees Used for Associative Searching. *Communication of the ACM* 18, 9 (September 1975), 509–517. https://doi.org/10.1145/361002.361007

[5] Christian Böhm, Stefan Berchtold, and Daniel A. Keim. 2001. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *Comput. Surveys* 33, 3 (Sept. 2001), 322–373. https://doi.org/10.1145/502807.502809

[6] Christian Böhm, Robert Noll, Claudia Plant, and Bianca Wackersreuther. 2009. Density-based clustering using graphics processors. In *Proceedings of the 18th ACM conference on Information and knowledge management (CIKM '09)*. Association for Computing Machinery, Hong Kong, China, 661–670. https://doi.org/10.1145/1645953.1646038

[7] Ricardo J. G. B. Campello, Davoud Moulavi, and Joerg Sander. 2013. Density-Based Clustering Based on Hierarchical Density Estimates. In *Advances in Knowledge Discovery and Data Mining (Lecture Notes in Computer Science)*, Jian Pei, Vincent S. Tseng, Longbing Cao, Hiroshi Motoda, and Guandong Xu (Eds.). Springer, Berlin, Heidelberg, 160–172. https://doi.org/10.1007/978-3-642-37456-2_14

[8] Yewang Chen, Lida Zhou, Songwen Pei, Zhiwen Yu, Yi Chen, Xin Liu, Jixiang Du, and Naixue Xiong. 2019. KNN-BLOCK DBSCAN: Fast Clustering for Large-Scale Data. *IEEE Transactions on Systems, Man, and Cybernetics: Systems* (2019), 1–15. https://doi.org/10.1109/TSMC.2019.2956527

[9] B. Dai and I. Lin. 2012. Efficient Map/Reduce-Based DBSCAN Algorithm with Optimized Data Partition. In *2012 IEEE Fifth International Conference on Cloud Computing*. 59–66. https://doi.org/10.1109/CLOUD.2012.42

[10] Dheeru Dua and Casey Graff. 2017. UCI Machine Learning Repository. http://archive.ics.uci.edu/ml

[11] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD'96)*. AAAI Press, 226–231.

[12] Junhao Gan and Yufei Tao. 2017. On the Hardness and Approximation of Euclidean DBSCAN. *ACM Transactions on Database Systems* 42, 3 (July 2017), 14:1–14:45. https://doi.org/10.1145/3083897

[13] Michael Gowanlock. 2019. Hybrid CPU/GPU clustering in shared memory on the billion point scale. In *Proceedings of the ACM International Conference on Supercomputing (ICS '19)*. Association for Computing Machinery, Phoenix, Arizona, 35–45. https://doi.org/10.1145/3330345.3330349

[14] Michael Gowanlock, Cody M. Rude, David M. Blair, Justin D. Li, and Victor Pankratius. 2017. Clustering Throughput Optimization on the GPU. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 832–841. https://doi.org/10.1109/IPDPS.2017.17

[15] M. Gowanlock, C. M. Rude, D. M. Blair, J. D. Li, and V. Pankratius. 2019. A Hybrid Approach for Optimizing Parallel Clustering Throughput using the GPU. *IEEE Transactions on Parallel and Distributed Systems* 30, 4 (April 2019), 766–777. https://doi.org/10.1109/TPDS.2018.2869777

[16] A. Guttman. 1984. R-trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data* (Boston, Massachusetts) *(SIGMOD '84)*. ACM, New York, NY, USA, 47–57. https://doi.org/10.1145/602259.602266

[17] Markus Götz, Christian Bodenstein, and Morris Riedel. 2015. HPDBSCAN: highly parallel DBSCAN. In *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments (MLHPC '15)*. Association for Computing Machinery, Austin, Texas, 1–10. https://doi.org/10.1145/2834892.2834894

[18] Salman Habib, Adrian Pope, Hal Finkel, Nicholas Frontiere, Katrin Heitmann, David Daniel, Patricia Fasel, Vitali Morozov, George Zagaris, Tom Peterka, et al. 2016. HACC: Simulating sky surveys on state-of-the-art supercomputing architectures. *New Astronomy* 42 (2016), 49–65.

[19] Yaobin He, Haoyu Tan, Wuman Luo, Huajian Mao, Di Ma, Shengzhong Feng, and Jianping Fan. 2011. MR-DBSCAN: An Efficient Parallel Density-Based Clustering Algorithm Using MapReduce. In *2011 IEEE 17th International Conference on Parallel and Distributed Systems*. 473–480. https://doi.org/10.1109/ICPADS.2011.83

[20] Xu Hu, Jun Huang, and Minghui Qiu. 2017. A Communication Efficient Parallel DBSCAN Algorithm based on Parameter Server. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management (CIKM '17)*. Association for Computing Machinery, Singapore, Singapore, 2107–2110. https://doi.org/10.1145/3132847.3133112

[21] Jayadharini Jaiganesh and Martin Burtscher. 2018. A High-performance Connected Components Implementation for GPUs. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '18)*. ACM, New York, NY, USA, 92–104. https://doi.org/10.1145/3208040.3208041

[22] T. Karras. 2012. Maximizing Parallelism in the Construction of BVHs, Octrees, and K-d Trees. In *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics (EGGH-HPG'12)*. Eurographics Association, Goslar Germany, Germany, 33–37. https://doi.org/10.2312/EGGH/HPG12/033-037

[23] Sonal Kumari, Poonam Goyal, Ankit Sood, Dhruv Kumar, Sundar Balasubramaniam, and Navneet Goyal. 2017. Exact, Fast and Scalable Parallel DBSCAN for Commodity Platforms. In *Proceedings of the 18th International Conference on Distributed Computing and Networking (ICDCN '17)*. Association for Computing Machinery, Hyderabad, India, 1–10. https://doi.org/10.1145/3007748.3007773

[24] D. Lebrun-Grandié, A. Prokopenko, B. Turcksin, and S. R. Slattery. 2020. ArborX: A Performance Portable Geometric Search Library. *ACM Trans. Math. Software* 47, 1 (Dec. 2020), 2:1–2:15. https://doi.org/10.1145/3412558

[25] Woong-Kee Loh and Hwanjo Yu. 2015. Fast density-based clustering through dataset partition using graphics processing units. *Information Sciences* 308 (July 2015), 94–112. https://doi.org/10.1016/j.ins.2014.10.023

[26] Alessandro Lulli, Matteo Dell'Amico, Pietro Michiardi, and Laura Ricci. 2016. NG-DBSCAN: scalable density-based clustering for arbitrary data. *Proceedings of the VLDB Endowment* 10, 3 (Nov. 2016), 157–168. https://doi.org/10.14778/3021924.3021932

[27] Daniel Meister, Shinji Ogaki, Carsten Benthin, Michael J. Doyle, Michael Guthe, and Jiří Bittner. 2021. A Survey on Bounding Volume Hierarchies for Ray Tracing. *Computer Graphics Forum* 40, 2 (2021), 683–712. https://doi.org/10.1111/cgf.142662

[28] Luis Moreira-Matias, Joao Gama, Michel Ferreira, Joao Mendes-Moreira, and Luis Damas. 2013. Predicting taxi–passenger demand using streaming data. *IEEE Transactions on Intelligent Transportation Systems* 14, 3 (2013), 1393–1402.

[29] Hamza Mustafa, Eleazar Leal, and Le Gruenwald. 2019. An Experimental Comparison of GPU Techniques for DBSCAN Clustering. In *2019 IEEE International Conference on Big Data (Big Data)*. 3701–3710. https://doi.org/10.1109/BigData47090.2019.9006169

[30] Vani Nagarajan and Milind Kulkarni. 2023. RT-DBSCAN: Accelerating DBSCAN using Ray Tracing Hardware. https://doi.org/10.48550/arXiv.2303.09655

[31] Md. Mostofa Ali Patwary, Suren Byna, Nadathur Rajagopalan Satish, Narayanan Sundaram, Zarija Lukić, Vadim Roytershteyn, Michael J. Anderson, Yushu Yao, Prabhat, and Pradeep Dubey. 2015. BD-CATS: big data clustering at trillion particle scale. In *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12. https://doi.org/10.1145/2807591.2807616

[32] Md. Mostofa Ali Patwary, Diana Palsetia, Ankit Agrawal, Wei-keng Liao, Fredrik Manne, and Alok Choudhary. 2012. A new scalable parallel DBSCAN algorithm using the disjoint-set data structure. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–11. https://doi.org/10.1109/SC.2012.9

[33] Md. Mostofa Ali Patwary, Nadathur Satish, Narayanan Sundaram, Fredrik Manne, Salman Habib, and Pradeep Dubey. 2014. Pardicle: Parallel Approximate Density-Based Clustering. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 560–571. https://doi.org/10.1109/SC.2014.51

[34] Madhav Poudel and Michael Gowanlock. 2021. CUDA-DClust+: Revisiting Early GPU-Accelerated DBSCAN Clustering Designs. In *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. 354–363. https://doi.org/10.1109/HiPC53243.2021.00049

[35] Christopher Sewell, Li-ta Lo, Katrin Heitmann, Salman Habib, and James Ahrens. 2015. Utilizing many-core accelerators for halo and center finding within a cosmology simulation. In *2015 IEEE 5th Symposium on Large Data Analysis and Visualization (LDAV)*. 91–98. https://doi.org/10.1109/LDAV.2015.7348076

[36] Hwanjun Song and Jae-Gil Lee. 2018. RP-DBSCAN: A Superfast Parallel DBSCAN Algorithm Based on Random Partitioning. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. Association for Computing Machinery, Houston, TX, USA, 1173–1187. https://doi.org/10.1145/3183713.3196887

[37] Robert Endre Tarjan. 1979. A class of algorithms which require nonlinear time to maintain disjoint sets. *J. Comput. System Sci.* 18, 2 (April 1979), 110–127. https://doi.org/10.1016/0022-0000(79)90042-4

[38] Rajeev J. Thapa, Christian Trefftz, and Greg Wolffe. 2010. Memory-efficient implementation of a graphics processor-based cluster detection algorithm for large spatial databases. In *2010 IEEE International Conference on Electro/Information Technology*. 1–5. https://doi.org/10.1109/EIT.2010.5612134

[39] Christian R. Trott, Damien Lebrun-Grandié, Daniel Arndt, Jan Ciesko, Vinh Dang, Nathan Ellingwood, Rahulkumar Gayatri, Evan Harvey, Daisy S. Hollman, Dan Ibanez, Nevin Liber, Jonathan Madsen, Jeff Miles, David Poliakoff, Amy Powell, Sivasankaran Rajamanickam, Mikael Simberg, Dan Sunderland, Bruno Turcksin, and Jeremiah Wilke. 2022. Kokkos 3: programming model extensions for the exascale era. *IEEE Transactions on Parallel and Distributed Systems* 33, 4 (April 2022), 805–817. https://doi.org/10.1109/TPDS.2021.3097283 Conference Name: IEEE Transactions on Parallel and Distributed Systems.

[40] Yiqiu Wang, Yan Gu, and Julian Shun. 2020. Theoretically-Efficient and Practical Parallel DBSCAN. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. Association for Computing Machinery, Portland, OR, USA, 2555–2571. https://doi.org/10.1145/3318464.3380582

[41] Benjamin Welton and Barton P. Miller. 2014. The Anatomy of Mr. Scan: A Dissection of Performance of an Extreme Scale GPU-Based Clustering Algorithm. In *2014 5th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*. 54–60. https://doi.org/10.1109/ScalA.2014.10

[42] Benjamin Welton, Evan Samanas, and Barton P. Miller. 2013. 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*. 1–11. https://doi.org/10.1145/2503210.2503262

[43] Xiaowei Xu, Jochen Jäger, and Hans-Peter Kriegel. 1999. A Fast Parallel Clustering Algorithm for Large Spatial Databases. *Data Mining and Knowledge Discovery* 3, 3 (Sept. 1999), 263–290. https://doi.org/10.1023/A:1009884809343