CONTENTS

GPU-INSCY: A GPU-Parallel Algorithm and Tree Structure for Efficient Density-based Subspace Clustering

Jakob Rødsgaard Jørgensen
Department of Computer Science
Aarhus University, Denmark
jakobrj@cs.au.dk

Katrine Scheel
Department of Computer Science
Aarhus University, Denmark
scheel@cs.au.dk

Ira Assent
Department of Computer Science
DIGIT Aarhus University Centre for Digitalisation, Big Data and Data Analytics
Aarhus University, Denmark
ira@cs.au.dk

ABSTRACT

Subspace clustering is the task of grouping objects based on mutual similarity in subspaces of the full-dimensional space. The INSCY algorithm extends the well-known density-based clustering algorithm DBSCAN. It finds dimensionality-unbiased non-redundant subspace clusters using a tree structure to speed up the processing of subspaces. Still, finding density-based clusters in all subspaces implies an exponential search space in the number of dimensions. Thus, the running time of INSCY is still measured in hours on even small datasets of 2000 points. For larger datasets, it becomes prohibitively expensive.

To benefit from INSCY for real-world sized datasets, we propose a novel GPU-parallel approach that runs on standard graphics cards. To utilize the many cores of the GPU, we need new algorithmic strategies that fit the computational model of the GPU. While the GPU provides a large number of threads, traditional algorithms incur diverging threads and poor memory alignment, both of which lead to idle time and poor runtime performance. In INSCY, extracting subspace regions from the SCY-tree structure and the density-based clustering of regions itself are thus unfit for the GPU.

Our novel GPU-friendly algorithm GPU-INSCY computes the same subspace clustering as INSCY at dramatically reduced runtimes. To achieve this, we devise a restructured SCY-tree index-structure and associated operations for the GPU, as well as a GPU-parallel density-based subspace clustering.

We experimentally show that GPU-INSCY scales well with the size of the dataset and the number of dimensions, and improves the running time of INSCY by a factor of several thousand for large datasets of high dimensionality.

1 INTRODUCTION

Clustering, i.e., grouping data points based on mutual similarity, is a widely used data mining task, e.g., for grouping customers to allow for targeted marketing. However, real-world data is often high-dimensional, and a higher number of dimensions means that there are more possibilities for points to seem dissimilar. This is known as the curse of dimensionality. Due to this effect, points tend to group within a subspace of the full-dimensional space, leading to the task of subspace clustering [2, 4, 15], where we search for clusters with all possible subspaces. To search for such clusters, we often employ density-based clustering similar to DBSCAN [12]. Most subspace clustering algorithms, e.g., SUBCLU [15], use a fixed density threshold independent of the subspace's dimensionality. When finding clusters, the density threshold needs to match the expected density such that we can find all points within clusters, but without including everything. However, the expected density is lower for higher-dimensional subspaces than it is for lower-dimensional subspaces. For density-based subspace clustering, this problem implies that density-measures that do not take the subspace's dimensionality into account are biased toward lower subspaces. To address this problem, Assent et al. [6] formulated a dimensionality-unbiased density-measure and utilized this in the algorithm INSCY [8]. INSCY, furthermore, removes redundancy and provides an index-structure called SCY-tree used to partition and prune regions of density-connected data points. A drawback that remains, is that the running time is still measured in hours on even small datasets of a couple of thousands of points.

To reduce the runtime of dimensionality-unbiased density-based subspace clustering, we exploit modern graphics cards (GPUs), capable of general-purpose computations, fast context switches, and parallelizing over many cores, but with a restrictive computational model and limited memory. The high computational throughput of GPUs has been utilized to improve clustering runtimes [1, 5, 10]. However, to our knowledge, there exists no GPU-parallelization of a dimensionality-unbiased index-supported algorithm like INSCY, which is challenging to GPU-parallelize due to index and depth-first subspace search being optimized for (sequential) CPU processing.

Contributions. In this work, we present a novel GPU-parallel algorithm, called GPU-INSCY, which provides the same clusterings as INSCY at substantially reduced runtimes. To achieve this, we restructure several major parts of INSCY, the index-structure SCY-tree, the operations used to partition regions of data, and the clustering of points. INSCY partitions regions represented by SCY-trees through a sequence of operations. We show how to make these operations parallel and combine several partitions into one process. Combining these allows us to avoid many redundant iterations and temporary copies. The clustering step is also GPU-parallelized and improved further by utilizing the density monotonicity for neighborhoods in increasing subspaces.

This paper is organized as follows: Section 2 discusses related work, Section 3 gives the background of subspace clustering and INSCY, Section 4 describes our new parallel algorithm GPU-INSCY, Section 5 presents the experimental comparison of INSCY and GPU-INSCY, and Section 6 concludes our work.

2 RELATED WORK

Subspace clustering is the task of grouping points based on mutual similarity in any possible subspace of the full-dimensional space, hence its worst case complexity is exponential in the number of dimensions.

Algorithms for subspace clustering [2, 4, 6–8, 11, 14, 15, 25] are often categorized into bottom-up or top-down approaches [16, 21, 24, 26]. Bottom-up approaches start with clustering in 1-dimensional subspaces, iteratively combining k-dimensional subspace clusters into $(k+1)$-dimensional subspace clusters. CLIQUE [4] and MAFIA [14] are grid-based approaches that may miss subspace clusters spanning across grid cells. Instead of clustering dense cells, SUBCLU [15] clusters dense points, as in the density-based full space clustering algorithm DBSCAN[12]. An issue with SUBCLU and other density-based subspace clustering approaches is that they use a fixed density-threshold for all subspaces. Therefore, they do not take dimensionality into account and are biased towards lower-dimensional subspace clusters. INSCY is an extension of SUBCLU that mitigates this problem by introducing a density measure normalized by a subspace's expected density.

Top-down approaches start by clustering the full-dimensional space and iteratively refine the subset of dimensions associated with each subspace cluster [2, 3, 29]. These approaches limit subspace clusters by assigning each point in the data to exactly one subspace cluster. Due to the exponential search for subspaces, many of the algorithms take an approximate approach to subspace clustering [2, 14, 20]. They do so using a heuristic to pick the subspaces that are examined or only compute clusterings of dense regions instead of single dense points. These approaches might miss clusters that exact algorithms like INSCY capture.

Even though exact subspace clustering algorithms are time consuming, few algorithms have been proposed to reduce the running time by exploiting the high computational throughput of the GPU. Utilizing the many cores of the GPU is highly challenging because of the distinct and limited computational model, as well as limited memory. There have been proposed several GPU-parallelized full-space clustering algorithms [5, 10, 13, 17, 19]. One of the earliest GPU versions of the full-space clustering algorithm DBSCAN was CUDA-DClust* [10], which starts multiple searches for clusters in parallel. If multiple searches start within the same cluster, they are merged. Multiple other GPU-versions of DBSCAN have been developed [5, 18, 19, 28]. Our assessment of self-reported results suggest that G-DBSCAN[5] and CUDA-DClust*[10] are the best performing options. An experimental evaluation [22] studies three of these GPU-versions and finds that G-DBSCAN is the fastest and CUDA-DClust* uses less memory.

Only one GPU-parallelization of a well-known subspace clustering approach has been proposed [1] for grid-based MAFIA. GPUMAFIA parallelizes one operation at a time, mapping nested for-loops of minor computations directly to parallel threads. Our restructuring of INSCY lets us GPU-parallelize GPU-INSCY even further such that we can even parallelize operations performed at different points of the process. We completely restructure the algorithm and its underlying SCY-tree structure to fit the computational model and the memory structure of the GPU.

To the best of our knowledge, we are the first to develop a GPU-parallelized version of a density-based subspace clustering algorithm, in particular an algorithm that supports dimensionality-unbiased density measures and exploits indexing structures for efficient computation.

3 BACKGROUND

3.1 The graphics processing unit

We give a short introduction to graphics processing units (GPUs) and their computational model. When using a GPU for general-purpose computation, the GPU is *co-processor*, and the CPU is *main processor*. Throughout the paper, we use the term *parallel* to denote parallel execution under the GPU's computational model. The main difference between a multi-core CPU and a GPU is that GPUs can perform fast context switches and that several cores on the GPU uses the same program counter and, therefore, must perform the same operations.

CUDA is NVIDIA's framework for using their line of GPUs. It uses the concept of a kernel, which is a function executed on multiple threads in parallel. Threads are organized into blocks, and all threads within a single block are capable of synchronizing, share fast accessible memory, and use atomic operations. However, there is a physical limit to the number of threads a block can contain, and the communication between threads comes at a time-cost. Each block is further separated into warps. All threads within a warp share a program counter, implying that they must perform the same instructions (SIMD) at all times. In the case of branch-diversion, threads in different branches will remain idle until the other branch has finished.

When parallelizing operations on the GPU, we are not guaranteed any order of executions. Therefore, our goal is to identify *independent* operations, i.e., operations that do not use the partial result of each other and therefore can be run in any order without changing the final result. All allocation of memory and calls to kernels are done by the CPU and executed on the GPU. All communication with the GPU comes with a time-cost due to the large latency of data transfer. Therefore, it is essential to balance where data is processed and how long it takes to transfer.

3.2 INSCY

We describe INSCY briefly. For further details please see [8]. We use the following terminology: let $X \in \mathbb{R}^{n \times d}$ be a d-dimensional dataset with n points, $D = \{0, \ldots, d-1\}$ an index set for the full dimensional space, $S \subseteq D$ a subspace of D, and $N_\varepsilon^S(p)$ the neighborhood with radius ε of a point p in subspace S.

According to INSCY [6], a subspace cluster is a maximal set of points of at least min_C, which are density-connected in a subspace according to some density measure, and which is not redundant w.r.t. a higher dimensional subspace projection:

Definition 3.1. **INSCY Subspace Cluster**
A set of points $C \subseteq X$ in subspace $S \subseteq D$ is a subspace cluster if:

- objects in C are **S-connected**: $\forall p, q \in C : \exists o_1, \ldots, o_m \in C : p = o_1 \wedge q = o_m \wedge \forall i \in \{2, \ldots, m\} : o_i \in N_\varepsilon^S(o_{i-1})$
- all points fulfill the **density criterion**: $\forall p \in C : dc^S(p)$,
- C is **maximal**, i.e., contains all S-connected objects: $\forall p, q \in X : p, q$ S-connected $\Rightarrow p \in C \wedge q \in C$,
- **minimum cluster size**: $|C| \geq min_C$,
- **not redundant**: $\nexists C', S'$ subspace cluster with $C' \subseteq C \wedge S \subset S' \wedge |C'| \geq r \times |C|$

where r is the redundancy parameter, min_C is the minimum size of a cluster, and $dc^S(p)$ is any dimensionality-unbiased density criterion within subspace S.

In this paper, we use the dimensionality-unbiased rectangular density measure for the density criterion $dc^S(p) := |N_\varepsilon^S(p)| \geq \max(F \cdot \alpha(S), \mu)$, where F is the density factor threshold, $\alpha(S) = \mathbb{E}_S\left[|N_\varepsilon^S(p)|\right] = |X| \frac{c(S) \times \varepsilon^{|S|}}{v_S}$ is the expected density, $c(S) = \pi^{\frac{|S|}{2}} / \Gamma\left(\frac{|S|}{2} + 1\right)$ with $\Gamma(n+1) = n \times \Gamma(n), \Gamma(1) = 1, \Gamma(1/2) = \sqrt{\pi}$, v_S is the volume of subspace S, and μ is the minimum number of points required for not just being pseudodense. Other density measures can also be used. For further details see [6]. Note that

Def. 3.1 is similar to density-based clustering in DBSCAN [12], but with an unbiased density notion wrt. subspaces.

SUBCLU [15] uses monotonicity of density-connectivity to prune points that lie outside clusters in a lower-dimensional subspace projection. However, for INSCY's unbiased density measure that scales with the expected density of a subspace, monotonicity is lost. Still, as [6] observes, pruning can be done by discarding points that are not dense w.r.t. the lowest possible density threshold, i.e., for the full-space. INSCY finds such points, called not weak-dense, which can safely be pruned before searching for clusters within superspaces of the current space. A point is weak-dense if $|N_\varepsilon^S(p)| \geq \max(F \times \alpha(D), \mu)$.

3.2.1 The INSCY algorithm. The idea of INSCY is to bound the search for subspace clusters by identifying regions that fully contain potential clusters. INSCY describes such a region by the dimensions it spans and the respective intervals in these dimensions, and call it a subspace region. INSCY performs a depth-first search (DFS) of the subspace regions, i.e., enumerating all possible subspace regions. INSCY does so by recursively extending with one dimension at a time and partitioning the region into intervals along that dimension. When INSCY returns from the recursion, it performs density-based clustering within the current subspace region to obtain the clusters. This implies that INSCY cluster points within all superspaces of the current space first.

Each dimension is partitioned into a fixed number of cells. As a cluster likely spans multiple cells, INSCY register this by having a border between each cell at the size of the neighborhood radius ε. When performing density-based clustering, it follows that if there are no points within this border, the two cells' points cannot be density-connected. Otherwise, a cluster may span both cells. Such connected cells are referred to as S-connected. S-connected cells must be merged into a density-connected interval to ensure that no clusters are split. An interval spanning multiple cells is identified by the first cell. A dimension might have multiple density-connected intervals, and INSCY is called recursively on each interval in a depth-first manner. The whole process of expanding with a new dimension and bounding to a density-connected interval is referred to as restricting w.r.t. a new dimension and the cell identifying the interval. The pair of dimension d and cell c is called a descriptor (d, c). When expanding with a new dimension, we expand one region at a time. Figure 1 shows a 1-dimensional example, and the expansion into two dimensions. On the left, the dimension is split into three cells, where two are S-connected and merged into one interval marked by green. On the right, we see the expansion. The red region is split into cells along the added dimension and connected with any S-connected cells, and likewise for the green region.

To keep track of the possible dimensions and cells that can be restricted, INSCY introduces an index-structure called SCY-tree. The idea of SCY-tree is to precompute the number of points within cells along a dimension such that restricting becomes easier. The

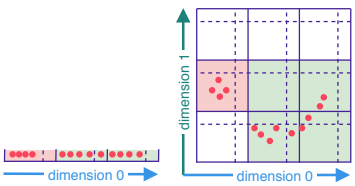


Figure 1: Expansion of 1-d regions into 2-d

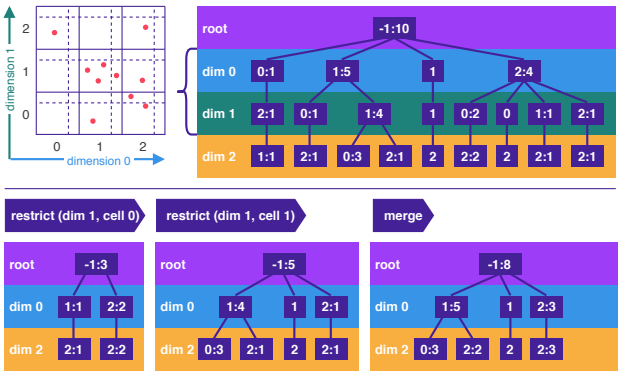


Figure 2: SCY-tree for examples in [8]; node values *cell* : *count*; **dimensions and points colored as in later figures**

SCY-tree, therefore, represents the dimensions and cells not yet restricted. The SCY-tree is a tree-structure containing nodes that represent a partition of a space along a specific dimension. All nodes regarding a specific dimension are located at the same height in the SCY-tree, which we call a layer. The children of a node represent splits into cells along a dimension, one child per cell. Each node contains its cell number and the count of points within the cell it represents. A cell with an S-connection is represented by adding a sibling with the same cell number, but with the count of points set to -1. Such a node is called an S-connector node. INSCY keeps track of S-connections by continuing the path of S-connector nodes down to the leaf layer. The root node of the SCY-tree represents a restricted subspace region. SCY-trees that represent regions that share a border are called neighboring SCY-trees. For further details, see [8].

Figure 2 (top) shows an example of an initial SCY-tree for the full-dimensional space. In this example, the space is first partitioned along dimension 0, creating three cells noted by the cell number and the count of points in that cell *cell* : *count*. Cell 1 has an S-connection, which is represented by a node without a count of points. Each cell is then further partitioned along dimension 1, discarding cells that do not contain any points.

INSCY proceeds as in Algorithm 1. For each descriptor, create a restricted SCY-tree. If cells in the SCY-tree are S-connected, merge connected restricted SCY-trees into one final restricted SCY-tree. INSCY prune the final restricted SCY-tree for redundancy, call recursively, and cluster the points if there is a possibility for non-redundant clusters.

Restrict. INSCY restricts a SCY-tree by identifying nodes matching the current descriptor, i.e., the nodes residing on the layer of the restricted dimension and with the same cell number as the descriptor. For each matching node, copy the node's path to the root and subtrees below the node into a new restricted

Algorithm 1 INSCY($scytree$, d_f, X, d, r, F, μ, ε, min_C, R)

```
1: for d_re = d_f to d do
2:   for c_re = 0 to n_cells do
3:     scytree' ← restrict(scytree, d_re, c_re)
4:     scytree' ← mergeNeighbors(scytree, scytree', d_re, c_re)
5:     if prune_recursion(scytree', F, μ, ε, min_C) then
6:       INSCY(scytree', d_re + 1, X, d, r, F, μ, ε, min_C, R)
7:       if prune_redundancy(scytree', r, R) then
8:         R ← R ∪ clustering(scytree', X, F, μ, ε)
```

SCY-tree. Since the SCY-tree keeps track of not yet restricted dimensions, the matching node itself is not copied. The node's children are now children of the node's parent. The count of points is also updated to reflect the number of points in the restricted region. Figure 2 (bottom) contains two restricted SCY-trees for descriptors $(1, 0)$ and $(1, 1)$ and the merged result. For descriptor $(1, 0)$ only 2 nodes match, leading to a small SCY-tree.

Merge. INSCY merges neighboring restricted SCY-trees if there exists an S-connection, i.e. when an S-connector path starts at dimension d and has cell number c that matches the current descriptor (d, c). Merge is done by going through the two restricted SCY-trees and copying the nodes in both. A node can be represented in several SCY-trees. During the merge, nodes with the same cell number and the same parent are merged. Figure 2 (bottom), shows that the descriptor $(1, 0)$ matches an S-connector node, the node represented by only a 0 on dimension 1, and therefore INSCY restricts the neighboring descriptor $(1, 1)$ and merges the two restricted SCY-trees.

Pruning recursion. To reduce the search space, INSCY prunes the final restricted SCY-tree before calling recursively, as follows: Remove non-weak dense points and check if the region's number of points still exceeds min_C. INSCY only proceeds with the recursion if this is the case, as further restrictions will only reduce the number of points.

Pruning redundancy. When returning from the recursive call INSCY has found clusters in all superspaces of the current subspace. The current region can therefore be pruned by redundancy. INSCY prunes by redundancy by checking if the result already contains a cluster covering a factor r of the points in the restricted region. If the number of points in the region is large enough, INSCY computes the density-based clustering on all points in the final restricted SCY-tree and adds all non-redundant clusters to the result.

4 GPU-INSCY ALGORITHM

INSCY is inherently computationally expensive, making it infeasible to run on large real-world datasets. As mentioned in the introduction, GPUs provide computational power that algorithms designed for a different computational model of single-core CPUs, as INSCY, cannot utilize. We design an algorithm for the GPU that reduces the running time of INSCY substantially, making it feasible to run on much larger datasets. To summarize the notation found in this section we provide Table 1 for ease of reading. Recall that threads in a warp must execute the same instructions to fully utilize the GPU's computational power. INSCY does not group similar operations and would perform poorly on the GPU.

The idea of each iteration in INSCY is to bound a subspace region by restricting and merging, prune that region, and perform clustering in that region. This process is repeated until all clusters in all subspace regions are found. This approach is efficient for a sequential algorithm. However, when parallelizing for the GPU, we prefer grouping identical and independent operations to make each kernel call utilize as many cores as possible. Making INSCY run parallel on the GPU is not straightforward since many partial computations depend on previous results. E.g., in the alternation between restricting and merging SCY-trees, we need the previous merged SCY-tree and the neighboring restricted SCY-tree before continuing to merge.

Table 1: Notation

n_{nodes}	number of nodes
n_{pts}	number of points
n_{cells}	number of cells
n_{dims}	number of dimensions
n_{r_dims}	number of restricted dims
$pa \in \mathbb{N}^{n_{nodes}}$	parent array
$ce \in \mathbb{N}^{n_{nodes}}$	cell array
$co \in \mathbb{N}^{n_{nodes}}$	count array
$la \in \mathbb{N}^{n_{dims}}$	layer-indexing array
$dims \in \mathbb{N}^{n_{dims}}$	dimension array
$r\_dims \in \mathbb{N}^{n_{r\_dims}}$	restricted dims array
$po \in \mathbb{N}^{n_{pts}}$	point-id array
$pl \in \mathbb{N}^{n_{pts}}$	point-placement array
$incl \in \{\mathbf{0}, \mathbf{1}\}^{n_{dims} \times n_{cells} \times n_{nodes}}$	node inclusion array
$incl_{pts} \in \{\mathbf{0}, \mathbf{1}\}^{n_{dims} \times n_{cells} \times n_{pts}}$	point inclusion array
$idx \in \mathbb{N}^{n_{dims} \times n_{cells} \times n_{nodes}}$	node new-index array
$idx_{pts} \in \mathbb{N}^{n_{dims} \times n_{cells} \times n_{pts}}$	point new-index array
$n\_co \in \mathbb{N}^{n_{dims} \times n_{cells} \times n_{nodes}}$	new-count array
$is\_S(i)$	is S-connection
$s\_incl(j, i, c)$	should be included
$S \in \{\mathbf{0}, \mathbf{1}\}^{n_{dims} \times n_{cells}}$	S-connection array
$M \in \mathbb{N}^{n_{dims} \times n_{cells}}$	merge map
$n\_pa \in \mathbb{N}^{n_{dims} \times n_{cells} \times n_{nodes}}$	new-parent array
$n\_ch \in \mathbb{N}^{n_{dims} \times n_{cells} \times n_{nodes} \times n_{cells} \times 2}$	new-children array
$rep(j, c, i)$	representative node

Algorithm 2 GPU-INSCY($scytree'$, d_f, X, d, r, F, μ, ε, min_C, R)

```
1:  L ← GPU_restrict_and_merge(scytree, d_f, d)
2:  precompute_neighborhoods(X, L, ε)
3:  for d_re ← d_f to d − d_f do
4:      C ← 1d array of size |X| initialized to −1
5:      for ∀scytree' ∈ L[d_re] do
6:          if prune_recursion(scytree', F, μ, ε, min_C) then
7:              GPU-INSCY(scytree', d_re + 1, X, d, r, F, μ, ε, min_C, R)
8:              if prune_redundancy(scytree', r, R) then
9:                  L' ← L' ∪ {(scytree', C)}
10: R ← R ∪ GPU_clustering(L', X, F, μ, ε)
```

In this section, we present a new algorithm called GPU-INSCY, in which we tackle the problem of identifying and reorganizing the operations that can be performed in parallel to reduce running time. Contrary to INSCY, GPU-INSCY aims to perform similar and independent operations simultaneously for multiple final restricted SCY-trees to utilize multiple thread blocks. Remember that this allows us to use more cores, but it is only possible if the threads in different blocks do not need to communicate.

We first outline the general order of computations in GPU-INSCY, and we later explain this reordering. These reorderings do not affect the result since the reordered operations are independent of each other as discussed below for each change we introduce. GPU-INSCY can be seen in Algorithm 2. First, compute the set L of all final restricted SCY-trees. Precompute the neighborhoods for all points in all final restricted SCY-trees. For each final restricted SCY-tree, prune the recursion, call GPU-INSCY recursively, and prune for redundancy. All non-pruned final restricted SCY-trees are added to L'. Finally, we cluster all points in each of the final restricted SCY-trees in L'.

Restrict and merge. In GPU-INSCY, we isolate all restrict and merge operations at the beginning of the algorithm, whereas INSCY performs them ad hoc. We isolate the operations such that we can parallelize them in different thread blocks. The result of

each restrict and merge operation only depends on the information parsed to the recursion. Computing all restricted SCY-trees at the beginning does, therefore, not change the final result. Parallelizing within each thread block is not a simple task due to both the alternation between restrict and merge and the fact that INSCY only visits nodes in the SCY-trees one by one when restricting and merging. We discuss how to parallelize restrict and merge in Section 4.1.2, after introducing a representation of the SCY-tree index-structure for the GPU in Section 4.1.1.

Precomputing the neighborhoods. Computing the neighborhoods is an expensive task, and it is used both for the clustering and when computing weak-density while pruning a recursion. In Section 4.2, we describe how to precompute the neighborhoods in parallel and how we take advantage of having direct access to the neighborhoods in a subspace of the current space.

Pruning. In Section 4.3, we parallelize both pruning phases following the same approach as for restrict and merge.

Clustering. In Section 4.2, we change the sequential way of expanding the clusters [12] with one density-connected point at a time, to obtain a more efficient clustering algorithm.

4.1 SCY-tree on the GPU

The SCY-tree representation and the associated operations are not very suited for the GPU. Section 4.1.1 describes how to represent the SCY-tree in a GPU friendly fashion and Section 4.1.2 describes how to perform the restrict and merge operations in parallel.

4.1.1 Representing the SCY-tree on the GPU. Handling memory on the GPU is more restrictive than on the CPU, and allocating memory can only be done from the CPU. Furthermore, it is expensive to alternate between calling kernels, transferring data, and allocating memory. Therefore, we prefer to allocate memory and transfer data as few times as possible. GPU memory is loaded one block at a time to reduce latency, implying that data used close together in time should be placed close together in memory. If the data we use is not placed in the same block, we get cache misses, i.e., not using the loaded data, which we would like to reduce. For ease of reference, we call the GPU friendly representation of the SCY-tree GPU-SCY-tree. A way to represent tree structures on the CPU is to create an object for each node with pointers to its children, parent, and other values in the tree. This structure is very flexible and allows adding nodes on the fly. However, this does not fit well with the restrictions on the GPU.

Remember, all nodes for a particular dimension are placed on the same layer in the SCY-tree. These layers are indexed by j starting with $j = -1$ for the root and incrementing toward the leaf layer $j = n_{dims} - 1$, implying that lower indices are above the higher indices in the SCY-tree. In Section 4.1.2, we describe how we handle all nodes on the same layer simultaneously, and we would therefore like to place these nodes close together in memory. The same is the case for points contained in the tree.

Instead of representing nodes as objects, we choose to represent the GPU-SCY-tree as arrays, with an entry for each node. Each array represents the kind of pointer or values that a node contains. In the arrays, we locate nodes on the same layer in the SCY-tree next to each other and order the layers by their index j. In this way, data for nodes on the same layer is placed close together in memory, making it more likely to avoid cache-misses. We organize points using the same reasoning. To represent the GPU-SCY-tree, we use a total of eight arrays with one entry per node, point, or dimension. An example is given in Figure 3. Besides the arrays we also keep count of the number of nodes

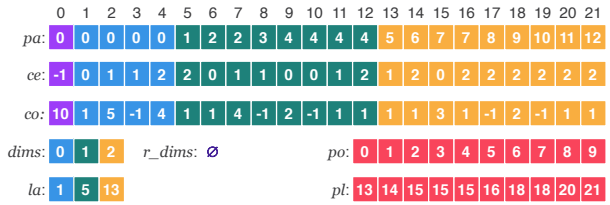


Figure 3: GPU-SCY-tree for SCY-tree in Figure 2.

n_{nodes}, number of points n_{pts}, number of cells n_{cells}, number of dimensions in the SCY-tree n_{dims}, and number of restricted dimensions n_{r_dims}.

The nodes are represented using three arrays: the parent pointer $pa \in \mathbb{N}^{n_{nodes}}$, the cell number $ce \in \mathbb{N}^{n_{nodes}}$, and the count of points $co \in \mathbb{N}^{n_{nodes}}$. Notice that we do not keep pointers to children, see Section 4.1.2 for reasoning. To access each layer j we have an array with the starting index of each layer $la \in \mathbb{N}^{n_{dims}}$ and an array with the dimensions that the layers represent $dims \in \mathbb{N}^{n_{dims}}$. We furthermore keep an array of the restricted dimensions $r\_dims \in \mathbb{N}^{n_{r\_dims}}$, however, for the GPU-SCY-tree in Figure 3 this is empty. To keep track of the points in the GPU-SCY-tree, we have two arrays with an entry for each point. One keeps track of the points' index in the dataset $po \in \mathbb{N}^{n_{pts}}$, and the other keeps track of which leaf-node each point is placed in $pl \in \mathbb{N}^{n_{pts}}$.

4.1.2 Restrict and merge on the GPU. When parallelizing for the GPU, we identify: (i) ways to reorder independent tasks that can be performed in parallel, (ii) similar tasks that can be performed by a warp, and (iii) ways to allocate memory as few times as possible. Restrict and merge for a SCY-tree are sequential operations where we look at one node at a time, check if it should be included, and copy all information to the temporary or final result. Running this in parallel on the GPU requires a substantial restructuring due to two things: The alternation between restrict and merge and a node's inclusion being dependent on the inclusion of either the parent or one of its children. As mentioned before, such a dependency makes the process sequential, which is not suitable for the GPU.

In Section 4, we state that all final restricted SCY-trees can be computed first in the recursion since the computation only requires the descriptors and the SCY-tree parsed to the recursion. But to parallelize the restrict and merge operation, we need several observations and restructuring that we now provide.

Allocating once. To allocate memory only once per restricted GPU-SCY-tree, we first compute which nodes and points are included in the restricted SCY-trees. This information is kept in two temporary binary arrays both initialized to **0**. One for nodes $incl \in \{\mathbf{0}, \mathbf{1}\}^{n_{dims} \times n_{cells} \times n_{nodes}}$ with entries for each descriptor and node combination. And one for points $incl_{pts} \in \{\mathbf{0}, \mathbf{1}\}^{n_{dims} \times n_{cells} \times n_{pts}}$ with entries for each descriptor and point combination. Here **0** and **1** represent false and true, respectively. In Figure 2, we show the restriction for descriptor (1,0). In Figure 4 we show the same restriction in GPU-SCY-tree representation, and the temporary arrays. Here the five included nodes are marked with a **1** in $incl$. Knowing which nodes and points are included allows us to compute the new indices of the nodes and points in the restricted SCY-trees. We compute the indices fornodes and points using inclusive scan (cumulative sum) of $incl$ for each descriptor. The result is kept in $idx \in \mathbb{N}^{n_{dims} \times n_{cells} \times n_{nodes}}$

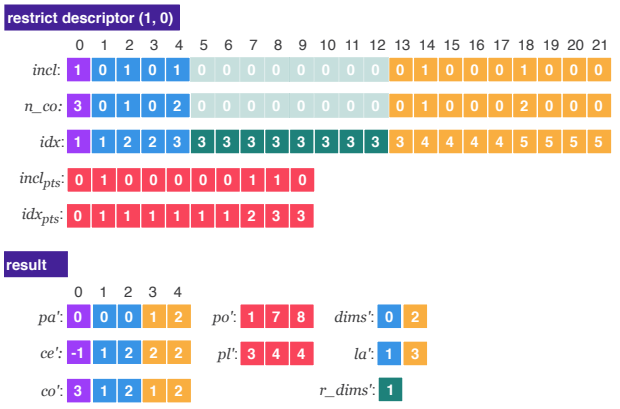


Figure 4: Restrict example before combining with merge.

and $idx_{pts} \in \mathbb{N}^{n_{dims} \times n_{cells} \times n_{pts}}$. This is used to determine where each node is placed in the resulting SCY-tree. E.g. in Figure 4, the last included node is placed at entry $4 = idx(18) - 1$. Furthermore, for each descriptor, we use the last index to allocate the needed memory for the restricted SCY-trees.In Figure 4 we need to allocate space for $5 = idx(|idx| - 1)$ nodes. After allocating memory, we copy all included nodes and points to the restricted SCY-trees. To copy, we need the new count of points in the subtrees starting at each node $n\_co \in \mathbb{N}^{n_{dims} \times n_{cells} \times n_{nodes}}$ which we compute along side the inclusion of each node.

Restrict is independent. We observe that the restrict operation only requires the SCY-tree parsed to the recursion and the descriptor it is restricting w.r.t.. Both the descriptor and the SCY-tree are not changed during the recursion. Therefore, the restrict operations of each recursion are completely independent of each other and all other operations. Consequently, the final result does not depend on the order of restriction, and we can parallelize the restrict operations over different thread blocks, which allows us to utilize more cores.

Restrict - similar tasks and restructuring. INSCY restricts by identifying all nodes matching a descriptor and then visiting upward and downward in the layers of the SCY-tree from there. INSCY copies all nodes on the path to the root and the subtree below the matching nodes to the new restricted SCY-tree. We take advantage of the SCY-tree being a well-balanced tree with a layer for each dimension. Observe that nodes on layers above the restricted dimension are included if any of its children is included in the restricted SCY-tree. The nodes on layers below are included if their parent is included. Because of the dependency w.r.t. inclusion between parents and children, we have a dependency between layers where we need to compute the inclusion of nodes up- and downwards in the GPU-SCY-tree starting from the restricted dimension. However, observe that computing the inclusion of each node on a layer is independent of the other nodes on that layer. Using this observation, we suggest computing the inclusion of nodes one layer at a time, making the computation of node inclusion parallel over each node on a layer. Since we keep the ordering between parents and children, we do not violate the dependency, and hence we compute the same result as INSCY.

When computing the inclusion of nodes, we have four cases, where the computation is different for each of them. One for nodes directly above the restricted dimension, one for the nodes on the remaining layers above, one for nodes directly below the restricted dimension, and one for the nodes on the remaining layers below. We handle each of the cases in their own kernel, to avoid branch-divergence that would lead to idle threads.

We compute the inclusion array *incl* in parallel with thread blocks for each descriptor $(dims(j), c)$ where j is the layer representing the restricted dimension and c is the cell number.Within each block, we process sequentially over each layer $j + k$ where $-j \leq k < n_{dims} - j$, starting from $k = 0$ and incrementing/decrementing from there. For all nodes i on a given layer we parallelize using threads.

When we compute the inclusion array *incl*, we treat normal nodes and S-connector nodes slightly differently. An S-connection is only used to enforce a merge along the restricted dimension. Therefore, we discard the S-connector path starting at the restricted dimension. Remember, we have an S-connection on the restricted dimension, when an S-connector node i has a normal node as the parent:

$$is\_S(i) := (co(i) < 0) \wedge (co(pa(i)) \geq 0). \tag{1}$$

In Figure 3, node 10 represents an S-connection since it has a negative count and its parent, node 4, has a positive count.

We can now use this when searching for nodes i matching the descriptor $(dims(j), c)$. A node i on layer j matches the descriptor $(dims(j), c)$ if its cell number matches the cell number of the descriptor $ce(i) = c$ and it is not an S-connector node starting at the restricted dimension $\neg is\_S(i)$:

$$s\_incl(j, i, c) := (ce(i) = c) \wedge (\neg is\_S(i)). \tag{2}$$

In Figure 3, for descriptor (1, 0), node 6 should be treated as a match since it is in dimension 1 and has cell number 0 and does not represent an S-connection. Node 10 also matches the descriptor, but it represents an S-connection, so it should not be treated as a match.

We wish to compute inclusion for all nodes on the layers above the restricted dimension. This requires us to look at each child of a given node. As the number of children can vary from node to node, threads in the same warp would stay idle until the other threads have visited all their children. We address this by parallelizing over all children instead and letting the children mark if their parent is included. Observe that now each thread only visits the current node and its parent, instead of a varying number of children.

Starting from layer j we compute inclusion for the nodes on layer $j-1$ just above the restricted dimension $dims(j)$. The parent $pa(i)$ of a node i is marked as included if the node i matches the descriptor $s\_incl(j, i, c)$:

$$\begin{aligned} &\forall 0 \leq j < n_{dims}, 0 \leq c < n_{cells}, \\ &\quad la(j) \leq i < la(j+1), s\_incl(j, i, c) : \\ &\qquad incl(j, c, pa(i)) := \mathbf{1}. \end{aligned} \tag{3}$$

In Figure 4 node 2 is included since node 6 matches the descriptor.

Sequentially moving towards the root, we can now compute inclusion for nodes on layer $j - k$ where $2 \leq k < j$. The parent $pa(i)$ is now included if the node i is marked as included:

$$\begin{aligned} &\forall 0 \leq j < n_{dims}, 0 \leq c < n_{cells}, 1 \leq k < j - 1, \\ &\quad la(j-k) \leq i < la(j-k+1), incl(j, c, i) : \\ &\qquad incl(j, c, pa(i)) := \mathbf{1}. \end{aligned} \tag{4}$$

In Figure 4 the root, node 0, is included since node 2 is included.

Similarly, we include nodes on the layer $j+1$ directly below the restricted dimension $dims(j)$ if the parent matches the descriptor:

$$\forall 0 \le j < n_{dims}, 0 \le c < n_{cells}, la(j+1) \le i < la(j+2): \\ incl(j,c,i) := s\_incl(j, pa(i), c). \tag{5}$$

In Figure 4 node 14 is included as node 6 matches the descriptor.

Moving towards the leaves, we compute inclusion for nodes on layer $j+k$ where $2 \le k < n_{dims} - j$ by checking if a node's parent is marked as included:

$$\forall 0 \le j < n_{dims}, 0 \le c < n_{cells}, 2 \le k < n_{dims} - j, \\ la(j+k) \le i < la(j+k+1): \\ incl(j,c,i) := incl(j,c,pa(i)). \tag{6}$$

After we have computed the inclusion of the nodes on the leaf layer, we can compute which points are included. A point p is included if the leaf node where it is located $pl(p)$ is included:

$$\forall 0 \le j < n_{dims}, 0 \le c < n_{cells}, 0 \le p < n_{pts}: \\ incl_{pts}(j,c,p) := \begin{cases} incl(j,c,pl(p)) & \textbf{if } j < n_{dims} - 1 \\ ce(pl(p)) = c & \textbf{else} \end{cases} \tag{7}$$

E.g. in Figure 4 point 1 is included since the leaf node 14 where the point is placed is included. The computation is done in parallel over each descriptor as blocks and each point p as threads. We handle the case of restricting the leaf layer by directly checking if the placement node's cell number matches the descriptor.

Restrict and merge combined. INSCY alternates between restricting and merging as long as S-connections are found. The merge operation only merges restricted SCY-trees that represent subspace regions within the same subspace. Therefore, merges are independent between restricted dimensions in the same recursion. However, remember that the merge operation merges the newly restricted SCY-tree with the previous merged SCY-tree. Instead of this sequential process, we devise a strategy to perform merges and restrictions simultaneously. Implying that we avoid allocating space for the temporary restricted and merged SCY-trees, and by that, save time.

Precomputing SCY-trees to merge. Observe that in INSCY, what makes the merge process sequential, is that we do not know in advance which SCY-trees need to be merged for a given descriptor. However, this only depends on the S-connections along the restricted dimension. A merge is only necessary if there is an S-connection between two cells on the restricted dimension. We suggest precomputing which SCY-trees need to be merged for each descriptor in advance. First, check if there is an S-connection for the given descriptor, then compute from which descriptor the merging process should start. The first check for S-connections can be parallelized as follows. We define $S \in \{\mathbf{0}, \mathbf{1}\}^{n_{dims} \times n_{cells}}$, a table of whether there exists an S-connection for a given descriptor. Each entry of S is initialized to $\mathbf{0}$ and updated in parallel over each layer j as thread blocks and each node i as threads. The update entails writing $\mathbf{1}$ if the node i is the start of an S-connector path.

$$\forall 0 \le j < n_{dims}, la(j) \le i < la(j+1), is\_S(i): \\ S(j, cells(i)) := \mathbf{1}. \tag{8}$$

We use S to compute from which descriptor each merge sequence starts. This information is saved in $M \in \mathbb{N}^{n_{dims} \times n_{cells}}$, where each entry represents a descriptor. For each entry, we compute which restricted SCY-trees should be merged, denoted by the cell number c of the descriptor associated with the first SCY-tree in that merge sequence. Remember that we start a new sequence

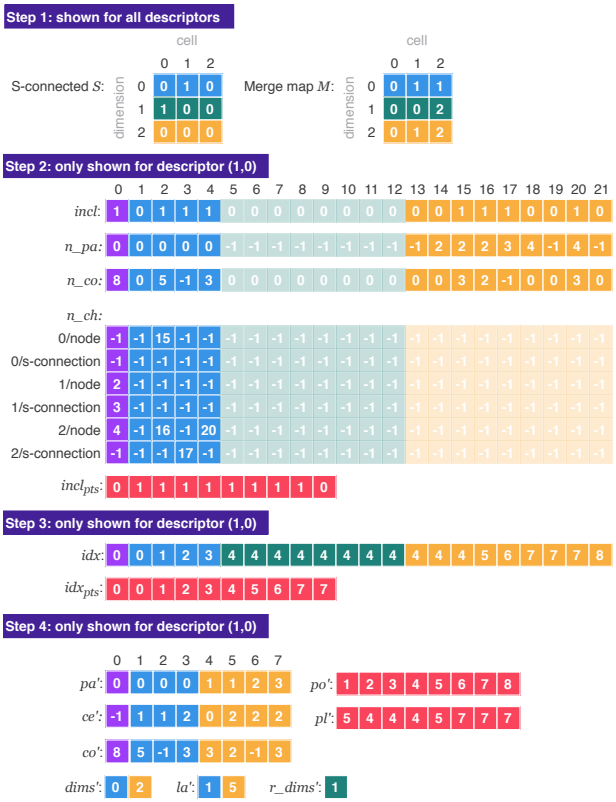


Figure 5: Restrict after combining with merge.

of merges whenever there was no S-connection from the previous cell $S(j, c-1)$. In other words, if there is an S-connection between two cells, we continue the sequence with identifier $M(j, c-1)$. If not, we start a new sequence with the identifier c.

$$\forall 0 \le j < n_{dims}, 0 \le c < n_{cells}: \\ M(j,c) := \begin{cases} M(j,c-1) & \textbf{if } (c > 0) \wedge S(j,c-1) \\ c & \textbf{else} \end{cases} \tag{9}$$

Equation 9 is parallelized over layers j but remains sequential over cell numbers c since we need to know the preceding entry $M(j, c-1)$ to compute $M(j,c)$.

The table with S-connections S and merge map M for the GPU-SCY-tree in Figure 3 are shown in Figure 5. S contains an S-connection in dimension 1, starting at cell 0. Therefore, in M, a merge sequence starts at cell 0, continuing to cell 1.

Avoiding merge sequences. The merge map M allows us to avoid the merge sequence and instead directly include nodes that would be in the final restricted SCY-tree for a given descriptor. More concretely, when checking if a node i on the restricted dimensions $d = dims(j)$ matches the descriptor, we instead look up the restricted dimension and the current node's cell number in the merge map M. We treat node i as a match if $M(j, ce(i))$ matches the cell number c of the descriptor. This changes Equation 2 into:

$$s\_incl(j,i,c) := (M(j, ce(i)) = c) \wedge (\neg is\_S(i)), \tag{10}$$

and Equation 7 into

$$\forall 0 \le j < n_{dims}, 0 \le c < n_{cells}, p < n_{pts}: \\ incl_{pts}(j,c,p) := \begin{cases} incl(j,c,pl(p)) & \textbf{if } j < n_{dims} - 1 \\ M(j,c,ce(pl(p))) = c & \textbf{else} \end{cases} \tag{11}$$

E.g. for descriptor (1, 0), we now also treat node 7 in Figure 3 and 5 as a match, since cell 1 in dimension 1 has a merge sequence starting at cell number 0.

Since nodes on the restricted dimension are not included, nodes directly below that dimension will become their grandparents' children instead. This implies that the grandparent can end up with multiple children with the same cell number. Nodes with the same parent and cell number would have been merged in INSCY and must also be merged in GPU-INSCY to ensure that INSCY and GPU-INSCY still compute the same final restricted SCY-trees. However, INSCY merges these one by one and GPU-INSCY merges them all simultaneously. In Figure 5, nodes 14 and 16 will now both be children of node 2, and they have the same cell number, so they must be merged.

Merging nodes can propagate the problem of children, with the same cell number, down towards the leaves. We merge such nodes during our new restrict phase. We keep track of nodes that need to be merged in the restricted SCY-trees by computing two things: each node's new parent $n\_pa \in \mathbb{N}^{n_{dims} \times n_{cells} \times n_{nodes}}$ and the node's new children $n\_ch \in \mathbb{N}^{n_{dims} \times n_{cells} \times n_{nodes} \times n_{cells} \times 2}$. Examples of both arrays are shown in Figure 5. All entries of n_pa and n_ch are initialized to -1. For each descriptor, n_pa holds the new parents of all nodes. Likewise for n_ch, except that we make room for all possible children by $n_{cells} \times 2$. A node can have two types of children: normal or S-connector nodes. For both types, we can have a node for each cell. To look up the type of a node we use:

$$S\_idx(i) := \begin{cases} \mathbf{0} & \textbf{if } co(i) \geq 0 \\ \mathbf{1} & \textbf{else} \end{cases} \tag{12}$$

Merge representatives. When merging nodes in the SCY-tree, we pick one of the nodes to be the representative, which is the node that will actually be included in the final restricted SCY-tree. We will lookup the representative node $rep(j, c, i)$ by

$$rep(j, c, i) := n\_ch(j, c, n\_pa(j, c, i), ce(i), S\_idx(i)).$$

If a node should be represented in the final restricted SCY-tree we say that it is fused into that SCY-tree. We call it fused if it is either merged or included in the SCY-tree. If a node is merged into the SCY-tree, the count of points and children is added to the representative node. In Figure 5, nodes 14 and 16 should be fused, but only node 16 is included as the representative.

We assign a new parent to all nodes that are fused into the final restricted SCY-tree. This implies that iff n_pa has a value that is not -1, the associated node has been fused into the final restricted SCY-tree. Notice that we can use $n\_pa(j, c, i) \geq 0$ to check if the parent has been fused instead of just checking if it has been included $incl(j, c, i)$.

When identifying the new parent of a node i, below the restricted dimension, we look up which node the old parent has been merged into. This will be one of the children of the new grandparent of node i, which is identified as the representative node for the parent:

$$\begin{aligned} &\forall 0 \leq j < n_{dims}, 0 \leq c < n_{cells}, 2 \leq k < n_{dims} - j, \\ &\quad la(j+k) \leq i < la(j+k+1), n\_pa(j, c, pa(i)) \geq 0 : \\ &\qquad n\_pa(j, c, i) := rep(j, c, pa(i)). \end{aligned} \tag{13}$$

When computing the new parent for nodes just below the restricted dimension, we need to skip the nodes on the restricted dimension, since the restricted layer is removed from the result. However, for a node above the restricted dimension, there are no changes. Therefore, no merge of nodes can occur, and we do not need to check which child has been picked:

$$\begin{aligned} &\forall 0 \leq j < n_{dims}, 0 \leq c < n_{cells}, \\ &\quad la(j+1) \leq i < la(j+2), s\_incl(j, pa(i), c) : \\ &\qquad n\_pa(j, c, i) := pa(pa(i)). \end{aligned} \tag{14}$$

E.g., the parent of node 14 is node 6, and the parent of node 6 is node 2. Therefore, the new parent of node 14 is node 2.

For all nodes above the restricted dimension, we do not change the child-parent relationship, and they can be copied in parallel.

$$\begin{aligned} &\forall 0 \leq j < n_{dims}, 0 \leq c < n_{cells}, 1 \leq k < j, \\ &\quad la(j-k) \leq i < la(j-k+1), I(j, c, i) : \\ &\qquad n\_pa(j, c, i) := pa(i), \\ &\qquad n\_ch(j, c, pa(i), ce(i), S\_idx(i)) := i. \end{aligned} \tag{15}$$

Below the restricted dimension, we need to decide which of the merged nodes is the representative. It is not important which of the nodes is picked, but all threads involved in the merge must agree on just one node. We do this by letting each node i, that is fused, write its id as the representative, i.e., the new child:

$$\begin{aligned} &\forall 0 \leq j < n_{dims}, 0 \leq c < n_{cells}, 1 \leq k < n_{dims} - j, \\ &\quad la(j+k) \leq i < la(j+k+1), n\_pa(j, c, i) \geq 0 : \\ &\qquad rep(j, c, i) := i. \end{aligned} \tag{16}$$

We synchronize such that all threads see the same node id, and only include that node as the new child. E.g., in Figure 5 both node 14 and 16 would vote for themselves as the representative. In our example, node 16 was the last to write. Therefore, node 16 becomes the representative. This expands Equation 5 into:

$$\begin{aligned} &\forall 0 \leq j < n_{dims}, 0 \leq c < n_{cells}, la(j+1) \leq i < la(j+2) : \\ &\quad incl(j, c, i) := s\_incl(j, pa(i), c) \wedge (rep(j, c, i) = i), \end{aligned} \tag{17}$$

and Equation 6 into:

$$\begin{aligned} &\forall 0 \leq j < n_{dims}, 0 \leq c < n_{cells}, 2 \leq k < n_{dims} - j, \\ &\quad la(j+k) \leq i < la(j+k+1) : \\ &\qquad incl(j, c, i) := (n\_pa(j, c, i) \geq 0) \wedge (rep(j, c, i) = i). \end{aligned} \tag{18}$$

For a point, the placement can change since nodes are merged. Therefore, we check if the node where the point is placed is fused into the final restricted SCY-tree. This is the case if the node has been assigned a new parent. Equation 11 changes into:

$$\forall 0 \leq j < n_{dims}, 0 \leq c < n_{cells}, p < n_{pts} :$$

$$incl_{pts}(j, c, p) := \begin{cases} n\_pa(j, c, pl(p)) \geq 0 & \textbf{if } j < n_{dims} - 1 \\ M(j, c, ce(pl(p))) = c & \textbf{else} \end{cases} \tag{19}$$

Accumulating count. Now that we know which nodes are fused into the SCY-tree, we can accumulate the count of points in the subtree of each node i. For nodes on the same layer, the entry in n_co might be incremented by different threads. Therefore, we need to use atomic addition, implying that threads handling nodes on the same layer must be in the same thread block. For the layer just above the restricted dimension, we sum the old count of all children that are normal nodes and fused. If the parent is included and an S-connector node, we set the count to -1:

$$\forall 0 \leq j < n_{dims}, 0 \leq c < n_{cells}, la(j) \leq i < la(j+1), s\_incl(j, i, c) :$$

$$n\_co(j, c, pa(i)) := \begin{cases} n\_co(j, c, pa(i)) + co(i) & \textbf{if } co(i) \geq 0 \\ -1 & \textbf{if } co(pa(i)) < 0 \end{cases} \tag{20}$$

For the nodes on the remaining layers above the restricted dimension, we iteratively sum the new count of points of the children:

$$\begin{aligned}&\forall 0 \le j < n_{dims}, 0 \le c < n_{cells}, 1 \le k < j-1,\\ &\quad la(j-k) \le i < la(j-k+1), incl(j,c,i):\\ &\qquad n\_co(j,c,n\_pa(j,c,i)) :=\\ &\qquad\quad + \begin{cases} n\_co(j,c,pa(i)) + n\_co(j,c,i) & \textbf{if } co(i) \ge 0\\ -1 & \textbf{if } co(pa(i)) < 0 \end{cases}\end{aligned} \tag{21}$$

For all layers below the restricted dimension, the new count is a sum of the old counts of all fused nodes:

$$\begin{aligned}&\forall 0 \le j < n_{dims}, 0 \le c < n_{cells}, 1 \le k < n_{dims} - j,\\ &\quad la(j+k) \le i < la(j+k+1), n\_pa(i) \ge 0:\\ &\qquad n\_co(j,c,rep(j,c,i)) := \begin{cases} -1 \textbf{ if } co(i) < 0\\ n\_co(j,c,rep(j,c,i)) + co(i) \textbf{ else} \end{cases}\end{aligned} \tag{22}$$

Overview of restrict and merge operations. To summarize, the restricting and merging for all descriptors is done by

- Initialization: Each entry of $incl$, $incl_{pts}$, idx, idx_{pts}, and n_co is initialized to 0. Each entry of n_ch and n_pa is initialized to -1.
- Step 1: Compute for which descriptors the associated SCY-trees will be merged using two kernels; one that checks for each descriptor if there is an S-connection, using Equation 8, and one that uses this information to compute which SCY-trees will be merged, using Equation 9.
- Step 2: Compute which nodes are included in the final restricted and merged SCY-trees, and accumulate the count of points in the subtrees. We compute the inclusion in the restriction using five kernels. First, directly above the restricted dimension we use Equations 3, 15, and 20, second, for the remaining layers above we use Equations 4, 15, and 21, third, directly below we use Equations 17, 14, 16, and 22, fourth, for the remainder below we use Equations 18, 13, 16, and 22, and at last, we compute inclusion of points by checking if the leaf-node where the point is placed is included using Equation 19.
- Step 3: We now know which nodes and points are included in the final restricted SCY-trees. We do an inclusive scan and decrement each entry with 1 to compute the new indices for nodes idx and points idx_{pts}. This is also used to allocate the arrays for all final restricted SCY-trees.
- Step 4: All needed information has been precomputed, and we now copy all nodes, points, dimensions, and restricted dimensions to the final restricted SCY-trees. Each copy is independent and can be done completely in parallel.

4.2 Density-based clustering on the GPU

In this section, we discuss how to find the subspace clusters for all points in each SCY-tree. For each subspace region, the clustering process of INSCY is similar to that of DBSCAN [12]. The main difference is that INSCY supports different density measures and that clustering is done in a subspace projection. DBSCAN, and other density-based clustering methods, find clusters by expanding chains of density-connected points. This is a sequential process that we would like to replace with a parallelized process.

As discussed in related work, G-DBSCAN [5] is a competitive parallelization of full-space DBSCAN with rectangle kernel for density assessment. To support INSCY subspace clustering and further improve runtime performance, we introduce three major algorithmic solutions: supporting a different unbiased, i.e., subspace-dependent density-measure, reduced neighborhood searches, and expanding several clusters at once.

Precomputing the neighborhoods. To compute the neighborhood without allocating worst-case sizes, G-DBSCAN first computes the neighborhoods' size, then allocates space, and at last populates the neighborhoods with the neighboring points. For GPU-INSCY, the neighborhood of each point in all SCY-trees can be computed independently of other points and can therefore be computed in parallel over different thread blocks.

GPU-INSCY additionally takes advantage of already having computed the neighborhoods in the lower-dimensional subspace projections of the current subspace. Since adding a dimension to a subspace only increases the distance between points, previous neighborhoods can be used to bound the search for neighbors effectively. We demonstrate that this is an efficient strategy in the experiments, see Section 5.

Collecting the clusters. Using the precomputed neighborhoods, G-DBSCAN proceeds as follows. While there are still unclustered points, pick a random point to expand a cluster from. While that cluster is still being expanded, look at all points in parallel. If a point has just been added to the cluster, add its neighbors that have not yet been clustered to the current cluster. Since G-DBSCAN run in parallel for all points, but only a few points actually expand a single cluster each iteration, many threads are left idle. We suggest instead that a point adds itself to a cluster. Furthermore, we expand all clusters simultaneously for each point p in parallel as threads and over each descriptor in parallel as blocks. We precompute for each point if it is dense and only perform the following for dense points. For each descriptor, let $C \in \mathbb{N}^{n_{pts}}$ be clustering labels for each point p in the SCY-tree associated with that descriptor. Start by assigning all points to a singleton cluster, letting the cluster id be the point id, $C(p) := p$. While any cluster is still being expanded, look at all points in parallel. If the point p can reach a cluster with a lower cluster-id through its neighborhood, add the current point to that cluster $C(p) := \min_{q \in N_\varepsilon(p) \cup \{p\}} C(q)$. Between each iteration, we synchronize such that all threads know if any cluster has been expanded. For each iteration we check for all points if they can be expanded, thus we ensure that all density connected clusters have been found.

Clustering of each subspace region (SCY-tree) is independent of each other since the subspace regions are not S-connected, meaning that no density-connected clusters can span multiple subspace regions. Therefore, since no communication is needed, we can compute the clustering in parallel for each SCY-tree using different thread blocks. However, since we want to perform all clusterings in parallel and each SCY-tree must have been pruned first, we can only perform clusterings in parallel at the end.

4.3 Pruning on the GPU

As previously mentioned, we parallelize both pruning phases. In the interest of space, we keep the discussion brief as it follows the same approach as for restricting and merging the GPU-SCY-trees.

When pruning the recursion, we compute in parallel for each point if it is weak-dense. If it is not, mark it as not-included and propagate the count up in the SCY-tree layer by layer. We also parallelize the propagation over all nodes on a layer. If the count in the root is below min_C, then we do not continue with the recursion for this SCY-tree.

Pruning for redundancy is done as follows. For each superspace of the current subspace, we execute three kernels: Find the size of each cluster, find all clusters that overlap with points in the current SCY-tree, and find the smallest cluster that overlaps with the points in the current SCY-tree. Update the largest smallest cluster *max_min_cluster* that overlaps with the current SCY-tree. If the number of points in the SCY-tree scaled by the parameter r is smaller than *max_min_cluster*, we do not perform clustering for this SCY-tree because it can only contain redundant clusters.

4.4 Trading off speed for memory usage

Each recursive call of GPU-INSCY is parallelized over all descriptors simultaneously. This requires that we keep all final restricted SCY-trees, neighborhoods, and clusters in memory for all descriptors. However, memory on the GPU is limited, putting a bound on how large inputs we can process in parallel. There is, therefore, a natural trade-off between memory usage and how many descriptors we efficiently parallelize over simultaneously. To support efficient processing of larger inputs, we devise a version of GPU-INSCY called GPU-INSCY-memory that iterates over subsets of descriptors that we then parallelize over. We study this trade-off experimentally in Section 5.

5 EXPERIMENTS

5.1 Experimental setup

We conduct experiments for comparison of GPU-INSCY with INSCY on synthetic and on real-world data, and study impact of parameters on a workstation with Intel Core i7-9750HF CPU 2.60GHz × 12 cores, 16 GB RAM, GeForce GTX 1660 TI 6 GB dedicated RAM. The large scale experiments in Section 5.4 are executed on NVIDIA TITAN V 12 GB dedicated RAM, Intel Core E5-2687W 3.100GHz × 10 cores, 400 GB RAM.

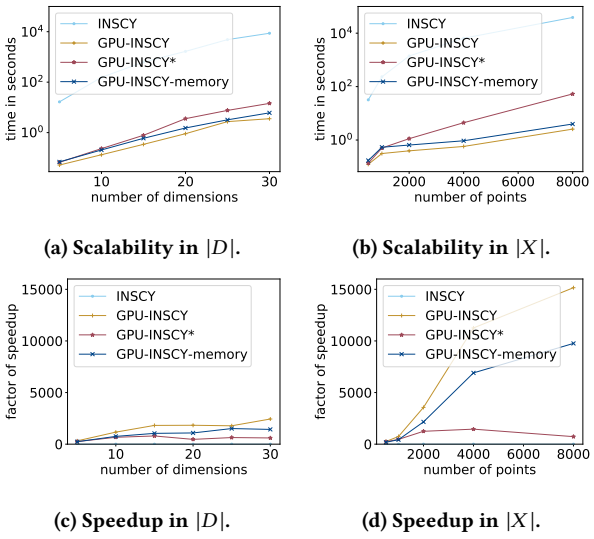
We use a search-tree for efficient neighborhood search in INSCY, which provides a large speedup and makes it a fairer comparison. We have experimentally validated that GPU-INSCY and INSCY produce identical subspace clusterings. Plots and further details have been omitted due to the space limit. We provide our source code at: https://github.com/jakobrj/GPU_INSCY.

5.2 Comparison with INSCY.

For subspace clustering, the dimensionality and size of the dataset are dominating factors regarding runtime. Especially dimensionality since, as the number of dimensions increases, the number of possible subspaces increases exponentially.

To compare INSCY and GPU-INSCY and the impact of input data, we use the data generator provided by [1] to generate synthetic datasets with dense clusters in arbitrary subspaces that may overlap and have a small percentage of noise. As in [7], we generate different datasets with four hidden subspace clusters. All runtimes are averages of three runs on datasets with the same generator settings. All dataset have been min/max-normalized. The default parameters for INSCY and GPU-INSCY in these experiments are $F = 1$, $R = 1$,$\mu = 8$, $\varepsilon = 0.01$, $n_{cells} = 4$, and min_C is set to 5% of the data points.

To analyze components of our algorithm, we also test GPU-INSCY* and GPU-INSCY-memory. GPU-INSCY* is a version of GPU-INSCY that does not bound the neighborhood search, so that we can study the effect of bounding the neighborhood search. GPU-INSCY-memory is described in Section 4.4. For our experiments we group the descriptors by the dimensions such that each iteration of the recursions is only parallel over the cells.



(a) Scalability in $|D|$. (b) Scalability in $|X|$.

(c) Speedup in $|D|$. (d) Speedup in $|X|$.

Figure 6: Scalability in size and dimensionality

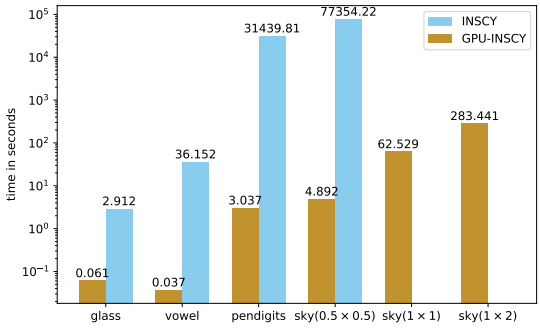


Figure 7: Real world data; INSCY aborted after 24 hours

Comparison of INSCY and GPU-INSCY. In Figure 6a, the running time for INSCY is decent for lower dimensions but increases rapidly for higher dimensions. GPU-INSCY reduces the running time to a point where it is faster to find subspace clusters for 25 dimensions using GPU-INSCY than finding subspace clusters for two dimensions using INSCY. In fact, in Figure 6c, the speedup of GPU-INSCY relative to INSCY keeps increasing. For 30 dimensions we achieve a factor of speedup of more than 2000×. A similar effect is observed for increasing the number of points. In Figure 6b, INSCYs runtime again increases faster than for GPU-INSCY. In Figure 6d, we see that the speedup becomes a factor of several thousand. This speedup is much higher than expected for the relatively low number of 1536 cores on our GPU.

Comparison of versions of GPU-INSCY. As mentioned in Section 4.2, we attribute this dramatic speedup to our bounding of the neighborhood searches. This effect is also clear in Figure 6c and 6d, where we see that GPU-INSCY* achieves a 500-1000× speedup, corresponding to a good use of the cores, and GPU-INSCY achieves a substantially larger speedup of up to 14'000× obtained by our improved neighborhood search. GPU-INSCY-memory allows us to run on larger datasets, with only a slight reduction of factor 2 in speedup, which is a reasonable trade-off.

5.3 Real world datasets

We also demonstrate GPU-INSCY speedups for real-world datasets. We report runtimes on the three datasets (glass, vowel, pendigits) [23] also studied in [7, 8]. The glass dataset $X_{glass} \in \mathbb{R}^{214\times9}$, vowel $X_{vowel} \in \mathbb{R}^{990\times10}$, and pendigits $X_{pendigits} \in \mathbb{R}^{7494\times16}$. Furthermore, we also evaluate on a more sizable higher dimensional real world data set, part of the SkyServer dataset[27] that contains measurements of objects in the sky, e.g., stars and galaxies. We select three different areas of size 0.5×0.5, 1×1, and 1×2, measured in spherical coordinates (RA/Dec): $X_{sky(0.5\times0.5)} \in \mathbb{R}^{7253\times17}$, $X_{sky(1\times1)} \in \mathbb{R}^{29627\times17}$, and $X_{sky(1\times2)} \in \mathbb{R}^{59285\times17}$. Experiments are aborted if they run for more than 24 hours, as INSCY does for larger setups. In Figure 7, we see that we obtain high speedups on all datasets, but much higher for larger datasets up to 15′000× speedup.

5.4 Effect of parameters

In this section, we study the effect of parameters for the density criterion, ε, μ, F and the model parameter n_{cells}.

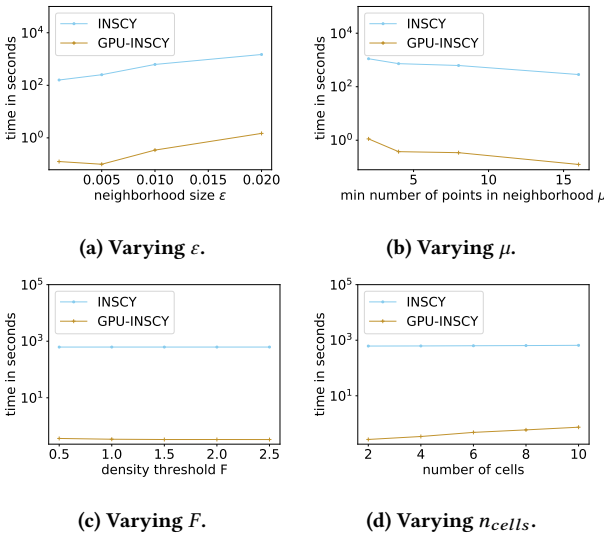
In particular, the parameters for the density criterion are expected to impact the running time. Especially the neighborhood radius ε is interesting since GPU-INSCY uses a strategy for reducing the neighborhood search that INSCY does not employ. The bigger the part of the subspace region that the neighborhood radius covers, the less we save by reducing the search area for the neighborhoods. Therefore, we expect that GPU-INSCY will obtain the greatest speedup for smaller values of ε. In Figure 8a, we study the range of ε between 0 and 0.02, and see that this is the case, but that the speedup remains large for the entire range.

The minimum number of points in the neighborhood μ and density threshold F only affect the number of points that are dense and weak-dense. The fewer points that are dense or weak-dense, the fewer points INSCY and GPU-INSCY need to process. As this is the same fraction of points for both INSCY and GPU-INSCY, we, therefore, expect to see a similar reduction in time for both algorithms. For μ, we study the range between 2 and 16, as this parameter is intended as a cut-off for avoiding tiny subspace clusters in very high-dimensional subspace projections (called pseudodense in INSCY). The factor F that governs the extent to which expected density is exceeded is evaluated in the range between 0.5 and 2.5. A value of 0.5 implies that we only expect a point to be half as dense as the expected density, which is a very low criterion, and 2.5 is more than twice the expected density, which is rather high. In Figure 8b and 8c, we see that the speedup for the density parameters remains stable for both criteria. As expected, we see that the running time decreases equally for both INSCY and GPU-INSCY as μ increases.

The parameter number of cells n_{cells} does not change the result, but only how we partition the subspace into cells and regions. We can, therefore, pick whichever number of cells INSCY or GPU-INSCY perform the best at. In Figure 8d, we study a range between 2 and 10 cells. Here both INSCY and GPU-INSCY perform best at a lower number of cells, especially GPU-INSCY.

5.5 Scalability and different distributions

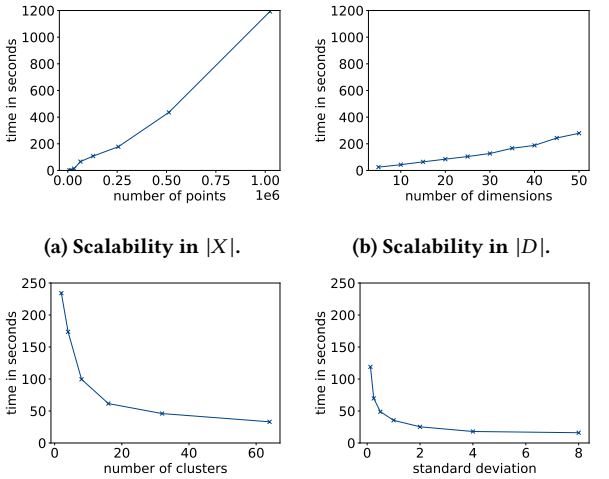
We evaluate scalability and different data distributions for GPU-INSCY alone. The running time of INSCY quickly becomes too high, e.g., more than 10 hours for 8000 points and 15 dimensions, which makes experiments for large inputs infeasible. In this section, we use GPU-INSCY-memory.



(a) Varying ε. (b) Varying μ.

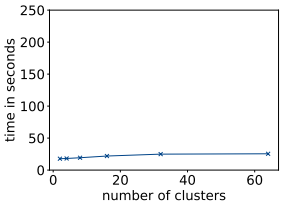
(c) Varying F. (d) Varying n_{cells}.

Figure 8: Runtimes for different parameter values



(a) Scalability in $|X|$. (b) Scalability in $|D|$.

(c) Scalability in number of clusters. (d) Scalability in cluster spread (standard deviation).



(e) Scalability in number of clusters with stable density.

Figure 9: Runtimes for scalability experiments

To test various distributions, we use the generator provided by [9], but modify it to generate clusters in random subspaces and not just the first k dimensions. The default settings used for the dataset generator are 64‘000 points with 4000 points for each cluster, except 1%, which is uniformly distributed noise. The dataset values range from −100 to 100, and the full space consists of 15 dimensions. Each cluster is normally distributed with a

standard deviation of 0.3 in a random 3-dimensional subspace. All datasets have been min/max-normalized. The default parameters for GPU-INSCY in these experiments are $F = 0.1$, $R = 1$, $\mu = 1$, $\varepsilon = 0.0003$, $n_{cells} = 4$, and $min_C = 500$ points.

Scalability. Figure 9a shows runtimes with increasing dataset size $|X|$ up to 1‘024‘000. The figure shows that GPU-INSCY performs subspace clustering on 1‘024‘000 points in less that 20 minutes. We also run experiments for increasing dimensionality $|D|$ up to 50, as shown in Figure 9b. GPU-INSCY can perform subspace clustering in 50 dimensions (and on 64‘000 points) in less than 6 minutes.

Data distribution. We evaluate performance on data with different distributions using the same setting as for scalability. In Figure 9c, we increase the number of clusters, keeping cluster distribution (standard deviation) and total number of points fixed. As we can see, large numbers of clusters further reduce the runtime of GPU-INSCY, as it finds fewer points in each neighborhood when the number of points per cluster decreases. In Figure 9d, we increase the spread of clusters (standard deviation). Again, the runtime of GPU-INSCY further improves, as neighborhoods are again less populated. Finally, we conduct an experiment with stable density. As the number of clusters is doubled, we increase cluster density accordingly by halving standard deviation. As Figure 9e confirms, similar density results in stable runtime when scaling number of clusters.

To summarize, the trend is that a lower density implies fewer points in each neighborhood and, therefore, a lower runtime. This means that GPU-INSCY scales particularly well for large numbers of clusters and clusters that are spread.

Overall, GPU-INSCY outperforms INSCY by two-four orders of magnitude with respect to runtimes for all tested settings. Even on our small GPU, we measure the running time in seconds instead of hours for smaller datasets ($<$ 10‘000 points and 15 dimensions) and minutes instead of days for larger datasets.

6 CONCLUSION

In this paper, we propose GPU-INSCY, a novel GPU-parallel algorithm for dimensionality-unbiased density-based subspace clustering, following INSCY. GPU-INSCY outperforms INSCY by several orders of magnitude. To achieve this, we utilize GPU cores by restructuring both the algorithmic processing and the data structure SCY-tree used in INSCY to fit the GPU computational model. Furthermore, GPU-INSCY proposes a further reduction of the time spent on neighborhood searches. Our experiments show that GPU-INSCY scales well w.r.t. dimensionality and size of the dataset, and compared to INSCY, the gap even continues to grow with the scale of data.

ACKNOWLEDGMENTS

This work was supported by Independent Research Fund Denmark.

REFERENCES

[1] Andrew Adinetz, Jiri Kraus, Jan Meinke, and Dirk Pleiter. 2013. GPUMAFIA: Efficient subspace clustering with MAFIA on GPUs. In *European Conference on Parallel Processing*. Springer, 838–849.

[2] Charu C Aggarwal, Joel L Wolf, Philip S Yu, Cecilia Procopiuc, and Jong Soo Park. 1999. Fast algorithms for projected clustering. *ACM SIGMoD Record* 28, 2 (1999), 61–72.

[3] Charu C Aggarwal and Philip S Yu. 2000. Finding generalized projected clusters in high dimensional spaces. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*. 70–81.

[4] Rakesh Agrawal, Johannes Gehrke, Dimitrios Gunopulos, and Prabhakar Raghavan. 1998. Automatic subspace clustering of high dimensional data for data mining applications. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*. 94–105.

[5] 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 (2013), 369–378.

[6] Ira Assent, Ralph Krieger, Emmanuel Müller, and Thomas Seidl. 2007. DUSC: Dimensionality unbiased subspace clustering. In *seventh IEEE international conference on data mining (ICDM 2007)*. IEEE, 409–414.

[7] Ira Assent, Ralph Krieger, Emmanuel Müller, and Thomas Seidl. 2008. EDSC: efficient density-based subspace clustering. In *Proceedings of the 17th ACM conference on Information and knowledge management*. 1093–1102.

[8] Ira Assent, Ralph Krieger, Emmanuel Müller, and Thomas Seidl. 2008. INSCY: Indexing subspace clusters with in-process-removal of redundancy. In *2008 Eighth IEEE International Conference on Data Mining*. IEEE, 719–724.

[9] Anna Beer, Nadine Sarah Schüler, and Thomas Seidl. 2019. A Generator for Subspace Clusters.. In *LWDA*. 69–73.

[10] 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*. 661–670.

[11] Chun-Hung Cheng, Ada Waichee Fu, and Yi Zhang. 1999. Entropy-based subspace clustering for mining numerical data. In *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*. 84–93.

[12] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise.. In *Kdd*, Vol. 96. 226–231.

[13] Reza Farivar, Daniel Rebolledo, Ellick Chan, and Roy H Campbell. 2008. A Parallel Implementation of K-Means Clustering on GPUs.. In *Pdpta*, Vol. 13. 212–312.

[14] Sanjay Goil, Harsha Nagesh, and Alok Choudhary. 1999. MAFIA: Efficient and scalable subspace clustering for very large data sets. In *Proceedings of the 5th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Vol. 443. ACM, 452.

[15] Karin Kailing, Hans-Peter Kriegel, and Peer Kröger. 2004. Density-connected subspace clustering for high-dimensional data. In *Proceedings of the 2004 SIAM international conference on data mining*. SIAM, 246–256.

[16] Hans-Peter Kriegel, Peer Kröger, and Arthur Zimek. 2009. Clustering high-dimensional data: A survey on subspace clustering, pattern-based clustering, and correlation clustering. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 3, 1 (2009), 1–58.

[17] You Li, Kaiyong Zhao, Xiaowen Chu, and Jiming Liu. 2013. Speeding up k-means algorithm by gpus. *J. Comput. System Sci.* 79, 2 (2013), 216–229.

[18] Woong-Kee Loh, Yang-Sae Moon, and Young-Ho Park. 2014. Erratum: Fast Density-Based Clustering Using Graphics Processing Units [IEICE Transactions on Information and Systems Vol. E97. D (2014), No. 5 pp. 1349-1352]. *IEICE TRANSACTIONS on Information and Systems* 97, 7 (2014), 1947–1951.

[19] Woong-Kee Loh and Hwanjo Yu. 2015. Fast density-based clustering through dataset partition using graphics processing units. *Information Sciences* 308 (2015), 94–112.

[20] Gabriela Moise and Jörg Sander. 2008. Finding non-redundant, statistically significant regions in high dimensional data: a novel approach to projected and subspace clustering. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*. 533–541.

[21] Emmanuel Müller, Stephan Günnemann, Ira Assent, and Thomas Seidl. 2009. Evaluating clustering in subspace projections of high dimensional data. *Proceedings of the VLDB Endowment* 2, 1 (2009), 1270–1281.

[22] 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)*. IEEE, 3701–3710.

[23] David J Newman, SCLB Hettich, Cason L Blake, and Christopher J Merz. 1998. UCI repository of machine learning databases, 1998.

[24] Lance Parsons, Ehtesham Haque, and Huan Liu. 2004. Subspace clustering for high dimensional data: a review. *Acm Sigkdd Explorations Newsletter* 6, 1 (2004), 90–105.

[25] Karlton Sequeira and Mohammed Zaki. 2005. SCHISM: a new approach to interesting subspace mining. *International Journal of Business Intelligence and Data Mining* 1, 2 (2005), 137–160.

[26] Kelvin Sim, Vivekanand Gopalkrishnan, Arthur Zimek, and Gao Cong. 2013. A survey on enhanced subspace clustering. *Data mining and knowledge discovery* 26, 2 (2013), 332–397.

[27] Alexander S Szalay, Jim Gray, Ani R Thakar, Peter Z Kunszt, Tanu Malik, Jordan Raddick, Christopher Stoughton, and Jan vandenBerg. 2002. The SDSS skyserver: public access to the sloan digital sky server data. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. 570–581.

[28] 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*. IEEE, 1–5.

[29] Kyoung-Gu Woo, Jeong-Hoon Lee, Myoung-Ho Kim, and Yoon-Joon Lee. 2004. FINDIT: a fast and intelligent subspace clustering algorithm using dimension voting. *Information and Software Technology* 46, 4 (2004), 255–271.