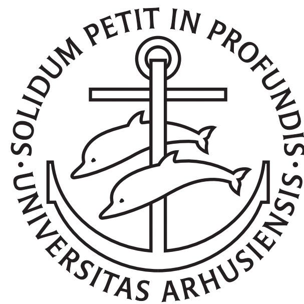
Parallel Algorithms for Clustering, Subspace Clustering, and Projected Clustering on the GPU

Jakob Rødsgaard Jørgensen

PhD Dissertation



Department of Computer Science
Aarhus University
Denmark

Parallel Algorithms for Clustering, Subspace Clustering, and Projected Clustering on the GPU

A Dissertation
Presented to the Faculty of Natural Sciences
of Aarhus University
in Partial Fulfillment of the Requirements
for the PhD Degree

by
Jakob Rødsgaard Jørgensen
July 27, 2022

Abstract

In our age, digital data is becoming increasingly available. A vast array of data mining algorithms have been developed to gain valuable information from this data. These data mining algorithms detect previously unknown patterns such as clustering, subspace clustering, projected clustering, outliers, cluster trajectories, association rules, and frequent patterns. Some of these patterns can take a long time to identify for most of the algorithms developed, and they, therefore, need a speed increase. We see potential in developing algorithms suited for modern hardware. Modern hardware promises a high computational throughput but achieves this at the cost of using multiple computational cores. This effectively changes the computational model, and we must develop algorithms that fit with these models to utilize the high computational throughput. We see the greatest potential in the modern graphics processing units (GPUs) with its thousands of cores.

The clustering algorithms, especially the subspace clustering algorithms, are very compute-heavy. Therefore, in this thesis, we focus on transforming these to fit the computational model of the GPU. We, furthermore, aim to improve the most promising algorithms within the category. The first algorithm we propose is GPU-INSCY, a GPU-parallelized version of the subspace clustering algorithm INSCY, which is an adaptation of the well-known SUBCLU that tried to reduce the size of the result set. GPU-INSCY includes a GPU-parallelized indexing structure for pruning regions we can cluster within subsequently. Furthermore, we improve density-based clustering on the GPU for subspace clustering, e.g., GPU-INSCY. We achieve this using a strategy for pruning the neighborhood search and a more efficient way of gathering the clusters. Next, we propose GPU-FAST-PROCLUS, a GPU-parallelized adaptation of the axis-aligned projected clustering algorithm PROCLUS, with some additional strategies that allow for the reuse of computations to gain even further speedup. GPU-FAST-PROCLUS is now so fast that it can perform axis-aligned projected clustering within a real-time interactive scenario. Therefore, we construct a data visualization tool where we implement the entire visualization pipeline on the GPU, called AVID. At last, we propose EGG-SynC, a GPU-parallelized variation of the clustering algorithm SynC. SynC automatically drags the clusters apart and only requires one user-defined parameter. Besides parallelizing SynC, we propose an indexing structure, a summarization strategy, and a correct termination criterion. Our proposed algorithms have expanded the range of data mining algorithms usable on standard desktop hardware.

Resumé

I vores tid, bliver digital data i stigende grad tilgængelig. En lang række *data mining* algoritmer er blevet udviklet for at opnå værdifuld information fra dette data. Disse *data mining* algoritmer leder efter tidligere ukendte mønstre, såsom *clustering*, *subspace clustering*, *projected clustering*, *outliers*, *cluster trajectories*, *association rules* og *frequent patterns*. Nogle af disse mønstre tager desværre lang tid at identificere, for de fleste af de udviklede algoritmer, og derfor er der et behov for hastighedsforøgelse. Vi ser derfor potentiale i at udvikle algoritmer til moderne hardware. Moderne hardware lover en høj beregningskraft, men på bekostning af, at regnekraften er fordelt på flere beregningsenheder. Dette ændrer beregningsmodellen, og vi må derfor udvikle algoritmer, der passer til disse modeller, for at udnytte den høje regnekraft. Vi ser det største potentiale i det moderne grafikkort (GPU), med dens tusindvis af enheder.

Clustering algoritmerne, specielt *subspace clustering*, er meget beregningstunge. I denne afhandling fokuserer vi derfor på at transformere disse til, at passe til GPU'ens beregningsmodel. Derudover går vi efter at forbedre de mest lovende algoritmer indenfor disse kategorier. Den første algoritme vi foreslår, er GPU-INSCY. Dette er en GPU-paralleliseret variation af *subspace clustering* algoritmen INSCY, som er en adaption af den kendte algoritme SUBCLU, som prøver at reducere størrelsen af resultatet. GPU-INSCY inkluder en GPU-paralleliseret indekseringsstruktur til begrænsning af regioner, som vi derefter kan finde *clusters* indenfor. Derudover forbedrer vi densitetbaseret *clustering* opdagelse på GPU'en, når det bruges i forbindelse med opdagelsen af *subspace clustering*, fx i GPU-INSCY. Dette opnås via en strategi til at begrænse naboskabssøgningen og via en mere effektiv måde at samle *clusters* på. Derefter foreslår vi GPU-FAST-PROCLUS, som er en GPU-paralleliseret adaptering af den *axis-aligned projected clustering* algoritme PROCLUS, som med ekstra strategier, tillader genbrug af beregninger, som øger hastigheden yderligere. GPU-FAST-PROCLUS' hastighed er nu øget i en sådan grad, at den kan udføre *axis-aligned projected clustering* opdagelse i et tidsrum, der muliggør realtidsinteraktion. Derfor har vi også konstrueret et datavisualiseringsværktøj, kaldet AVID, hvori hele visualiseringsprocessen er implementeret på GPU'en. Til sidst foreslår vi EGG-SynC, en GPU-paralleliseret variation af *clustering* algoritmen SynC. Sync trækker automatisk clusters fra hinanden og kræver kun én parameter af brugeren. Udover paralleliseringen af SynC, foreslår vi også en indekseringsstruktur, en opsummeringsstrategi, og et korrekt termineringskriterium. Igennem disse algoritmer udvider vi sortimentet af *data mining* algoritmer, der kan bruges på skrivebordscomputere.

Acknowledgments

Throughout my Ph.D. study, I have experienced a lot of help and support from the people around me. First, I would like to thank my advisor Ira Assent, who has encouraged, supported, advised, and collaborated with me doing my Ph.D. I want to thank Katrine Scheel, with whom I have spent countless hours discoursing and generating ideas. Furthermore, I would like to thank Anne C. Elster, who was kind enough to host, advise, and collaborate with me during my stay abroad in Trondheim, especially for allowing me to leverage her knowledge in the field of GPUs. I also thank Hans-Jörg Schulz for being a great lecturer in the data visualization course and for collaborating with me on a demo paper. I thank all members of the Data-intensive system group for a lot of great research discussions and social activities. Finally, I would like to give the biggest thanks to my girlfriend Emma Løper, who has been great support doing my Ph.D.

Jakob Rødsgaard Jørgensen,
Aarhus, July 27, 2022.

Contents

Abstract **i**

Resumé **iii**

Acknowledgments **v**

Contents **vii**

I Overview **1**

1 Introduction **3**
1.1 Thesis outline 5

2 Modern parallel architectures **7**
2.1 The Graphics Processing Unit (GPU) 7
2.2 Our common approaches and strategies 8

3 General notations and definitions **11**
3.1 Distances 11
3.2 Neighborhoods 12

4 Clustering **13**
4.1 k-means and k-medoids 13
4.2 DBSCAN 14
4.3 G-DBSCAN 16
4.4 Density peak clustering (DPC) 16
4.5 Clustering by synchronization (SynC) 17
4.6 Our contribution: EGG-SynC 18

5 Subspace Clustering **23**
5.1 SUBCLU 23
5.2 INSCY 24
5.3 Our contribution: GPU-INSCY 29

6 Projected Clustering **39**
6.1 PROCLUS 39
6.2 Our contribution: GPU-FAST-PROCLUS 41
6.3 Our contribution: AVID 44

7 Discussion and future work **47**

II Publications **51**

8 GPU-INSCY: A GPU-Parallel Algorithm and Tree Structure for Efficient Density-based Subspace Clustering **53**
8.1 Introduction 54
8.2 Related Work 55
8.3 Background 56
8.4 GPU-INSCY algorithm 60
8.5 Experiments 75
8.6 Conclusion 80
8.7 Acknowledgments 80

9 GPU-FAST-PROCLUS: A Fast GPU-parallelized Approach to Projected Clustering **83**
9.1 Introduction 83
9.2 Background 84
9.3 FAST-PROCLUS 88
9.4 GPU-PROCLUS 91
9.5 Experiments 97
9.6 Related work 102
9.7 Conclusions 104
9.8 Acknowledgments 104

10 AVID: GPU-enabled Visual Analytics with GPU-FAST-PROCLUS **105**
10.1 Introduction 105
10.2 PROCLUS and GPU-FAST-PROCLUS 106
10.3 Analysis and Visualization In Device 107
10.4 Demonstration Plan 110
10.5 Conclusions 112
10.6 Acknowledgments 112

11 EGG-SynC: Exact GPU-parallelized Grid-based Clustering by Synchronization **113**
11.1 Introduction 113
11.2 Related Work 115
11.3 Clustering by synchronization 116
11.4 EGG-SynC 118

11.5 Experiments 132
11.6 Conclusion 136
11.7 Acknowledgments 137
11.8 Proof of Lemma 11.4.3 137
11.9 Proof of Lemma 11.4.4 137
11.10 Proof of Lemma 11.4.5 137
11.11 Proof of Lemma 11.4.6 138

Bibliography **141**

Part I

Overview

Chapter 1

Introduction

We live in a world that is increasingly becoming digital, implying that we gather an ever-increasing amount of data. Understanding and gaining valuable insight from such data becomes overwhelming as the amount grows. Therefore, researchers have developed a vast range of data mining algorithms that can identify previously unknown patterns in the data. Some of the most common tasks within data mining are clustering, projected clustering, subspace clustering, outlier detection, trajectory mining, association rule mining, and frequent pattern mining. Some of these patterns take a long time to detect for algorithms; this is especially true for the clustering tasks and even more for subspace clustering. Clustering is the task of grouping data points based on some similarity. A user could, e.g., use clustering to identify customer groupings. However, due to the curse of dimensionality [14], points seem increasingly dissimilar as the dimensions of datasets increase. The curse of dimensionality implies that a cluster may only exist within a subspace of the full-dimensional space. This behavior naturally gives rise to two related tasks projected and subspace clustering. In projected clustering, each cluster exists within a subspace projection of the full-dimensional space, and in subspace clustering, clusters are identified in all possible subspaces. Clustering, projected clustering, and subspace clustering can provide significant new insight. However, these patterns can take a long time to identify.

Modern hardware can provide increasingly high computational power, but due to physical limitations, we can only gain such improvements by utilizing several computational cores. The many cores significantly change the computational model algorithms must follow to utilize the full computational power of such modern hardware. Most wildly used are the multi-core CPU and the graphics processing unit. The CPU is often equipped with more capable cores, whereas the GPU consists of thousands of more primitive and dependent cores. GPUs, therefore, provide the highest computational throughput but is more restrictive. Despite the more restrictive computational model, we see the greatest potential in the GPU.

This Ph.D. thesis aims to decrease the runtime of some of the most used or interesting clustering approaches. To achieve this, we propose utilizing modern hardware, specifically the GPU.

Research objective. Our research objective is to identify general principles for transforming traditional data mining algorithms. We see potential in several aspects. They are finding alternative processes to make an algorithm parallelizable under the GPU's model of computation, inventing new data structures suitable for the GPU, and balancing the workflow and storage requirements. To that end, we propose several GPU-parallelized algorithms, each requiring adaptations to processing orders, data structures, and balancing memory usage and workflow.

We propose several algorithmic and heuristic strategies that improve the performance of algorithms on the GPU and the CPU. In general, if utilized correctly, we show that the GPU can lead to several orders of magnitude speedup compared to CPU algorithms, both single-core and multi-core. More concretely, we study several clustering algorithms.

First, we consider the subspace clustering algorithm INSCY [11] and aim to make a GPU equivalent algorithm that we call GPU-INSCY [39]. INSCY makes use of the data structure SCY-tree for pruning dense regions. The SCY-tree is not suited for the GPU, so to transform the INSCY algorithm into a GPU equivalent, we propose a data structure that supports the same task but fits the architecture of the GPU. To that end, we developed a data structure called the GPU-SCY-tree. INSCY, furthermore, proceeds in a recursive fashion ill-suited for the GPU. We restructure this process to be able to perform more in parallel. INSCY used a clustering definition similar to that of DBSCAN [29], and we, therefore, base the clustering on the state-of-the-art GPU-parallelized variation of DBSCAN, G-DBSCAN [7]. However, we adapt it to fit the INSCY definition and propose improvements that mainly utilize that we perform DBSCAN in subspace clustering.

We also study the projected clustering algorithm PROCLUS [4] and propose a GPU-parallelized equivalent called GPU-FAST-PROCLUS [41]. For GPU-FAST-PROCLUS, we propose several algorithmic and heuristic improvements that provide CPU and GPU speedups. We also propose space-saving variations of these strategies to balance storage and runtime. However, the most noticeable speedups are those gained from tailoring each subroutine to fit the GPU's model of computation. GPU-FAST-PROCLUS is now so fast that it allows for real-time interaction, even for millions of points. We, therefore, propose a data visualization tool where we implement the entire pipeline on the GPU, called AVID.

Finally, we also study the newly proposed concept of clustering by synchronization and aim to transform the clustering algorithm SynC [18] into a GPU-equivalent. For this, we provide a multi-core CPU and GPU parallelized baseline. Since standard indexing structure such as the R-Tree does not fit well with the GPU, we propose a new grid-based indexing structure alongside a novel summarization strategy that summarizes all points in each grid cell as $O(d)$ values. These values allow our novel EGG-SynC algorithm [42] to compute the clustering by synchronization definition without looking at all points in each iteration. Besides the runtime improvements, the original SynC algorithm has an approximative termination criterion based on a single value. However, this termination criterion comes without any approximation guarantees, and we, therefore, propose a new termination criterion based on a state

instead of a value and prove that when the algorithm is in this state, it is safe to terminate. We also show how to gather the clustering correctly using our grid structure when the algorithm has terminated.

1.1 Thesis outline

This thesis is structured as follows. Chapter 2 introduces the multi-core CPU and CPU's vector operations and goes into detail with the GPU. Chapter 3 provides the most common clustering notations and definitions used throughout this thesis. This chapter is mainly regarding distance and neighborhood functions. Chapter 4 provides an overview of related clustering algorithms, definitions, and our contributions. This chapter includes the clustering definitions and algorithms; k-means, k-medoids, DBSCAN, G-DBSCAN, DPC, SynC, and our EGG-SynC. Chapter 5 provides an overview of related subspace clustering algorithms, definitions, and our contributions. This chapter includes the subspace clustering algorithms SUBCLU, INSCY, and our GPU-INSCY. Chapter 6 provides an overview of related axis-aligned projected clustering algorithms, definitions, and our contributions. This chapter includes PROLCUS, our GPU-FAST-PROCLUS, and our data-visualization tool AVID. Besides the format, we have not made any changes to these papers. The papers are as follows:

- Chapter 8: GPU-INSCY: A GPU-Parallel Algorithm and Tree Structure for Efficient Density-based Subspace Clustering.
  Jakob Rødsgaard Jørgensen, Katrine Scheel, and Ira Assent. EDBT 2021.
- Chapter 9: GPU-FAST-PROCLUS: A Fast GPU-parallelized Approach to Projected Clustering.
  Jakob Rødsgaard Jørgensen, Katrine Scheel, Ira Assent, Ajeet Ram Pathak, and Anne C. Elster. EDBT 2022.
- Chapter 10: AVID: GPU-enabled Visual Analytics with GPU-FAST-PROCLUS.
  Jakob Rødsgaard Jørgensen, Ira Assent, and Hans-Jörg Schulz. EDBT 2022.
- Chapter 11: EGG-SynC: Exact GPU-parallelized Grid-based Clustering by Synchronization.
  Jakob Rødsgaard Jørgensen, and Ira Assent. EDBT 2023.

Please note that the paper provided in Chapter 10 is a demo paper showcasing a data visualization tool for the paper in Chapter 9.

During my Ph.D. I have co-authored several papers that are not part of this thesis. These papers are regarding; dimensionality reduction, reparameterization of a neural network, and trajectory mining, all parallelized for the GPU. One published at NeurIPS, one under review, and one still being written. The published paper is:

- What if neural networks had SVDs?
  Alexander Mathiasen, Frederik Hvilshøj, Jakob Rødsgaard Jørgensen, Anshul Nasery, and Davide Mottin. NeurIPS 2020.

Chapter 2

Modern parallel architectures

Throughout the early history of computing, software developers have counted on single-core architectures to become exponentially faster over the years. However, the exponential improvement of a single-core has been dwindling. Instead, hardware vendors have relied on multi-core architectures to provide higher throughput. Especially Graphics Processing Units (GPUs) and multi-core CPUs are widely adopted as consumer hardware. Sadly, the higher throughput provided by the multi-core architectures does not come without a cost. Researchers must develop new specialized parallelized algorithms that compute the same result as the sequential counterparts to utilize both multi-core CPUs and GPUs. To clarify, we use the term sequential when instructions of a process are executed after each other and parallel when instructions are executed concurrently. There are significant differences in how multi-core architectures work; therefore, researchers must develop unique algorithms to utilize each. The main difference between the CPU and the GPU is that the CPU has few highly independent and fast cores, whereas the GPU has many simple and more dependent cores. Since data mining algorithms often perform similar operations on many data points, we are confident the GPU provides the best opportunity for high throughput.

2.1 The Graphics Processing Unit (GPU)

The GPU is, as the name suggests, developed to process graphics. However, GPUs have become more and more flexible over the years, so performing general-purpose computations has become possible. Nevertheless, the GPU architecture still varies a lot from the modern CPU.

The modern CPU contains tens of cores on which threads can be executed individually and concurrently with different instructions, known as Simultaneous Multi-Threading (SMT). Furthermore, the CPU has vector operations that can be executed on multiple data entries concurrently, known as Single Instruction, Multiple Data (SIMD). In contrast, the modern GPU contains thousands of cores that can execute threads concurrently, but groups of threads must execute the same instruction simultaneously, known as Single Instruction, Multiple Threads (SIMT). The difference between these

can seem minuscule but significantly impact how researchers design algorithms.

NVIDIA[1] and the CUDA[2] programming environment have slight differences between how the cores and memory are physically organized and how the threads executed on these cores are logically structured. We here simplify the abstractions a bit. NVIDIA group cores into streaming multiprocessors (SMs), where groups of cores in each SM have a common program counter, implying that they must perform the same instruction at all times. The cores in an SM share access to fast shared memory and can synchronize with each other without synchronizing the entire device. All cores on the GPU share access to the slower global memory. CUDA group threads into warps executed on cores in an SM and therefore have the same restrictions and capabilities. CUDA, furthermore, group warps into blocks and execute warps in a block within the same SM, but different warps are guaranteed not to share a program counter. Therefore, the number of cores in an SM limits the number of threads in a block. Furthermore, CUDA organizes the threads in each block in a 3-dimensional grid. Similarly, the blocks are also organized in a 3-dimensional grid, referred to as the grid. The process executed on multiple threads at once is called a kernel. During a kernel call, the coordinates for each thread and block are available at execution time and can be used to identify what memory a specific thread handles. Since only smaller groups share a program counter, CUDA still allows running multiple streams of threads executing different kernels. Streams, e.g., allow loading memory from the main RAM to global memory on the GPU in one stream while a different stream executes kernels.

As mentioned, the GPU have shared memory and global memory. Share memory is much faster than global memory but is also more restricted in its small size of only a few megabytes, and only threads in the same block can access the same area. Global memory is much slower but can be accessed by all threads and are much larger with a couple of gigabytes of memory. Like for the CPU, memory on the GPU is loaded in blocks to make the loading faster. Also, similar to the CPU, race conditions can occur if multiple threads read and write to the same memory address. To counter this, the GPU provides atomic operations; however, they should be used with care since they can be expensive to perform.

2.2 Our common approaches and strategies

Throughout our papers, we use several repeated approaches and strategies to adhere to the computational model of the GPU. Instead of repeating ourselves, we provide an overview here.

General. The first conceptualization to consider is which computations we can perform or alter to perform in parallel. Remember, the GPU executes threads concurrently; operations must, therefore, be independent of each other. If only parts of the computations are independent, the GPU provides some synchronization

[1] https://www.nvidia.com/

[2] https://developer.nvidia.com/cuda-toolkit

capabilities, but it should be used with care since it takes time to perform. Furthermore, since the GPU execute threads in warps that must perform the same instructions at all time, we must identify identical operations that we can perform in this fashion. Such independent and identical operations can be hard to identify in many cases but are necessary to isolate to utilize the GPU well. Even though context switching is fast on the GPU, it still takes a bit of time; therefore, handling multiple elements per thread makes for faster runtime. We only put weights on such optimizations in the descriptions if they impact the algorithm.

Memory. It is essential to consider how we can utilize different types of memory, how they work, and their limitations. As mentioned, shared memory is many times faster than global memory. When we compute a global result, it is, therefore, an excellent strategy to keep temporary results in shared memory or local variables and then later combine the result in global memory. However, a limitation of shared memory is that we can only allocate it during a kernel call, and only threads within the same block can access it. This implies that if we want to keep a temporary result in shared memory, it must be within the same kernel call, e.g., by combining kernels that perform different computations. Furthermore, the threads that compute the temporary result and those using it must be within the same block. On the other hand, combining kernels may not always lead to speedups, i.e., it can lead to a less parallelizable kernel. An additional benefit of using local temporary results is reducing the number of atomic operations we would need to perform. The relatively small memory size available in shared memory is also a limitation we must consider. Most often, the size implies we have tens of values available for each thread, limiting what we can save in shared memory, e.g., the neighborhood of a point would be too large to keep in shared memory. Similarly, even though the global memory is many times larger, it is still limited; therefore, when handling massive datasets, it can be necessary to offload memory from the GPU to the main memory. Like the CPU architecture, the GPU loads data as coalesced blocks from global memory issued by threads in a warp. Therefore, having misaligned data access increases the running time significantly.

For-loop notation. In pseudo-code, we often formulate threads that are being invoked concurrently as a for-loop. This abstraction is solely a means of notation; in reality, we invoke multiple threads and distribute the computations associated with elements in the for-loop. We handle this distribution through what is commonly known as striding. Each thread contains a for-loop that starts at the thread ID and increments with the number of threads invoked. This pattern ensures that all elements are handled and makes it easy to access the memory coalesced.

List construction. A common operation we use is to construct a list. However, how we can do this on the GPU might not be clear. On the GPU, we generally want to do the same for many elements simultaneously, and we want to allocate memory as few times as possible. Therefore, we keep many lists of elements in one allocated array and keep track of each list's start and end. We use the C++ notation, where the end is the last index plus one. Notice that the end (minus one) of one list is the start of the next; therefore, it is redundant to keep both. To construct the list in parallel and avoid allocating on the fly, we first compute the size of each list, allocate the total

amount of space, and populate each list. First, for each element in parallel, increment the size of the list it belongs to atomically. Then, perform an inclusive scan to obtain the ends of each list and allocate the total required space. At last, for each element in parallel, increment the location it belongs to in the list atomically and place the element at that location.

Chapter 3

General notations and definitions

This chapter presents standard notation used throughout the overview in Part I. In data mining, the primary assumption is that we are given a dataset with some unknown knowledge. We assume that a dataset $data$ has d dimensions and n data points. The full set of dimensions is denoted as D, and the subset of these dimensions is denoted as $S \subseteq D$. We use some notation that has several meanings. However, we believe that the meaning is clear from the context. We denote both the absolute value and the size of a set as $|a|$. We use superscript to denote exponent and indicate a version at a specific iteration, e.g., p^t. For simplicity, we use the notation point p to refer to the point-vector in the dataset $data$ and the index of that point in the dataset. To distinguish, we use $:=$ for assignment and $=$ for equality. Furthermore, we note the boolean value true as $\mathbb{1}$ and false as $\mathbb{0}$. We use this notation because we often perform inclusive scans on the boolean values, where we interpret true as 1 and false as 0. Throughout the overview in Part I, we note a function using brackets, e.g. $f(a)$, list or arrays using squared brackets, e.g. $L[a]$, and matrices using subscript, e.g. M_a. However, in practice, we represent matrices as arrays.

3.1 Distances

All algorithms in this thesis apply a notion of similarity, or opposite, distance, between each pair of points $p, q \in data$. The most utilized distance measure is the Euclidean distance, i.e., the L2-norm of the distances between two points $p, q \in data$.

Definition 3.1.1 (Euclidean distance)**.** *Given a dataset data and two points* $p, q \in data$*. The Euclidean distance is:*

$$||p-q||_2 := \sqrt{\sum_{j \in D} |p_j - q_j|^2}. \tag{3.1}$$

Other methods use the Manhattan distance, which is similar but uses the L1-norm.

Definition 3.1.2 (Manhattan distance)**.** *Given a dataset data and two points $p, q \in data$. The Manhattan distance is:*

$$||p-q||_1 := \sum_{j \in D} |p_j - q_j|. \tag{3.2}$$

The Manhattan segmental distance is also used and is the Manhattan distance normalized by the number of dimensions in a given subspace.

Definition 3.1.3 (Manhattan segmental distance)**.** *Given a dataset data, two points $p, q \in data$, and a subspace projection $S \subseteq D$. The Manhattan segmental distance is:*

$$\frac{1}{|S|}||p-q||_1^S := \frac{1}{|S|} \sum_{j \in S} |p_j - q_j|. \tag{3.3}$$

When any distance measure can be used, we simply denote it as $||p-q||$.

3.2 Neighborhoods

Many data mining algorithms use the notion of the neighborhood of a point. A neighborhood contains the closest points of a given point, either within a fixed radius or with a fixed number of closest points. We start by defining the ε-neighborhood that contains all points within a radius ε.

Definition 3.2.1 (ε-neighborhood)**.** *Given a dataset data and a neighborhood radius ε, the ε-neighborhood of a point $p \in data$ is:*

$$N_\varepsilon(p) := \{q \in data \mid ||p-q|| \leq \varepsilon\}. \tag{3.4}$$

.

Opposite, we define the k-nearest neighbors as the ε-neighborhood with a radius that contains exactly k points.

Definition 3.2.2 (k-nearest neighbors)**.** *Given a dataset data and a number of nearest neighbors k, the k-nearest neighbors of a point $p \in data$ is:*

$$kNN(p) := N_\varepsilon(p) \ : \ |N_\varepsilon(p)| = k. \tag{3.5}$$

We now have some of the most essential and common concepts defined, and we will build upon this as we proceed with the thesis.

Chapter 4

Clustering

Clustering is the task of grouping points based on some similarity. However, this is a relative loos definition, and the best concrete definition is often use-case dependent, giving rise to several definitions of clustering. Some main categories are centroid-based, density-based, connection-based, distribution-based, and fuzzy clustering. However, this thesis only touches on the density-based and centroid-based clustering methods. This chapter presents the work within clustering that we build upon before introducing our contribution, clustering algorithm EGG-SynC [42], and in later chapters, our contributions within subspace and projected clustering. Starting with probably the most well-known clustering algorithms, k-means [51], and k-medoids [44, 61], and then going onto the density-based clustering methods that allow for arbitrarily shaped clusters. The first of these is the DBSCAN clustering algorithm [29]; the first to introduce the notation of density-based clustering. Furthermore, we describe what we deem the best GPU-parallelized version of DBSCAN, called G-DBSCAN [7]. Next, we shortly present density peak clustering (DPC) [67], a clustering method that removes the predefined density threshold and instead aids the user in picking the cluster centers. We then present clustering by synchronization (SynC) [18], a clustering concept and algorithm that automatically separates clusters of various densities. This concept is motivated by a different idea than DBSCAN, but we provide a formal definition that shows the similarities to DBSCAN and enables us to fix inaccuracies of the SynC algorithm. Building upon SynC and our formal definition of the concept, we propose EGG-SynC, an exact variation that first terminates when the clustering can be correctly gathered. To make EGG-SynC, we provide an indexing structure and a strategy that summarizes regions in the dataset based on a grid structure without loss in accuracy. We propose to parallelize EGG-SynC for the GPU and design the grid structure with this in mind.

4.1 k-means and k-medoids

Probably the most well-known definition of clustering is the k-means [54]. It is defined by partitioning the data points into k regions where the total distance for all

points to the center of their cluster is as small as possible.

Definition 4.1.1 (k-means)**.** *Given a dataset data $\in \mathbb{R}^{n\times d}$ and number of clusters k. The clustering C is k partitions of the dataset data where the points in each cluster has the shortest distance to the mean of that cluster:*

$$C := \arg\min_{C} \sum_{C_i\in C} \sum_{p\in C_i} ||p-\mu_i||, \tag{4.1}$$

where $\mu_{i,j} := \frac{1}{|C_i|}\sum_{p\in C_i} p_j$ *is the mean of cluster* C_i*, and* $\bigcup_{C_i\in C} C_i = data$*.*

A similar definition is the k-medoids [44, 61], which makes it possible to cluster a dataset that does not allow for computing the mean of a cluster, e.g., documents, but instead uses the medoid of each cluster. The medoid is a representative point in the cluster with the smallest distance to all other points in that cluster. This definition makes it possible to cluster datasets even if they do not exist in a Euclidean space.

Definition 4.1.2 (k-medoids)**.** *Given a dataset data and the number of clusters k. The clustering C is k non-empty partitions of the dataset data where the points in each cluster have the shortest distance to the medoid of that cluster:*

$$C := \arg\min_{C} \sum_{C_i\in C} \sum_{p\in C_i} ||p-m_i||, \tag{4.2}$$

where $m_i := \arg\min_{q\in C_i}\sum_{p\in C_i} ||p-q||$ *is the medoid of cluster* C_i *and* $\bigcup_{C_i\in C} C_i = data$*.*

Both k-means and k-medoids can have n^k partitions and non-convex cost functions, making a global minimum expensive to find; therefore, most algorithms that identify such clusters settle for a local minimum instead. The approximative method can quickly find a result and, therefore, scale well to large datasets. However, these clustering definitions produce a data partition as a Voronoi-diagram and, therefore, prefer spherical clusters and do not automatically separate outliers. These definitions minimize the intra-cluster distance but do not consider the inter-cluster distance. This implies that they can separate points close to each other and place them in different clusters. At last, they also require the user the select the number of clusters in advance.

4.2 DBSCAN

Instead of just minimizing the distance to the center of a cluster, a different approach is to define density-connected areas as clusters, i.e., clusters are dense areas separated by less dense areas. To achieve this, Ester et al. [29] first define the terms core-point, directly density-reachable, density-reachable, and density-connected.

A point is considered a core-point if it has more than the minimum required points *minPts* in its ε-neighborhood. These are the main building blocks of a cluster.

Definition 4.2.1 (Core-points)**.** *A point $p \in Data$ is a core-point if:*

$$|N_\varepsilon(p)| \geq minPts. \tag{4.3}$$

A point that can be reached by the ε-neighborhood of a core-point is called directly density-reachable and should be part of the same cluster.

Definition 4.2.2 (Directly density-reachable)**.** *A point $p \in Data$ is directly density-reachable from point q if:*

$$dirReach(p,q) := p \in N_\varepsilon(q) \wedge |N_\varepsilon(q)| \geq minPts. \tag{4.4}$$

Core-points that are directly density-reachable from other core-points create a chain of points that should be in the same cluster. A point that can be reached from a core-point through such a chain is said to be density-reachable.

Definition 4.2.3 (Density-reachable)**.** *A point $p_1 \in Data$ is density-reachable from a point $p_j \in Data$ if:*

$$denReach(p_1, p_j) := \exists p_1, \ldots, p_j \in data : dirReach(p_i, p_{i+1}) \forall i < j. \tag{4.5}$$

Similarly, two points that are density-reachable from a common point are called density-connected, even if none of them are core-points.

Definition 4.2.4 (Density-connected)**.** *A point $p \in Data$ is density-connected to point $q \in Data$ if:*

$$denConnect(p,q) := \exists o \in data : denReach(p,o) \wedge \tag{4.6}$$
$$denReach(q,o). \tag{4.7}$$

According to the DBSCAN definition, clusters are sets of points in the dataset that are density-connected and where each cluster contains all dense points reachable from that cluster.

Definition 4.2.5 (DBSCAN)**.** *Given a dataset data, a ε neighborhood radius, and a minimum number of points minPts. A clustering C is a set of cluster $C_i \in C$ when the following conditions holds:*

- $\forall p,q \in data : p \in C_i \wedge denReach(q,p) \Rightarrow q \in C_i$
- $\forall p,q \in C_i : denConnect(p,q)$.

All points not in C is considered to be noise.

The DBSCAN clustering definitions can create arbitrary-shaped clusters separated by less dense areas through the density-connected points. Furthermore, the number of clusters is automatically determined but at the cost of using a density criterion instead. The fixed ε neighborhood radius and the minimum number of points *minPts* constitute a global threshold for how dense areas should be connected. However, it is not always the case that such a global threshold exists in the data.

Algorithm 1 G-DBSCAN($data, \varepsilon, minPts$)
Our interpretation of the algorithm by Andrade et al. [7].

1: Precompute neighborhoods
2: **while** still exists, pick un-clustered point p_l **do**
3: $exp[l] := \mathbb{1}$
4: **for** point $p_i \in data$ - in parallel **do**
5: **while** still expanding **do**
6: $exp[i] := \bigvee_{p_j \in N^S_\varepsilon(p_i) | \ |N^S_\varepsilon(p_j)| \geq minPts} exp[j]$
7: copy exp to host
8: assign cluster ID to $C[i]$ where $exp[i] = \mathbb{1}$

4.3 G-DBSCAN

Several GPU-parallelizations of DBSCAN exist [7, 52, 53, 78], but experimental comparisons of these are sparse. To the best of our judgment, G-DBSCAN [7] is the variation that reports the best running times and, therefore, the variation we consider. Andrade et al. [7] propose precomputing the neighborhoods as a single preprocessing task and then use this to perform DBSCAN clustering. The lists of points in the neighborhoods are constructed as we describe in Section 2.2. The G-DBSCAN clustering proceeds as in Algorithm 1. While there are still un-clustered points, pick an un-clustered point p_l. G-DBSCAN then iteratively marks in the array *exp* which points are expanding the cluster starting at point p_l. Initially, mark point p_l as part of the cluster $exp[l] := \mathbb{1}$. Then in parallel across points p_i, while the cluster is still expanding, assign all points p_j in the ε-neighborhood of p_i to the cluster. When no more new cluster members are found, the array *exp* is copied to the host, and all point p_i where $exp[i] = \mathbb{1}$ is assigned the same cluster ID.

Checking all points to see if they can expand the cluster instead of just checking the neighborhoods of each point already assigned to the cluster is more work. However, this is a common strategy when developing for the GPU and one that allows for more parallelized work; this is a trade-off that is often worth taking. Even though this is the case, we still see a potential for improvement that we discuss in Section 5.3.

4.4 Density peak clustering (DPC)

Rodriguez et al. [67] propose a clustering definition that removes the global density threshold *minPts* but instead requires the user to select the cluster centers. However, DPC aids the selection using a 2-dimensional visual representation of possible cluster centers; by plotting the size of the ε-neighborhood of point p (local density) against the minimum distance to a point q with a higher local density. The points with both a high local density and a large distance to a point with a higher local density are the candidates for cluster centers. Connecting each point to the closest point with a higher local density naturally creates a tree with the point with the highest density as

the root. Cutting off the subtrees starting at the points picked as the centers create a set of trees. The points in each of these trees constitute the clusters. Like DBSCAN, DPC allows for arbitrary-shaped clusters. Furthermore, DPC provides a density-based clustering that does not require a density criterion. However, they have replaced the density criterion with requiring the user to select the cluster centers manually.

4.5 Clustering by synchronization (SynC)

Instead of providing a fixed density threshold or needing the user to select where to separate the clusters, it would be nice to have a density-based clustering method that could drag the clusters apart automatically. clustering by synchronization (SynC)[18] is a new concept based on attracting points to their ε-neighborhood until it reaches a state where points have synchronized, i.e., do not move around any longer. Inspired by the Kuramoto Model, Böhm et al. [18] define how the SynC algorithm updates the points in each iteration:

Definition 4.5.1 (Update)**.** *Given a point $p \in data^t$ at iteration t, the point p is updated for iteration $t+1$ as:*

$$p_i^{t+1} := p_i^t + \frac{1}{|N_\varepsilon(p^t)|} \sum_{q \in N_\varepsilon(p^t)} \sin(q_i^t - p_i^t). \tag{4.8}$$

However, Böhm et al. do not provide a formal definition of clustering by synchronization. Therefore, we provide this in Jørgensen et al. [42], see Section 4.6. Instead of terminating when a clustering definition is satisfied, Böhm et al. [18] propose a measure of orderedness called Cluster Order Parameter r_c.

Definition 4.5.2 (Cluster Order Parameter)**.** *Given dataset data, the cluster order parameter r_c is defined as:*

$$r_c := \frac{1}{n} \sum_{p \in data} \frac{1}{|N_\varepsilon(p)|} \sum_{q \in N_\varepsilon(p)} e^{-||q-p||}. \tag{4.9}$$

The cluster order parameter r_c increases towards 1 as points in the neighborhoods move closer. If $r_c = 1$, all points have synchronized, i.e., are at the same location as the points in their neighborhoods. However, since the update function Definition 4.5.2 uses the non-linear sin-functions to update the location of a point, that point never reaches the center of the neighborhood, implying that r_c never reaches 1. Instead, SynC terminates whenever $r_c \geq \lambda$ where λ is an extra parameter the user must provide. Even though it would seem like a high r_c would imply that the points are close to synchronizing, this is not guaranteed. In Section 4.6, we discuss this problem and propose a way to solve it. Another drawback is that clustering by synchronization takes a long time to compute, even with the premature termination criterion. Since SynC moves points closer to each other, the clusters cover a smaller and smaller area until the neighborhood of a point becomes the entire cluster. The large neighborhoods imply that each iteration can take $O(n^2)$, even if the algorithm uses an efficient indexing structure, like R-Tree used in the FSynC algorithm [22].

4.6 Our contribution: EGG-SynC

SynC has a lot of desirable properties; however, it has a considerable drawback of a long runtime due to the $O(n^2)$ time complexity for each iteration. Xinquan Chen [22] investigates reducing the running time using a data structure called the R-Tree [34]. However, this mainly leads to speedup in the first couple of iterations while points are spread-out. Therefore, utilizing the R-Tree only leads to a relatively low speedup in most cases. Consequently, we still see the need to speed up clustering by synchronization but using a different approach.

In Jørgensen et al. [42], we propose EGG-SynC; the main goal is to reduce the running time of clustering by synchronization using both algorithmic strategies and by utilizing the GPU. We provide a formal clustering definition and use this to define an accurate termination criterion in Section 4.6. We then provide a grid structure that can be used to make range queries on the GPU, Section 4.6. Furthermore, we propose a summarization strategy supported by the grid structure, Section 4.6. At last, we provide a GPU-parallelized execution of the updates that use both our summarization strategy and grid structure.

Formal definition and accurate termination criterion

As mentioned, Böhm et al. [18] do not provide a formal definition of clustering by synchronization. Therefore, we start by formulating this in Jørgensen et al. [42]. The formal definition becomes very similar to DBSCAN; however, it replaces the notion of density-connected points with points moving closer over iterations (synchronizing). For some iteration t and all future iterations $t' \geq t$, for maximality, two points $p,q \in data$ if p is in a cluster C_i and q is in the neighborhood of p, then q must be in the same cluster and, for connectivity, two points $p,q \in C_i$ in the same cluster must also be in each others neighborhood.

Definition 4.6.1 (Clustering by synchronization)**.** *Given a dataset data, a ε neighborhood radius, and the iterative update of points Definition 4.5.1, a clustering C is a set of clusters $C_i \subseteq data$ where the following conditions are satisfied after iteration t:*

- $\forall t' \geq t, \forall p,q \in data \; : \; p \in C_i \wedge q \in N_\varepsilon(p) \Rightarrow q \in C_i$
- $\forall t' \geq t, \forall p,q \in C_i \; : \; q \in N_\varepsilon(p)$.

While it is true that when the cluster order parameter Definition 4.5.2 $r_c = 1$ the clustering by synchronization definition is satisfied, it never actually reaches 1, making the original SynC algorithm inaccurate. Definition 4.6.1 makes it possible to devise a termination criterion that guarantees that EGG-SynC finds a clustering that follows clustering by synchronization. In Jørgensen et al. [41], we propose the following criterion for when the points have synchronized and prove that we can safely terminate and gather the cluster defined by Definition 4.6.1.

Definition 4.6.2 (Synchronization criterion)**.** *Given a dataset data and a neighborhood radius ε, the synchronization criterion is defined as follows:*

$$\begin{aligned}\forall p \in data : \nexists q \in data :& (\varepsilon/2 < ||p-q|| \leq \varepsilon)\\ \wedge\ \nexists q \in data :& (\varepsilon < ||p-q|| \leq \varepsilon + \delta\\ & \wedge (dist(MBR(N_{\varepsilon/2}(q)), p) \leq \varepsilon),\end{aligned}$$

where $MBR(S)$ is the minimum bounding rectangle of a set of points, $dist(MBR, p)$ is the distance from a minimum bounding rectangle to a point p, and $\delta := \varepsilon - \varepsilon \times \sqrt{15/16} + \varepsilon/-\sin(\varepsilon/2)$ is an extra distance from the neighborhood that, must be checked to ensure that no point can move into the neighborhood in later iterations.

In Jørgensen et al. [41], we prove that when the synchronization criterion is satisfied, no points can leave or enter any neighborhoods, and therefore the points that end up synchronizing are the points that are in each other's neighborhood.

The grid structure

We aim to create an indexing structure that is easy to access and construct in parallel on the GPU. Furthermore, this indexing structure should support our summarized statistics, Section 4.6, and new termination criterion, Section 4.6. With these goals in mind, we propose to use a grid structure with a fixed cell width c_w. Having a fixed cell width implies that we can easily enumerate the cells and compute the ID of the cell a point p is located within. Even though a grid structure is a simple concept, we can still represent it in many ways. We propose a mixed representation using two ways of representing a grid structure to balance the advantages and disadvantages. We aim to break down the explanation of our mixed access grid structure, first explaining a random access, then a sequential access, and finally a mixed grid structure.

The random access

To create a grid structure with random access, we propose representing all cells in an enumerable order in an array. This would imply that we can compute the index of a specific cell in the array and look up the result using random access. Accompanying the grid structure, we have an array of the list of points in each grid cell. For each grid cell, we have the location of where each list starts and end.

Construction. We construct this grid structure similar to the lists described in Section 2.2. For each point in parallel, increment the count of points *gridSizes* in the grid cell it resides within. Perform an inclusive scan of the count *gridSizes* to compute the end of each list *gridEnds*. For each point in parallel, add it to the list of points *gridPoints* of the grid cell it resides within.

Access time. To access a grid cell, we compute the index of the cell; then, we can access each grid cell in $O(1)$. To perform a range query, we need to access all cells within the range, also the empty once, implying an $O(v^d)$ query time, where

$v := \lceil \varepsilon / c_w \rceil \times 2 + 1$ is the maximal number of cells that the range cross on the diagonal. Later we pick c_w to be dependent on ε, implying that v becomes a constant.

Space usage. Since we represent all cells, also empty once, the space usage becomes exponential in the number of dimensions $O(w^d + n)$, where $w := \lceil 1/c_w \rceil$ is the number of cells along each dimension. This complexity is not a problem for lower-dimensional datasets, but we run out of space for higher-dimensional datasets.

The sequential access

To use less space, we propose to represent only the non-empty cells. Since we do not know in advance which cells are non-empty, we can not compute the index of a cell given its ID and, therefore, not access the cell using random access. Instead, we propose a sequential access representation, where we simply have a list of all non-empty cells.

Construction. To create the grid structure, we first need to identify all non-empty nodes. For each point p_i in parallel, we write which cell it resides within $gridId[i]$. For each point p_i in parallel, find the location j where the value matches the cell number of the point and mark it as used $gridIncl[j] := \mathbb{1}$. Compute an inclusive scan on *gridIncl* to obtain the index of all non-empty cells *gridIdx*. For each non-empty cell in parallel, move it to its new location. At last, we add the points to each cell's list of points *gridPoints* as in the random access grid structure.

Access time. Accessing a single cell implies that we must traverse all cells, which we can have at most n of, and we, therefore, get a $O(n \times d)$ time complexity. However, to perform a range query, we only need to traverse the cells once to find the ones within the range; therefore, the complexity is the same.

Space usage. The space usage becomes $O(n \times d)$ since we have at most n cells.

The mixed access

Both the random and sequential access grid structures have their advantages and disadvantages. The random access representation is fast to access for lower dimensional datasets but uses considerable space for higher dimensional datasets. On the other hand, the sequential access representation uses less space but is slower to access. We propose a representation that balances both by creating two levels of the grid structure. One that split the dataset into grid cells along the first d' dimensions using the random access representation and one that further split the cells along the rest of the dimensions using the sequential access representation. Note that instead of the random access level containing a list of points, it contains a list of grid cells of the sequential structure.

Space usage. To make the space usage of the first random access level the same complexity as the dataset, we pick d' such that $w^{d'} \leq n \times d$. Therefore, the space usage becomes $O(n \times d)$. The sequential access level already uses $O(n \times d)$; therefore, the total space usage becomes $O(n \times d)$.

Access time. The query time for the mixed access structure is also bounded by $O(n)$. To query this representation, we first identify the random access grid cells within the ε range; this takes $O(v^{d'}) = O(n)$. Next, we sequentially go through all non-empty full-dimensional cells within the lower-dimensional ones to find the overlapping; this also takes at most $O(n)$. At last, we can have at most n points in the result set, implying a $O(n)$ time. We, therefore, have a total time complexity of $O(n)$.

Compared to just having the dataset or an R-Tree, this grid structure does not improve the time or space complexity. However, our heuristic is that, most likely, the points will be distributed among a few cells, making it much faster in practice. Furthermore, we propose a strategy for summarizing the points in a cell if the range entirely includes that cell. As the iterations progress and the points become denser, it is likely that most points can be summarized this way.

Grid structured summarized statistics

We have concluded that it is impossible to lower the complexity of SynC by using an indexing structure because the neighborhood quickly becomes $O(n)$ points. We, therefore, look for ways to reduce the running time by not having to retrieve all the points in the neighborhood. We suggest reusing partial results that is common among multiple points. However, the challenge is that there are no obvious partial results to reuse since the update is a non-linear function. Still, instead of retrieving all points in a ε-neighborhood to compute the update and termination criterion, we propose to compute summarized statistics that we can use to compute the update. To do this, we must reformulate the update function into an equivalent that can use partial results. The main idea is to use the relationship $\sin(y-x) = \sin(y)\cos(x) - \cos(y)\sin(x)$ to reformulate it; for proof, see our paper. We propose the equivalent update function in Definition 4.6.3 that uses the summed *sin* for each dimension j of each point p in each grid cell g, $gridSin[g]_j := \sum_{p \in g} \sin(p_j)$, and analogically for *cos*, $gridCos[g] := \sum_{p \in g} \cos(p_j)$. We precompute these alongside the construction of the grid structure; this only takes $O(n \times d)$ time.

Definition 4.6.3 (EGGUpdate)**.** *Given a point $p^t \in data^t$ at iteration t and a grid structure grid, the point p is updated for iteration $t+1$ as:*

$$p_i^{t+1} := p_i^t + \frac{1}{|N_\varepsilon(p^t)|}\left(\cos\left(p_i^t\right)\left(\sum_{g \in GF} gridSin[g]_i + \sum_{g \in GP}\sum_{q \in g \cap N_\varepsilon(p)} \sin(q_i)\right)\right.$$
$$\left. - \sin\left(p_i^t\right)\left(\sum_{g \in GF} gridCos[g]_i + \sum_{g \in GP}\sum_{q \in g \cap N_\varepsilon(p)} \cos(q_i)\right)\right).$$

where GF is the set of grid cells fully intersecting the ε-neighborhood, and GP is the set of grid cells that only partially intersect.

GPU-parallelized execution of update

We now have the foundation to compute the update to the point using our grid structure and summarization. We call this update function EGGSynC update. When performing the update, we need to perform many range queries. These range queries can have a varying result size and, therefore, a varying workload.

For the random access level, there can be a lot of empty nodes; if threads in a warp handle empty nodes and non-empty nodes simultaneously, those handling the empty would need to wait until the others have finished. To reduce this wasteful behavior, we propose to decouple the random access and the traversal of the cells. We do this by identifying all the non-empty random access cells and saving the indices in a list, as in Section 2.2, and then use this list when performing the range query.

The size of each neighborhood can vary a lot between points, leading to an unbalanced workload. On the GPU, threads in a warp must perform the same operations at all times; therefore, if the workload is unbalanced, the threads with fewer points can make the other threads wait. Our summarization, of course, mitigate this by summarizing the dense centers of the range query, but for the early iterations, it still makes sense to consider this problem. To mitigate this problem further, we suggest letting threads in the same warp handle points more likely to have the same neighborhood. Observe that the points in the grid structure are ordered by the cell number (the location in the data). Handling the points in this order implies that threads are likelier to handle points in the same grid cell and, therefore, a similar neighborhood.

Summarized key contributions and insights

We propose a new GPU-parallelized algorithm for clustering by synchronization called EGG-SynC. To achieve much-needed speedup, we propose a summarization strategy that allows EGG-SynC to compute the update in each iteration faster without loss in accuracy. Furthermore, we propose a grid structure that supports the summarization strategy and range queries for the points that can not be summarized.

We identify an undesired behavior of the method SynC used to terminate that results in an inaccurate clustering. Therefore, we propose a new termination criterion and prove that when this is met, the correct clustering can be gathered. We compare EGG-SynC against the original SynC, FSynC, and both a straightforward CPU and GPU-parallelized version of SynC. EGG-SynC outperforms the CPU-based version by several orders of magnitude in all experiments. However, the straightforward GPU-parallelized SynC terminates faster in a few experiments. This is due to EGG-SynC running until the correct clustering has been found where SynC terminates preemptively.

Chapter 5

Subspace Clustering

As mentioned, the task of subspace clustering is to identify clusters in all subspaces of the full-dimensional space. The concept of subspace clustering sounds very promising, but identifying clusterings in all possible subspaces naturally comes with the drawbacks of a potentially huge result set and a runtime that can be exponential in the number of dimensions. In this chapter, we present the background for our contribution. First, we present the earliest exact density-based subspace clustering algorithm SUBCLU. Then we introduce the extension of SUBCLU, called INSCY, that aims to make the subspace clustering unbiased w.r.t. dimensionality and reduce the result set. At last, we present our contributions, GPU-INSCY, a GPU-parallelized variation of INSCY, including a GPU-parallelized indexing structure and an improved adaptation of DBSCAN.

5.1 SUBCLU

Likely due to the high time complexity, early subspace clustering methods only aimed to compute an approximative result like the grid-based subspace clustering algorithms CLIQUE [6], ENCLUS [24], and MAFIA [33]. To obtain an exact subspace clustering algorithm, Kailing et al. [43] propose SUBCLU. SUBCLU is a subspace clustering algorithm that uses the DBSCAN clustering algorithm to find the clustering for all possible subspaces. Instead of just naively clustering all combinations of subspaces, SUBCLU employs an Apriori-like strategy [5] to prune the points in higher-dimensional subspaces using the result of lower-dimensional subspaces. The idea is that, since the distance between points only increases for superspaces, the clusters can only be separated and never become larger. This is known as the monotonicity properties, Lemma 5.1.1, proven by Kailing et al. [43].

Lemma 5.1.1 (Monotonicity)**.** *Given neighborhood radius ε, number of points minPts, points $p, q \in data$, a cluster $C_i \subseteq data$, then the following monotonicity properties holds for all $T \subseteq S$:*

1. $|N_\varepsilon^S(p)| \geq minPts \Rightarrow |N_\varepsilon^T(p)| \geq minPts$

2. $dirReach^S(p,q) \Rightarrow dirReach^T(p,q)$

3. $denReach^S(p,q) \Rightarrow denReach^T(p,q)$

4. $denConnect^S(p,q) \Rightarrow denConnect^T(p,q)$.

More concretely, SUBCLU starts by clustering all 1-dimensional subspaces of the full-dimensional space. The s-dimensional clusters $\mathbb{C}_s$ and associated subspaces $\mathbb{S}_s$ are maintained and used for pruning the search for clusters. While SUBCLU still finds clusters in increasing sizes of subspaces s, SUBCLU generates candidate subspaces of size $s+1$ to be clustered. The candidate subspaces $cand \in cands_{s+1}$ are generated by combining each pair of $S_1 \in \mathbb{S}_s, S_2 \in \mathbb{S}_s$ that has $s-1$ common dimensions. For each candidate $cand \in cands$, SUBCLU then finds the subspace with the smallest amount of points in its clustering and performs DBSCAN on each of the clusters C_i in the candidate subspace. This way, the SUBCLU algorithm directly adapts the DBSCAN algorithm into the subspace clustering domain and provides an efficient pruning strategy.

5.2 INSCY

The main drawback of the SUBCLU algorithm is that it assumes a fixed density for all subspaces, which implies that it prefers lower dimensional subspaces, produces large result sets, and has a relatively high runtime. Through three incremental steps, Assent et al. [9–11] propose the algorithms DUSC, EDSC, and INSCY to mitigate these drawbacks. Here, we will describe the concept as one algorithm and refer to it as INSCY. The first paper [9] proposes the clustering definition, the second [10] proposes the overall algorithm and pruning techniques, and the last [11] proposes a data structure that supports this pruning scheme.

The INSCY algorithm address the density threshold $minPts$, that is biased towards lower dimensional subspaces, by defining what they call an unbiased-density Definition 5.2.1 to replace the definition of a core point in DBSCAN, Definition 4.2.1. The density threshold is made unbiased by normalizing it with the expected density of a subspace S:

$$expDen(S) := \mathbb{E}_S\left[|N_\varepsilon^S(p)|\right] = n \times \frac{c(S) \times \varepsilon^{|S|}}{v_S}, \tag{5.1}$$

where $c(S) := \pi^{\frac{|S|}{2}} / \Gamma\left(\frac{|S|}{2}+1\right), \Gamma(n+1) := n \times \Gamma(n), \Gamma(1) := 1, \Gamma(1/2) := \sqrt{\pi}$, and v_S is the volume of subspace S.

Definition 5.2.1 (Unbiased-density). *Given a point $p \in data$, a subspace S, a neighborhood radius ε, and density factor threshold F, the point p is unbiased-dense in the subspace S if:*

$$unbiasedDense^S(p) := |N_\varepsilon^S(p)| \geq \max(minPts, F \times expDen(S)). \tag{5.2}$$

In DBSCAN, a cluster can be of size *minPts* since it requires only one point to be a core point. Assent et al. [9] propose requiring a minimum cluster size min_C to reduce the number of small clusters found. Of course, the user still has the option to set $min_C := minPts$, if the user wishes to find the small clusters. This is introduced in the first variation [9] of the INSCY algorithm; in the last paper [11], this is used to prune points by removing density-connected regions, called subspace regions, with fewer than min_C.

DBSCAN clustering, Definition 4.2.5, uses the terms density-reachable, Definition 4.2.3, and density-connected, Definition 4.2.4. This implies an inconsistency where a border point can be connected to multiple clusters, but DBSCAN only assigns each to one. To fix this, Assent et al. [9] replace both definitions with a term they call S-connected, Definition 5.2.2, and require that all points should be unbiased-dense.

Definition 5.2.2 (S-connected)**.** *Given points $p_1, p_j \in data$ in subspace S, the points p_1, p_j is S-connected if:*

$$S\text{-}connected^S(p_1, p_j) := \exists p_1, \ldots, p_j : p_{i+1} \in N_\varepsilon^S(p_i) \forall i < j. \tag{5.3}$$

To reduce the size of the result set, Assent et al. [9] propose a notion of redundancy by introducing a redundancy factor *r*.

Definition 5.2.3 (Redundancy)**.** *Given a cluster C in subspace S and redundancy factor r, C is redundant if there exists a cluster C' in subspace S' where:*

$$redundant(C, S) := \exists C' \subseteq C, S' \supset S : |C'| \geq r \times |C|. \tag{5.4}$$

To incorporate all of these concepts, Assent et al [9] propose a new clustering definition based on DBSCAN, Definition 4.2.5.

Definition 5.2.4 (INSCY clustering)**.** *Given a dataset data, a redundancy factor r, a density factor threshold F, a minimum cluster size min_C, a minimum number of points minPts, and a neighborhood radius ε, a subspace clustering C is a set of clusters C_i in a subspace S_i when the following conditions hold:*

- $\forall p, q \in data : p \in C_i \wedge S\text{-}connected^{S_i}(p, q) \Rightarrow q \in C_i$
- $\forall p, q \in C_i : S\text{-}connected^{S_i}(p, q)$
- $\forall p \in C_i : unbiasedDense^{S_i}(p)$
- $|C_i| \geq min_C$
- $\neg redundant(C_i, S_i)$.

Density pruning

The goal of the unbiased-density Definition 5.2.1 is to allow for a lower density for higher-dimensional subspaces; this also implies that the monotonicity, Lemma 5.1.1, utilized by SUBCLU to prune the points in the superspaces, no longer applies. Therefore Assent et al. [9] developed a new pruning strategy that prunes using the density threshold of the full-dimensional space D; they refer to it as the weak-density, Definition 5.2.5. For weak-density, monotonicity applies and can therefore be used for pruning.

Definition 5.2.5 (Weak-density)**.** *Given a point $p \in data$, a subspace S, a minimum number of points minPts, a neighborhood radius ε, and a density factor threshold F, the point p is weak-dense in the subspace S if:*

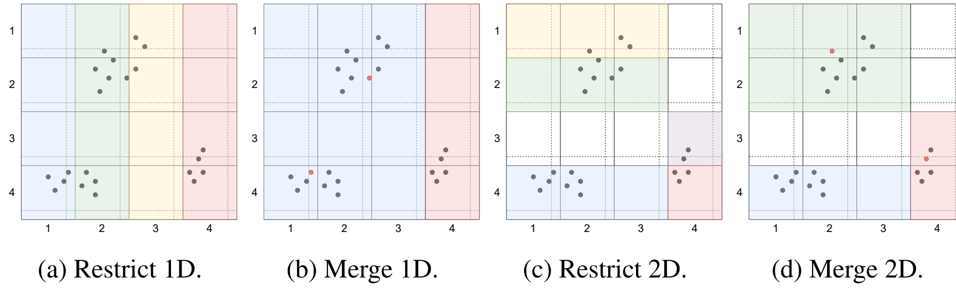
$$weakDense^{S}(p) := |N_{\varepsilon}^{S}(p)| \geq \max(minPts, F \times expDen(D)). \tag{5.5}$$

Using Definition 5.2.5 and Lemma 5.1.1, a weak-dense point can never become unbiased-dense for any superspace, and INSCY can therefore safely remove it before clustering any superspace.

Restricting and pruning subspace regions

Similarly, since the monotonicity does not hold for the unbiased-density, neither does it for clusters; they can, therefore, not be used to pruning the search for clusters in the superspaces. In the second paper, Assent et al. [10] instead propose bounding dense regions in the subspaces. Assent et al. [10] construct these subspace regions from a grid structure with an even cell width. A cluster is likely to span multiple cells, and the cells should, therefore, be merged into large density-connected subspace regions. INSCY checks a ε border at the edge of the cell. If there are no points within this border, there can be no ε-neighborhood connecting a cluster that goes across the two cells; hence it is safe to keep them as two separate subspace regions. If the opposite is the case, Assent et al. call them S-connected, and they merge the connected subspace regions one by one. Since the subspace regions are only defined by the ε-neighborhood, the monotonicity properties hold; it implies that the higher-dimensional subspace regions can be identified by restricting cells within the lower-dimensional subspace regions. The processing order starts in the 1-dimensional space and then recursively restricts the subspace regions to the $(k+1)$-dimensional subspaces.

We provide an example in Figure 5.1. In the first iteration, we restrict each one-dimensional cell; these are the first subspace regions. In Figure 5.1a, we see the grid structure restricted along the horizontal axis; this creates four cells, each marked with a color. INSCY merges them if there are points within the border between the subspace regions. In Figure 5.1b, there are points within the two borders of the first three subspace regions, and they are, therefore, merged into one. Next, the subspace regions are restricted along a second dimension and merged if there are points on the



(a) Restrict 1D. (b) Merge 1D. (c) Restrict 2D. (d) Merge 2D.

Figure 5.1: Restrict and merge example.

borders. In Figure 5.1c, the blue 1-dimensional subspace region is split into three regions, and the red is split into two. At last, in Figure 5.1d, the green and yellow subspace regions have a point on the border between them, and INSCY merges them into one. The same is true for the red and purple subspace regions.

After restricting and clustering in the superspaces of the current space, these subspace regions can be pruned for redundancy. An entire subspace region R can be prune if it contains less then the required cluster size min_C or if there already exists a cluster C' in a superspace where $|C'| \geq r \times |R| \wedge C' \subseteq R$.

The SCY-tree

In the third paper, Assent et al. [11] propose a tree-structure, called the SCY-tree, that maintains the restricted subspace regions, or rather, everything that has not yet been restricted. The idea is first to precompute a tree structure that splits the full-dimensional dataset into cells one dimension at a time. This information then allows for faster restriction of the subspace regions.

In the SCY-tree, a node has two attributes; the cell number and the number of points in the cell, represented by the subtree. The root node of the SCY-tree has no specific cell number, and the amount of points is the size of the dataset. The children of a node represent the further division of a cell into cells along a new dimension. This implies that all layers in the tree correspond to splitting the dataset into cells with increasing dimensionality. They add an extra sibling with that cell number but no count of points to keep track of S-connections. They furthermore add descendants to the S-connection node for each of the remaining dimensions, marking which of the cells the points creating the S-connection reside inside. This is to track where the S-connection is located along other dimensions.

Figure 5.2 shows a small example of a SCY-tree for a small 3-dimensional dataset before any restriction has occurred. The first layer is the root layer and consists of one root node. This layer represents the dataset that has not been split along any dimensions. The root node, therefore, contains no cell number, and the count of points is the entire dataset. The children of this node represent the first splitting of the dataset into cells along dimension 0. In the dataset, there are points in all cells along this dimension, and we, therefore, have a node for each cell, with the cell number and

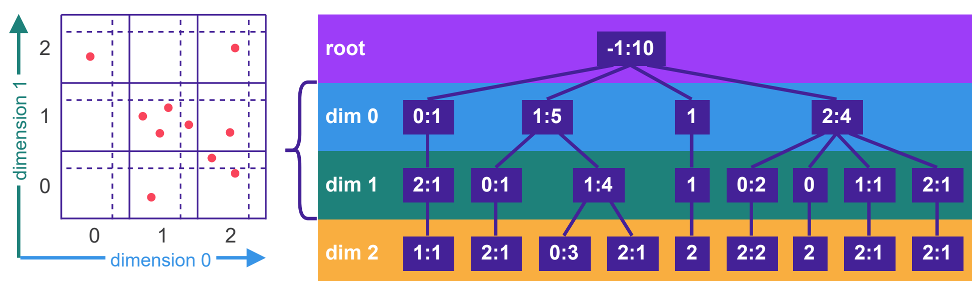


Figure 5.2: Example of the SCY-tree from our paper [39].

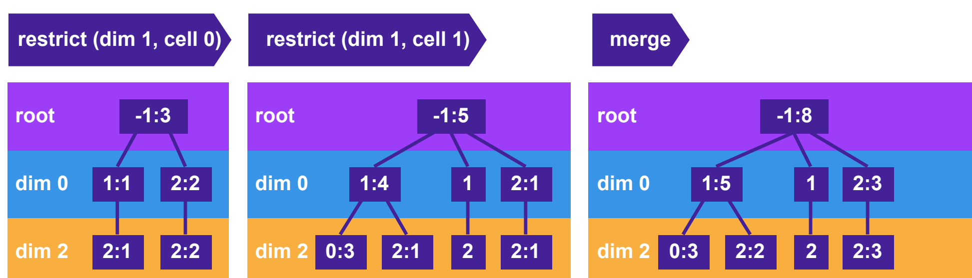


Figure 5.3: Example of restricting the SCY-tree to descriptor (1, 0) and then merge with the neighboring region. From our paper [39].

the number of points in this cell. Furthermore, in cell 1, there exists an S-connection, and we, therefore, add a sibling representing this. The next layer further represents the splitting of each cell, represented by nodes on the previous layer, into cells along dimension 1. For the first cell along dimension 0, when splitting along dimension one, there is only one cell with a single point; therefore, the corresponding node gets only one child with the cell number 2 and the amount of points 1. Similarly, when cell number 1 along dimension 0 is further divided along dimension 1, the points are located within two cells, and the corresponding nodes get two children. This continues for all cells until they have been split along all dimensions.

Restricting the subspace region, represented by the SCY-tree, to a cell ce along a new dimension d_r is done by identifying all nodes matching this descriptor (ce, d_r). For each node matching this descriptor, the part to the root and the subtree is copied to the new SCY-tree. If one of the matching nodes represents an S-connection, the restricted SCY-tree must be merged with its neighbor cell. Note that it is only the first node on an S-connection path that represents an S-connection. An example of such a restricted SCY-tree for descriptor $(1, 0)$ is shown in Figure 5.3. Since this descriptor matches an S-connection, the result must be merged with the neighboring cell.

Two SCY-trees are merged by combining the identical nodes into one and summing the count of points. Two nodes are identical if they have the same cell number and an identical path to the root node. For example, in Figure 5.3, the rightmost path in both restricted SCY-tree has the exact same cell numbers on the path from the root, and the

nodes on these paths are, therefore, merged into one.

Processing order

Each of the two pruning steps has an order to them. Pruning for weak-density implies that if a point is not even weak-dense, then it will never be unbiased-dense in any superspace, and it should, therefore, be pruned before proceeding with clustering in superspaces. On the other hand, pruning for redundancy requires that we know the size of the clusters in the superspaces; it should therefore be done after clustering the superspaces. This gives rise to the processing order in Algorithm 2. Here the idea is that INSCY restricts to a new subspace region, prunes for weak-density, call recursively to cluster all superspaces, prunes for redundancy, and if INSCY does not determine the subspace region as redundant, INSCY performs clustering on the subspace region.

Algorithm 2 INSCY$(scytree, d_f, data, d, r, F, minPts, \varepsilon, min_C, C)$
From our paper [39].

1: **for** $d_{re} := d_f$ **to** d **do**
2: **for** $c_{re} := 0$ **to** n_{cells} **do**
3: $SCY-tree' \leftarrow$ restrict$(scytree, d_{re}, c_{re})$
4: $scytree' \leftarrow$ mergeNeighbors$(scytree, scytree', d_{re}, c_{re})$
5: **if** prune_weak_density$(scytree', F, minPts, \varepsilon, min_C)$ **then**
6: INSCY$(scytree', d_{re}+1, data, d, r, F, minPts, \varepsilon, min_C, C)$
7: **if** prune_redundancy$(scytree', r, C)$ **then**
8: $C \leftarrow C \cup$ clustering$(scytree', data, F, minPts, \varepsilon)$

5.3 Our contribution: GPU-INSCY

The INSCY algorithm and subspace clustering definition have some excellent properties, and Assent et al. [11] have shown that it is faster than SUBCLU [43]. However, the same experiments show that INSCY still takes an hour to cluster a small dataset of $4,000$ points. The high runtime naturally brings the aim of a faster algorithm that can produce the INSCY clustering, and we, therefore, propose utilizing the GPU's computational power. However, the INSCY algorithm is not very suited for the GPU due to the processing order of the algorithms, the manipulation of the SCY-tree, and the clustering process. Therefore, our contribution is a GPU-parallelized dimensionality-unbiased density-based subspace clustering algorithm called GPU-INSCY. This includes a GPU-efficient clustering search and indexing structure.

The overview

The original processing order only allows for parallelization across one restriction and one clustering at a time. For subspaces of subspace regions, both the restrictions and

the clusterings are independent and ideal candidates for parallelizations. Therefore, we propose changing the processing order of INSCY for the GPU-INSCY Algorithm 3. We introduce the functions used in this algorithm in the following sections. Since the restriction of subspace regions are independent across dimensions and cells, we propose to do this as the first step Line 1. Since the neighborhood of each point is used both for clustering and pruning for weak-density, we precompute this in parallel across all subspace regions Line 2. We then prune for weak-density Line 6, call recursively for each subspace region Line 7, and prune for redundancy given the superspaces Line 8. When all subspace regions have been pruned, we identify clusters in each region in parallel Line 10.

Algorithm 3 GPU-INSCY($scytree'$, d_f, $data$, d, r, F, $minPts$, ε, min_C, R)
Algorithm from Jørgensen et al. [39].

1: $L \leftarrow$ GPU_restrict_and_merge($scytree, d_f, d$)
2: precompute_neighborhoods($data$, L, ε)
3: **for** $d_{re} \leftarrow d_f$ **to** $d - d_f$ **do**
4: $\quad C \leftarrow$ 1d array of size $|data|$ initialized to -1
5: $\quad$ **for** $\forall scytree' \in L[d_{re}]$ **do**
6: $\quad\quad$ **if** prune_recursion($scytree'$, F, $minPts$, ε, min_C) **then**
7: $\quad\quad\quad$ GPU-INSCY($scytree', d_{re}+1, data, d, r, F, minPts, \varepsilon, min_C$, R)
8: $\quad\quad\quad$ **if** prune_redundancy($scytree'$, r, R) **then**
9: $\quad\quad\quad\quad$ $L' \leftarrow L' \cup \{(scytree', C)\}$
10: $R \leftarrow R \cup$ GPU_clustering($L', data, F, minPts, \varepsilon$)

The GPU-SCY-tree

The original SCY-tree is a well-suited structure for the CPU; however, it is unsuitable for the GPU both in the way it is represented and in the way INSCY operates on it. The original SCY-tree is represented as multiple independent nodes allocated on the fly. However, allocating memory is an expensive operation, and if a thread in a warp spends a long time allocating, the other threads would need to wait until the thread has finished. Furthermore, we must also consider the access pattern when proposing a new representation. Restricting and merging the SCY-tree happens through sequential and alternating operations, where each operation also sequentially traverses the trees one node at a time. While this is a valid CPU strategy, this sequential approach does not fit well with the GPU. The original approach clearly has two sequential dependent levels: among descriptors and nodes. To make the restrict and merge operations suitable for the GPU, we need to tackle both levels and for both merge and restrict.

We propose a new indexing structure based on the SCY-tree but suitable for the GPU, called the GPU-SCY-tree. The decisions of how the GPU-SCY-tree is represented and operated on are very entangled since changing the representation affects how it is manipulated, and changing how it is manipulated affects the optimal representation. However, to simplify the discussion, we will first describe the representation

and then how the GPU-SCY-tree is manipulated. To make the restrict and merge operations suitable for the GPU, we will propose an entirely new operation that combines multiple operations into one. To simplify the abstraction, we will first describe how we compute restrict operations on the GPU in parallel across both descriptors and nodes. We then justify combining multiple restrict and merge operations into one parse and propose a method to do exactly that.

The GPU-SCY-tree representation

On the CPU, the SCY-tree is represented as nodes containing a parent pointer, children, and two attributes. This object-oriented representation is not well suited for the GPU since it requires many memory allocations and has a lousy memory alignment. Instead, we suggest a different representation that will dramatically reduce the number of allocations and allow for a more parallelized traversal of the tree doing restricting and merging. Furthermore, our representation also considers how memory is loaded and located in memory.

We propose to represent the tree structure as three arrays representing the nodes, three arrays representing the layer, and two arrays representing the points, each with an entry per node, layer, or point. The arrays representing the nodes are arrays with the index of the parent node of each node pa, an array with the cell numbers ce, and an array with the count of points in the subtree co. The arrays representing the layers are an array with the dimensions the layers represent $dims$, an array with the end index of the nodes on that layer la, and an array containing the dimensions that have been restricted r_{dims}. Lastly, the arrays representing the points are an array with the ID of the points po and an array with the index of the leaf node it is located in pl. We locate all nodes on the same layer next to each other to ensure memory alignment when accessing the layers of the tree concurrently. The GPU-SCY-tree representation implies that the layer of a node i is:

$$layerOf(i) := j | la[j-1] \leq i < la[j]. \tag{5.6}$$

An GPU-SCY-tree representation of the SCY-tree example shown in Figure 5.2 is provided in Figure 5.4. Here we see each layer color-coded as in the original SCY-tree. Entry 0 of the arrays pa, ce, co gives us the root node, with a parent pointer pointing to the node itself, the cell number of -1, and a count of points equal to 10. On the next layer, all nodes point to the root node, the next again, all points to nodes in the previous layer, and so on. The three layers correspond to all three dimensions, which are represented in $dims$ and at which entry the layer end is noted in the array la. This example has not been restricted to any dimensions yet; therefore, the r_dims are empty. At last, po shows the point IDs, and the corresponding entry in pl shows which nodes the points are placed in.

The GPU-SCY-tree restricting

Restricting the SCY-tree is a simple task on the CPU; the nodes matching the descriptor are identified, and the path above and their subtrees are copied to the restricted SCY-

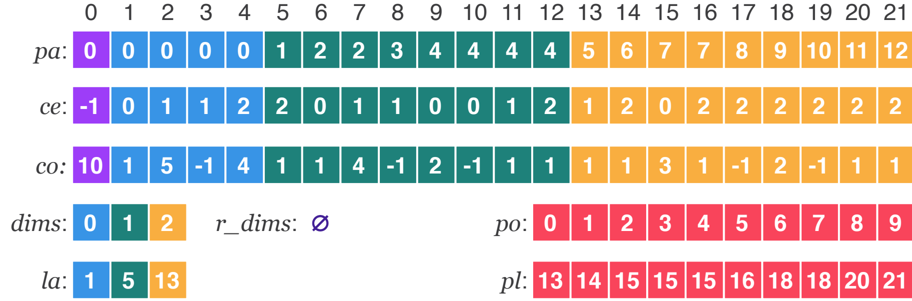


Figure 5.4: Representation of the GPU-SCY-tree, from our paper [39].

tree. However, this is a very sequential task, and it is unclear how to do this in parallel on the GPU. We observe that two levels of sequential dependencies exist when operating on the SCY-tree. For the descriptor-level dependency, restricting to each of the descriptors is entirely independent when looking at only the restrict operations. Therefore, we can compute them in parallel and, even better, within different thread blocks, making it easier to utilize all cores. For the node-level, we observe that determining which nodes are being included in the restricted SCY-tree occurs when the node matching the descriptor has been identified, and the path above and the subtree below are copied. This relationship implies that for the nodes on the layers above, the restricted dimension is included if the children are included, and for the layers below, if the parent is included. There is, therefore, a clear dependency between nodes on different layers. However, it also implies that there are no dependencies between nodes on the same layer, and they can be handled entirely in parallel. This gives rise to four different kernels; for the layer just below the restricted dimension, we check for each node if the parent node matches the descriptor; for the rest below, we check if the parent node is included. Similarly, for the layer just above the restricted dimension, we check if one of the children matches the descriptor, and for the rest above, we check if one of the children is included:

$$incl[j,c,i] := \begin{cases} cells[pa[i]] = c & \textbf{if } layerOf(i) = j-1 \\ incl[j,c,pa[i]] & \textbf{if } layerOf(i) < j-1 \\ \bigvee_{ch|i=pa[ch]} cells[ch] = c & \textbf{if } layerOf(i) = j+1 \\ \bigvee_{ch|i=pa[ch]} incl[j,c,ch] & \textbf{if } layerOf(i) > j+1 \\ \mathbb{0} & \textbf{else} \end{cases}, \tag{5.7}$$

where i is the node index, j is the dimension, and c is the cell number being restricted. However, performing a check for each child a node contains can lead to an unbalanced workload since the number of children can vary a lot from node to node. Instead, we propose changing the point of view such that each node on the layer below checks if the node matches the descriptor or is included and then marks the parent as included.

Alongside computing whether or not a node is included, we also accumulate the count of points in the subtree. Using the *incl* array, we compute the new indices of

the nodes *idx* using an inclusive scan. Using these indices, we can copy all included nodes from the GPU-SCY-tree to the new restricted GPU-SCY-tree. Furthermore, we also maintain which dimensions are restricted and which are not, at what index each layer starts, which nodes are contained in the tree, and in which node they reside. However, all this is relatively straightforward and not time-consuming to compute; for further details, see the original paper [39].

The GPU-SCY-tree restricting and merging at the same time

Whenever we encounter an S-connection while restricting the SCY-tree, the restricted region must be merged with the next neighboring region. As long as we identify S-connections, this alternation between restricting and merging continues creating a sequential process. This sequential and alternating process also implies many temporary memory allocations and computations. Furthermore, when merging SCY-trees, we need to merge nodes considered identical, which is when they have the same cell number and all their ancestors are identical. On the CPU, we can easily keep track of this by recursively visiting the two SCY-trees simultaneously, ensuring that if we visit two nodes at the same time, then they are identical. However, keeping track of the ancestors the same way on the GPU would imply only one active thread. In contrast, letting each thread compare its own pair of nodes would imply that we need to identify the ancestors to check if the nodes should be considered identical. Comparing all $O(n^2)$ pairs would imply much redundant work. Instead, we have the idea of sorting the nodes on each layer depending on the ancestors; this would quire only $O(n \times \lg(n))$ comparisons, but still of all the ancestors of the nodes. Neither solution provides substantial speedups. More concretely, the problem is that we need to compare ancestors across restricted SCY-trees. We aim to mitigate all of these problems simultaneously in this section.

Opposite to the restrict operations, the merge operations are dependent on each other since INSCY merges the previous result with the neighboring SCY-tree and, in this way, combines continuously restricted SCY-trees. To avoid this inherently sequential process, we suggest combining multiple restrict and merge operations into one operation. However, how to do this is not entirely clear. We first identify the single information that forces INSCY to merge the regions, making it sequential: the S-connections. To reduce the sequential work, we propose precomputing an array S with all the S-connections:

$$S[j,c] := \bigvee_{i \in [la[j-1], la[j])} c = cells[i], \tag{5.8}$$

and then, using only this small array, compute which cells GPU-INSCY should merge M; we call this the merge map:

$$M[j,c] := \begin{cases} M[j,c-1] & \textbf{if } c > 0 \wedge S[j,c-1] \\ c & \textbf{else} \end{cases}. \tag{5.9}$$

Therefore, the S and M arrays have an entry for each descriptor (pair of dimensions and cell number). The entries of S mark if there exists an S-connection for the associated descriptor, and M contains the cell number of the first restricted SCY-tree in the chain of restricted SCY-trees GPU-INSCY should merge. To achieve the highest parallelism, we do not check for each entry of S if there is an S-connection among the nodes. Instead, we check for all nodes; if an S-connection exists, mark it in the corresponding index.

Instead of only restricting the cells matching the descriptor, we propose restricting all cells that will merge into one region simultaneously, using the merge map M:

$$incl[j,c,i] := \begin{cases} M[j,ce[pa[i]]] = c & \textbf{if } layerOf(i) = j+1 \\ incl[j,c,pa[i]] & \textbf{if } layerOf(i) > j+1 \\ \bigvee_{ch|i=pa[ch]} M[j,ce[ch]] = c & \textbf{if } layerOf(i) = j-1 \\ \bigvee_{ch|i=pa[ch]} incl[j,c,ch] & \textbf{if } layerOf(i) < j-1 \\ \mathbb{0} & \textbf{else} \end{cases}, \quad (5.10)$$

where i is the node index, j is the restricted dimension, and c is the restricted cell number.

Removing a layer may still course new nodes to become identical; therefore, we still need to merge nodes. At the beginning of this section, we discussed several naive solutions that would be suboptimal. Instead, we propose to merge nodes while we restrict by maintaining only one representative node among the identical ones. New identical nodes can only be introduced under the restricted layer, and we already compute inclusion from that layer and downwards. If threads agree on which node represents the parent, then two nodes are identical if the representative parent nodes are the same and the current cell number are identical. However, we use a different approach to avoid comparing all pairs of nodes. Instead, we keep an array of nodes representing the new children n_ch of different cell numbers. When we have this information, we can use this to identify the representative node:

$$rep(j,c,i) := n_ch[j,c,n_pa[j,c,i],ce[i],sCon(i)), \quad (5.11)$$

where we use $sCon$ to distinguish between S-connections and ordinary nodes. Since we only want to include the representative node and not the other identical nodes, the definition of when a node is included becomes:

$$incl[j,c,i] := \begin{cases} M[j,ce[pa[i]]] = c & \textbf{if } layerOf(i) = j+1 \\ incl[j,c,pa[i]] & \textbf{if } layerOf(i) > j+1 \\ \bigvee_{ch|i=pa[ch]} M[j,ce[ch]] = c & \textbf{if } layerOf(i) = j-1 \wedge rep(j,c,i) = i \\ \bigvee_{ch|i=pa[ch]} incl[j,c,ch] & \textbf{if } layerOf(i) < j-1 \wedge rep(j,c,i) = i \\ \mathbb{0} & \textbf{else} \end{cases}, \quad (5.12)$$

where i is the node index, j is the restricted dimension, and c is the restricted cell number. Notice that only nodes below the restricted dimension may have duplicates.

We refer to Equation 5.10 as the nodes that should be included or represented, and Equation 5.12 as the nodes that are actually included and represent the other identical nodes.

To compute the representative children, we observe that only one child of each cell number can be included; however, it also does not matter which one. Therefore, for each node in parallel, we compute if the node should be included, Equation 5.10, in the restricted SCY-tree, we write its index as the representative child. Since multiple threads can write the node index as the representative, we have no guarantee which index is there at the moment. However, it does not matter which one is the representative, but all threads must agree on which, and we synchronize all threads to ensure they see the same index.

Parents. After restricting the GPU-SCY-tree, some nodes may need to be assigned a new parent. For the included nodes above the restricted dimension j, the new parent is the same parent as before. For the included nodes just below the restricted dimension j, the new parent is the previous grandparent. For the rest of the included nodes below the restricted dimension j, the new parent is the representative node of the old parent:

$$n_pa[j,c,i] := \begin{cases} pa[i] & \textbf{if } layerOf(i) < j \\ pa[pa[i]] & \textbf{if } layerOf(i) = j+1 \\ rep(j,c,pa[i]) & \textbf{else} \end{cases}, \tag{5.13}$$

where i is the node index, j is the restricted dimension, and c is the restricted cell number.

Counts. When copying the nodes that are included in the restricted SCY-tree, a new count of points in each subtree must be computed. The count of points in a subtree is the sum of points in the subtree of the children:

$$n_co[j,c,i] := \begin{cases} \sum_{ch|i=n_pa[ch]\wedge\neg sCon(ch)} n_co[j,c,ch] & \textbf{if } layerOf(i) < j \wedge \neg sCon(i) \\ co[i] & \textbf{if } s_incl[j,c,i] \wedge \neg sCon(i) \\ -1 & \textbf{else} \end{cases}, \tag{5.14}$$

where i is the node index, j is the restricted dimension, and c is the restricted cell number. This information is, therefore, propagated from the leaf layer op wards.

Copy to the new restricted SCY-tree. We create each final restricted and merged GPU-SCY-tree starting at descriptor with dimension j and cell number c:

$$pa'[idxs[i]] := n_pa[j,c,i] \quad \forall i|incl[i] \tag{5.15}$$

$$co'[idxs[i]] := n_co[j,c,i] \quad \forall i|incl[i] \tag{5.16}$$

$$ce'[idxs[i]] := ce[i] \quad \forall i|incl[i]. \tag{5.17}$$

Moreover, we similarly maintain the restricted and not-restricted dimensions, the layers end-index, the included points, and the nodes they are included within.

Pruning

The pruning phases are neither time-consuming nor the most interesting algorithmically. However, to not create new bottlenecks, we GPU-parallelize these as well.

Pruning for weak-density and minimum cluster size. To prune for weak-density, we compute in parallel for each point if it is weak-dense. We then, layer by layer, propagate the count upwards in the GPU-SCY-tree by following the parent pointers. This is parallelized across nodes on each layer following the same logic as in Section 5.3. The root node at the end contains the total count of points in the subspace region, and we can prune the entire subspace region if it contains less than the minimum required cluster size min_C.

Pruning for redundancy. To prune w.r.t redundancy, we identify the cluster size, the cluster overlapping with the current subspace region, and the largest cluster overlapping with the subspace region. GPU-INSCY does this in three different kernels. If the number of points in the cluster exceeds the number of points in the GPU-SCY-tree times the redundancy factor r, then clustering is not performed.

Clustering using an adaptation of G-DBSCAN

INSCY uses a clustering definition very similar to that of DBSCAN [29]. We, therefore, propose to adapt a GPU-parallelized variation of DBSCAN analogically. In Section 4.3, we concluded that G-DBSCAN is the fastest GPU-parallelized variation of DBSCAN and, therefore, the one that we adapt to fit the clustering definition of INSCY. G-DBSCAN has not been optimized w.r.t. it is being used as a subroutine in subspace clustering. This implies that there could still exist several relationships that can be leveraged to make G-DBSCAN faster. We investigate this and suggest several improvements that make clustering much faster in our scenario.

G-DBSCAN continuously check for all points in parallel to see if they can expand a specific cluster through the immediate neighborhood. While this strategy is valid and leads to speedup, a lot of extra computations still go to waste. We observe that this process happens for each cluster that is being expanded. Instead of just expanding one cluster at a time, we propose expanding all clusters simultaneously, potentially saving a factor equal to the number of clusters k. To do this, we propose Algorithm 4. The algorithm provided here is from our paper [39], but the paper do not provide pseudo-code. The idea is to assign each point to a singleton cluster, then for each point, and while there are still clusters being expanded, we go through the neighborhood of each point and check if it can expand any cluster with a lower cluster ID. As an additional benefit, we only transfer the entire clustering once instead of transferring an array for each cluster. The running time of Algorithms 1 and 4 are both dominated by the $O(n^2)$ time complexity of computing the neighborhoods. The improvements from Algorithm 1 to Algorithm 4 are therefore only significant when the time-complexity of the neighborhoods can be reduced.

We have further improvements that leverage that the clustering is being performed as part of the INSCY subspace clustering. First, we can use previously computer

Algorithm 4 G-DBSCAN*$(data, \varepsilon, minPts)$

```
1: Precompute neighborhoods
2: for all regions R - in parallel do
3:    for point p_i ∈ R - in parallel do
4:       C_R[i] := i
5:       while still expanding do
6:          C_R[i] := min_{p_j ∈ N_ε^S(p_i) | |N_ε^S(p_j)| ≥ minPts} C_R[j]
7:    copy each C_R to host
```

neighborhoods to prune new neighborhoods. Starting at the 1-dimensional subspace regions, we compute the neighborhoods as G-DBSCAN. For subspace regions with two or more dimensions, we have already computed the neighborhoods of a lower subspace, and we can utilize the monotonicity of the neighborhoods in a subspace to prune the points in the current subspace region. This provides a huge speedup but can only be used when DBSCAN is used in a subspace clustering scenario. The second and most obvious improvement is that we can parallelize across multiple subspace regions. Keeping previous neighborhoods in memory uses extra space and is, therefore, a trade-off between space and speed. To remedy this, we can either off-load memory from the GPU to the main memory or compute across fewer regions in parallel.

Summarized key contributions and insights

The most important, but also the most space-vise expensive, idea is to use the monotonicity of neighborhoods to prune the computation of the neighborhoods in a superspace using the neighborhoods in a subspace. This pruning strategy, of course, only applies when DBSCAN is used in the subspace clustering and could be used to speed up similar subspace clustering algorithms, e.g., to speed up variations of SUBCLU that use G-DBSCAN to cluster. The neighborhood computations dominate the runtime of G-DBSCAN and not the in-efficient gathering of clusters; our adaption would not lead to significant speedups unless the neighborhoods can be efficiently pruned.

The clustering clearly dominates the running time of INSCY; however, our adaptation of G-DBSCAN provides the clustering within a must shorter time. Therefore, handling the SCY-tree becomes the new bottleneck and requires a lot more effort to reduce. We first propose a GPU-friendly representation of the SCY-tree, called the GPU-SCY-tree, that takes the memory alignment and the manipulation into account. Furthermore, restricting and merging the GPU-SCY-tree comes with multiple challenges. During the merge operation, it is unclear how to merge identical nodes best since we need to keep track of which nodes are identical or identify them on the fly, likely increasing the workload. Alternating between restricting and merging also creates a sequential process ill-suited for the GPU. Furthermore, this alternating process creates many redundant allocations of temporary results and parses through

the tree. We suggest a method that allows us to combine the alternating process of restricting and merging into one operation and easily keep track of the nodes that need to be merged.

Putting all of our contributions together creates the GPU-INSCY algorithm, which provides an ever-increasing speedup of four orders of magnitude. This implies that the GPU-INSCY can cluster a million points within 20 minutes, whereas it takes more than 10 hours for INSCY to cluster just $8,000$ points. In experiments, we also provide a variation of GPU-INSCY that does not prune using the monotonicity of the neighborhoods. This shows that the GPU-parallelizations, including our GPU-SCY-tree and adapted clustering method, provide around a constant $500\times$ to $1,000\times$ speedup, but that the pruning of neighborhoods provides up to an extra $30\times$ speedup which seems to continuously increase as the number of points grows, implying a total of $15,000\times$ speedup.

Chapter 6

Projected Clustering

The result set of subspace clustering algorithms can become exponentially large, which is one of the problems that INSCY also tries to mitigate. However, we are sometimes only interested in having one clustering assignment per point to make the result as interpretable as possible. Projected clustering is similar to regular clustering in that it assigns each point to one cluster; however, each cluster is projected within a subspace of the full-dimensional space. In this chapter, we first provide the background for our contribution; GPU-FAST-PROCLUS and AVID. This is primarily the axis-aligned projected clustering algorithm, PROCLUS. We then discuss our contributions. Our contributions, GPU-FAST-PROCLUS, makes PROCLUS fast enough for real-time interaction using the GPU, and AVID demonstrates this as a data visualization tool where the entire visualization pipeline is implemented on the GPU to make it responsive enough.

6.1 PROCLUS

PROCLUS [4] is an axis-aligned projected clustering algorithm. An axis-aligned projection is where either a dimension is fully included or excluded. This stands in contrast to non-axis-aligned projection, where the projected dimensions are linear combinations of the dimensions in the full-dimensional space. That PROCLUS identifies axis-aligned subspace projections implies that the result is easier to interpret for the user, but at the cost of being less flexible than non-axis-aligned projected clustering algorithms [3, 15, 36, 68]. Even though PROCLUS is relatively old, to the best of our knowledge, there does not exist a better performing axis-aligned projected clustering algorithm.

In Jørgensen et al. [41], we reformulate the cost to make it more parallelizable. However, this also reveals that the cost-function that PROCLUS aims to minimize is similar to that of k-means but uses the Manhattan segmental distance given the subspace projection D_i of a cluster C_i instead of the Euclidean distance.

Definition 6.1.1 (PROCLUS cost function)**.** *Given a dataset data, a clustering C, and subspace projections D for the clusters, the cost of PROCLUS is as follows:*

$$cost(data,C,D) := \frac{1}{n}\sum_{C_i\in C}\sum_{p\in C_i}\frac{1}{|D_i|}||p-\mu_i||_1^{D_i}, \tag{6.1}$$

where $\mu_{i,j} := \frac{1}{|C_i|}\sum_{p\in C_i} p_j$ *is the mean of cluster* C_i.

However, the complexity of identifying the best clustering and subspace projections becomes even worse since we now also need to optimize the subspace projection. PROCLUS approximate the cost function Definition 6.1.1 using a random search approach. Given parameters, number of cluster k, average number of dimensions per subspace l, scalars A and B, the patient $itrPat$, and minimum derivation $minDev$, PROCLUS proceeds in three phases: initialization, iterative, and refinement.

In the initialization phase PROCLUS first select a random subset S of size $A\times k$ of the dataset $data$. Then pick a potential set of medoids M as a subset of size $B\times k$ from S by iteratively selecting the point $m_i \in S$ with the first distance to all points M already selected. From the potential set of medoids M, PROCLUS picks a random initial set of current medoids $MCur \subseteq M$.

PROCLUS then proceeds to the iterative phase where the current set of medoids $MCur$ is iteratively improved. In each iteration, PROCLUS computes a sphere of influence L_i for each current medoid $m_i \in MCur$. From these spheres of influence, PROCLUS estimates the best subspace projection D_i with an average of l dimensions. Within these subspaces, PROCLUS assigns each point to the closest current medoid. The cost of this clustering is evaluated; if a better clustering is found, PROCLUS keeps it for later use. At last, PROCLUS construct the next current medoids $MCur$ by replacing the medoids $m_i \in mBest$ where $|C_i| < n/k\times minDev$; if no such exists the medoid with the smallest cluster is replaced.

The sphere of influence L_i is computed using `computeL` as:

$$L_i := \{p \in Data \mid ||p-m_i||_2 \leq \delta_i\}, \tag{6.2}$$

where $\delta_i := \min_{m_j\neq m_i}||m_i-m_j||_2$.

The subspace projections are estimated using `FindDimensions` and proceed in multiple steps. First, computing the average distance $X_{i,j} := \frac{1}{|L_i|}\sum_{p\in L_i}|p_j - m_{i,j}|$ to the medoid m_i with in each sphere of influence L_i along each dimensions j. Then compute the average distance $Y_i := \frac{1}{d}\sum_{j:=1}^{d} X_{i,j}$ for each medoids m_i and the standard deviation $\sigma_i := \sqrt{\frac{1}{d-1}\sum_{j:=1}^{d} X_{i,j}}$. At last, compute a measure of spread $Z_{i,j} := \frac{X_{i,j}-Y_i}{\sigma_i}$. PROCLUS then picks the pairs of medoids m_i and dimensions i with the lowest Z_i, j value to be the subspace projection D_i; however, PROCLUS pick at least two dimensions per medoid.

PROCLUS construct the clustering C using `AssignPoints`; assigning each point p to the cluster m_i within subspace projection D_i:

$$C_i := \{p \in data \mid ||p-m_i||_2^{D_i} \leq \min_{m_i\in m}||p-m_i||_2^{D_i}\}. \tag{6.3}$$

PROCLUS then evaluates the cost Equation 6.1 of clustering C; if the current clustering is better than the previously found, PROCLUS saves it as the best clustering together with the associated medoids and subspaces. The iterative phase stops if no better clustering has been found for *itrPat* iterations. Before continuing to the next iteration, PROCLUS replaces bad medoids with random medoids from M. The bad medoids are the medoid associated with the smallest cluster and medoids associated with a smaller cluster than $n/k \times minDev$.

The final clustering is at last computed in the refinement phase. The best clustering *CBest* is used as the sphere of influence to compute the final subspace projections using `findDimensions`. The points are re-assigned to clusters using these subspaces. At last, the outliers are identified by defining spheres for each medoid in the associated subspace with radius $\Delta_i := \min_{j \neq i} ||m_i - m_j||^{D_i} / D_i$; points that do not lay within any spheres are considered outliers.

6.2 Our contribution: GPU-FAST-PROCLUS

Even though PROCLUS is becoming old, it still performs well in evaluations [58]. However, PROCLUS can still take several minutes to perform on just a few thousand data points [4, 58]. Applying PROCLUS in an interactive scenario on hundreds of thousands or even millions of points is impossible. We, therefore, propose GPU-FAST-PROCLUS in Jørgensen et al. [41]. GPU-FAST-PROCLUS includes both algorithmic and GPU-parallelization strategies that accelerate PROCLUS. Even though the focus is on the most time-consuming functions, the entire PROCLUS algorithm has been GPU-parallelized to reduce memory transfer.

Strategies

We propose four algorithmic strategies to speed up PROCLUS and provide space-saving variations. We call the algorithm using the original strategies FAST-PROCLUS and the algorithm using the space-saving variations FAST*-PROCLUS.

Strategy 1. In `ComputeL`, the distances between all points and the current medoids are computed. Computing the distances takes $O(n \times k \times d)$ but seems necessary since the medoids change in each iteration. However, we observe that not all medoids are replaced in each iteration but that some medoids will reappear, implying that many of the distances are recomputed in later iterations. Therefore, as our first strategy, we propose to keep these distances in case the medoids reappear.

Strategy 1*. This strategy requires $O(n \times B \times k)$ space. Therefore, we propose a space-saving variation of this strategy that only saves the distances from the previous iteration, requiring only $O(n \times k)$ space. The space-saving variation likely will not lose too much speedup since the only medoids that are not kept only reappear if they are randomly chosen.

Strategy 2. In `FindDimensions`, the new best subspace projections are computed. The most expensive part is to compute the average distances $X_{i,j}$ to the medoid m_i to all points in the sphere of influence L_i along each dimension j. Since the current

medoids changes in each iteration, so do L_i and $X_{i,j}$. While all L_i are likely to change in each iteration, they are not likely to change much. We propose to, instead of saving the spheres of influence L_i, only save the change in the sphere:

$$\Delta L_i := \{p \in Data \mid \delta_i^{t'} < ||p - m_i||_2 \leq \delta_i^t \vee \delta_i^{t'} \geq ||p - m_i||_2 > \delta_i^t\}, \tag{6.4}$$

where t is the current iteration and t' is the previous iterations where the specific medoid was used. We then maintain the sum of distances to the medoid m_i:

$$H^t_{MIdx_i,j} = \sum_{p \in L_i} |p_j - m_{i,j}|, \tag{6.5}$$

where $MIdx_i$ is the index of medoid m_i into M. When we have $H^t_{MIdx_i,j}$ we can compute $X_{i,j} := H^t_{MIdx_i,j}/|L_i|$. We can update $H^t_{MIdx_i,j}$ to the current iteration t as:

$$H^t_{MIdx_i,j} := H^{t'}_{MIdx_i,j} + \lambda_i \times \sum_{p \in \Delta L_i} |p_j - m_{i,j}|, \tag{6.6}$$

where λ is 1 if the change is an increase and -1 if it is a decrease.

Strategy 2*. Again this requires $O(n \times B \times k)$ space; therefore, we again propose a space-saving variation that builds on the same idea of only saving the information for the k current methods $MCur$.

Strategy 3. Like any other projected clustering or subspace clustering algorithm, PROCLUS must be provided parameters that define the outcome. For PROCLUS, the most important parameters are the number of clusters k and the average number of dimensions per subspace l. Projected and subspace clustering algorithms are often used in an information retrieval process, where the user investigates different parameter settings to find the best clustering. Each time the user runs PROCLUS, everything is computed again. This implies that the temporary results from strategies 1 and 2 are not used. One reason the temporary results can not be reused is that the potential medoids M are picked at random for each run. We, therefore, suggest keeping the same potential medoids M across parameter settings, allowing us to reuse the computations from previous runs. The only side effect is that the size of M is fixed even though it is normally sub-sampled to be of size $B \times k$. This restricts the choice of B and the size A of the downsampling. However, this downsampling is already a heuristic speedup, and our speedups allow the user to pick a larger B.

Strategy 4. For each parameter setting, we start from a random initialization of the current medoids $MCur$, implying that we spend some time doing a random search until we start finding good medoids. However, we suggest the heuristic that a good set of current medoids for one parameter setting might also be a good initialization for a different parameter setting. We simply let the following parameter setting continue on the same set of current medoids. When the number of medoids increases or decreases, we randomly remove or add medoids.

FAST-PROCLUS uses strategies 1, 2, 3, and 4, and FAST*-PROCLUS uses strategies 1*, 2*, 3, and 4. The same is analogically the case for the GPU-variation, GPU-FAST-PROCLUS, and GPU-FAST*-PROCLUS.

GPU-Parallelization

So far, we have only discussed algorithmic improvements. However, there are also several considerations to make regarding GPU parallelization. The entire algorithm is parallelized for the GPU; however, a couple of functions are the most time-consuming steps as they use $O(n \times d \times k)$ in each iteration. These functions will be the focus here and include ComputeL, FindDimensions, AssignPoints, and EvaluateCluster.

ComputeL. To make the computation for Equation 6.2 fit the architecture of the GPU, we identify three tasks that each can be computed independently across points. First, we compute all pair-wise distances in parallel among all points and medoids. Next, in parallel across pair of medoids, we find the smallest distance δ_i. At last, in parallel across each pair of points p and medoids m_i, we assign each point to the sets L_i where the distance to the medoid m_i is less than δ_i. The assignment is performed in parallel by atomically incrementing a location, and the point is saved at that location. This guarantees a unique location for each point. To accommodate strategies 1 and 1*, in algorithms GPU-FAST-PROCLUS and GPU-FAST*-PROCLUS, the computation of the distances is only computed if it has not been computed before. To accommodate strategy 2 and 2*, in algorithms GPU-FAST-PROCLUS and GPU-FAST*-PROCLUS, the old $\delta_i^{t'}$ from previous iterations is saved, and the conditions for which points are saved is changed to:

$$\delta_i^{t'} < ||p - m_i||_2 \leq \delta_i^t \vee \delta_i^{t'} \geq ||p - m_i||_2 > \delta_i^t. \tag{6.7}$$

Furthermore, we also record if it is an increase or decrease λ_i.

FindDimensions. Finding the dimensions are already nicely separated into several small steps. Each of these steps contains mostly independent computations that we can easily parallelize a cross and use atomic operations to sum across threads. However, since so many threads are accessing the same memory address at the same time, we keep a temporary result for each thread and then, in the end, combine the temporary results using the atomic addition operation. Computing Y requires X, σ requires Y, and Z requires σ. We combine these three computations into one kernel to avoid transferring memory to and back from global memory. To avoid expensive memory transfer between global and shared memory, we propose to combine the computations of Y, σ, and Z into one kernel. To accommodate strategies 2 and 2* in algorithms GPU-FAST-PROCLUS and GPU-FAST*-PROCLUS, the matrix H is updated using ΔL instead of recomputing the entire X.

AssignPoints. The points are assigned to the closest medoids, within the subspace, in parallel across both points and medoids. This is done by computing the distance from each point to all current medoids within the same thread block, using the atomic minimum operation. Then synchronizing each thread block to ensure that distances have been computed when checking the minimum distance. At last, assign it to that medoid if it is the one with the smallest distance.

EvaluateCluster. The original formulation of the cost function in PROCLUS consists of several parts, Equation 6.1. Separately computing each part would imply saving the temporary results to the slow global memory. We, therefore, propose to

reformulate the cost function to a single sum across clusters, points, and dimensions, to achieve higher parallelism:

$$\sum_{i}^{k}\sum_{p\in C_i}\sum_{j\in D_i}\frac{|p_j-\mu_{i,j}|}{|D_i|\times n}. \tag{6.8}$$

However, the centroid $\mu_{i,j}$ is still a separate part. We notice that the centroid $\mu_{i,j}$ is computed across clusters, points, and dimensions, like the cost. We, therefore, propose to compute it in the same kernel to avoid even more reads and writes to global memory. Furthermore, to not synchronize the whole GPU, but just each thread block, we parallelize across clusters and dimensions in blocks and points across threads within each block. This implies that we can compute $\mu_{i,j}$, synchronize, and then compute the cost within the same kernel call. When summing $\mu_{i,j}$ and the cost, we keep a temporary result tmp for each thread to reduce the use of atomic operations.

Summarized key contributions and insights

In this section, we propose several heuristic strategies for improving the running time of PROCLUS. Each of these provides additional speedup to both PROCLUS and our GPU-parallelized variation. However, what provides the greatest speedup is the GPU-parallelization of each function. Where the algorithmic strategies provide 1-order magnitude speedup, the parallelizing of everything provides 3-orders of magnitude speedup. Each function requires its own strategy to perform well and achieve this speedup. However, especially reducing memory transfer provides great speedups but often requires some good ideas to be successful.

6.3 Our contribution: AVID

We now have an algorithm that performs axis-aligned projected clustering on a million data points within a $1/10$ second. However, PROCLUS and other clustering algorithms are rarely just used as a background process but instead as part of an information retrieval task. Before, when performing PROCLUS on such large datasets, the user would need to wait for minutes before actually seeing the result, whereas GPU-FAST-PROCLUS now provides a result fast enough to allow real-time interaction [74]. However, to the best of our knowledge, there does not exist a data visualization framework for visualizing data already located on the GPU. Using standard data visualization frameworks would imply that throughout the visualization pipeline [35] we would perform the analysis on the GPU, transfer the result to the CPU where the visual mapping is performed, and then back to the GPU to be rendered on the screen. The CPU in this setup would clearly be a bottleneck that would make the interactive experience tedious at best. To actually support a real-time data visualization tool for GPU-FAST-PROCLUS, we propose to use the idea of "what happens on the GPU, stays on the GPU", i.e., perform the entire visualization pipeline on the GPU without the data ever leaving the GPU.

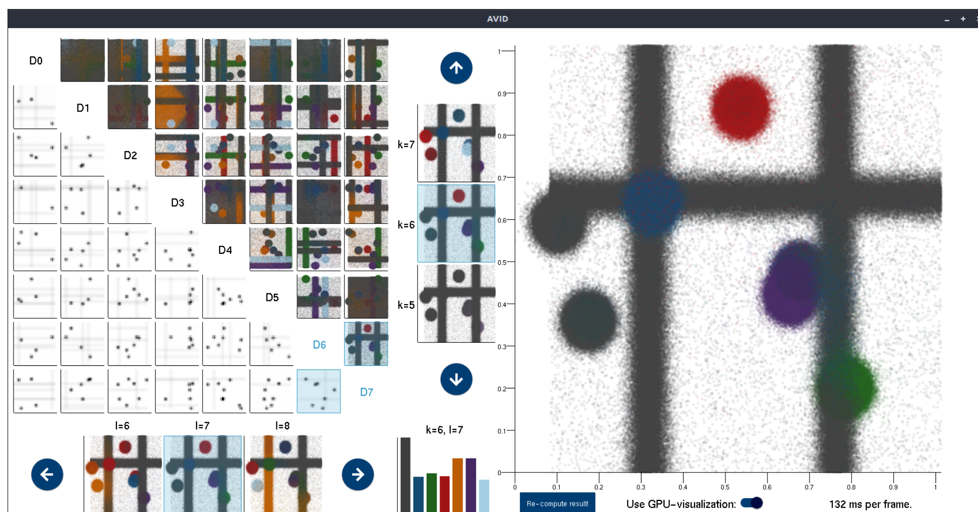


Figure 6.1: Layout of the data visualization AVID. From Jørgensen et al. [40].

Implementation. The visualization is implemented using CUDA[1] and OpenGL[2], together with the extensions GLUT[3] and GLEW[4]. OpenGL, GLUT, and GLEW are APIs for graphics rendering on the GPU. Furthermore, OpenGL allows for interpolation between CUDA and OpenGL resources, which implies that CUDA kernels can manipulate graphics. However, these are not frameworks for data visualization but only handle the rendering. Through GPU-FAST-PROCLUS, we also have the data analysis on the GPU. Therefore, the remaining is the filtering and mapping that must be implemented on the GPU. Furthermore, all interactions have also been implemented; however, these just instruct which filtering, selection, and mapping should be performed on the GPU.

Visualization. For the visualization, we propose to have a multiple-coordinated view with a scatter-plot-matrix as an overview, previews of changes in parameters, a column chart with the distribution of points among clusters, and a detailed view, Figure 6.1. In the overview, the user can select a specific scatter plot to show in the detailed view and the different previews. A scatter-plot matrix upper and lower triangle shows the mirrored image of the same information. We have therefore replaced the scatter plots, in the lower triangle, with heat maps to show the dense regions clearly. In the detailed view, the user can filter to specific points by brushing an area, and in the column chart, the user can filter to a specific cluster by clicking the corresponding column. There are two previews, one for the parameter k and one for l. Each consists of three scatter plots that show the selected scatter plot but at varying parameter settings. The parameters shown in the previews can be adjusted by clicking either a

[1] https://developer.nvidia.com/cuda-downloads
[2] https://www.opengl.org/
[3] http://freeglut.sourceforge.net/
[4] http://glew.sourceforge.net/

bottom to increase or decrease. The user can change the parameters used in the entire visualization by clicking on one of the scatter plots in the previews.

Demonstration. We provide a demonstration of our tool both as a textual demonstration in the paper and a video presentation at: `https://au-dis.github.io/publications/AVID/`. This demonstration shows how the user can quickly and easily fine-tune the parameters until the best clustering has been found. Doing the parameter fine-tuning, we show how a user can investigate a single cluster to see if it, along some dimensions, should be split into multiple clusters, increase k, if the subspace does not include a dimension where it is dense, increase l, or if the subspace includes a dimension where the cluster is not dense, decrease l. In contrast, if a cluster is not found or is split into multiple, the user can filter the dense area by brushing in the detailed view. If this confirms that multiple clusters should indeed be just a single cluster, decrease k, and if the cluster does not exist, increase k.

Chapter 7

Discussion and future work

Many supervised machine learning algorithms are based on linear algebra, which in most cases have been efficiently parallelized on the GPU. Several plug-and-play frameworks have been developed and integrated into the most used programming languages for data analysis, like python. However, there is still a lack of data mining (unsupervised machine learning) algorithms that are transformed to fit the computational model of the GPU. This is likely because many data mining algorithms are not based on linear algebra and are, therefore, not as low-hanging fruit to adapt to the GPU. The development of such algorithms enables fast analysis of big data on modern consumer-end hardware, without the need for high-end servers, but of course, also reduces the running time significantly on servers equipped with GPUs. In our data visualization tool AVID, we demonstrate how fast the interaction becomes for GPU-FAST-PROCLUS. A fast projected clustering could be beneficial in biological or medical research, where it could be used to identify groups with common genes, e.g., to investigate gene-specific diseases. However, similar scenarios could be imagined for most data mining tasks.

We, alongside others within the field of data mining, have started to develop algorithms that fit the GPU; however, there is still much work to be done before even the most used data mining algorithms have been adapted to fit the GPU. Some of the most used clustering algorithms have been adapted to fit the GPU, like DBSCAN [7, 17, 52, 53], k-means [16, 83], mean-shift [87], OPTICS [55], DPC [48], and now SynC [42]. However, for subspace clustering and projected clustering, even fewer have been developed. For subspace clustering, to the best of our knowledge, only MAFIA [1] and now INSCY [39] are adapted to the GPU. Furthermore, for axis-aligned projected clustering, we only know of our GPU adaptation of PROCLUS [41]; however, many non-axis-aligned projected clustering algorithms are based on linear algebra and would therefore be a simple task to fit the GPU. Other data mining tasks lay mainly untouched, e.g., within trajectory mining, only a GPU-based algorithm [32] for the simple flock pattern exist, while researchers have yet to develop algorithms for other more enticing patterns.

To enable the end-user to utilize our proposed algorithms easily, we provide both

c++ implementation at Github (`https://github.com/mrjakobdk/datamining`) and python packages (`https://anaconda.org/mrjakobdk/datamining`) that interact well with the standard packages for data analysis like numpy[1] and scikit-learn[2].

Throughout this thesis, we have shown several transformations and trade-offs relevant when developing GPU-parallelized equivalent algorithms of their sequential counterparts. Within data mining and especially throughout our papers, one of the most important concepts is the range query, i.e., the neighborhood search. In many scenarios, precomputing range queries, such as the neighborhoods, can reduce redundant work and, more importantly, ensure a less diverging execution on the GPU. However, if the neighborhoods contain many points due to a large radius or a dense dataset, this can lead to space issues. In some cases, this is not too bad of a trade-off, like for G-DBSCAN, GPU-INSCY, and GPU-FAST-PROCLUS, where we can still reach millions of points without running into problems. For GPU-INSCY, having the neighborhoods precomputed provides the additional benefit of enabling the pruning of the neighborhoods in the superspaces. In GPU-FAST-PROCLUS, we also perform range queries to find each sphere of influence. However, this is only computed for each medoid and, therefore, does not require significant space. In other cases, like for EGG-SynC, this would be a limiting factor, and we need to find a less space-heavy alternative instead. We propose to use a grid structure that both support the range search and our summarization strategy.

We have explored some strategies for speeding up the computation of neighborhoods that can be used in multiple algorithms, e.g., pruning neighborhoods using the neighborhoods in a subspace could easily be employed in other subspace clustering algorithms similar to INSCY, e.g, SUBCLU. However, other strategies like the summarization of areas within a neighborhood in EGG-SynC or only updating previous partial computations with the change in GPU-FAST-PROCLUS can most likely not be directly applied for other algorithms. Even if a sufficient indexing structure supporting range-queries is available, the result can vary greatly between threads, leading to an unbalanced workload. In EGG-SynC, we explore a strategy for balancing the workload of threads performing range queries. The idea is that points located close to each other are more likely to have a similar neighborhood size, and threads in the same warp should handle points located close to each other. In EGG-SynC, our grid structure already contains a list of points ordered by the grid cells they are located within, and we let the threads handle points in that order. This strategy can be used with any indexing structure relying on an ordered set of points.

Most data mining algorithms utilize range queries; more often than not, it is one of the most time-consuming tasks. Therefore, we see great potential in inventing more general data structures and work balancing strategies for performing and traversing range queries. Likely, two different strategies would need to be developed, one where many points need to find their neighborhoods and one where only a few points need to do it. If we need to find the neighborhood of many points, each thread could find

[1] https://numpy.org/

[2] https://scikit-learn.org/stable/

the entire neighborhood of a point. What we then would need to keep in mind is to balance it such that it takes the same instructions for all threads in a warp to identify the neighborhood and traverse it. On the other hand, if we only need to find the neighborhood of a few points, it would be beneficial to spread out the identification of a neighborhood among multiple threads. Even in the optimistic case where a generalized indexing structure for range queries on the GPU has been found, it might still take considerable effort to transform a sequential algorithm into a parallel one that can use such an indexing structure.

To summarize, we have created three new algorithms for clustering, subspace clustering, and projected clustering. However, there are still many categories of data mining algorithms and still more algorithms for different clustering definitions that could benefit from utilizing the GPU. This would greatly ease data analysis on the desktop, both within information retrieval and automated processes.

Part II

Publications

Chapter 8

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

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.

8.1 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 [4, 6, 43], where we search for clusters with all possible subspaces. To search for such clusters, we often employ density-based clustering similar to DBSCAN [29]. Most subspace clustering algorithms, e.g., SUBCLU [43], 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. [9] formulated a dimensionality-unbiased density-measure and utilized this in the algorithm INSCY [11]. 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, 7, 17]. 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 8.2 discusses related work, Section 8.3 gives the background of subspace clustering and INSCY, Section 8.4 describes our new parallel algorithm GPU-INSCY, Section 8.5 presents the experimental comparison of INSCY and GPU-INSCY, and Section 8.6 concludes our work.

8.2 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 [4, 6, 9–11, 24, 33, 43, 69] are often categorized into bottom-up or top-down approaches [47, 58, 64, 75]. Bottom-up approaches start with clustering in 1-dimensional subspaces, iteratively combining k-dimensional subspace clusters into $(k+1)$-dimensional subspace clusters. CLIQUE [6] and MAFIA [33] are grid-based approaches that may miss subspace clusters spanning across grid cells. Instead of clustering dense cells, SUBCLU [43] clusters dense points, as in the density-based full space clustering algorithm DBSCAN[29]. 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 [3, 4, 81]. 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 [4, 33, 56]. 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 [7, 17, 30, 49, 52]. One of the earliest GPU versions of the full-space clustering algorithm DBSCAN was CUDA-DClust* [17], 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 [7, 52, 53, 78]. Our assessment of self-reported results suggest that G-DBSCAN[7] and CUDA-DClust*[17] are the best performing options. An experimental evaluation [59] 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.

8.3 Background

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.

INSCY

We describe INSCY briefly. For further details please see [11]. We use the following terminology: let $X \in \mathbb{R}^{n \times d}$ be a d-dimensional dataset with n points, $D = \{0, \dots, d-1\}$ 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 [9], 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 8.3.1. ***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**:* $\forall p,q \in C : \exists o_1,\dots,o_m \in C : p = o_1 \wedge q = o_m \wedge \forall i \in \{2,\dots,m\} : \ o_i \in N_\varepsilon^S(o_{i-1})$
- *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 [9]. Note that Def. 8.3.1 is similar to density-based clustering in DBSCAN [29], but with an unbiased density notion wrt. subspaces.

SUBCLU [43] 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 [9] 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 $|N_\varepsilon^S(p)| \geq \max(F \times \alpha(D), \mu)$.

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.

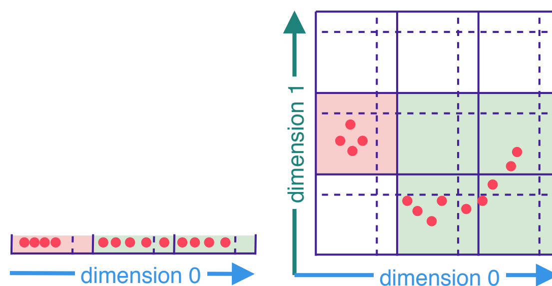


Figure 8.1: Expansion of 1-d regions into 2-d

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 8.1 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 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.

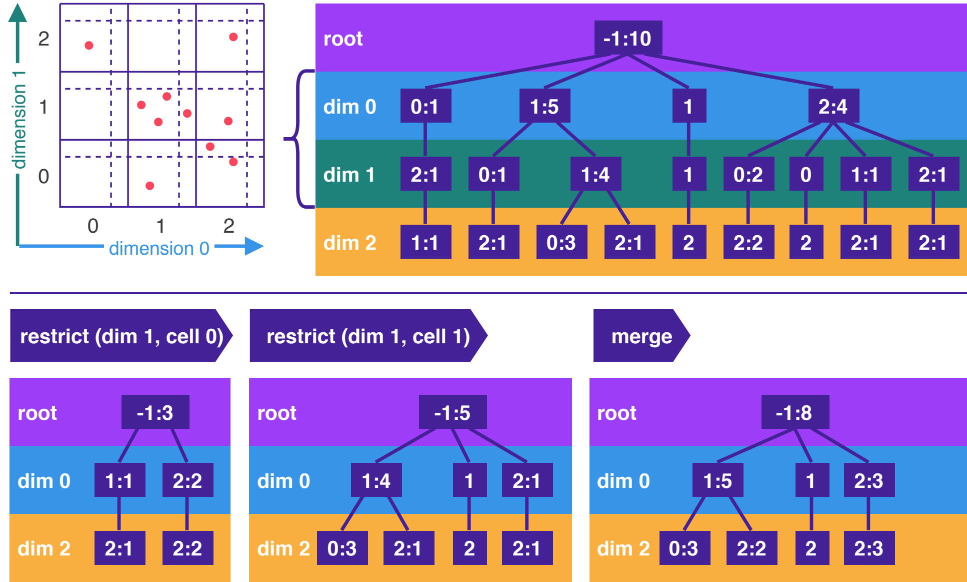


Figure 8.2: SCY-tree for examples in [11]; node values *cell* : *count*; dimensions and points colored as in later figures

For further details, see [11].

Figure 8.2 (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 5. 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.

Algorithm 5 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:     $scytree' \leftarrow$ restrict$(scytree, d_{re}, c_{re})$
4:     $scytree' \leftarrow$ mergeNeighbors$(scytree, scytree', d_{re}, c_{re})$
5:     **if** prune_recursion($scytree'$, F, μ, ε, min_C) **then**
6:       INSCY$(scytree', d_{re}+1, X, d, r, F, \mu, \varepsilon, min_C, R)$
7:       **if** prune_redundancy($scytree'$, r, R) **then**
8:         $R \leftarrow R \cup$ clustering($scytree'$, X, F, μ, ε)

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 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 8.2 (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 8.2 (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.

8.4 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 8.1 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.

Table 8.1: 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

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.

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 6. 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 8.4, after introducing a representation of the SCY-tree index-structure for the GPU in Section 8.4.

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 8.4, 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 8.4, we parallelize both pruning phases following the same approach as for restrict and merge.

Clustering. In Section 8.4, we change the sequential way of expanding the clusters [29] with one density-connected point at a time, to obtain a more efficient clustering algorithm.

Algorithm 6 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)
```

SCY-tree on the GPU

The SCY-tree representation and the associated operations are not very suited for the GPU. Section 8.4 describes how to represent the SCY-tree in a GPU friendly fashion and Section 8.4 describes how to perform the restrict and merge operations in parallel.

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 8.4, 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 8.3. Besides the arrays we also keep count of the number of nodes 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 8.4 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 8.3 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

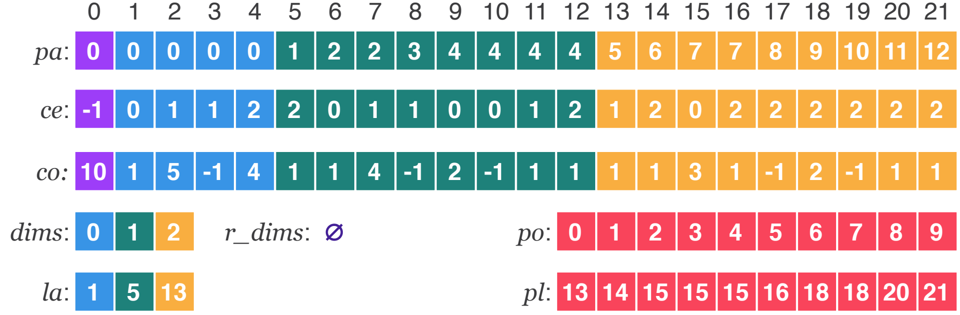


Figure 8.3: GPU-SCY-tree for SCY-tree in Figure 8.2.

dataset $po \in \mathbb{N}^{n_{pts}}$, and the other keeps track of which leaf-node each point is placed in $pl \in \mathbb{N}^{n_{pts}}$.

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 8.4, 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 $\mathbf{0}$. 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 $\mathbf{0}$ and $\mathbf{1}$ represent false and true, respectively. In Figure 8.2, we show the restriction for descriptor (1,0). In Figure 8.4 we show the same restriction in GPU-SCY-tree representation, and the temporary arrays. Here the five included nodes are marked with a $\mathbf{1}$ 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 $idx \in \mathbb{N}^{n_{dims}\times n_{cells}\times n_{nodes}}$ 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 8.4, the last included node is placed at entry

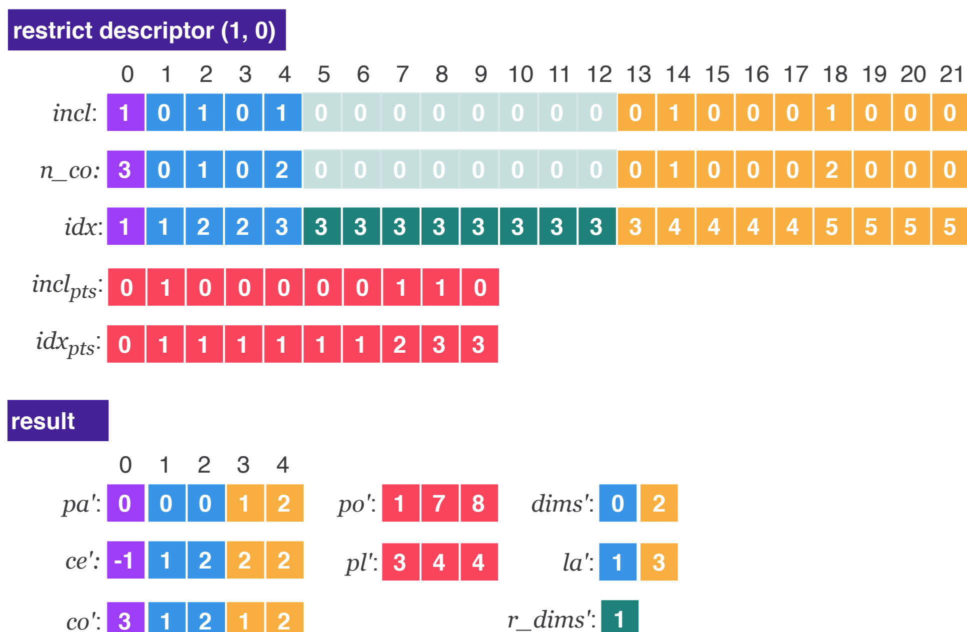


Figure 8.4: Restrict example before combining with merge.

$4 = idx(18) - 1$. Furthermore, for each descriptor, we use the last index to allocate the needed memory for the restricted SCY-trees. In Figure 8.4 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:

$$is_S(i) := (co(i) < 0) \land (co(pa(i)) \geq 0). \tag{8.1}$$

In Figure 8.3, 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 $\neg is_S(i)$:

$$s_incl(j,i,c) := (ce(i) = c) \land (\neg is_S(i)). \tag{8.2}$$

In Figure 8.3, 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 $s_incl(j,i,c)$:

$$\begin{aligned} &\forall 0 \leq j < n_{dims}, 0 \leq c < n_{cells}, \\ &\quad la(j) \leq i < la(j+1), s_incl(j,i,c) : \\ &\qquad incl(j,c,pa(i)) := \mathbf{1}. \end{aligned} \tag{8.3}$$

In Figure 8.4 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:

$$\begin{aligned} &\forall 0 \leq j < n_{dims}, 0 \leq c < n_{cells}, 1 \leq k < j-1, \\ &\quad la(j-k) \leq i < la(j-k+1), incl(j,c,i) : \\ &\qquad incl(j,c,pa(i)) := \mathbf{1}. \end{aligned} \tag{8.4}$$

In Figure 8.4 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:

$$\begin{aligned} &\forall 0 \leq j < n_{dims}, 0 \leq c < n_{cells}, la(j+1) \leq i < la(j+2) : \\ &\quad incl(j,c,i) := s_incl(j,pa(i),c). \end{aligned} \tag{8.5}$$

In Figure 8.4 node 14 is included as node 6 matches the descriptor.

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

$$\begin{aligned} &\forall 0 \leq j < n_{dims}, 0 \leq c < n_{cells}, 2 \leq k < n_{dims} - j, \\ &\quad la(j+k) \leq i < la(j+k+1) : \\ &\qquad incl(j,c,i) := incl(j,c,pa(i)). \end{aligned} \tag{8.6}$$

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:

$$\begin{aligned} &\forall 0 \leq j < n_{dims}, 0 \leq c < n_{cells}, 0 \leq p < n_{pts} : \\ &incl_{pts}(j,c,p) := \begin{cases} incl(j,c,pl(p)) & \textbf{if } j < n_{dims} - 1 \\ ce(pl(p)) = c & \textbf{else} \end{cases} \end{aligned} \tag{8.7}$$

E.g. in Figure 8.4 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.

$$\begin{aligned}&\forall 0 \leq j < n_{dims}, la(j) \leq i < la(j+1), is_S(i): \\ &\quad S(j, cells(i)) := \mathbf{1}.\end{aligned} \tag{8.8}$$

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 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.

$$\begin{aligned}&\forall 0 \leq j < n_{dims}, 0 \leq c < n_{cells}: \\ &M(j,c) := \begin{cases} M(j,c-1) & \textbf{if } (c>0) \wedge S(j,c-1) \\ c & \textbf{else} \end{cases}\end{aligned} \tag{8.9}$$

Equation 8.9 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 8.3 are shown in Figure 8.5. 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

Step 1: shown for all descriptors

S-connected S (rows: dimension, columns: cell):

dimension	0	1	2
0	0	1	0
1	1	0	0
2	0	0	0

Merge map M (rows: dimension, columns: cell):

dimension	0	1	2
0	0	1	1
1	0	0	2
2	0	1	2

Step 2: only shown for descriptor (1,0)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
incl:	1	0	1	1	1	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	1	0
n_pa:	0	0	0	0	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	2	2	2	3	4	-1	4	-1
n_co:	8	0	5	-1	3	0	0	0	0	0	0	0	0	0	0	3	2	-1	0	0	3	0
n_ch:																						
0/node	-1	-1	15	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
0/s-connection	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
1/node	2	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
1/s-connection	3	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
2/node	4	-1	16	-1	20	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
2/s-connection	-1	-1	-1	17	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

$incl_{pts}$: 0 1 1 1 1 1 1 1 1 0

Step 3: only shown for descriptor (1,0)

idx: 0 0 1 2 3 4 4 4 4 4 4 4 4 4 4 4 5 6 7 7 7 8

idx_{pts}: 0 0 1 2 3 4 5 6 7 7

Step 4: only shown for descriptor (1,0)

	0	1	2	3	4	5	6	7
pa':	0	0	0	0	1	1	2	3
ce':	-1	1	1	2	0	2	2	2
co':	8	5	-1	3	3	2	-1	3

po': 1 2 3 4 5 6 7 8

pl': 5 4 4 4 5 7 7 7

dims': 0 2   *la'*: 1 5   *r_dims'*: 1

Figure 8.5: Restrict after combining with merge.

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 8.2 into:

$$s_incl(j,i,c) := (M(j,ce(i)) = c) \wedge (\neg is_S(i)), \qquad (8.10)$$

and Equation 8.7 into

$$\forall 0 \leq j < n_{dims}, 0 \leq c < n_{cells}, p < n_{pts}:$$
$$incl_{pts}(j,c,p) := \begin{cases} incl(j,c,pl(p)) & \textbf{if } j < n_{dims}-1 \\ M(j,c,ce(pl(p))) = c & \textbf{else} \end{cases} \tag{8.11}$$

E.g. for descriptor (1, 0), we now also treat node 7 in Figure 8.3 and 8.5 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 8.5, 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 8.5. 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:

$$S_idx(i) := \begin{cases} \mathbf{0} & \textbf{if } co(i) \geq 0 \\ \mathbf{1} & \textbf{else} \end{cases} \tag{8.12}$$

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 8.5, 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:

$$\begin{aligned}&\forall 0 \leq j < n_{dims}, 0 \leq c < n_{cells}, 2 \leq k < n_{dims} - j,\\&\quad la(j+k) \leq i < la(j+k+1), n_pa(j,c,pa(i)) \geq 0 :\\&\qquad n_pa(j,c,i) := rep(j,c,pa(i)).\end{aligned} \tag{8.13}$$

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:

$$\begin{aligned}&\forall 0 \leq j < n_{dims}, 0 \leq c < n_{cells},\\&\quad la(j+1) \leq i < la(j+2), s_incl(j,pa(i),c) :\\&\qquad n_pa(j,c,i) := pa(pa(i)).\end{aligned} \tag{8.14}$$

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.

$$\begin{aligned}&\forall 0 \leq j < n_{dims}, 0 \leq c < n_{cells}, 1 \leq k < j,\\&\quad la(j-k) \leq i < la(j-k+1), I(j,c,i) :\\&\qquad n_pa(j,c,i) := pa(i),\\&\qquad n_ch(j,c,pa(i),ce(i),S_idx(i)) := i.\end{aligned} \tag{8.15}$$

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:

$$\begin{aligned}&\forall 0 \leq j < n_{dims}, 0 \leq c < n_{cells}, 1 \leq k < n_{dims} - j,\\&\quad la(j+k) \leq i < la(j+k+1), n_pa(j,c,i) \geq 0 :\\&\qquad rep(j,c,i) := i.\end{aligned} \tag{8.16}$$

We synchronize such that all threads see the same node id, and only include that node as the new child. E.g., in Figure 8.5 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 8.5 into:

$$\begin{aligned} &\forall 0 \leq j < n_{dims}, 0 \leq c < n_{cells}, la(j+1) \leq i < la(j+2): \\ &\quad incl(j,c,i) := s_incl(j,pa(i),c) \wedge (rep(j,c,i) = i), \end{aligned} \tag{8.17}$$

and Equation 8.6 into:

$$\begin{aligned} &\forall 0 \leq j < n_{dims}, 0 \leq c < n_{cells}, 2 \leq k < n_{dims} - j, \\ &\quad la(j+k) \leq i < la(j+k+1): \\ &\qquad incl(j,c,i) := (n_pa(j,c,i) \geq 0) \wedge (rep(j,c,i) = i). \end{aligned} \tag{8.18}$$

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 8.11 changes into:

$$\begin{aligned} &\forall 0 \leq j < n_{dims}, 0 \leq c < n_{cells}, p < n_{pts}: \\ &incl_{pts}(j,c,p) := \begin{cases} n_pa(j,c,pl(p)) \geq 0 & \textbf{if } j < n_{dims} - 1 \\ M(j,c,ce(pl(p))) = c & \textbf{else} \end{cases} \end{aligned} \tag{8.19}$$

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:

$$\begin{aligned} &\forall 0 \leq j < n_{dims}, 0 \leq c < n_{cells}, la(j) \leq i < la(j+1), s_incl(j,i,c): \\ &n_co(j,c,pa(i)) := \begin{cases} n_co(j,c,pa(i)) + co(i) & \textbf{if } co(i) \geq 0 \\ -1 & \textbf{if } co(pa(i)) < 0 \end{cases} \end{aligned} \tag{8.20}$$

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

$$\begin{aligned} &\forall 0 \leq j < n_{dims}, 0 \leq c < n_{cells}, 1 \leq k < j-1, \\ &\quad la(j-k) \leq i < la(j-k+1), incl(j,c,i): \\ &\qquad n_co(j,c,n_pa(j,c,i)) := \\ &\qquad\quad + \begin{cases} n_co(j,c,pa(i)) + n_co(j,c,i) & \textbf{if } co(i) \geq 0 \\ -1 & \textbf{if } co(pa(i)) < 0 \end{cases} \end{aligned} \tag{8.21}$$

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

$$\begin{aligned}
&\forall 0 \leq j < n_{dims}, 0 \leq c < n_{cells}, 1 \leq k < n_{dims} - j, \\
&\quad la(j+k) \leq i < la(j+k+1), n_pa(i) \geq 0 : \\
&\quad\quad n_co(j,c,rep(j,c,i)) := \begin{cases} -1 \textbf{ if } co(i) < 0 \\ n_co(j,c,rep(j,c,i)) + co(i) \textbf{ else} \end{cases}
\end{aligned} \tag{8.22}$$

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 8.8, and one that uses this information to compute which SCY-trees will be merged, using Equation 8.9.
- 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 8.3, 8.15, and 8.20, second, for the remaining layers above we use Equations 8.4, 8.15, and 8.21, third, directly below we use Equations 8.17, 8.14, 8.16, and 8.22, fourth, for the remainder below we use Equations 8.18, 8.13, 8.16, and 8.22, and at last, we compute inclusion of points by checking if the leaf-node where the point is placed is included using Equation 8.19.
- 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.

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 [29]. 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 [7] 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 8.5.

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.

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.

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 8.5.

8.5 Experiments

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 $\times$ 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 $\times$ 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`.

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 [10], 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.

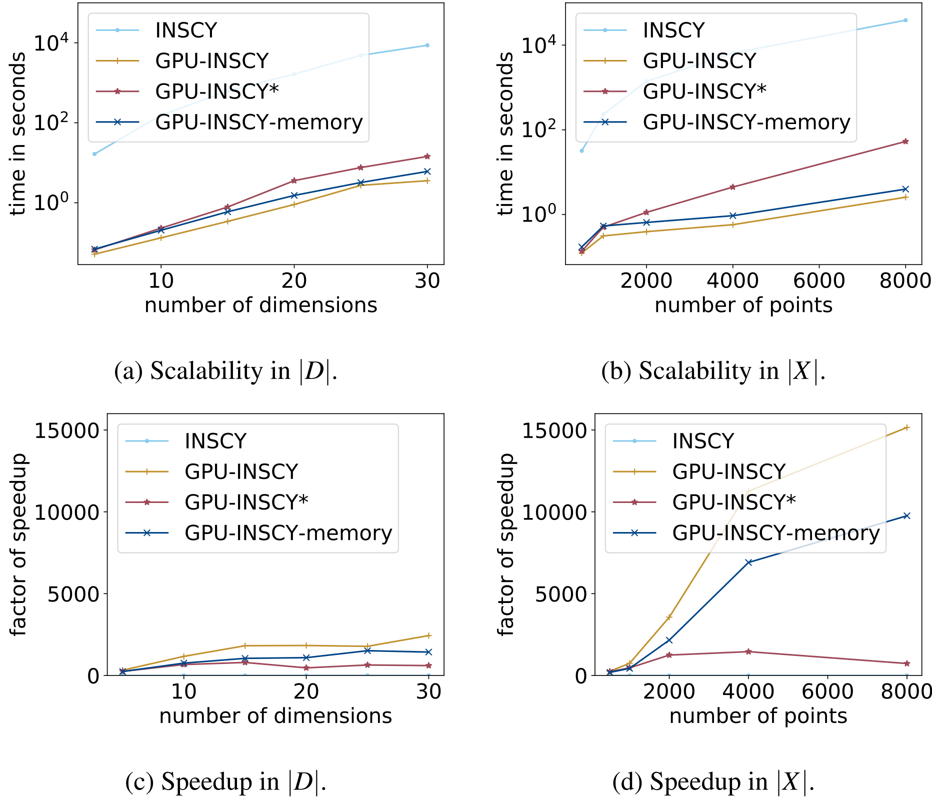
To analyze components of our algorithm, we also test GPU-INSCY* and GPU-INSCY-memory. GPU-INSCY* is a version of GPU-INSCY that does not bound the neighborhood search, so that we can study the effect of bounding the neighborhood search. GPU-INSCY-memory is described in Section 8.4. For our experiments we group the descriptors by the dimensions such that each iteration of the recursions is only parallel over the cells.

Comparison of INSCY and GPU-INSCY. In Figure 8.6a, 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 8.6c, the speedup of GPU-INSCY relative to INSCY keeps increasing. For 30 dimensions we achieve a factor of speedup of more than 2000$\times$. A similar effect is observed for increasing the number of points. In Figure 8.6b, INSCYs runtime again increases faster than for GPU-INSCY. In Figure 8.6d, 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 8.4, we attribute this dramatic speedup to our bounding of the neighborhood searches. This effect is also clear in Figure 8.6c and 8.6d, where we see that GPU-INSCY* achieves a 500-1000$\times$ speedup, corresponding to a good use of the cores, and GPU-INSCY achieves a substantially larger speedup of up to 14'000$\times$ 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.

Real world datasets

We also demonstrate GPU-INSCY speedups for real-world datasets. We report runtimes on the three datasets (glass, vowel, pendigits) [60] also studied in [10, 11].



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

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

Figure 8.6: Scalability in size and dimensionality

The glass dataset $X_{glass} \in \mathbb{R}^{214\times 9}$, vowel $X_{vowel} \in \mathbb{R}^{990\times 10}$, and pendigits $X_{pendigits} \in \mathbb{R}^{7494\times 16}$. Furthermore, we also evaluate on a more sizable higher dimensional real world data set, part of the SkyServer dataset[76] 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): $X_{sky(0.5\times 0.5)} \in \mathbb{R}^{7253\times 17}$, $X_{sky(1\times 1)} \in \mathbb{R}^{29627\times 17}$, and $X_{sky(1\times 2)} \in \mathbb{R}^{59285\times 17}$. Experiments are aborted if they run for more than 24 hours, as INSCY does for larger setups. In Figure 8.7, we see that we obtain high speedups on all datasets, but much higher for larger datasets up to $15'000\times$ speedup.

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

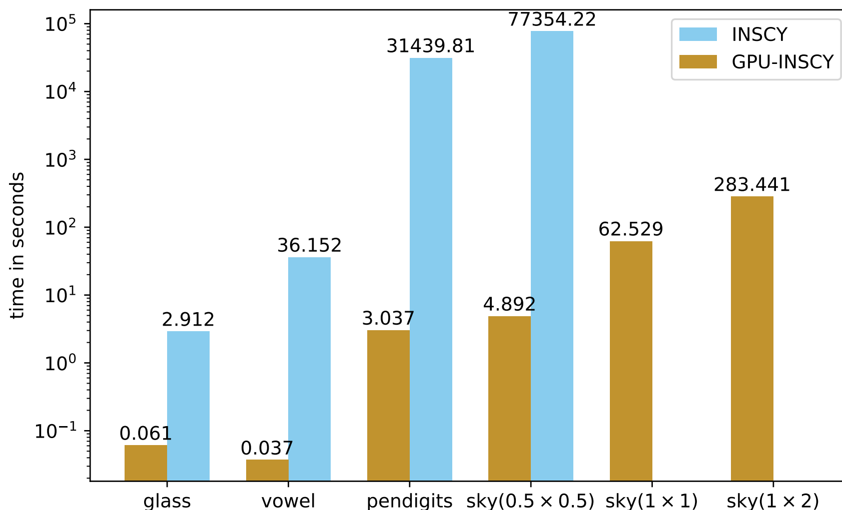
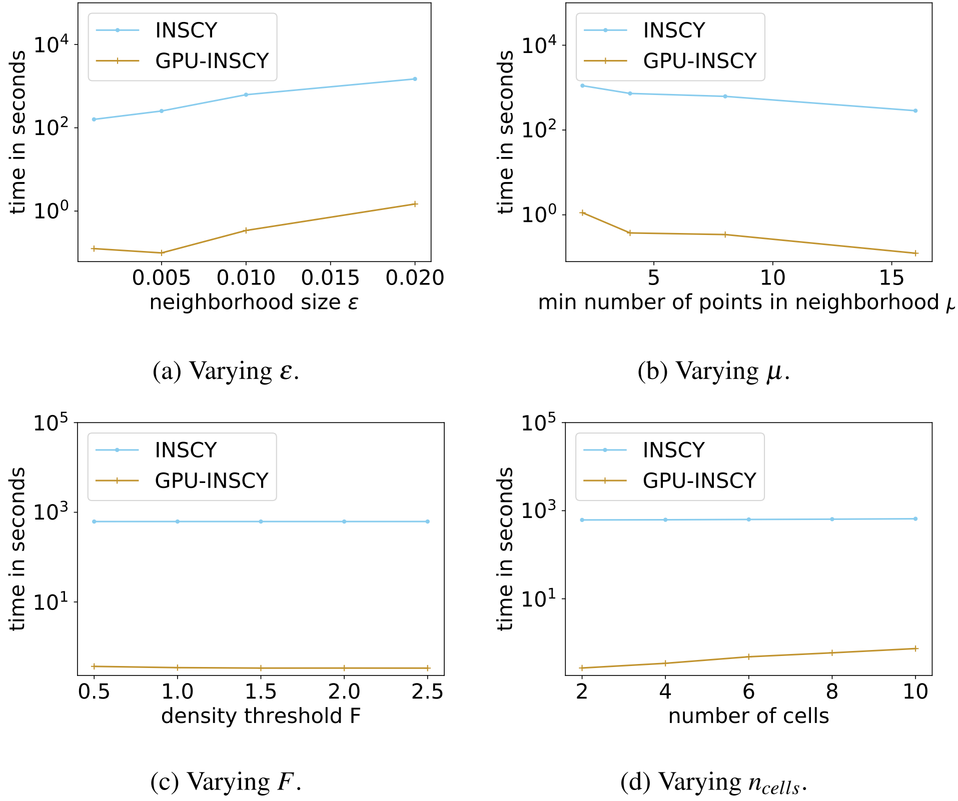


Figure 8.7: Real world data; INSCY aborted after 24 hours

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 8.8a, 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 8.8b and 8.8c, 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 8.8d, 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.



(a) Varying ε.

(b) Varying μ.

(c) Varying F.

(d) Varying n_{cells}.

Figure 8.8: Runtimes for different parameter values

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.

To test various distributions, we use the generator provided by [12], 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 8.9a 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 8.9b. 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 8.9c, 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 8.9d, 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 8.9e 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.

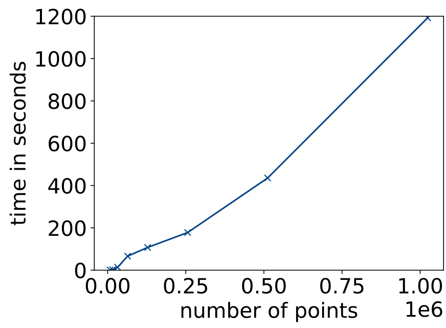
Overall, GPU-INSCY outperforms INSCY by two-four orders of magnitude with respect to runtimes for all tested settings. Even on our small GPU, we measure the running time in seconds instead of hours for smaller datasets ($<$ 10‘000 points and 15 dimensions) and minutes instead of days for larger datasets.

8.6 Conclusion

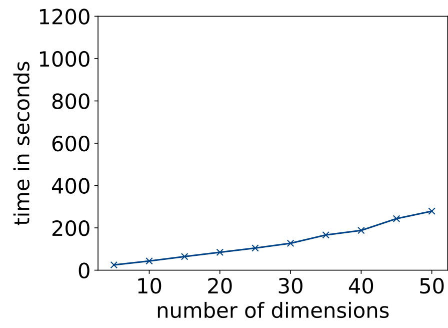
In this paper, we propose GPU-INSCY, a novel GPU-parallel algorithm for dimensionality-unbiased density-based subspace clustering, following INSCY. GPU-INSCY outperforms INSCY by several orders of magnitude. To achieve this, we utilize GPU cores by restructuring both the algorithmic processing and the 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.

8.7 Acknowledgments

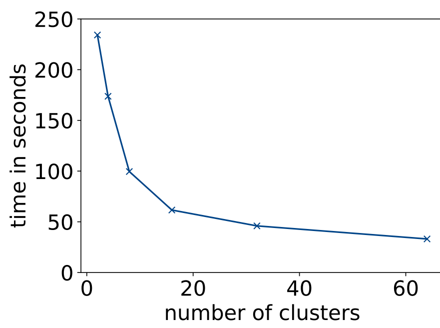
This work was supported by Independent Research Fund Denmark.



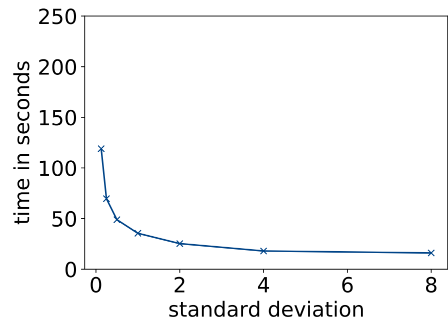
(a) Scalability in $|X|$.



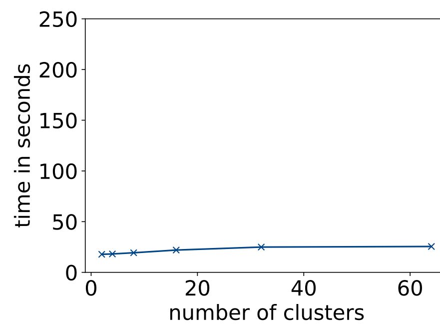
(b) Scalability in $|D|$.



(c) Scalability in number of clusters.



(d) Scalability in cluster spread (standard deviation).



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

Figure 8.9: Runtimes for scalability experiments

Chapter 9

GPU-FAST-PROCLUS: A Fast GPU-parallelized Approach to Projected Clustering

Abstract

Projected and subspace clustering aim to find groups of similar objects within a subspace of the full-dimensional space. Where subspace clustering tries to identify clusters in all possible subspaces, projected clustering assigns each point to a single cluster within one projected subspace, resulting in a much smaller result set. PROCLUS is an adaptation of the k-medoids clustering algorithm, CLARANS, to projected clustering. Even though PROCLUS is the first projected clustering algorithm, it is still competitive in comparative empirical studies.

PROCLUS is, however, still too slow for large-scale data or real-time interaction when used in information retrieval processes. Therefore, we propose novel algorithmic strategies to reduce computations and exploit the massive parallelism offered by modern graphical processing units (GPUs). To take advantage of their high degree of parallelism, standard sequential algorithms need to be significantly restructured. We therefore also propose a novel GPU-parallelized algorithm, GPU-FAST-PROCLUS, that takes advantage of the computational power of modern GPUs.

We provide experimental studies that demonstrate the benefit of our proposed strategies and GPU-parallelizations. In this experimental evaluation, we obtain 3 orders of magnitude speedup compared to PROCLUS.

9.1 Introduction

Clustering, the task of grouping similar objects, is an often employed data mining task, e.g., for finding groups of customers that exhibit similar traits. However, clustering within the full-dimensional space becomes meaningless for higher-dimensional data as distances become increasingly similar [14]. This implies that clusters might only

exist within subspace projections of the full-dimensional space, e.g., for a group of customers, a trait like height might not be important for the grouping. The discovery of such clusters can be made by subspace clustering [6, 43], which finds clusters that exist within all possible subspaces. Projected clustering [4], like subspace clustering, aims to find clusters within a subspace of the full-dimensional space but only reports disjoint clusters, each in one subspace projection. By that, projected clustering reduces the size of the result set and makes it easier to understand for the user.

The first projected clustering algorithm is PROCLUS [4], an adaptation of the K-medoids approach, CLARANS [61], to find clusters in projected subspaces. Today, PROCLUS is still one of the fastest subspace or projected clustering algorithms while remaining competitive in evaluations [58]. However, it still takes up to several minutes to perform PROCLUS on small datasets of just a few thousand data points [4, 58].

Data mining is usually part of information retrieval and data science processes. A successful process produces the information for a task quickly [19]. For real-time interaction, this means executing data analysis within 100*ms* [74]. We propose algorithmic strategies to make PROCLUS fast enough for real-time interaction on even a million data points. This includes strategies to reduce computations and to utilize modern multi-core hardware. Furthermore, we address the challenge of determining the right set of parameters for a clustering algorithm, and leverage the fact that we can reuse partial results between parameter settings to provide even higher speedups when computing several PROCLUS clusterings with different parameter settings.

Modern GPUs with their thousands of cores provide high computational throughput. However, this throughput comes with a vastly different computational model. Since PROCLUS is developed for a single-core model, a novel algorithmic approach must be taken for it to benefit from the computational power offered by GPUs.

Our contributions include:

- Algorithmic strategies to reduce the number of computations by reusing distances and partial results between iterations and parameter settings, as well as by an alternative trade-off between speed and space.
- GPU-parallelized versions of PROCLUS and of our proposed algorithmic strategies.
- An experimental evaluation of PROCLUS and our proposed strategies showing 3 orders of magnitude speedup across parameters and data distributions.

9.2 Background

We provide important notations in Table 9.1. $|A|$ denotes the size of a set A, $||a||$ is the absolute value of a scalar a, $||p||_1$ is the L1-norm of a point p, and $||p||_2$ is the L2-norm. The norm in subspace projection D_i is denoted using a superscript, e.g., $||p-q||_1^{D_i}$. PROCLUS uses Manhattan distance $||p-q||_1$, Manhattan segmental distance $||p-q||_1^{D_i}/|D_i|$, and Euclidean distance $||p-q||_2$.

We use subscript to index matrices, sets, lists, and points, e.g., $Data_{p,j}$ refers to dimension j of data point p. Superscript is used to refer to a version of a matrix, a set, or a list at a specific iteration, e.g., H^t is H in the current iteration t, H^{t-1} is H in the previous iteration.

We use $\leftarrow$ for assignment and $=$ for equality. For simplicity, we use the index of point p as the actual data point $Data_p$, and vice versa. E.g. when computing a distance between point p and medoid m_i we write $Dist_{m_i,p} \leftarrow ||p - m_i||_2$ as shorthand for $Dist_{m_i,p} \leftarrow ||Data_p - Data_{m_i}||_2$.

PROCLUS

We briefly describe PROCLUS, details are found in [4]. PROCLUS is an adaptation of the k-medoids algorithm, CLARANS [61], to projected clustering. Given input $Data \in \mathbb{R}^{n \times d}$ a d-dimensional dataset with n points, the number of clusters k, the average number of dimensions l, two scalars A, B, minimum deviation *minDev*, and number of rounds without improvement *itrPat*, PROCLUS proceeds in three phases initialization, iterative, and refinement. PROCLUS outputs k disjoint clusters in different subspace projections.

Initialization phase. First, greedily pick potential medoids M from a subset $Data'$ of the dataset $Data$. The set of points $Data'$ is a random sample of size $A \times k$ from $Data$. From $Data'$ PROCLUS greedily selects $B \times k$ points one by one, as the one with the largest Euclidean distance to all other points in M, see Algorithm 7 line 2-3.

Iterative phase. PROCLUS iteratively optimizes the set of medoids $MCur \subset M$ of size k. This is done through multiple sub-phases, see lines 5-14. First, compute the set of points L_i within a sphere centered at each medoid m_i. Second, find the optimal sets of dimensions for the sets of points L. Third, assign points to the closest medoid within the selected subspace projection for each medoid. Fourth, evaluate the clustering. If the cost of the clustering is smaller than the best found so far, keep the current set of medoids *MCur* as the best set of medoids *MBest* and the corresponding clustering as *CBest*. Compute a new *MCur* for the next iteration by replacing the bad medoids in *MBest*. The bad medoids *MBad* are the medoids with a cluster size smaller than $|Data|/k \times minDev$ or if no such exists, the medoid with the smallest cluster. The iterative phase stops when no new *MBest* has been found for *itrPat* iterations.

Refinement phase. The last phase refines the clusters found so far. First, let $L \leftarrow CBest$ instead of the spheres and use L to find the best set of dimensions for the medoids. Within these dimensions, assign points to the closest medoid. At last, define a sphere for each medoid m_i with radius $\Delta_i \leftarrow \min_{j \neq i} ||m_i - m_j||^{D_i}/D_i$ in subspace D_i. A point is an outlier if it lies outside the sphere for all medoids.

ComputeL. Compute the set of points L_i close to each medoid $m_i \in MCur$. For each medoid m_i, let $\delta_i \leftarrow \min_{j!=i} ||m_j - m_i||_2$ be the distance to the nearest medoid $m_j \in MCur$ and $L_i \leftarrow \{p \in Data| \; ||p - m_i||_2 \leq \delta_i\}$ be the set of points within the sphere centered at m_i with radius δ_i.

FindDimensions. Find the best subspace projections. Compute the average distances $X_{i,j}$ from all points in L_i to m_i along dimensions j. Next, compute the

Table 9.1: Notation

Symbol	Description
$Data \in \mathbb{R}^{n \times d}$	Dataset with n points and d dimensions
k	Number of clusters
l	Average number of dimensions
$itrPat$	Max number of iterations w/o changes
$minDev$	Threshold to identify bad medoids
A	Constant to determine size of $Data'$
B	Constant to determine size of M
itr	Iteration counter
t, t'	The current and previous usage
p	Point in $Data$
$Data' \subseteq Data$	Random subset of full dataset
$M \subset Data'$	Greedily selected subset of $Data'$
$MCur \subset M$	Set of current medoids selected from M
$MBest \subset M$	Set of best medoids
$MBad \subset MCur$	Set of replaced medoids in $MCur$
$m_i \in MCur$	i'th medoid
$Dist$	Distance matrix
δ_i	Distance to the closest medoid
$L_i \subseteq Data$	Set of points w/ radius δ_i of medoid m_i
$\Delta L_i \subseteq L_i$	Change in set L_i between iterations
λ_i	Indicates increase or decrease in Δ_i
$MIdx_i$	The index of m_i in M
$DistFound$	Indicates the distances computed
$H_{i,j}$	Sum of dist. of $p \in L_i$ to m_i in dim. j
$X_{i,j}$	Avg. dist. to medoid m_i in dim. j
Y_i	Average distance to medoid m_i
σ_i	Standard deviation of $X_{i,j}$
Z_{ij}	Measure of spread for C_i in dim. j
$D_i \subseteq \{1, \ldots, d\}$	i'th subspace projection
$C_i \subseteq Data$	i'th cluster
$CBest$	Best clustering so far
μ_i	Centroid of cluster C_i
w_i	Cost of a cluster i
$cost$	Weighted cost of the full clustering
$costBest$	Lowest cost found so far
Δ_i	Dist. to closest medoid in subspace D_i

Algorithm 7 PROCLUS$(Data, A, B, k, l, itrPat, minDev)$

```
1: // Initialization Phase
2: Data' ← random sample from Data of size A × k
3: M ← Greedy(Data', A, B, k)
4: // Iterative Phase
5: while itr < itrPat do
6:     L ← ComputeL(Data, MCur)
7:     D ← FindDimensions(Data, MCur, L, k, l)
8:     C ← AssignPoints(Data, MCur, D)
9:     cost ← EvaluateClusters(Data, C, D, k)
10:    itr ← itr + 1
11:    if cost < costBest then
12:        itr ← 0, costBest ← cost, MBest ← MCur
13:        MBad ← ComputeBadMedoids(MBest, minDev)
14:    Compute MCur by replacing the bad medoids in MBest with random points
       from M
15: // Refinement Phase
16: L ← CBest
17: D ← FindDimensions(Data, L, MBest, d, k, l)
18: C ← AssignPoints(Data, MBest, D)
19: C ← RemoveOutliers(Data, C, MBest, D)
20: return C, D, M
```

average distances
$Y_i \leftarrow \left(\sum_{j=1}^{d} X_{i,j}\right)/d$ for each medoid m_i across all dimensions j, the standard deviation $\sigma_i \leftarrow \sqrt{\left(\sum_{j=1}^{d} X_{i,j}\right)/(d-1)}$ for each medoid m_i across all dimensions j, and a relative measure of spread $Z_{i,j} \leftarrow (X_{i,j} - Y_i)/\sigma_i$ for each pair of dimension j and medoid m_i.

At last, for each medoid m_i pick the two dimensions with the smallest $Z_{i,j}$, and after that pick the dimensions j corresponding to the lowest $Z_{i,j}$ and add them to the subspace D_i until a total of $k \times l$ dimensions has been picked.

AssignPoints. Assign each point p to cluster C_i with smallest Manhattan segmental distance to medoid m_i within the subspace projection D_i.

EvaluateCluster. The cost of the clustering is the average Manhattan segmental distance from centroid $\mu_i \leftarrow \sum_{p \in C_i} p/|C_i|$ of cluster C_i within subspace D_i:

$$w_i \leftarrow \frac{\sum_{j \in D_i} V_{i,j}}{|D_i|}, V_{i,j} \leftarrow \frac{\sum_{p \in C_i} ||p_j - \mu_{i,j}||}{|C_i|}, \tag{9.1}$$

$$cost \leftarrow \frac{\sum_i^k |C_i| \times w_i}{|Data|}. \tag{9.2}$$

Note that PROCLUS uses localized random search for the best set of medoids and subspaces within a random subset of the data. This implies that PROCLUS is non-deterministic, so results can differ with the same parameters and dataset if

the random seeds differ. Any of these results are equally correct according to the PROCLUS algorithm.

9.3 FAST-PROCLUS

We observe that PROCLUS performs many similar computations that we exploit when we devise parallel algorithmic strategies. Some of these computations are not just similar but repeated between iteration and function calls. We propose two strategies: (1) thorough analysis of distance computations and storing partial results for re-use, (2) parallel algorithms for the GPU. To make a clear distinction between the speedup gained by reducing computations performed, and the speedup gained by parallelization, we describe our FAST-PROCLUS for the algorithmic improvements, and in Section 9.4, we parallelize both PROCLUS and FAST-PROCLUS.

The result of PROCLUS depends on user-selected parameters, so the user usually does multiple runs to find the best parameter setting. We propose a strategy that reuses temporary results between iterations and parameter settings to reduce computations. This includes a heuristic to reduce conversion time by reusing the best set of medoids found for the previous parameter setting.

The iterative phase of PROCLUS has several steps with a $O(n \times k \times d)$ running time. These steps are the most time-consuming and, therefore, the focus for improvement.

Compute distances to potential medoids only once. When computing the set of points L_i within the radius δ_i, PROCLUS computes the distances from each medoid m_i to each point p. These computations have a $O(n \times k \times d)$ run-time and are, therefore, one of the more expensive in PROCLUS.

For the medoids used in earlier iterations, the distances have already been computed and do not change between iterations since the distance measure is the Euclidean distance in full-dimensional space. Furthermore, PROCLUS only picks the current medoids *MCur* from a small set of $B \times k$ potential medoids M where B should be a small number. This implies that the reuse of medoids is likely. We, therefore, propose to save the distances across iterations. For this purpose, we introduce a distance matrix $Dist \in \mathbb{R}^{Bk \times n}$ with all pairs of distances between medoids and points. This requires $O(B \times k \times n)$ space and is, therefore, a trade-off between running time and space. For the cases where space is a limiting factor, we later propose an adaptation of this strategy that reduces the memory used by a factor B, at the cost of a small increase in running time.

The distance matrix *Dist* allows us to only compute the distances the first time a medoid is used, and therefore reduce the computations in *ComputeL*. To keep track of which distances have been computed, we maintain an indicator vector $DistFound \in \{true, false\}^{Bk}$ that indicates if the distances to a medoid have been computed. Furthermore, we introduce an index $MIdx_i$ to indicate which of the potential medoids the i'th medoid $m_i \in MCur$ corresponds to from the potential medoids M and the distance matrix $Dist$. For each iteration t we check *DistFound* to see if

the distances from each m_i to all points p has been computed, if not we compute the distances $Dist_{MIdx_i,p} \leftarrow ||p - m_i||_2$ for all p. Afterward, we set $DistFound_{MIdx_i} \leftarrow true$ to indicate that the distances have been computed.

Introduce sum of distances to medoids as temporary result. As part of selecting subspaces, we compute the average distances $X_{i,j}$ to each medoid m_i from all points $p \in L_i$ along dimension j. We observe that it is often the case that the set L_i only changes for a fraction of the points between iterations since the potential medoids are selected to be far apart. We denote the change in L_i by ΔL_i.

Theorem 9.3.1 (Computing the change ΔL_i in L_i between iterations)**.** *For medoid m_i and the set of points L_i in the sphere centered at m_i with radius δ_i, let:*

$$\begin{aligned} \Delta L_i \leftarrow \{p \in Data|\ \delta_i^{t'} < ||p - m_i||_2 \leq \delta_i^t \vee \\ \delta_i^{t'} \geq ||p - m_i||_2 > \delta_i^t\}. \end{aligned} \tag{9.3}$$

Then ΔL_i is the change in set L_i between the current iteration t and the previous usage t'.

Proof. We have two cases, either the radius δ_i has increased, implying that $\delta_i^t > \delta_i^{t'}$, or it has decreased, implying that $\delta_i^t < \delta_i^{t'}$. In the first case, we can split the set L_i^t at the current iteration t into two disjoint sets, the old set $L_i^{t'}$ at iteration t' union with the change ΔL_i:

$$\begin{aligned} L_i^t &= \{p \in Data|\ ||p - m_i||_2 \leq \delta_i^t\} \\ &= \{p \in Data|\ ||p - m_i||_2 \leq \delta_i^{t'}\} \cup \\ &\quad \{p \in Data|\ \delta_i^{t'} < ||p - m_i||_2 \leq \delta_i^t\} \\ &= L_i^{t'} \cup \{p \in Data|\ \delta_i^{t'} < ||p - m_i||_2 \leq \delta_i^t \vee \\ &\qquad \delta_i^{t'} \geq ||p - m_i||_2 > \delta_i^t\} \\ &= L_i^{t'} \cup \Delta L_i. \end{aligned} \tag{9.4}$$

And analogously for the decrease in L_i. □

To avoid recomputing the entire $X_{i,j}$, we propose to maintain a matrix $H^{t'} \in \mathbb{R}^{Bk \times d}$ with the sum of distances to each medoid m_i to all points in $L_i^{t'}$ along dimension j from the previous usage t':

$$H^{t'}_{MIdx_i,j} = \sum_{p \in L_i^{t'}} ||p_j - m_{ij}||. \tag{9.5}$$

Theorem 9.3.2 (Computing H iteratively)**.** *For medoid $m_i \in MCur$, set L_i^t in radius δ_i^t at iteration t and change ΔL_i since the previous usage t' we can split the sum of distances $H^t_{MIdx_i,j}$ into two parts:*

$$H^t_{MIdx_i,j} \leftarrow H^{t'}_{MIdx_i,j} + \lambda_i \times \sum_{p \in \Delta L_i} ||p_j - m_{ij}||, \tag{9.6}$$

where λ_i is 1 if the sphere increases in size and -1 if it decreases.

Proof. We again have two cases. Either the change is an increase or decrease. For increase, using Theorem 9.3.1 and since $L_i^{t'}$ and ΔL_i are disjoint, we have:

$$\begin{aligned} H^t_{MIdx_i,j} &= \sum_{p \in L_i^t} ||p_j - m_{ij}|| = \sum_{p \in L_i^{t'} \cup \Delta L_i} ||p_j - m_{ij}|| \\ &= \sum_{p \in L_i^{t'}} ||p_j - m_{ij}|| + \sum_{p \in \Delta L_i} ||p_j - m_{ij}||. \end{aligned} \quad (9.7)$$

Analogously for decrease. □

We use Theorem 9.3.2 to update H and then compute the average distance $X_{i,j} \leftarrow H^t_{MIdx_i,j}/|L_i|$ across dimension j from all points in L_i to m_i.

Only updating H with the change in L_i requires that during *ComputeL* we do not calculate the set L_i, but instead the change ΔL_i in points as in Theorem 9.3.1. Notice that ΔL_i can be computed in the same way as L_i, the only difference being the condition that we keep points between the current δ_i^t and the previous $\delta_i^{t'}$ radius of the set L_i. To maintain radius $\delta_i^{t'}$ we must keep the previous radius for any of the $B \times k$ potential medoids. Furthermore, we maintain the size of the set L_i, $|L^t_{MIdx_i}| \leftarrow |L^{t'}_{MIdx_i}| + \lambda_i \times |\Delta L_i|$, for all potential medoids.

We can now update H between iterations instead of recomputing the sum of distances for each iteration. This implies that we can reuse computations and reduce the running time.

Multiple parameter settings

A drawback for most subspace and projected clustering algorithms, including PROCLUS, is that the result depends on the selected parameters. In practice, these algorithms are typically run multiple times with different parameters. For PROCLUS, the important parameters are the number of clusters k and the average number of dimensions l.

When running for multiple different parameter settings, PROCLUS performs many similar computations. We observe that if we have the same potential medoids M for all parameter settings, both the distance matrix *Dist* and the sum of distances H can be reused. To achieve this, we only once greedily pick potential medoids for the largest k and use this set M for all parameter settings. Having a constant $|M| = B \times k$ picked from a set of size $|S| = A \times k$ corresponds to an increase in $A = |S|/k$ and $B = |M|/k$ as k decreases. In other words, the first selection of A, B, and k impacts the values for A and B in subsequent parameter settings for different k. Please note that A, B, k, and l are only used to determine the size of M and S and do not otherwise impact greedy picking, so the likelihood of picking a specific M for any given execution is unchanged. We thus trade-off selection of A and B for speed. We implement both this faster version reusing the computations saved in *Dist*, H, M, and S, as well as one that follows the original PROCLUS sampling strategy, and study the speedup gained in the experiments, Section 9.5.

As an additional speedup option, we introduce a heuristic that reusing medoids found to be good in one setting as initialization in other settings, as this may lead to faster convergence. Therefore, instead of initializing each parameter setting with the current medoids *MCur* as a random subset of the potential medoids M, we initialize the current medoids as a random subset of the previous best medoids *MBest*. In the experiments, Section 9.5, we study the speedup this initialization provides.

Trade-off between running time and space

We propose an adaptation of FAST-PROCLUS, called FAST*-PROCLUS, that reduces the space complexity at the cost of a slight increase in running time. Instead of saving the distance matrix *Dist* and the sum of distances H to all potential medoids, which require $O(B \times k \times n)$ and $O(B \times k \times d)$ space, respectively, we only keep these temporary results from the previous iteration $t-1$. This requires only $O(k \times n)$ space but implies that we can only reuse the computations in *Dist* and H from iteration $t-1$ instead of any earlier iteration t'. However, it is often the case that few of the current medoids are replaced, and we can, therefore, still reuse most of the computed distances.

Since we no longer need to keep track of which of the potential medoids are in use, we do not use the index *MIdx*. Instead, we use $i \in MBad$ to identify for which of the medoids we need to recompute the distance matrix *Dist* and reset the previous δ_i^{t-1}, size of L_i and the sum of distances H before we compute ΔL_i.

9.4 GPU-PROCLUS

Modern GPUs provide high computational power through thousands of cores at the cost of a restrictive parallel computational model. This stands in contrast to the sequential model of the CPU that most algorithms follow. Therefore, when developing algorithms for the GPU, several properties must be considered. We use the term parallel to denote parallel execution under the GPU's computational model.

Programming the GPU is more like using vector registers, where the same operation is performed on each element on a given execution cycle. This is known as the Single Instruction Multiple Data (SIMD) model. GPUs similarly use a Single Instruction Multiple Thread (SIMT) model where hundreds or thousands of threads are executed at the same time.

On NVIDIA GPUs, instructions are grouped into vector instructions known as warps, where 32 threads are scheduled together and share a program counter. This implies that all cores in a warp must perform the same instruction at all times. Furthermore, cores are grouped into streaming multiprocessors (SMs). The warps are scheduled on a given SM. Warps on the same SM can share fast access L1-cache associated with that SM and can also synchronize.

In the CUDA programming environment, the CPU program spawns functions, known as kernels, onto the GPU. The functions spawn on invocation of a given number of threads to be executed concurrently. These threads may be organized into

blocks. All threads within a thread block are executed within the same SM. Threads in different blocks cannot synchronize automatically, so computations performed in different blocks should be independent to avoid a global synchronization. By independent we mean computations that do not use the partial result of each other. Data accessed by threads in different blocks must be located in global memory, which is slower than the shared memory.

When multiple threads write to the same memory address, race conditions can occur, where changes by one of them may be lost. To avoid such behavior, the GPU provides atomic versions of increment, addition, maximum, etc. However, these are more expensive and should be avoided, if possible.

In this paper, our algorithms are structured such that they may perform for-loops in parallel as threads. This entails that each step of the for-loop iterations is performed concurrently by different threads. We use a similar notation for thread blocks. If the for-loop has more iterations than threads per thread block, each thread handles multiple iterations.

GPU-friendly parallelization

PROCLUS has a long running time for larger datasets and thus is too slow for interactive settings. We, therefore, propose an algorithm capable of utilizing the computational power of the GPU to reduce the running time. We present GPU-parallelized versions of PROCLUS, FAST*-PROCLUS, and FAST-PROCLUS, called GPU-PROCLUS, GPU-FAST*-PROCLUS and GPU-FAST-PROCLUS.

In our GPU-parallelization approaches, we ensure a correct PROCLUS result by only parallelizing independent computations, or else by synchronizing to ensure that all threads within a block are at the same state. Furthermore, to avoid race conditions between threads that compute part of a common result, we use atomic operations. As mentioned above, PROCLUS is non-deterministic due to local randomization, so results between runs may differ both for the GPU versions and the CPU versions of PROCLUS, but all our results are fully correct with respect to the PROCLUS definition.

Each of the sub-functions *Greedy*, *ComputeL*, *FindDimensions*, *AssignPoints*, and *EvaluateCluster* has a high time complexity and will therefore be the focus of this section.

To avoid costly memory transfers between the CPU and the GPU, all other computations are also performed on the GPU. Each sub-function is formulated as a separate algorithm for readability. However, since it is time-consuming to allocate and free memory on the GPUs, we allocate all required memory at the beginning of GPU-PROCLUS and reuse the same allocated memory for all of the iterations.

Greedy. In PROCLUS [4], the greedy selection of potential medoids repeatedly selects the point that is furthest away from all other potential medoids. Algorithm 8 shows our GPU-parallelized version of the greedy function. At line 1-4 we first pick a random point M_1 in $Data'$ as part of M and in parallel we compute the Euclidean distance $Dist_p \leftarrow ||M_i - p||_2$ to all points p in $Data'$. Each distance computation

is completely independent of others and can therefore be computed using different thread blocks. To reduce the number of accesses to global memory, we compute the maximal distance *maxDist* within the same kernel call, line 5. However, to guarantee that the correct maximum *maxDist* is computed, we must ensure that all blocks have finished before using the global maximum *maxDist*. We, therefore, need to check which points have the largest distance in a separate kernel call at line 6-9.

Algorithm 8 Greedy$(Data', A, B, k)$

1: Pick M_1 at random from $Data'$.
2: $maxDist \leftarrow 0$ // shared variable
3: **for** $p \in Data'$ - in parallel as threads and blocks **do**
4: $\quad Dist_p \leftarrow ||M_i - p||_2$
5: $\quad maxDist \leftarrow \max(maxDist, Dist_p)$ // atomic
6: **for** $i \leftarrow 2, \ldots, Bk$ **do**
7: $\quad$ **for** $p \in Data'$ - in parallel as threads and blocks **do**
8: $\quad\quad$ **if** $maxDist = Dist_p$ **then**
9: $\quad\quad\quad M_i \leftarrow p$
10: $\quad maxDist \leftarrow 0$
11: $\quad$ **for** $p \in Data'$ - in parallel as threads and blocks **do**
12: $\quad\quad Dist_p \leftarrow \min(Dist_p, ||M_i - p||_2)$
13: $\quad\quad maxDist \leftarrow \max(maxDist, Dist_p)$ // atomic
14: **return** M

At line 6 to 13, we repeat this procedure until $B \times k$ potential medoids has been picked. The only change is that we keep the smallest distance to any already picked potential medoids from point p.

ComputeL. Computing the set of points L_i for each medoid m_i is done on the GPU as in Algorithm 9. First, pre-compute the distances $Dist_{i,p}$ between each medoid m_i and each point p. Each distance computation is completely independent and can be performed completely in parallel, see line 1-3. Next, in parallel over each pair of medoids m_i, m_j, find the distance δ_i from m_i to the closest medoid $m_{j'}$, see line 4-7. For each medoid m_i, we compute the set of points L_i in the sphere with radius δ_i centered at m_i. This is done by checking if the distance to each point p is within δ_i.

To save time, we allocate memory for the worst-case size of L_i and add points to the first available location in the allocated array. Adding each point p to the set L_i is done using *atomicInc* to increment the location of points without race conditions, see line 8-12.

FindDimensions. Finding the dimensions for the projected subspaces follows the formula for the original PROCLUS closely. Each entry of X, Y, σ, Z can be computed completely independently and therefore by different thread blocks. To avoid saving Y and σ to global memory we combine the computation of Y, σ, and Z into one kernel call. This reduces the running time substantially, but since these kernel calls are small the impact on the overall running time is minor.

Algorithm 9 ComputeL$(Data, MCur)$

```
1: for m_i ∈ MCur - in parallel as blocks do
2:   for p ∈ Data - in parallel as threads and blocks do
3:     Dist_{m_i,p} ← ||m_i − p||_2
4: for m_i ∈ MCur - in parallel as blocks do
5:   for m_j ∈ MCur - in parallel as threads do
6:     if m_i ≠ m_j then
7:       δ_i ← min(δ_i, Dist_{m_i,m_j}) // atomic
8: for m_i ∈ MCur - in parallel as blocks do
9:   for p ∈ Data - in parallel as threads and blocks do
10:    if Dist_{m_i,p} ≤ δ_i then
11:      l ← increment(|L_i|) // atomic
12:      L_{i,l} ← p
13: return L
```

Algorithm 10 FindDimensions$(Data, MCur, L, k, l)$

```
1: for m_i ∈ MCur - in parallel, as blocks do
2:   for j ← 1,...,d - in parallel, as blocks do
3:     sum ← 0 // local variable
4:     for p ∈ L_i - in parallel, as threads do
5:       sum ← sum + ||p_j − m_{i,j}||
6:     X_{i,j} ← X_{i,j} + sum/|L_i| // atomic
7: for m_i ∈ MCur - in parallel, as blocks do
8:   for j ← 1,...,d - in parallel, as threads do
9:     Y_i ← Y_i + X_{i,j}/d // atomic
10:    σ_i ← σ_i + (X_{i,j} − Y_i)^2 // atomic
11:    synchronize threads
12:    σ_i ← sqrt(σ_i/(d − 1))
13:    synchronize threads
14:    Z_{i,j} ← (X_{i,j} − Y_i)/σ_i
15: Pick the dimensions j with the two smallest Z_{i,j} values for each medoids m_i.
16: Pick next k × l − 2 × k smallest Z_{i,j}, append associated dimensions j to subspace
    of corresponding medoids m_i
17: return D
```

When computing $X_{i,j}$, we sum across a large number of points. To avoid race conditions each addition to the global memory location must be performed by *atomicAdd*. Instead of performing the expensive atomic operations for each point, we let each thread compute a part of the sum locally. Afterward, each thread adds the local sum atomically to $X_{i,j}$ in the global memory. To compute the average, the local sum is divided by $|L_i|$. At last, Z is used to pick the subspaces D.

AssignPoints. Assigning each point p to the closest medoid m_i is done by first computing the distance $Dist_{p,m_i}$ from each point p to each medoid m_i, which again can be done completely in parallel. Remember that the distance measure used to assign points is the Manhattan segmental distance in subspace D_i, and we can therefore not reuse the previously computed distances. Next, for each point p in parallel, we check which medoid is closest and assign the point p to the corresponding cluster C_i. Both steps are joined into one kernel to reduce the number of global memory accesses, see Algorithm 11. Remember that accessing shared memory is faster than accessing global memory, therefore, it has a large effect on the running time. We use *atomicMin* to find the smallest distance to a medoid and synchronize to ensure that all medoids have been checked before selecting the closest. This implies that we must compute the distances from each point to all medoids in the same thread block. Adding the points to set C_i is done the same way as for L_i.

Algorithm 11 AssignPoints$(Data, MCur, D)$

1: **for** $p \in Data$ - in parallel, as threads and blocks **do**
2:     **for** $m_i \in MCur$ - in parallel, as threads **do**
3:         $minDist_p \leftarrow \infty$ // shared variable
4:         $Dist_{p,m_i} \leftarrow ||p - m_i||_1^{D_i} / |D_i|$ // local variable
5:         $minDist_p \leftarrow \min(minDist_p, Dist_{p,m_i})$ // atomic
6:         synchronize threads
7:         **if** $minDist_p = Dist_{p,m_i}$ **then**
8:             $l \leftarrow increment(|C_i|)$ // atomic
9:             $C_{i,l} \leftarrow p$
10: **return** C

EvaluateCluster. The evaluation of the clustering is the average Manhattan segmental distance to the centroid, not to the medoid as in previous sub-functions. Therefore the first step is to compute the centroid of each cluster.

The formulation of the cost-function in Eq. 9.2 is separated into multiple steps that each could be parallelized in its own kernel call; computing the mean $\mu_{i,j}$ of each cluster, the average distance to the mean along each dimension $V_{i,j}$, the average distance among the dimensions for each cluster w_i, and finally the summed weighted cost for the entire clustering. However, this would require saving temporary results of each step to global memory, which is expensive to access. To avoid this, we reformulate the cost-function into a sum of values that can be computed in parallel

and where only the final cost must be written to global memory:

$$cost = \frac{\sum_i^k |C_i| \times \frac{\sum_{j \in D_i} \frac{\sum_{p \in C_i} ||p_j - \mu_{i,j}||}{|C_i|}}{|D_i|}}{|Data|} \tag{9.8}$$

$$= \sum_{i=1}^{k} \sum_{j \in D_i} \sum_{p \in C_i} \frac{||p_j - \mu_{i,j}||}{|D_i| \times |Data|}. \tag{9.9}$$

Eq. 9.9 allows both the mean $\mu_{i,j}$ and *cost* to be computed in parallel using a thread block for each pair of medoids m_i and dimensions j and then distribute the points among different threads within these thread blocks. Since this is the case, we can combine the computation of both into one kernel call, and synchronize all threads in each block to ensure that the computation of $\mu_{i,j}$ has finished before using it. By this, we can avoid writing $\mu_{i,j}$ to global memory but instead keep it as a shared variable. Writing to shared memory is much faster then writing to global memory and therefore provides a large reduction in running time.

To reduce the number of atomic operations in Algorithm 12, we use a local temporary variable that each thread can save its partial result to and then only perform one atomic operation per thread at the very end. This strategy is used both to compute the centroid μ_i and *cost*.

Algorithm 12 EvaluateCluster$(Data, C, D, k)$

1: **for** $i \leftarrow 1, \ldots, k$ - in parallel as blocks **do**
2: **for** $j \in D_i$ - in parallel as blocks **do**
3: $\mu_{i,j} \leftarrow 0$ // shared variable
4: $tmp \leftarrow 0$ // local variable
5: synchronize threads
6: **for** $p \in C_i$ - in parallel as threads **do**
7: $tmp \leftarrow tmp + p_j$
8: $\mu_{i,j} \leftarrow \mu_{i,j} + tmp/|C_i|$ // atomic
9: $tmp \leftarrow 0$
10: synchronize threads
11: **for** $p \in C_i$ - in parallel as threads **do**
12: $tmp \leftarrow tmp + ||p_j - \mu_{i,j}||$
13: $cost \leftarrow cost + tmp/(|D_i| \times |Data|)$ // atomic
14: **return** *cost*

Updated and iterations. We also update the best clustering, the best subspace, the best medoids, the current medoids, and the iteration counter for each iteration. However, this part is not time-consuming and details are therefore omitted.

RemoveOutliers. To remove ourliers, we compute the smallest distance between two medoids m_i, m_j, for each medoid m_i in parallel as block and each m_j using threads within that block and use atomics to find the smallest distance Δ_i. Then in parallel

across all points, we check if it lies within the Δ_i radius of any medoid m_i, else, it is reported as an outlier.

GPU-FAST-PROCLUS

The proposed strategies for reducing computations need modifications to fit the GPU. The lookup in the distance matrix *Dist* and H is as for FAST-PROCLUS. However, when computing the distances, in ComputeL, we must ensure that all threads have checked the flag $DistFound_{MIdx_i}$ before marking it as computed.

Since we would like to utilize as many cores as possible, we distribute the distance computations among multiple thread blocks. Instead of using community groups to synchronize across thread blocks, we set the flag afterward in a separate kernel call. Both λ_i and $|L_i^t|$ are computed as in FAST-PROCLUS, but only one thread per medoid m_i needs to compute this. Similarly for FindDimensions, when computing the average distance $X_{i,j}$, we must ensure that H is updated by all threads before computing $X_{i,j} \leftarrow H_{i,j}/|L_i|$. Therefore, $X_{i,j}$ is computed in a separate kernel call. The rest of GPU-FAST-PROCLUS proceeds as GPU-PROCLUS.

9.5 Experiments

We perform real-world experiments on a workstation with Intel Core i7-9750HF 2.6GHz 12-cores, 16 GB RAM, and a GeForce GTX 1660 TI 6 GB dedicated RAM. For the larger synthetic datasets, we move experiments to a workstation with an Intel Core i9 10940X 3.3GHz 14-Core, 258 GB RAM, and a GeForce RTX 3090 with 24 GB dedicated RAM. All algorithms have been implemented in C++ or CUDA. For repeatability, the source code is provided at: `https://au-dis.github.io/publications/GPU-FAST-PROCLUS/`.

Multi-core CPU-version. Some of the strategies proposed for GPU-parallelization are directly applicable to the CPU as well. We have therefore implemented multi-core CPU versions using OpenMP[1] to study the speedup of parallelization on the CPU vs. the GPU.

Algorithm parameters. The default parameters in all experiments are $k = 10$, $l = 5$, $A = 100$, $B = 10$, $minDev = 0.7$, and $itrPat = 5$.

CUDA kernel configurations. For the CUDA kernel configurations, the block size of 1024 threads is used. If fewer threads are required per block, only the required number of threads are started. To reduce unnecessary synchronizations in AssignPoints, Algorithm 11, 128 threads are used per block.

Synthetic data. For control of data distribution and size, we use the synthetic dataset generator provided by [12]. However, we modify the generator as [39] to generate clusters in any arbitrary subspace. The default parameters for the generated data are $64,000$ points with 15 dimensions, each dimension has values in the range

[1] https://www.openmp.org/

0 to 100. The points are distributed among 10 Gaussian distributed clusters in a subspace of 5 dimensions and with a standard deviation of 5.0.

Real-world data. For experiments on real-world datasets we use the datasets glass, vowel, pendigits [60] and part of the SkyServer dataset [76]. The glass dataset is of size 214 with 9 features, vowel is of size 990 with 10 features, and pendigits is of size $7,494$ with 16 features. From the SkyServer dataset, we use an area of size 1×1 measured in the spherical coordinates, referenced as sky 1×1. This subset contains $30,390$ points and we extract 17 features including the spherical coordinates. We also extract a 2×2 area with $133,095$ points and a 5×5 area with $934,073$ points.

All reported running times are averages of 10 runs on different generated datasets. The real-world and synthetic datasets are min-max normalized, such that all dimensions have values between 0 and 1.

Scalability

PROCLUS uses randomized search, but besides this random behavior, GPU-PROCLUS and all the algorithmic strategies produce the same clustering as PROCLUS. The important measure in this work is, therefore, not the accuracy but solely the running time. This section investigates how the size of the dataset and its dimensionality affect the running time of our proposed algorithms. We first compare against PROCLUS, where we run with just one parameter setting at a time. Later, in Section 9.5, we show how GPU-FAST-PROCLUS can achieve even higher speedup when allowed to reuse partial computations between parameter settings.

Figs. 9.2a-9.2b shows that the algorithmic strategies provide a factor of 1.2 to $1.4\times$ speedup for both PROCLUS and GPU-PROCLUS. However, the GPU-parallelization of each strategy provides an additional $2,000\times$ speedup. This speedup increases with the input size and stays constant after a certain input size. This is due to the more points, the easier it is to utilize all cores on the GPU. The speedup is so great that we can now perform PROCLUS in less than $100ms$, the limit for real-time interaction [74], for even $1,000,000$ data points. Similarly, the multi-core CPU-version provides up to $6\times$ speedup. The comparatively low utilization compared to the GPU could be due to the many context switches.

Figs. 9.2c-9.2d shows that the factor of speedup is higher for a lower number of dimensions, ranging from 896 to $1,265\times$ speedup. This could be caused by not all distance computations being parallelized across dimensions to avoid atomic operations and synchronizations.

Space usage. For FAST* compared to FAST, we see approximately 1.05 to $1.1\times$ slowdown, see Fig. 9.1, but with the benefit of a reduction in space usage. Fig. 9.3f investigate the reduction in space usage. Space usage of GPU-FAST*-PROCLUS is approximately half of that of GPU-FAST-PROCLUS and the space usage of GPU-PROCLUS and GPU-FAST*-PROCLUS is similar. Space usage of all algorithms increases linearly in n, which is inline with our space complexity analysis.

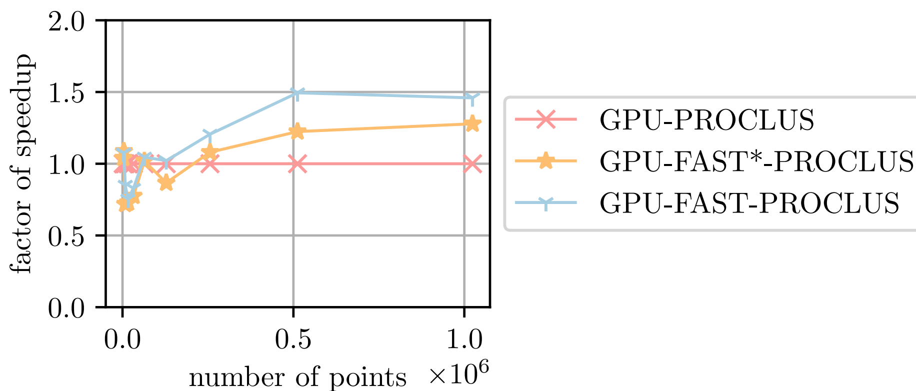


Figure 9.1: Speedup w.r.t. GPU-PROCLUS.

Effect of data distributions and parameters

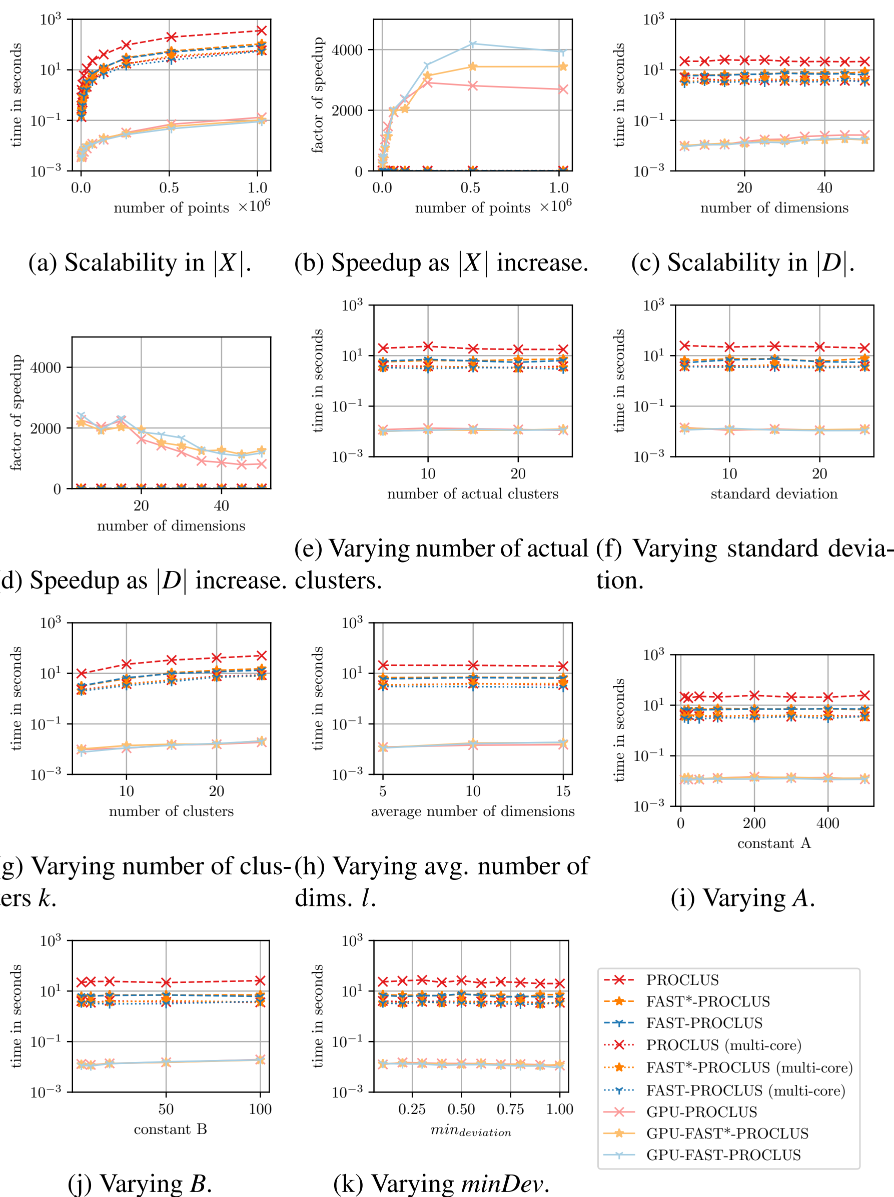
The performance of clustering, subspace, and projected clustering algorithms is affected by the data distribution. Therefore, we verify that GPU-PROCLUS performs well across data distributions. In Fig. 9.2e we vary the number of clusters and in Fig. 9.2f we vary the data distribution using different standard deviations. Here, we see that the running time of PROCLUS and GPU-PROCLUS is largely unaffected.

We show how different parameter settings affect the running time of PROCLUS and our proposed algorithms. In Fig. 9.2g-9.2k, we increase each of the parameters in PROCLUS one by one. We observe that the running time stays almost constant for most parameters, except for k and B, where we see that running time for both PROCLUS and GPU-PROCLUS increases with k or B. This is clearly because distances for a larger set of current medoids or potential medoids are computed. However, for all experiments, the factor of speedup remains relatively constant at around $1100\times$.

Multiple parameter settings simultaneously

As mentioned in Section 9.3, the result of PROCLUS depends on the parameters, so it is often run with multiple sets of parameters. GPU-FAST-PROCLUS uses this to reduce the number of distance computations. In Fig. 9.3a-9.3e, we show the average running time of testing 9 combinations of k and l. The reported running times are averages per combination to make it easier to compare with running times for just a single parameter setting. We see that GPU-FAST-PROCLUS provides up to around $7000\times$ speedup w.r.t PROCLUS. Furthermore, in Fig. 9.3e GPU-PROCLUS and GPU-FAST-PROCLUS run on more than $8,000,000$ points, and we see that the average execution time never exceeds a second. At $8,000,000$ points, space becomes the limiting factor, exceeding the 4.2 GB of free memory on our relatively small GPU.

The strategy of reusing computations between parameter settings consists of three



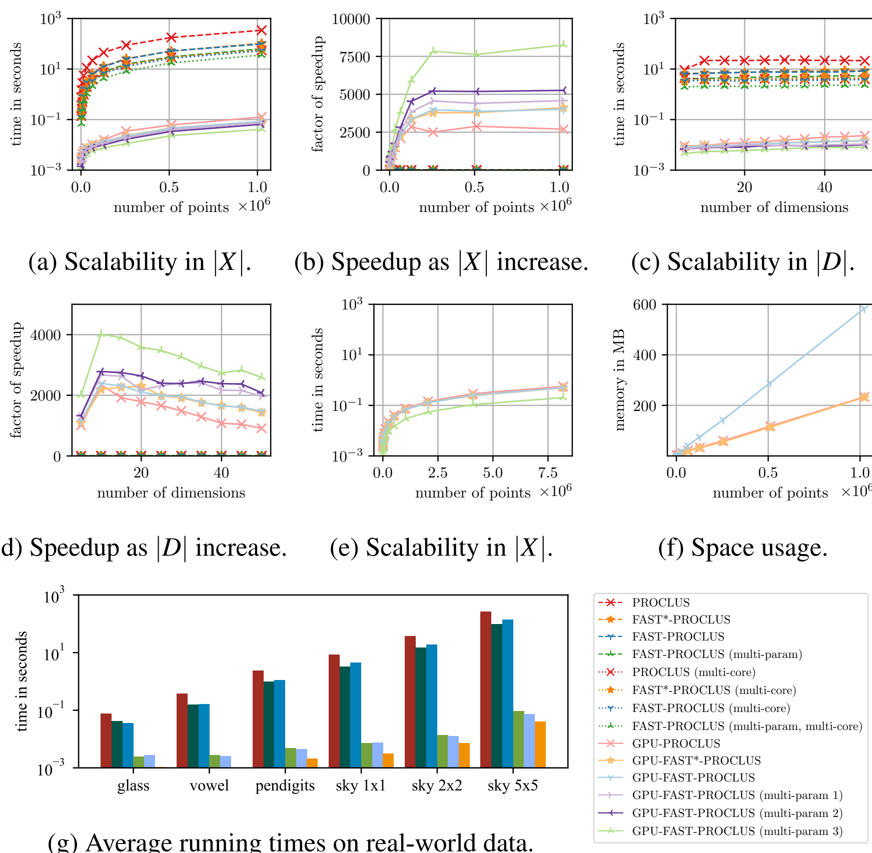
(a) Scalability in $|X|$. (b) Speedup as $|X|$ increase. (c) Scalability in $|D|$.

(d) Speedup as $|D|$ increase. (e) Varying number of actual clusters. (f) Varying standard deviation.

(g) Varying number of clusters k. (h) Varying avg. number of dims. l. (i) Varying A.

(j) Varying B. (k) Varying $minDev$.

Figure 9.2: Average running times of runs with a single parameter settings.



(a) Scalability in $|X|$. (b) Speedup as $|X|$ increase. (c) Scalability in $|D|$.

(d) Speedup as $|D|$ increase. (e) Scalability in $|X|$. (f) Space usage.

(g) Average running times on real-world data.

Figure 9.3: Average running times of runs with 9 parameter settings at a time.

parts: **multi-param 1** reuses partial compuations, **multi-param 2** reuses also greedy picking and **multi-param 3** reuses also the previous best set of medoids. Compared to GPU-FAST-PROCLUS executed with one parameter setting at a time, the reuse of partial computations provides approximately a factor $1.4\times$ speedup, also reusing the greedy picking provides approximately a factor $1.6\times$ speedup, and also initializing with the previous best set of medoids provides approximately a factor $2.3\times$ speedup.

GPU-utilization

The utilization of the GPU is dependent on many factors like memory throughput and and occupancy of the threads. Further more, it can very at lot between the dataset, parameters, and from kernel to kernel. We provide the utilization, inform of The memory throughput, theoretical occupancy, and achieved occupancy provided by NVIDIA Nsight Compute[2], for some of the most interesting kernels and extreme cases. An example of what could decrease the occupancy is if a block of threads

[2] `https://developer.nvidia.com/nsight-compute`

uses more than the registries that are available. The most time-consuming kernel is Algorithm 12. Given the parameter settings used in this section and a dataset with $4,096,000$ points and 10 dimensions, it has a theoretical occupancy of 100.00%, achieved occupancy of 99.99%, and memory throughput of 86.54%. Reducing the dataset size to $8,000$ points reduces the utilization to a theoretical occupancy of 78.12%, achieved occupancy of 77.98%, and memory throughput of 50.06%, this is because having $8,000$ points and 10 clusters implies that we spawn around 800 threads per block, which is not a good balance of the warps that can be executed per block. This kernel together with most of our other algorithms has a high utilization since we, in general, parallelize across a large number of points and try to keep the threads that need to communicate within the same block and do not exhaust the resources. On the other hand, a few kernels do not process all points, e.g., Algorithm 9 line 4-7 spans k blocks and k threads per block. If $k < 32$, we do not have enough threads per block to utilize a full warp and if $k \times k$ is less than the number of cores on the GPU, not all cores are engaged. If the preceding and the succeeding kernels were not depending on each other, streams could be used to run two kernels concurrently to engage more cores. This kernel has a theoretical occupancy of 50.00%, achieved occupancy of 3.12%, and memory throughput of 1.64%. This is not a good utilization, but not a time-consuming computation either, so these small kernels do not have a large effect on the overall running time.

Performance on real world datasets

Running with 9 parameter settings, we show that GPU-FAST-PROCLUS retains the high speedup on different real-world datasets. In Fig. 9.3g, we show the running time on different real-world datasets, and the experiments confirm that we obtain similar speedups for real-world datasets as we achieve for synthetic data. To be more specific, GPU-FAST-PROCLUS achieves $5490\times$ speedup compared to PROCLUS on the sky 5×5 dataset. As for the synthetic data, the speedup is greatest for large datasets.

9.6 Related work

Clustering is the task of grouping similar data points [82]. Distance-based methods like k-means [51] or k-medoids [61] try to minimize intracluster distance. Density-based methods like DBSCAN [29], DPC [67] and SynC [18] find clusters as high density regions separated by sparse regions.

In high dimensional data, clusters might only exist in subspaces of the full-dimensional space, giving rise to subspace and projected clustering [64]. Subspace clustering discovers clusters in any possible subspace projection, allowing a data point to participate in none or several clusters in different subspaces, whereas projected clustering assigns each point to exactly one cluster in one subspace projection. Subspace and projected clustering can be categorized as top-down or bottom-up algorithms.

Bottom-up approaches [6, 24, 33, 43] find clusters in $k = 1$ dimensional subspaces, then iteratively combine clusters in k-dimensional subspaces to find clusters in $(k+1)$-

dimensional subspaces. Top-down approaches [3, 4, 20, 26, 81] find clusters in the full-dimensional space and iteratively update assigned subspace projections to these clusters.

Subspace and projected clustering are time-consuming to compute due to the number of subspaces increasing exponentially in the number of dimensions, giving rise to works on efficient algorithms. Some algorithms reduce the running time by pruning subspace regions that cannot contain clusters [11, 43], approximating potentially dense areas by e.g. grid cells or histograms [6, 24, 33, 50], or by locally optimizing to iteratively improve candidate clusters and associated dimensions [3, 4, 20, 26, 56, 81].

While much effort has gone into algorithmic improvements of subspace and projected clustering algorithms, the high computational power of the graphics processing unit (GPU) remains largely unexplored. To the best of our knowledge, GPUMAFIA [1], a GPU version of SUBSCALE[25], and GPU-INSCY [39] are the only GPU algorithms proposed for subspace clustering. We review them in turn.

MAFIA is a bottom-up subspace clustering algorithm that combines histogram approximations of overlapping dense subspace regions as it moves from lower dimensional to higher dimensional subspaces [33]. GPUMAFIA efficiently parallelizes these core steps in different kernels. As PROCLUS uses a different clustering notion that makes use of neither histograms nor dense regions, and that cannot proceed in a bottom-up fashion on subspaces, the algorithmic parallelization strategies in GPUMAFIA are not applicable to the GPU-parallelization of PROCLUS.

SUBSCALE [45] is also density-based, but finds dense units, similar to neighborhoods in DBSCAN, per dimension. The dense units that overlap in dimensions can be combined and subjected to DBSCAN to derive the actual clusters. In the GPU-parallelized version of SUBSCALE[25], the identification of overlapping dense units across dimensions and dense units is parallelized, and possible dense units are precomputed. Again, the clustering notion and algorithmic strategy differ from PROCLUS such that it cannot serve as inspiration for a GPU-parallelization of PROCLUS.

Finally, INSCY [11] is also a density-based subspace clustering approach, but proceeds in a depth-first manner over potential dense subspace regions using a specialized tree structure. GPU-INSCY proposes a GPU-friendly version of the tree structure and devises algorithmic strategies for efficiently handling multiple dense regions in parallel. The GPU-parallelization is tailored to INSCY and does not fit for PROCLUS that does not operate on dense regions, and thus cannot benefit from a tree structured for managing them.

As PROCLUS is an adaptation of k-medoids to projected clustering [61], it is worth considering GPU-parallelized versions of full dimensional k-medoids clustering [46, 65], or of the similar k-means [30, 38, 49]. However, as opposed to k-medoids, which is based on distances between objects that reside in the same space, PROCLUS iteratively adds and removes dimensions from intermediate projected clusters, which results in changes of distance values and changes of projected subspaces. Thus, a GPU version of k-medoids cannot serve as a subroutine of a GPU version of PROCLUS.

9.7 Conclusions

We substantially improve the running time of PROCLUS for large-scale data to the extent that real-time interaction with PROCLUS projected clustering becomes possible. We achieve this in our GPU-FAST-PROCLUS, a GPU-parallelized version of PROCLUS that also contributes several algorithmic improvements. Our improvements are two-fold; efficient algorithmic strategies for PROCLUS, here termed FAST-PROCLUS, and an efficient parallelization on the GPU.

The algorithmic improvements target the most costly operations in PROCLUS, restructuring computations such that we can save and reuse distance computations and partial results. In addition, we introduce strategies that reuse partial computations across multiple parameter settings to further speed up FAST-PROCLUS. We demonstrate a trade-off between running time and space consumption, thereby allowing the user to adapt resource consumption as needed.

Our parallelized GPU-PROCLUS and GPU-FAST-PROCLUS are restructured to execute more operations in parallel, and to exploit the memory hierarchy of the GPU.

Our extensive experimental evaluation demonstrates 3 orders of magnitude speedup for GPU-FAST-PROCLUS. This speedup is stable across data distributions and parameter settings. The GPU-parallelizations provide up to 2000× speedup while the algorithmic strategies collectively provide around 2.5× extra speedup.

9.8 Acknowledgments

This work was supported by Independent Research Fund Denmark.

Chapter 10

AVID: GPU-enabled Visual Analytics with GPU-FAST-PROCLUS

Abstract

GPU-FAST-PROCLUS is a GPU-parallelized algorithm for projected clustering based on the k-medoids approach. It speeds up clustering to allow for real-time interaction – even for datasets of millions of items. Interactivity allows users to quickly determine sensible clustering parameters such as the number of clusters k, provided a suitable visualization is available. Yet, as clustering and visualization are usually decoupled, cluster results are funneled from the GPU back to the CPU, only to be mapped onto appropriate graphics, which are then rendered on the GPU again. This introduces a bottleneck that hinders fluid interaction with clustering.

As a solution to this, we propose AVID (Analysis and Visualization In Device). Following the principle "What happens on the GPU, stays on the GPU", AVID removes the round trip to the CPU and keeps clustering results on the GPU to render them on the GPU directly. By doing so, users can interactively tune projected clustering parameters and observe the effects without noticeable delay. In our demo system, we showcase the efficiency of our data management strategies for projected clustering as well as the efficacy of data visualization.

10.1 Introduction

Projected clustering aims to identify groups of similar objects in subspace projections of the full-dimensional space. Efficient algorithms for projected clustering are crucial as the number of possible subspace projections is exponential in the number of dimensions. Projected clustering algorithms must be provided with predefined parameters, but the best parameters are rarely known in advance. The choice of sensible parameters generally requires a human in the loop [28].

To enable interactive, human-in-the-loop parametrization of clustering, the effects of a change in parameters must be observable at interactive framerates. This usually means that results must be computed in around 100*ms* to reduce the *temporal separation* [79, p.140] between parameter change and visualization change,and thus providing the necessary "fluidity" [27]. In Jørgensen et al. [41], we present GPU-FAST-PROCLUS, a GPU-parallelized algorithm that computes projected clusters under the definition of the well-known PROCLUS approach [4], which extends k-medoids clustering to subspace projections. GPU-FAST-PROCLUS runs on a million points in around 100*ms*, and therefore theoretically allows for real-time interaction [74]. Yet, in order to visualize the results of GPU-FAST-PROCLUS to allow their interactive exploration under different parameterizations and in different projections – similar to the works by Tatu et al. [77] or Yuan et al. [85] – we would need to visualize these millions of points. To do so, the data would be clustered on the GPU (Graphics Processing Unit), then be transferred back to the CPU and mapped onto graphics primitives using some graphics framework, only to be then rendered again on the GPU.

To prevent the bottleneck of the CPU, we propose to compute both the cluster analysis and the visualization as a combined pipeline directly on the GPU. While GPU-based visualization is widely used [31, 66, 80], GPU-based Visual Analytics combining computational analysis and visualization on the GPU is still very rare with only a handful of systems having been published – e.g., [2, 57, 63]. To the best of our knowledge, no such purely GPU-based solution exists for computing and visualizing projected clusterings. Hence, we propose and demonstrate AVID (Analysis and Visualisation In Device), a real-time interactive data visualization for GPU-FAST-PROCLUS.

10.2 PROCLUS and GPU-FAST-PROCLUS

PROCLUS [4] is an axis-parallel projected clustering algorithm, inspired by the k-medoids algorithm CLARANS [61]. Given a dataset and the parameters

- number of clusters k,
- average number of dimensions l, and
- scalars A and B.

PROCLUS returns a cluster assignment for each point in some axis-aligned subspace projection for the respective cluster. To that end, PROCLUS proceeds in three phases:

1. Greedily picking potential medoids $M \subset Data$.
2. Iteratively improving the best set of current medoids $m \subset M$ that yields the best projected clustering
3. Further refining the best clustering.

The final result are k projected clusters within on average l-dimensional subspace. E.g., if we have $k = 3$ and $l = 4$, clusters could exist within subspaces of 2, 3, or 7 dimensions.

Our GPU-FAST-PROCLUS approach [41] provides efficient GPU-parallelization of PROCLUS clustering and even supports reusing computations between parameter settings, which is important in practice when determining the best set of parameters for a dataset and analysis task at hand. In Jørgensen et al. [41], we also provide an experimental evaluation on both real-world and synthetic datasets, and with varying size, dimensionality, distribution, and parameter settings. In the following, we provide a brief overview, with more details given in [41].

Speed-up is achieved by maintaining the distances $Dist$ from all points to all previously used medoids. Furthermore, the computation of scores $Z_{i,j}$, which indicate the suitability of medoid m_i in dimension j, is reorganized. The most expensive part of computing $Z_{i,j}$ is the sum of distances $H_{i,j}$ from each medoid m_i to all points that are within that medoid's sphere of influence L_i along each dimension j. The sphere of influence L_i is all points within a δ_i radius of m_i, where δ_i is the distance from m_i to the closest of the current medoids m_j. Since L_i is not likely to change much between iterations, the previously computed $H_{i,j}$ are stored, and updated only with the change ΔL_i in sphere L_i. To map between potential medoids M and the current medoids m, we use the index $MIdx$. All proposed reuses of computations can also be reused across parameter settings, provided that the selected sample of potential medoids M remains fixed.

In total, GPU-FAST-PROCLUS achieves up to 5000$\times$ speed-up over PROCLUS and can perform data analysis on a million points in around 100ms. Therefore, GPU-FAST-PROCLUS is admissible for real-time interaction [74]. Existing visualization frameworks, however, require data transfer to and from main memory and CPU involvement, dramatically increasing runtime beyond what users will accept in interactive settings.

10.3 Analysis and Visualization In Device

To leverage the speed of GPU-FAST-PROCLUS, we implement a "What happens on the GPU, stays on the GPU" data visualization supporting efficient visualization and interactive exploration of GPU-FAST-PROCLUS, called AVID (Analysis and Visualization In Device). We first discuss the implementation and then the data visualization.

Implementation

To the best of our knowledge, there does not exist a GPU-based data visualization framework that allows visualizing data directly located on the GPU. This implies that visualizing the result of GPU-FAST-PROCLUS must be funneled through the CPU and back to the GPU to be displayed. We, therefore, see the need to implement a visualization that does not use the standard data visualization frameworks to bypass

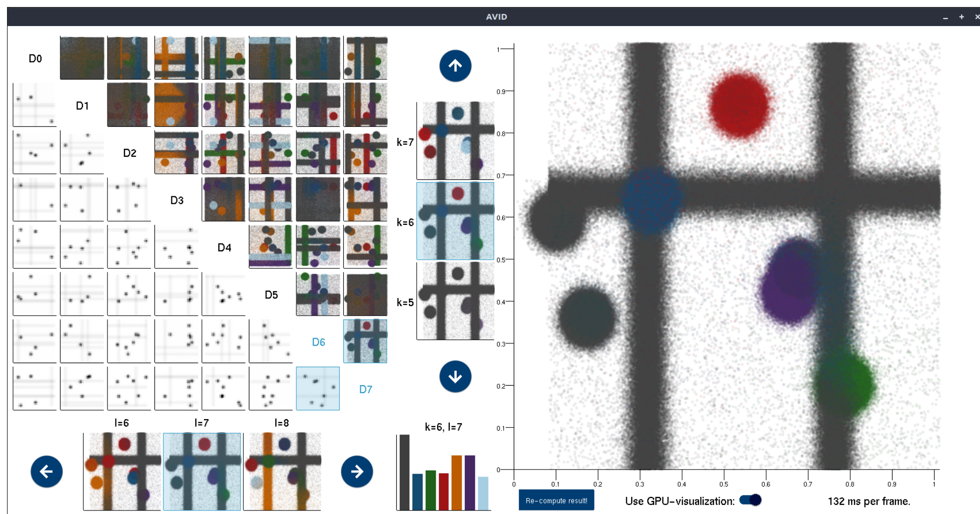


Figure 10.1: Layout of the data visualization AVID.

the CPU. AVID is implemented using OpenGL[1], GLEW[2], GLUT[3], and CUDA[4]. OpenGL, GLEW, and GLUT are application programming interfaces (APIs) for graphics rendering that can leverage the GPU to achieve acceleration. Furthermore, OpenGL and CUDA support resource interpolation between the interfaces which implies that the graphics displayed by OpenGL can be manipulated using CUDA kernels or the CUDA API directly, which is used to implement GPU-FAST-PROCLUS.

Like any graphics framework, OpenGL, GLEW, and GLUT support drawing dots, lines, polygons, and text. However, they do not directly support our needs for interactive data visualization. To enable brushing and selection of elements in the visualization, we have implemented layout components as a tree-structure and mouse move and click listeners for each component. The content of each plot, e.g., points, bars, and grid cells, is the part of the visualization that is related to the data and results, and therefore the part that must be computed on the GPU. For each content, the visual mapping to color, scale, and location is done in parallel for all data points using CUDA kernels and rendered directly on the GPU. This implies that the data and results never leave the GPU. For demonstration purposes, we also implement a second version where visualization is done on the CPU, but the data analysis and rendering are still performed on the GPU. The user can switch between implementations to experience the difference in runtime. To reuse computations between parameter settings, we keep $Dist$, $DistFound$, H, M, $MIdx$, $\delta^{t'}$, each $|L_i^{t'}|$, and the previous results in GPU-memory during the lifetime of the system.

[1]https://www.opengl.org/
[2]http://glew.sourceforge.net/
[3]http://freeglut.sourceforge.net/
[4]https://developer.nvidia.com/cuda-downloads

Effective and efficient visualization

Using these as a foundation, we can construct an interactive data visualization for GPU-FAST-PROCLUS. A common workflow, when exploring parameters for projected clustering, is for the user to select an initial set of parameters, wait for the result, plot the result in a scatter plot matrix, maybe zoom in to see a single scatter plot, adjust the parameters and repeat the whole process. Optimally, the user could see all possible results at once. However, having $k \times l$ d-dimensional scatter plot matrices, either requires a huge display or makes the scatter plots so small that they become indistinguishable. Instead of this tedious workflow, we propose a data visualization that has an overview, a detailed view, and previews of different parameter settings side by side. This allows the user to apply filters, selection in, and changing all views at once.

Layout: The layout consists of several components and can be seen in Figure 10.1. In the top left, we have an overview in form of a scatter plot matrix. The scatter plots in the lower triangle of the matrix provide the same information as the upper triangle and are therefore replaced with heat maps to allow the user to more clearly identify dense areas. In the scatter plots, the points in each cluster are assigned a unique color corresponding to that cluster, but only of the dimensions that the scatter plot represent are both in the subspace that the cluster represent. We further assign a gray color to all points, which are not part of a cluster within the subspace shown by a scatter plot.

Next, we have two menus for selecting parameters l and k, where l is the average number of dimensions in the subspace of the k clusters. These are located below and to the right of the overview, and we refer to them as l-menu and k-menu, respectively. Both menus consist of three scatter plots, each showing a change in the associated parameter and two buttons with an arrow. This allows the user to quickly see part of the result for the surrounding parameter settings, before changing the parameters for the whole view.

At the bottom right corner of the overview, we have a column chart displaying the size of each cluster and the outliers in the current result.

To the right, we have a detailed view that shows a larger version of a scatter plot selected in the scatter plot matrix. Below the detailed view, there is a set of tools: a button for recomputing the results, a switch to change where the visualization is being computed, and a display of the time it takes to compute each frame.

Selecting: In the scatter plot matrix, the user can select a scatter plot to show in the detailed view and the menus. The selected scatter plot is highlighted with light blue in the scatter plot matrix as well as the dimension id. In the menus, the user can click the buttons to increment or decrement the parameters previewed in the three scatter plots. When a scatter plot in the menu is clicked, the associated parameter is selected and all views are updated. Again, the selected scatter plot is marked with blue.

Filtering and linking: The user can filter points within a specific area by brushing the data points shown in the detailed view, or to a specific cluster by clicking on the bars in the column chart. Both filters can be applied at the same time and both are linked to all other scatter plots and heatmaps.

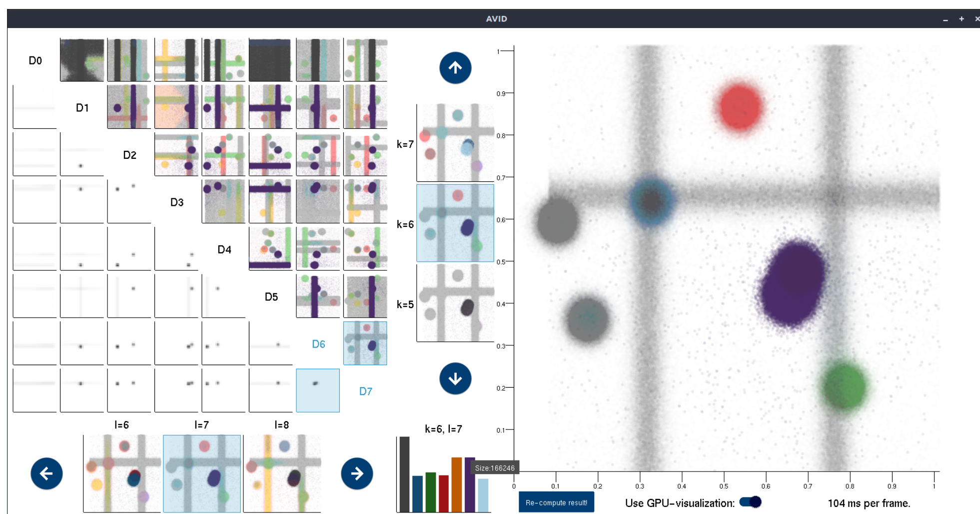


Figure 10.2: AVID showing a cluster that should be split in two.

10.4 Demonstration Plan

For the demonstration of AVID, we use a synthetic dataset to show the different mechanisms. However, the user can provide AVID with any dataset $Data \in \mathbb{R}^{n \times d}$ as a CSV-file. As examples, we provide real-world datasets with the source code. We here describe the demonstration, but a short video and source code is provided at `https://au-dis.github.io/publications/AVID/`.

We present the architecture and the frameworks used in AVID, as described in Section 10.3. Furthermore, we present the layout and options for selecting and filtering, as outlined in Section 10.3. To illustrate that AVID is fast enough for real-time interaction, we demonstrate how to use these mechanics to find a good set of parameters. An example could be the following (due to random initialization in PROCLUS, there can be variations):

Initially, we start with $k = 6$ and $l = 7$ as seen in Figure 10.1. We pick a scatter plot to show in the detailed view. We identify a cluster that maybe should have been split into two. Using the column chart, we filter to show only that cluster. The filter is applied in all plots and a look at the overview confirms that the cluster should be split in two, see Figure 10.2. In the k-menu, we investigate the previews. When changing the previews, a projected clustering is performed but happens so fast that it is not noticeable. We find at $k = 10$ the cluster is split, pick this parameter value and all views are updated accordingly. To update the previews for $l = 6$ and $l = 8$, two clusterings are computed using GPU-FAST-PROCLUS, and updated seamlessly. To investigate the old cluster, we filter using brushing in the detailed view, see Figure 10.3. In the overview, we see that it has indeed been split into two clusters. However, the clusters are also identified in subspaces that contains dimensions where the clusters are not dense. We conclude that l is too high and we look at the preview scatter plots in the l-menu. Again, changing the previews requires computing new projected

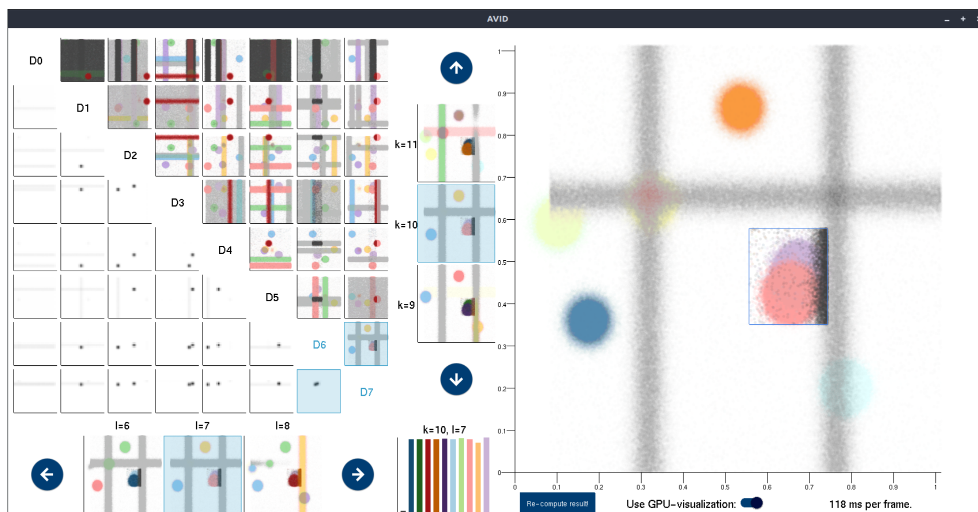


Figure 10.3: AVID showing brushing of two clusters.

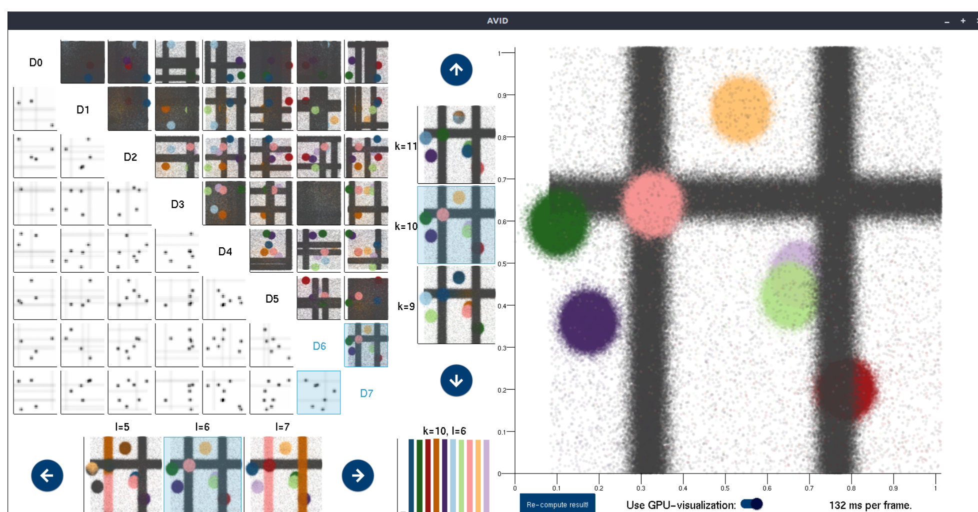


Figure 10.4: AVID showing the final clustering.

clusterings, but happens seamlessly. We pick $l = 6$ and in the overview, we see that the clusters only appear in the scatter plots where the clusters are dense, see Figure 10.4. Again, the last update requires two projected clusterings but occurs smoothly.

At last, we switch to computing the visualization on the CPU, to show the difference in interaction. Now the time to compute one frame increased from around a hundred milliseconds to a second.

10.5 Conclusions

We presented a demo of GPU-FAST-PROCLUS and a new data visualization that supports the exploration of parameters for the projected clustering algorithm PROCLUS. The fast GPU-parallelized version of PROCLUS makes the interaction real-time. To avoid funneling the results from the GPU through the bottleneck of the CPU and main memory just to transfer the visualization back to the GPU to be displayed, we implemented the pipeline for both clustering analysis and computing the data visualization fully on the GPU. This means that what happens on the GPU stays on the GPU. The proposed solution is a case study for GPU-FAST-PROCLUS, but could easily be adapted to fit any GPU-parallelized (projected) clustering approach.

10.6 Acknowledgments

This work was supported by Independent Research Fund Denmark.

Chapter 11

EGG-SynC: Exact GPU-parallelized Grid-based Clustering by Synchronization

Abstract

Clustering by synchronization (SynC) is a clustering method that is motivated by the natural phenomena of synchronization and is based on the Kuramoto model. The idea is to iteratively drag similar objects closer to each other until they have synchronized. SynC has been adapted to solve several well-known data mining tasks such as subspace clustering, hierarchical clustering, and streaming clustering. This shows that the SynC model is very versatile. Sadly, SynC has an $O(T \times n^2 \times d)$ complexity, which makes it impractical for larger datasets. E.g., Chen et al. [22] show runtimes of more than 10 hours for just $n = 70,000$ data points, but improve this to just above one hour by using R-Trees in their method FSynC. Both are still impractical in real-life scenarios. Furthermore, SynC uses a termination criterion that brings no guarantees that the points have synchronized but instead just stops when most points are close to synchronizing.

In this paper, our contributions are manifold. We propose a new termination criterion that guarantees that all points have synchronized. To achieve a much-needed reduction in runtime, we propose a strategy to summarize partitions of the data into a grid structure, a GPU-friendly grid structure to support this and neighborhood queries, and a GPU-parallelized algorithm for clustering by synchronization (EGG-SynC) that utilize these ideas. Furthermore, we provide an extensive evaluation against state-of-the-art showing 2 to 3 orders of magnitude speedup compared to SynC and FSynC.

11.1 Introduction

Clustering is the task of grouping similar objects to identify unknown structures in the data, e.g., customer groupings, and is one of the most common data mining tasks. Clustering by synchronization (SynC) [18] is a clustering definition that can capture

arbitrarily shaped clusters, requiring only a neighborhood radius ε and a threshold for the termination criterion. SynC is based on the Kuramoto model from physics which captures the natural phenomena of synchronization. In their paper, they show that SynC can capture clusters that visually stand out as actual clusters, which other clustering methods like DBSCAN [29] or k-means [30] do not. Furthermore, SynC [18] also provides a method to test increasing sizes of ε and only returns the clustering with the best score. This effectively hides ε for the user but at a much higher runtime. The SynC algorithm has shown to be versatile and has been used to solve several related data mining tasks such as outlier detection [70], hierarchical clustering [71], subspace clustering [72], and clustering streaming data [73].

The concept of clustering by synchronization is powerful, as seen in several papers [70–73], and experiments [18, 21]. However, SynC is very slow to compute due to the $O(T \times n^2 \times d)$ time complexity, where T is the number of iterations, n is the number of points, and d is the dimensionality. This complexity arises since, for each iteration, SynC computes the update using the ε-neighborhood by going through all points. FSynC [22] tries to remedy this by using an R-Tree to speedup the neighborhood query and achieves one order of magnitude speedup. However, the paper still reports that it takes more than an hour to cluster just $70,000$ points. Furthermore, SynC uses a measure r_c for synchronization; when r_c reaches 1, all points have synchronized with their neighborhoods. However, since the update never actually moves the points to the neighborhood's mean, r_c does not necessarily reach 1. SynC instead terminates whenever $r_c \geq \lambda$ which implies that not necessarily all points have synchronized and that SynC is effectively computing an approximation, with no bounds, of the definition of clustering by synchronization.

To get the necessary speed and accuracy, we provide an exact and fast algorithm. FSynC has investigated the use of data structures to speed up the computation, and this provides up to around $10\times$ speedup. To achieve further speedup, we see great potential in using modern hardware's high computational power, such as the graphic processing unit (GPU). However, to utilize the many cores of the GPU, algorithms and data structures must adhere to the computational model of the GPU, which is vastly different from the CPU model.

Our contributions. We propose:

- A new termination criterion for SynC that guarantees that the correct clustering is found,
- a strategy for partitioning the data into a grid of cells that can be summarized in a way that lets us compute the update of each point much faster,
- a GPU-friendly grid structure that supports this summarization and balances time and space efficiency,
- and an exact and fast GPU-parallelized algorithm for clustering by synchronization that utilizes these ideas.

All our contributions are manifested in a new exact and fast GPU-parallelized Grid-based algorithm for clustering by synchronization called EGG-SynC.

11.2 Related Work

In the literature, various approaches for data clustering are studied. SynC [18] is a clustering method that captures clusters revealed by synchronization, makes no assumption about data distribution, requires no human interaction, and allows the detection of clusters of arbitrary shape, size, and density. Density-based approaches like DBSCAN [29], and DENCLUE [37] also detect clusters of arbitrary shape and size but require a threshold for the global density of a cluster and do not capture clusters of varying density. DPC [67] and OPTICS [8] capture clusters of varying density, but require user selection of density parameters.

Some work on GPU-parallelizing clustering algorithms is based on analysis of neighborhoods, like G-DBSCAN [7], GPU accelerated OPTICS [55], and GPU-INSCY [39]. They mainly achieve speedup by precomputing neighborhoods in parallel in three stages; computing the size of each neighborhood, performing an inclusive scan to identify where each neighborhood should start and end in memory, and populating the neighborhoods. G-DBSCAN computes the neighborhood and then utilizes a GPU-parallelized breadth-first search (BFS) to assign the clusters. GPU-accelerated-OPTICS only computes the neighborhood on the GPU and the rest on the CPU. GPU-INSCY uses the neighborhoods in a subspace to prune the neighborhoods in its superspaces. An adaptation of G-DBSCAN processes multiple subspaces concurrently and grows clusters simultaneously instead of running BFS for each cluster. Precomputing neighborhoods comes at the cost of high space usage, $O(n \times \mathbb{E}[|N_\varepsilon(p)|])$. For SynC this would quickly become prohibitively large. Data points move closer to each other, implying that the expected neighborhood size becomes $O(\mathbb{E}[|N_\varepsilon(p)|]) = O(n)$. Therefore the space usage and runtime would become $O(n^2)$. Instead, we propose a space- and runtime-efficient method that both prunes and summarizes data points for computing the clustering. Moreover, because SynC changes the location of points, existing pruning strategies that rely on information about previously computed neighborhoods do not apply to SynC. We, therefore, propose a new strategy for pruning in this work.

The concept of clustering by synchronization is also used for outlier detection [70], hierarchical clustering [71], subspace clustering [72], and stream clustering [73]. However, a drawback of SynC is the complexity of $O(T \times n^2 \times d)$, which has given rise to works on improving its runtime. FSynC [22] is a version of SynC that uses the indexing structure R-Tree to support an efficient finding of the neighborhoods. This reduces the time it takes to find the neighborhoods from $O(n \times d)$ to $O(\log(n) \times d + |N_\varepsilon(x)| \times d)$. However, since SynC synchronizes the points of each cluster at a common location, the neighborhoods quickly become the size of each cluster. Even the best case, where the points are distributed equally among the k clusters, implies a $O(n/k \times d)$ runtime for each iteration. In their experiments, they show one order

of magnitude speedup. However, this still implies that it takes more than an hour to run FSynC on just 70,000 data points. As the neighborhoods become denser in the later iterations, FSynC becomes slower. We propose a strategy that leverages that the center of each neighborhood becomes denser to summarize regions fully within the neighborhood and avoid looking at the points in this region.

LSSPC [84] is a variation of SynC that can handle larger datasets. This is achieved by reducing the dataset using a method called CDC, running SynC on the reduced dataset, and assigning the remaining data points to clusters. CDC works by creating a minimum enclosing ball in an expanded feature space and, while there are still points outside the ball, expanding the ball to include the point farthest away from the center. The support vectors are returned as the reduced dataset when all points are covered. After running SynC, the remaining points are assigned to the cluster with the largest overlap with the neighborhoods. The remaining points are assigned to isolated clusters or outliers.

PSynC [21] is a CPU-parallelized SynC version that reduces the runtime by partitioning the dataset into areas with a roughly equal number of points, performing SynC on each partition in parallel, and merging the results to create the full clustering. Since the neighborhood of a point may span multiple partitions, PSynC computes the means of so-called K-neighborhood regions for each point as an approximation of their location. This approximation, unfortunately, comes without a guarantee as to the deviation from the correct result. PSynC also notes that the termination criterion is not exact and uses a criterion that considers the number of clusters but still does not provide an exact termination criterion. We propose the first exact SynC algorithm that provides efficient GPU-parallel computation and scalability to large datasets without the need for approximations.

Several clustering methods are based on k-nearest neighbors (kNN) [13, 62], which fixes the size of the neighborhood to k, instead of varying the number of points as in ε-neighborhoods. Some clustering definitions based on ε-neighborhood have been adapted to kNN-neighborhood instead. This provides results faster, but only approximately, e.g., the DPC approximation FastDPeak [23]. Another strategy is data summarization via some suitable set of statistics. BIRCH clustering [86] approximates sets of points as micro-cluster spheres, which are clustered to create the actual result. BIRCH assigns points to micro-clusters in their processing order, and the resulting approximation quality depends on this order. In this work, we focus on exact clustering according to the SynC cluster model without any loss in accuracy.

11.3 Clustering by synchronization

Clustering by synchronization (SynC) [18] is inspired by the natural phenomena of synchronization, e.g., a group of people with similar traits often come together and form common opinions; as time evolves, they become more similar and reach a state of local synchronization. The Kuramoto model from physics captures this interaction pattern. The basic idea of SynC is to iteratively move points closer to the points

Algorithm 13 SynC$(D, \varepsilon, \lambda)$

1: $t = 0$, $r_c = 0$
2: **while** $r_c < \lambda$ **do**
3: **for** $p \in D$ **do**
4: **for** $i = 0, ..., d-1$ **do**
5: $p_i^{t+1} = p_i^t + \frac{1}{|N_\varepsilon(p^t)|} \times \sum_{q \in N_\varepsilon(p^t)} \sin(q_i^t - p_i^t)$
6: $r_c = \frac{1}{|D|} \sum_{p \in D} \frac{1}{|N_\varepsilon(p^t)|} \sum_{q \in N_\varepsilon(p^t)} e^{-||q^t - p^t||}$
7: $t = t + 1$
8: **return** synCluster(D^t)

in their ε-neighborhood $N_\varepsilon(p) := \{q \in D|\ |p-q| \leq \varepsilon\}$, see Algorithm 13. Given a dataset $D \in \mathbb{R}^{n \times d}$, an ε radius, threshold λ, and a γ radius, the location of points $p \in D$ is iteratively updated using a function based on the Kuramoto model:

$$p_i^{t+1} = p_i^t + \frac{1}{|N_\varepsilon(p^t)|} \times \sum_{q \in N_\varepsilon(p^t)} \sin(q_i^t - p_i^t), \tag{11.1}$$

where i is the dimension and t is the current iteration. Since the sin function is used, distances must be within 0 and $\pi/2$ for points to approach each other. Böhm et al. [18], therefore, normalize the data between 0 and 1. Throughout this paper we use *drag* and *move* as synonyms for points being updated using Equation 11.1.

Instead of running the algorithm until the points have fully synchronized, Böhm et al. [18] compute what they call the Cluster Order Parameter:

$$r_c = \frac{1}{|D|} \sum_{p \in D} \frac{1}{|N_\varepsilon(p^t)|} \sum_{q \in N_\varepsilon(p^t)} e^{-||q^t - p^t||}. \tag{11.2}$$

Where r_c approach 1 when the points have synchronized. However, $r_c = 1$ is never reached, since we use sin to update the location, instead SynC terminates whenever $r_c \geq \lambda$. Böhm et al. [18] use $\lambda = 0.999$. When the points reach a state of local synchronization, the algorithm terminates, and the sets of points synchronizing together are returned as the clusters in the final clustering. SynC assigns the clusters by going through all not yet clustered points; all points within the γ-neighborhood constitute a cluster. However, the termination criterion does not guarantee that the points that synchronize are within a γ-neighborhood and, therefore, does not guarantee a correct result nor a bounded approximation quality.

SynC shows several desirable properties. It can capture arbitrarily shaped clusters with no assumption about data distribution, size, density, or the number of clusters. Furthermore, it naturally separates outliers from the cluster without specific measures, requiring no human interaction. As with all clustering methods, SynC also has its drawbacks. For SynC, we have identified the inaccuracy in termination and cluster gathering and the long runtime. We, therefore, strive to fix the inaccuracies and reduce the runtime.

FSynC [22] tries to reduce the runtime using R-Trees, an indexing structure that supports neighborhood queries, but their experiments show that it still takes more than an hour to cluster just $70,000$ points. We propose using modern hardware such as the Graphics Processing Unit to achieve a faster runtime. We also provide a GPU-friendly grid structure that can be used to correctly terminate when the points have synchronized, gather the clusters, and perform neighborhood queries. Furthermore, we propose a strategy to summarize the grid cells to achieve even higher speedup.

11.4 EGG-SynC

As mentioned in Section 11.1, SynC is a clustering concept with many advantages; however, SynC's biggest drawback is its slowness. As shown in Section 11.2, there have been multiple works on making SynC faster. PSynC gains up to $160\times$ speedup by partitioning the data and running SynC on each partition of different CPU threads, but at the cost of a less accurate result. On the other hand, FSynC does not lose accuracy and gains around $10\times$ speedup by using the R-Tree for indexing to speed up the neighborhood query used in the update function. However, FSynC still reports more than an hour of runtimes for just $70,000$ data points. Even though FSynC computes the same result as SynC, the λ-termination of SynC and FSynC does not guarantee a correct result. We are neither content with an approximative solution nor a runtime of hours for a relatively small dataset. To ensure a correct result, we propose a new termination criterion, and to achieve further speedup, we propose to utilize modern hardware such as the GPU; however, this requires developing an algorithm for a vastly different computational model. We thus present a new exact GPU-parallel grid-based algorithm for clustering by synchronization (EGG-SynC). Our proposed algorithm includes a novel exact termination criterion and proof of correctness. We also devise a GPU-friendly grid-based data structure to support neighborhood queries efficiently. To further reduce the runtime of the costly update function, we propose a strategy to summarize the points in the grid cells and use the precomputed values to reduce the number of points the update function needs to go through. We show how to compute the new update function in parallel across points. Furthermore, we use the grid structure to check the synchronization criterion, Definition 11.4.2, and gather the final clustering when the synchronization criterion is satisfied. At last, we collect the individual parts and propose our algorithm EGG-SynC.

Exact termination criterion

SynC aims to assign points that synchronize at the same location to the same cluster. Böhm et al. [18] define points synchronizing as a cluster. SynC uses a cluster order parameter r_c, Equation 11.2, as a measure of local synchronization. When r_c reaches 1, all points have synchronized. However, SynC uses the sin of the distance between the points to drag them closer, Equation 11.1, and for $0 < x \leq 1$, $\sin(x) < x$ implying that points that are not at the same location will never reach the same location, and r_c never reach 1. Instead SynC terminate whenever r_c exceeds λ where

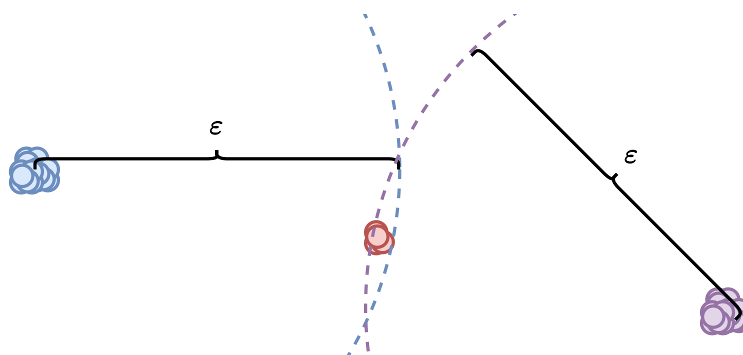


Figure 11.1: A cluster that should synchronize (be *dragged together* in later iterations), but where λ-termination incorrectly terminates with 3 separate clusters instead.

they use a value of $\lambda = 0.999$. The λ threshold provides a termination criterion, but the approximation quality of the clustering result depends on the choice of λ. Unfortunately, no guarantees on the quality of the approximation are provided. λ-termination, as we will call it for simplicity, may indeed produce incorrect results: consider, e.g., a small cluster on the border of the ε radius of larger clusters which in later iterations can "drag" the clusters together, see Figure 11.1. As we can see in the figure, the issue is that the impact of the smaller cluster on the termination condition in this iteration is small, but later iterations will make these clusters synchronize into a single cluster nonetheless. Consider a dataset of $1,000,000$ points, a $\lambda = 0.999$, and $\varepsilon = 0.025$, then there may be thousands of points in the small cluster, but the λ-termination criterion of r_c still reaches values above λ, even though the clusters should eventually be dragged into a single cluster. Furthermore, λ is an extra parameter that the user must set. The fact that λ-termination [18] does not indicate how close the points are to reaching their local synchronization point, is also noted by Chen et al. [21] who propose to add the number of clusters to the termination criterion. Still, their termination criterion suffers from the same fundamental problems. In this work, we propose a different approach to overcome these issues. Instead of defining an approximate measure of synchronization, we determine a state where the algorithm can safely terminate and gather points that eventually synchronize, thereby providing the first exact termination criterion. We begin with our formal definition of clustering by synchronization.

Definition 11.4.1 (Clustering by Synchronization)**.** *Given dataset D, parameter* ε*, and iterative updates using Equation 11.1. A non-empty* $C \subseteq D$ *is a cluster iff there exists an iteration t such that the following conditions are satisfied for all future iterations* t'*:*

1. $\forall t' \geq t, \forall p,q \in D : p \in C, q^{t'} \in N_\varepsilon(p^{t'}) \Rightarrow q \in C$

2. $\forall t' \geq t, \forall p,q \in C : q^{t'} \in N_\varepsilon(p^{t'})$

Given the Clustering by Synchronization, Definition 11.4.1, the state we want to capture in the synchronization criterion, Definition 11.4.2, is, therefore, when the neighborhoods do not change anymore, since this would imply that we know exactly which points will synchronize together when the iterations go towards infinity. To check this, we define two terms that should be satisfied for all points p. First term verifies that all points q within the ε-neighborhood of p are within the half radius $\varepsilon/2$ as well, implying that all neighborhoods either fully overlap $N_\varepsilon(q_1) \cap N_\varepsilon(q_2) = N_\varepsilon(q_1)$ or has no overlap at all $N_\varepsilon(q_1) \cap N_\varepsilon(q_2) = \emptyset$. The second term verifies that no points can be dragged into the ε-neighborhood of p using the update function. Since a point, q can only be dragged close to and not beyond all points within its neighborhood, the minimum bounding rectangle (MBR), the smallest axis-aligned hyper-cube that encloses a set of points, is a conservative approximation for where q can be moved. To conclude that q can not move into the neighborhood of p, therefore, it suffices to check that the $\varepsilon/2$-neighborhood's MBR of q does not intersect the ε radius of p.

Definition 11.4.2 (Synchronization Criterion)**.** *The criterion for termination is:*

$$\begin{aligned}\forall p \in D : \nexists q :& (\varepsilon/2 < ||p-q|| \leq \varepsilon)\\ \wedge\ \nexists q :& (\varepsilon < ||p-q|| \leq \varepsilon + \delta\\ &\wedge (dist(MBR(N_{\varepsilon/2}(q)), p) \leq \varepsilon),\end{aligned}$$

where $dist(MBR, p) = \sqrt{\sum_i^d \min_{c \in MBR} |p_i - c_i|^2}$ is the smallest Euclidean distance from any corner of the MBR to points p and $\delta = \varepsilon - \varepsilon \times \sqrt{\frac{15}{16}} + \varepsilon/2 - \sin(\varepsilon/2)$ is the extra radius that must be checked, see Appendix 11.11 for proof.

We provide the following theorem and lemmas to prove that this criterion ensures that the final clustering is correctly determined. We want to prove that the first term implies that all points sharing a common neighborhood always move closer to each other. First, Lemma 11.4.3 proves that if the first term is satisfied for all points, then all intersecting neighborhoods are fully intersecting. Next, Lemma 11.4.4 proves that all points with a common neighborhood move closer to each other.

Lemma 11.4.3 (Identical set of neighbors.)**.** *Given a point $p \in D$, if there does not exist a point $q \in D$ where $\varepsilon/2 \leq ||p-q|| \leq \varepsilon$, then all points $o \in N_\varepsilon(p)$ must have the same neighbors, i.e., $N_\varepsilon(p) = N_\varepsilon(o)$. See Appendix 11.8 for proof.*

Lemma 11.4.4 (Denser neighborhoods.)**.** *Given points $p, q \in D$ at iteration t, if $N_\varepsilon(p) = N_\varepsilon(q)$ then $||p^{t+1} - q^{t+1}|| \leq ||p^t - q^t||$, i.e., the distance between points is smaller in subsequent iterations. See Appendix 11.9 for proof.*

We now have the foundation to prove that no points leave the neighborhood. Even though points are being updated closer to their neighborhood and, therefore, further from other points, there is still a small chance that points in a neighborhood could drag themself into another neighborhood and, by that, merge the two neighborhoods. To prove that if the second term is satisfied this can not happen we provide Lemma

11.4.6. Since the update Equation 11.1 can tilt slightly from a straight line, we need to prove that this becomes smaller in later iterations, to support this we provide Lemma 11.4.5.

Lemma 11.4.5. *Given $x, y \in (0,1]$ and $y > x$, then:*

$$\frac{\sin(y - \sin(y))}{\sin(x - \sin(x))} > \frac{\sin(y)}{\sin(x)}. \tag{11.3}$$

See Appendix 11.10 for proof.

Lemma 11.4.6. *Given $p, q \in D$ at iteration t if the synchronization criterion, Definition 11.4.2, is met and $q^t \notin N_\varepsilon(p^t)$ then $\nexists t' > t : q^{t'} \in N_\varepsilon(p^{t'})$, i.e., no new point q can move into the neighborhood $N_\varepsilon(p)$ of any point p. See Appendix 11.11 for proof.*

We provide Theorem 11.4.7 to collect all the parts and conclude that when the synchronization criterion Definition 11.4.2 is met, we can correctly gather the final clustering.

Theorem 11.4.7 (Gathering Clusters)**.** *Given $p \in D$ at iteration t, if the synchronization criterion, Definition 11.4.2, is met then $N_\varepsilon(p^t) = C | p \in C$, i.e., $N_\varepsilon(p)$ is the set of points that p synchronizes with and, therefore, the final cluster that p belongs to.*

Proof. By Lemma 11.4.3, since all points only have neighbors within the $\varepsilon/2$ neighborhood, all points must have the same points in their neighborhoods as their neighbors do. By Lemma 11.4.4, all points with identical neighborhoods move closer; thus, the neighborhoods never lose points. Lastly, by Lemma 11.4.6, no points move into the neighborhood when the synchronization criterion, Definition 11.4.2, is met. This implies that the neighborhoods do not change anymore. Therefore, when the synchronization criterion is met, each point's p neighborhood is the set of points that p synchronizes with and, therefore, the final cluster that the point p belongs to according to Definition 11.4.1. □

With this proof, we establish the first exact termination criterion for clustering by synchronization, determining a state where the algorithm can safely terminate and gather the final clusters.

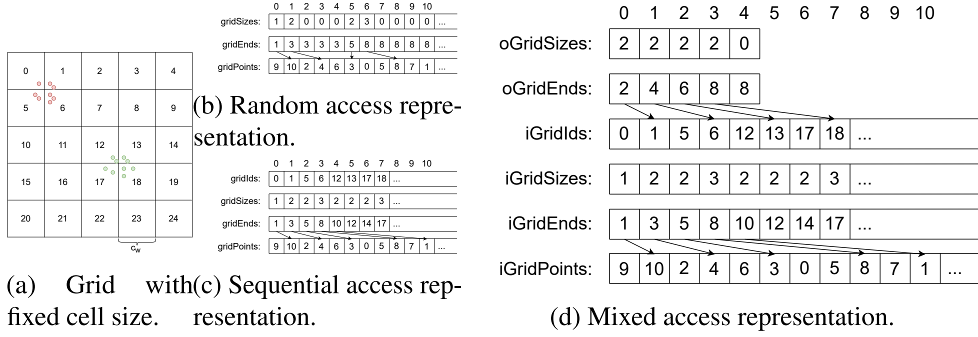
GPU-friendly grid structure

Clustering algorithms have been primarily developed with the implicit assumption of a sequential single-threaded model of the CPU. Modern hardware architectures, however, employ different computational models requiring different algorithmic solutions. The modern CPU contains up to tens of cores where threads can execute individual instructions concurrently as SMT (Simultaneous Multi-Threading). It provides hardware units that can execute a single instruction on hundreds of data entries simultaneously as SIMD (Single Instruction, Multiple Data).

In this work, we propose to exploit the massive parallelism in modern GPUs (Graphics Processing Units) for efficient SynC clustering. GPUs consist of thousands of cores that provide high computational power at the cost of a more restricted computational model, where warps, groups of 32 threads, execute with a shared program counter. All threads in a warp execute the same operation as SIMT (Single Instruction, Multiple Threads). In the CUDA programming environment, threads are organized into blocks and further distributed among warps. The blocks are, furthermore, organized in a grid. Physically, cores on the GPU are grouped in SMs (Streaming Multiprocessors), which share fast access to L1-cache and can synchronize during execution. Thread in a block is executed within the same SM and therefore has the capabilities of the SM. Furthermore, all threads can access the slower main memory of the GPU, known as global memory. Due to threads accessing memory concurrently, several considerations must be taken into account. Threads that access the same memory address can lead to race conditions, and atomic operations can be used with care to avoid these; however, the atomic operation takes longer to perform. Global memory access by the same warp can be combined into one transfer if the memory accesses coalesce; this requires that the memory accesses are consecutive and aligned with global memory. In this paper, *parallel* is used to denote parallel execution on the GPU unless specified otherwise. *Computed in parallel* means distributing a for-loop among thread blocks as well as threads within each block.

The main and most time-consuming operation of Equation 11.1 is to compute the neighborhood of each point. Using just the dataset requires going through the entire dataset each time the neighborhood is computed. Therefore, an indexing structure is often used to speed up the neighborhood query. Tree structures, such as the R-Tree used in FSynC, support fast neighborhood queries on the CPU; however, they are not constructed with the GPU in mind. When constructing a tree, as nodes in the tree reach their maximal capacity, they are split in two, and the tree's structure is altered. If we try to construct an R-Tree in parallel, points may be inserted by some threads while the tree is being altered by others and therefore could end up in the wrong node. Furthermore, when performing a neighborhood query, each thread may need to go through multiple different parts leading to branch-divergence, which slows down performance substantially. Instead, we propose to use a grid structure to speed up the neighborhood query. This naturally reduces the runtime of the neighborhood queries, but in addition, we devise a strategy to summarize the grid cells such that we do not have to access all points during the update, following Equation 11.1. We also show how to check the synchronization criterion using the grid structure and gather the final clustering.

When designing the grid structure for the GPU, the main considerations are access-time, query-time, construction-time, space-usage, and how to construct and access it in parallel using warps. To make it easy to compute which grid cell a point is located within, we decide to use a grid structure with a fixed cell width c_w, implying that the



(a) Grid with fixed cell size. (b) Random access representation. (c) Sequential access representation. (d) Mixed access representation.

Figure 11.2: Grid representations

ID of a cell containing a point p is

$$ID(p) = \sum_{i=0}^{d-1} \frac{p_i}{\lceil 1/c_w \rceil^i} \mod \lceil 1/c_w \rceil, \tag{11.4}$$

where IDs are enumerated as seen in Figure 11.2a. Similarly, this approach also makes it easy to determine the cells that possibly intersect the neighborhood radius ε.

In order to fully utilize the many GPU cores when constructing and accessing the grid structure, we must devise a GPU-friendly implementation. Recall that all threads within a warp must perform the same instruction at all times, or else branches diverge, slowing processing down. We must also ensure that a thread does not change part of the structure other concurrent treads are working on. Furthermore, we must also consider data access and workload distribution to achieve the highest performance. Likewise, dynamic memory allocation is relatively expensive to perform, and we aim to reduce this as much as possible.

List construction

In order to handle the frequent creation of lists or sets of elements, we need to consider memory allocation. In the sequential model, elements can be added to lists on the fly and memory allocated for lists is expanded when needed. Since dynamic memory allocation is expensive on the GPU and threads in a warp wait for each other, allocating memory on the fly may quickly lead to a very high runtime. Instead, we adopt the strategy of maintaining several lists per single memory allocation. First, compute the size *sizes* of each list, then perform an inclusive scan of the sizes to find the end index *ends* of each list, allocate the total space for the elements in all lists *elements*, and at last populate each list. This requires only one memory allocation for all lists combined and makes the memory coalesced, both of which are important for efficiency. If the total size is fixed between iterations, the memory allocation can even be reused. The start index of list number i is:

$$\text{getStart}(ends, i) = \begin{cases} 0 & \text{if } i = 0 \\ ends[i-1] & \text{else} \end{cases},$$

and end index as getEnd$(ends, i) = ends[i]$. Here, we use standard C++ notation, where the end is the index after the last entry.

Random access

The first implementation we propose represents all possible cells in the grid structure enumerated as in Figure 11.2a, where the grid cell of a point is determined using Equation 11.4. To access the points in each cell, we compute lists of points as in Section 11.4, with the lists of points in *gridPoints*, their sizes in *gridSizes*, and the end index of each list in *gridEnds*.

Access. Since the grid cell width is fixed, the index of each cell can be computed in $O(d)$ time and the cell can be located by random access. The total look-up time is, therefore, $O(d)$ to access the grid cells and $O(d + |g|)$ to get all points in the grid cell. Since we do not know the non-empty cells in advance, we must look at all cells intersecting the neighborhood of a point. This makes the complexity of updating each point p $O(v^d + |N_\varepsilon(p)|)$, where $v = \lceil \varepsilon / c_w \rceil \times 2 + 1$ is the possible number of grid cells along each dimension that the ε radius can overlap with.

Construction. The random access grid structure is a set of lists of points constructed as described in Section 11.4. This grid structure is illustrated in Figure 11.2b. We discuss how to add the summarized statistics for all cells in Section 11.4. Since we have an index for all cells, also non-empty ones, the number of cells increases exponentially with d, i.e., $O(w^d)$ space usage, where w is the number of cells along each dimension. For lower-dimensional data, this representation is efficient because the time complexity depends on the number of dimensions. However, for higher-dimensional datasets, it leads to space issues; therefore, we propose an alternative structure for higher-dimensional data in the following.

Sequential access

To reduce the space usage, we could represent the grid as a list of non-empty cells, this way we would never use more than $O(n \times d)$ space. Since there is no direct mapping between the array entries and the cells in this representation, we must keep track of which cell each entry represents. We, therefore, maintain an array *gridIDs* of the IDs of each non-empty cell, see also Figure 11.2c. In this paper, we view it as a single value for simplicity's sake, but in reality, these IDs can become quite large since the number of cells increases exponentially in the dimensionality. It is, therefore, represented using $O(d)$ integers in the implementation. Furthermore, we use two arrays for housekeeping to remove duplicates and tightly pack the non-empty cells. An array *gridIncl* to mark which cells should be included and an array *gridIdxs* containing the new index of the first occurrence of each non-empty cell.

Access. To find a specific cell, we scan the array *gridIDs*, implying a $O(n \times d)$ access-time. However, to retrieve the neighborhood of a point p, we only need to traverse the list once, and the complexity is $O(n \times d)$ as well.

Construction. In the construction of this implementation, we additionally keep a list of non-empty grid cells but otherwise again separate it into multiple steps. We need to create all cells in parallel, but we do not know which cells are empty in advance, and we do not want to create duplicates. First, we go through all points in parallel and compute the cell ID *cID*, which is saved at the index corresponding to the point's ID $gridIDs[p] = cID$. The array *gridIDs* is now a list of all non-empty cells but possibly with duplicates. Next, to remove duplicates and compute the location of the points in each grid cell, we go through each point in parallel, compute the cell ID *cID* it belongs to and find the first entry *idx* in *gridIDs* matching *cID*. At the corresponding location *idx* in *gridSizes*, we increment the size and set *gridIncl* to TRUE to mark that this cell ID is the first of multiple duplicates and, therefore, is the one that should be included. All other duplicates can be ignored. Next, we perform an inclusive scan on *gridIncl* to find the location idx' for where each non-empty cell needs to be placed to be tightly packed and save the result in *gridIdxs*. Similarly, we perform an inclusive scan on *gridSizes* to find the end index of the cell's list of points and save it in *gridEnds*. We set *gridSizes* to zero and populate the cells' list of points as in Section 11.4. At last, we repack the grid structure such that all first occurrences are tightly packed at the beginning. This is done for each point in parallel. If the cell is marked as included $gridIncl[p]$, we move the end index, the size, and the cell ID to the new location $idx' = gridIdx[p] - 1$.

Theoretically, the sequential and random access grid structures have the same worst-case access time since all points could be within the neighborhood, making the query time $O(n \times d)$. However, for most datasets, we do not just have a single dense area at the neighborhood's size; therefore, the random access representation would be the fastest. On the other hand, the space complexity of the random access structure is exponential in d making it impractical for higher-dimensional datasets, where the sequential access structure uses $O(n \times d)$.

Mixed access.

Both the random access and the sequential access representation have their drawbacks. To get the best of both worlds, we propose a mix of these representations that balances the access time and the space usage. It is a heuristic to distribute the long list of grid cells into as many buckets with random access cells that we can maintain in $O(n \times d)$ space.

To get a compact representation of the grid structure, we create a random access grid structure for the first d'-dimensions only, where $w^{d'} \leq n \times d$. We refer to this partial structure as the outer-grid *oGrid*. Then for each cell in the outer-grid, we keep a list of all full-dimensional non-empty cells that fall within the outer-grid cell. We refer to these full-dimensional cells as the inner-grid *iGrid*, which is implemented as a sequential access grid structure. The outer-grid allows us to quickly locate a subset of cells in the inner-grid that potentially intersect the neighborhood. We then sequentially check each cell if it intersects the neighborhood.

Similar to the random access, the outer-grid structure consists of an array *oGridSizes* with the number of inner-grid cells, and an array *oGridEnds* with the end-locations, see Figure 11.2d. However, instead of indexing into the end of a list of points, it indexes into the inner-grid cells within each outer-grid cell, see Figure 11.2d. The inner-grid is exactly the same as the sequential access, but the inner-grid cells are grouped by the outer-grid cells.

The space use of the outer-grid is $O(n \times d)$ by choice of d' and the inner-grid can at most have n non-empty grid cells and, therefore, uses $O(n \times d)$ space. Therefore, the total space usage is $O(n \times d)$. For higher-dimensional datasets, this is much better than the $O(w^d)$ for the random access representation and as good as the sequential access strategy, but we can still access portions of the cells by random access.

Access. To retrieve the neighborhood, we identify each of the outer-grid cells intersecting the neighborhood of p this is $O(v^{d'}) = O(n \times d)$. Then we traverse the list of inner-grid cells in each outer-grid cell, we can again at most have n non-empty inner-grid cells and, therefore, this also takes $O(n \times d)$ time. In total, we use worst-case $O(n \times d)$ to query the neighborhood. However, in practical experiments, EGG-SynC performs much faster, see Section 11.5.

Construction. The mixed access grid structure is constructed by first building the random access grid structure for the first d' dimensions, and then for each cell, building a sequential access grid structure for the full dimensional space, as described in Algorithm 14. All arrays are allocated at the beginning of Algorithm 16 and reused in all iterations to avoid expensive memory allocations. Alongside the construction description, we provide an example of how the arrays change in Figure 11.2d (small captions with respective algorithm lines). To construct the outer grid, we aim to fill it with the non-empty inner grid cells. However, it is challenging to avoid duplicates in parallel processing. In Lines 1-3, we instead temporarily accept potential duplicates, for each point adding the ID of the inner grid cell the point is located in, then removing duplicates in Lines 4 where we mark the first occurrence of each inner cell ID to be included. To save computations and memory accesses, we count the number of points in each inner grid cell at the same time. At this stage, the first occurrence of non-empty inner grid cells is spread out sparsely in the outer grid. To compute a compact index *iGridIdxs*, Line 5 performs an inclusive scan on *iGridIncl*. In Lines 6,7, the inner grid cells are populated as in Sect. 11.4. Lines 8-10 repack the grid using the computed indices *iGridIdxs* into new arrays to avoid breaking the old structure while reading from it.

To conclude, we now have a data structure that uses $O(n \times d)$ space and where a neighborhood can be found in worst-case $O(n \times d)$ time, but likely much faster. This is the same as the R-Tree used in FSynC; however, our grid structure can be constructed and accessed by multiple GPU threads in parallel and supports the summarization that we discuss in Section 11.4.

Algorithm 14 constructGrid(D, ε)

1: $\forall$ points $p \in D$ in parallel: atomically increment the size of each outer grid cell *oGridSizes*
2: inclusive scan of *oGridSizes* saved in *oGridEnds*
3: $\forall p \in D$ in parallel: atomically add the inner cell ID *iID* containing p to the list of inner grid cell in the outer grid cell containing p with ID *oID*
4: $\forall p \in D$ in parallel: compute outer cell with ID *oID* and inner cell with ID *iID* containing p, find the first occurrence of *iID* in the list of inner grid cell in *oID*, and mark it as included; atomically increment the size of each inner grid cell
5: inclusive scan of *iGridIncl* saved in *iGridIdxs*
6: inclusive scan of *iGridSizes* saved in *iGridEnds*
7: $\forall p \in D$ in parallel: atomically add p to the inner grid cell
8: $\forall iIdx$ in parallel: relocate ends $iGridEnds[iIdx]$ and ids $iGridIds[iIdx]$ to new location $iIdx' = iGridIdxs[iIdx] - 1$
9: $\forall oID$ in parallel: compute new end of each outer grid cells list of inner grid cells $oGridEnds'[oID] = getStart(iGridIdxs, oGridEnds[oID])$
10: swap(*iGridIDs*, *iGridIDs*$'$), swap(*iGridEnds*, *iGridEnds*$'$), swap(*oGridEnds*, *oGridEnds*$'$)

Precomputing the surrounding cells

When computing the update to a point p, the thread handling this point must access the surrounding grid cells to see if they contain any points that should be included in the neighborhood. Many of these cells are empty, and for the random access strategy, this implies that some threads, in a warp, access empty cells while other threads access non-empty cells. In turn, this results in some threads waiting on other threads finishing treating the non-empty cells before continuing to the next cell. To reduce the number of idle threads, we precompute the non-empty surrounding cells of each cell in advance. The ids of the non-empty cells are saved in an array *preGridCells* similar to the points in our grid structure.

First, we compute all non-empty cells *preGridNonEmpty* in parallel by atomically incrementing the number of non-empty cells *noOfNonEmpty* and saving the outer-grid cell ID *oID* at that location. Second, in parallel for each non-empty cell *cID*, we go through the surrounding cells and count the non-empty cells $preGridSizes[cID]$. Third, to get the start and end indices *preGridEnds* of each list of surrounding outer-grid cells in
preGridCells, we perform an inclusive scan on the counts of surrounding cells *preGridSizes* to get the end index of each list. At last, we set the counts *preGridSizes* to *zero* and, in parallel, for each non-empty cell *cID*, we go through the surrounding cells. If it is non-empty, we increment the count $preGridSizes[cID]$ to get the location $loc = atomicInc(preGridSizes[cID])$, compute the starting location $offset = getStart(preGridEnds, cID)$ in *preGridCells* and save the surrounding non-empty cells *oID* at that location loc plus the starting location $offset$. Having this precompu-

tation implies that threads always work with non-empty cells; however, the number of points contained in each cell can still differ. We address this issue in the following.

Execution order

Precomputing the surrounding non-empty grid cells implies that no threads have a non-empty workload per cell we handle. However, we still have an unbalanced workload since some cells can contain a lot of points and others only a few. To make it more likely that threads in the same warp have a balanced workload, we aim to have warps handle points that are located close to each other. To achieve this, we leverage that the array of all points in the grid structure *iGridPoints* is sorted in order of the grid cells. Instead of updating each point in the order given in the data set, we access them in sorted order in the grid structure. This implies that it is much more likely that all threads in a warp access the same surrounding grid cells and, therefore, have a more balanced workload since the threads are likely to access the same points. However, it is still possible that threads in a warp handle points in different grid cells. It would be possible to make a warp only handle points within the same grid cells and just let the remaining threads do nothing, but this would lead to even lower utilization of the threads.

Efficient cluster algorithm on the grid

We now have the definitions needed for an exact clustering by synchronization and a GPU-friendly grid structure to support our summarization strategy. This section proposes our summarization strategy and an exact GPU-parallelized grid-based algorithm for clustering by synchronization (EGG-SynC).

Summarized grid cells

Supporting neighborhood queries provide speedup, but the worst-case complexity of SynC using any indexing structure is still quadratic in the number of data points. In each iteration, to update each point p requires all points in the neighborhood $N_\varepsilon(p)$, Equation 11.1. This is an expensive task since, as the points synchronize, the neighborhoods become larger and larger until they contain an entire cluster. If one cluster contains the majority of the points, updating each point in this cluster would take $O(n \times d)$. Even if the k clusters are of equal size, the update still takes $O(n/k \times d)$. This implies that the complexity of SynC is still $O(T \times n^2 \times d)$. The challenge is, therefore, the inherent problem of SynC, that the neighborhoods become extremely dense and that SynC has to look at all points when computing Equation 11.1. We propose an entirely new approach that avoids these costly computations in many cases. The core idea is to precompute summarized statistics that fulfill several requirements. Since grid cells can be fully included within multiple neighborhoods, the summarized statistics should be computed per grid cell and be reusable among points when computing the update using Equation 11.1. To ensure an exact result, the summarized statistics should not provide an approximation but sufficient information

for correct updates. At last, it should be efficient to precompute the summarized statistics.

It is known that $\sin(y-x) = \sin(y)\cos(x) - \cos(y)\sin(x)$. We use this to rewrite the update of a point p:

$$
\begin{aligned}
p_i^{t+1} &= p_i^t + \frac{1}{|N_\varepsilon(p^t)|} \sum_{q \in N_\varepsilon(p)} \sin\left(q_i^t - p_i^t\right) \\
&= p_i^t + \frac{1}{|N_\varepsilon(p^t)|} \sum_{q \in N_\varepsilon(p)} \sin\left(q_i^t\right)\cos\left(p_i^t\right) \\
&\quad - \cos\left(q_i^t\right)\sin\left(p_i^t\right) \\
&= p_i^t + \frac{1}{|N_\varepsilon(p^t)|} \left(\cos\left(p_i^t\right) \left(\sum_{q \in N_\varepsilon(p)} \sin\left(q_i^t\right) \right) \right. \\
&\quad \left. - \sin\left(p_i^t\right) \left(\sum_{q \in N_\varepsilon(p)} \cos\left(q_i^t\right) \right) \right). \qquad (11.5)
\end{aligned}
$$

The sums $\sum_{q \in N_\varepsilon(p)} \sin(q_i^t)$ and $\sum_{q \in N_\varepsilon(p)} \cos(q_i^t)$ can then be separated into two parts, one for the points in the grid cells GF fully within the neighborhood, and one for the points in grid cells GP partially within the neighborhood:

$$
\sum_{q \in N_\varepsilon(p)} \sin(q_i^t) = \sum_{g \in GF} gridSin[g]_i + \sum_{g \in GP} \sum_{q \in g \cap N_\varepsilon(p)} \sin(q_i), \qquad (11.6)
$$

where $gridSin[g] = \sum_{p \in g} \sin(g)$ and analogically for cos, which are the terms that we can precompute and use in multiple updates. This implies that instead of going through all the points in the grid cell, we can look up the precomputed sums and use them, saving us a lot of time. This precomputation is computed in parallel across points, by computing the grid cell ID and adding the $\sin(p)$ and $\cos(p)$ atomically to $gridSin[g]$ and $gridCos[g]$. This makes it extremely fast in practice compared to the time it saves doing the update. The points are spread more or less equally across the neighborhood in the early iterations, but as the iterations progress, the points come closer and closer to the center. This implies that EGG-SynC becomes faster in the later iterations, as we also confirm empirically in the experiments.

Efficient EGG-update

Updating the location of each point p requires that we compute the ε-neighborhood of p and use it to compute the direction in which point p should move. As mentioned in related work Section 11.2, other GPU-parallelized algorithms precompute the neighborhood before use. However, as the neighborhoods of SynC increase in size for each iteration, the space usage becomes prohibitively expensive, and we must find alternative strategies. To balance the workload and reduce branch-divergence, we precompute the non-empty cells in Section 11.4, group the points by location in Section 11.4, and reduce point access using summarized statistics in Section 11.4.

Algorithm 15 EGG-update$(D, \varepsilon, grid, preGrid)$

1: **for** $p \in iGridPoints$ - in parallel **do**
2: compute center outer cell ID $cOID$ of p
3: compute center inner cell index $cIIdx$ of p
4: **for** $\forall oID \in preGridCells[cOID]$ **do**
5: **for** $\forall iIdx \in$ outer cell oID **do**
6: **if** inner cell at $iIdx$ fully within ε radius of p **then**
7: $sum_i = sum_i + \cos(p_i) \times iGridSin[iIdx]_i - \sin(p_i) \times iGridCos[iIdx]_i \forall i$
8: $neighbors = neighbors + iGridSizes[iIdx]$
9: **else if** inner cell at $iIdx$ intersect ε radius of p **then**
10: **for** $q \in$ inner cell at $iIdx$ **do**
11: $sum_i = sum_i + \sin(q_i^t - p_i^t)$
12: $neighbors = neighbors + 1$
13: $p_i^{t+1} = p_i^t + \frac{1}{neighbors} \times sum_i \forall i$
14: **if** $neighbors \neq iGridSizes[cIIdx]$ **then**
15: $r_c = 0$

The update of each point, using these concepts, proceeds as in Algorithm 15. For each point p in parallel, we compute the center outer-grid cell and the center inner cell where p lies. For each surrounding outer cell, we go through each inner-grid cell. If the inner-grid cell is fully within the ε radius of p, then we can use the summarized statistics of the inner-grid cell. Else if the inner-grid cell only overlaps, we must go through all points in that inner-grid cell.

Termination using the grid structure

To efficiently check the synchronization criterion, Definition 11.4.2, we propose leveraging our grid structure. We split the criterion into two checks to reduce the amount of work that must be performed in each iteration. First, we check the first term of the criterion of whether neighborhoods either fully intersect or not at all. If the first term is satisfied, we also check the second term: no new points can be dragged into any neighborhood.

First term. We need to check if $N_{\varepsilon/2}(p) = N_\varepsilon(p)$, naively this could be done by going through all points $q \in N_\varepsilon(p)$ and checking if $\varepsilon/2 \geq |p - q|$ for any point p. However, looking at all points in the neighborhood is to be avoided, as discussed before. Instead, we aim to find a method that does not need to look at all points. The core idea is to find a lower-bound of the size of $N_{\varepsilon/2}(p)$ that can easily be computed. We propose to make the cell width $c_w \leq \sqrt{(\varepsilon/2)^2/d}$, such that the diagonal is less than $\varepsilon/2$. The grid cell g containing p is then fully within $N_{\varepsilon/2}(p)$ and we can use $|g|$ as a lower-bound of $|N_{\varepsilon/2}(p)|$ to avoid computing the exact value by terminating if $|g| = |N_\varepsilon(p)|$. This still determines the termination criterion fully correctly since this lets the algorithm run until all points within $N_{\varepsilon/2}(p)$ are also within g.

Algorithm 16 EGG-SynC(D,ε)

```
1:  t = 0, r_c = 1
2:  while r_c ≠ 0 do
3:      r_c = 1
4:      grid = constructGrid(D^t, ε)
5:      computeSinAndCosSums(grid, D^t, ε)
6:      preGrid = preComputeNonEmptyCells(D^t, ε, grid)
7:      EGG-update(D, ε, grid, preGrid)
8:      t = t + 1
9:      if r_c = 1 then
10:         Check second term of Def. 11.4.2, if not satisfied, set r_c = 0.
11: grid = constructGrid(D^t, ε)
12: return gatherCluster(D^t, ε, grid)
```

Second term. The second term is more expensive to check but only needs to be checked when the first term is true. For all points p we must go through all points in the surrounding grid cells to check if there exists any points q_1 within $\varepsilon < ||p - q_1|| < \varepsilon + \delta$. This is done in parallel across points p. Then for each pair p, q_1 in parallel, we go through all points q_2 in the surrounding grid cells to check if the minimum bounding rectangle containing q_1, q_2 intersects the ε neighborhood of p.

Cluster gathering

Since the λ-termination criterion does not guarantee that the points have synchronized, some points may be left out when gathering the clusters. We, therefore, propose a new method for gathering the clusters, `gatherCluster`, that guarantees that all points are assigned to the correct cluster. Recall that we only terminate when all neighbors are within the center grid cell of the neighborhood. Theorem 11.4.7 states that when we terminate, the neighborhoods contain all the points that synchronize together. Thus, the center grid cell of each neighborhood must contain the final cluster. Therefore, we can return all non-empty grid cells as the clustering.

The full algorithm

The full overview of EGG-SynC is described in Algorithm 16. While the points have not yet synchronized, we construct the grid, compute the summarized statistics, and update the points. We check if all points in the neighborhood are within the center cell, and if so, we check if the surrounding points can be dragged into the neighborhood. At last, when the points have synchronized, we gather the clusters.

11.5 Experiments

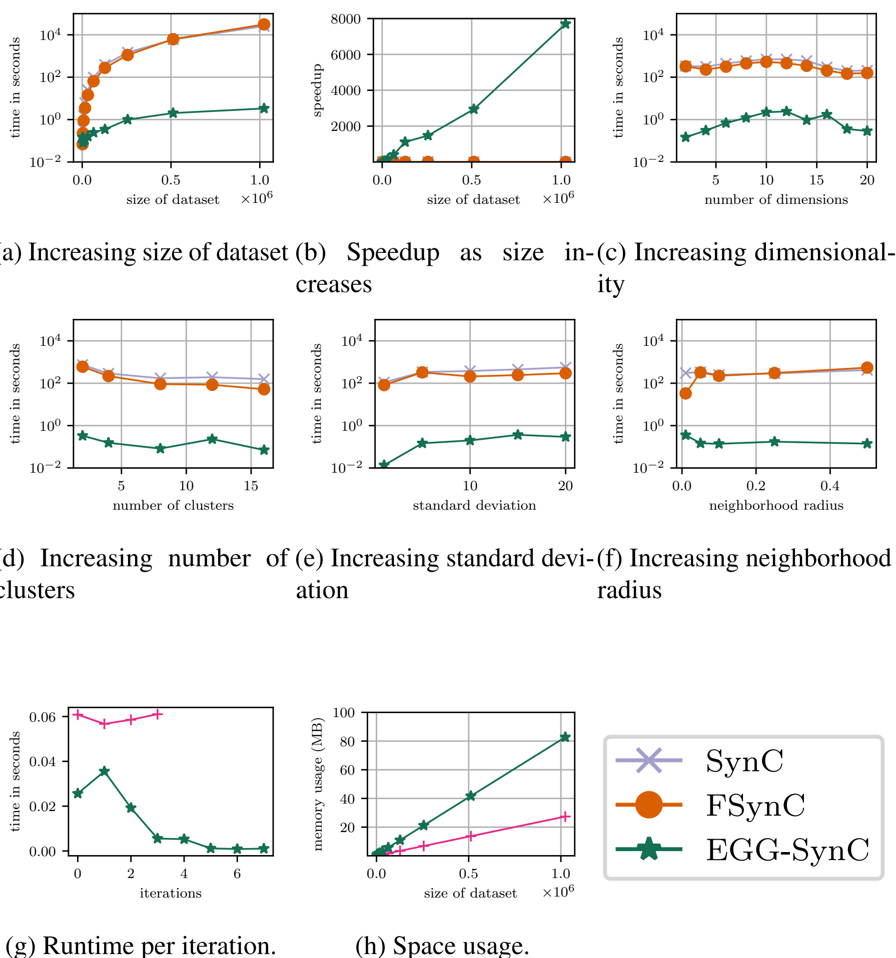
We perform the experimental evaluation on a workstation with Intel Core i9 10940X 3.3GHz 14-Core, 258 GB RAM, and a GeForce RTX 3090 with 24 GB dedicated RAM. All algorithms have been implemented in C++ or CUDA, where all CUDA experiments are run with a block size of 128. For repeatability, the source code is provided at: `https://au-dis.github.io/publications/EGG-SynC`.

Methods. Our proposed algorithm is compared against the original algorithm for clustering by synchronization, SynC [18], and the more recent speedup FSynC [22]. For a fair comparison, we have implemented both in C++ as well. In initial experiments, our implementation of SynC provides approximately $4\times$ speedup compared to the Java implementation provided by the authors. We have implemented straightforward SynC versions that are GPU-parallel (GPU-SynC) and CPU-parallel using multiprocessors (MP-SynC). Both parallelizations distribute updates of all points among threads. All runtime measurements for GPU algorithms also include data transfer time to GPU memory. Böhm et al. [18] employ a strategy for selecting the best ε, as Chen et al. [22], we do not include this in our experiments to make each runtime on different ε values transparent.

Hyperparameters. SynC takes parameters ε and λ; default values in all experiments are $\varepsilon = 0.05$, and $\lambda = 0.999$. EGG-SynC uses our new exact termination criterion and does therefore not need the λ parameter to terminate. FSynC, on the other hand, introduces an additional parameter, the maximum fanout B for the R-Tree; initial experiments suggest $B = 100$ performs best.

Synthetic data. We use the synthetic dataset generator provided by Beer et al. [12] to control data distribution and size, which produces Gaussian distributed clusters. The default parameters for the generated data are $100,000$ points with 2 dimensions, each dimension has values in the range -100 to 100. The points are distributed among 5 Gaussian distributed clusters existing in the full-dimensional space and with a standard deviation of 5.0.

Real-world data. We study the same seven real-world datasets as Chen 2018 [22] from the UCI repository [60]; *data banknote authentication (Bank)* with $1,372$ points and 4 dimensions, *Yeast* with $1,484$ points and 8 dimensions, *Wilt* with $4,838$ points and 5 dimensions, *CCPP* with $9,568$ points and 5 dimensions, *Tamilnadu Electricity Board Hourly Readings (EB)* with $45,781$ points and 2 dimensions, *Skin_NonSkin (Skin)* with $245,057$ points and 3 dimensions, *3D_spatial_network (Roads)* with $434,874$ points and 3 dimensions. In addition, we study higher-dimensional datasets, namely, *Eye State (EEG)* with $10,000$ points and 14 dimensions, *Letter Recognition (Letter)* with $20,000$ points and 16 dimensions, both also from the UCI repository. All datasets are min/max-normalized between 0 and 1.



(a) Increasing size of dataset (b) Speedup as size increases (c) Increasing dimensionality

(d) Increasing number of clusters (e) Increasing standard deviation (f) Increasing neighborhood radius

(g) Runtime per iteration. (h) Space usage.

Figure 11.3: Synthetic experiments

Performance comparison

Scalability

We first investigate the runtime when scaling the input size in the number of points and dimensions. In Figure 11.3a, we see that EGG-SynC is about 2-3 orders of magnitude faster than SynC, MP-SynC, and FSynC and almost a magnitude faster than GPU-SynC. Moreover, the speedup provided by EGG-SynC over SynC and GPU-SynC, see Figure 11.3b, keeps increasing as the number of points increases. This can be attributed to our summarized statistics strategy since the more points we have, the higher the probability that points fall within the same cells and can be effectively summarized by our algorithm. In Figure 11.3c, we see that the runtime increases with the dimensionality for all algorithms and that EGG-SynC has the largest speedup for lower dimensions. We observe that all algorithms show a drop in runtime for higher dimensional datasets. As the dimensionality increases, points are more

likely to be spread out, which in turn likely leads to an increased number of smaller clusters instead of a few large ones that require more synchronization, thus reducing the number of iterations required. This is in line with the effects of the curse of dimensionality [14]; when the number of dimensions increases, the points are further apart and cover more cells, meaning that speedup starts to converge at around 350× speedup. In all cases, EGG-SynC provides a substantial speedup, particularly for large datasets.

Distribution

Besides the size of the input data, data distribution may also affect the runtime of clustering algorithms. We, therefore, evaluate datasets with varying spread and number of clusters. In Figure 11.3d, we see that EGG-SynC maintains several orders of magnitude speedup compared to SynC and FSynC. Furthermore, as the number of clusters increases, all three algorithms become faster. This behavior is most apparent for FSynC and EGG-SynC and can be attributed to their use of an indexing structure for the neighborhood queries. When increasing the standard deviation of the generated clusters to study clusters with a larger spread in Figure 11.3e, we similarly see several orders of magnitude speedup for EGG-SynC compared to SynC and FSynC. Furthermore, the runtime is lowest for all three algorithms when the standard deviation is low. This makes sense since a smaller cluster would imply fewer iterations until the points reach the local synchronization.

Real-world data

We also evaluate the performance on benchmark data from the UCI repository (Figure 11.4), where we again see large speedups for the GPU-parallelized versions of SynC. From the synthetic experiments, we expect EGG-SynC to be faster than GPU-SynC for the larger dataset. This is true for Roads but not for Skin. This can be explained by recalling the example in Figure 11.1, where a smaller part connects bigger parts of a cluster. Such a case would have a high cluster order parameter and make SynC, FSynC, and GPU-SynC stop too early, even though it could require many more iterations to cluster correctly. This is exactly what happens for the Skin dataset in this experiment (Figure 11.4): several clusters are approximated incorrectly but found correctly by our method, at the cost of less speedup. More concretely, GPU-SynC stops after 7 iterations, whereas EGG-SynC continues for the 343 iterations needed to find a correct clustering in this case.

We demonstrate the impact of such cluster approximations for Skin by varying neighborhood radius ε, resulting in different clustering results. As we can see in Figure 11.5, for other values of ε, EGG-SynC is substantially faster than GPU-SynC when there is no need to resolve slowly converging clusters.

The main take-away is thus that EGG-SynC is most often substantially faster than GPU-SynC and especially SynC. In cases where it is not, this is due to more iterations

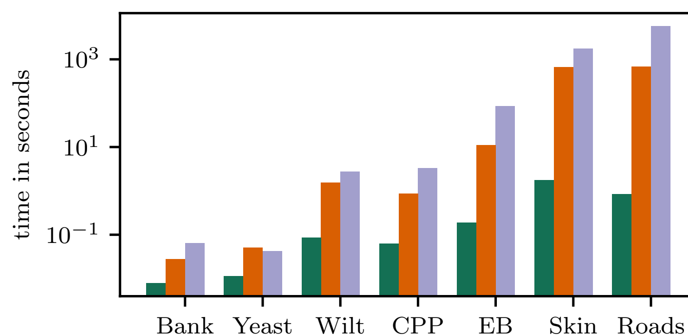


Figure 11.4: Real world datasets

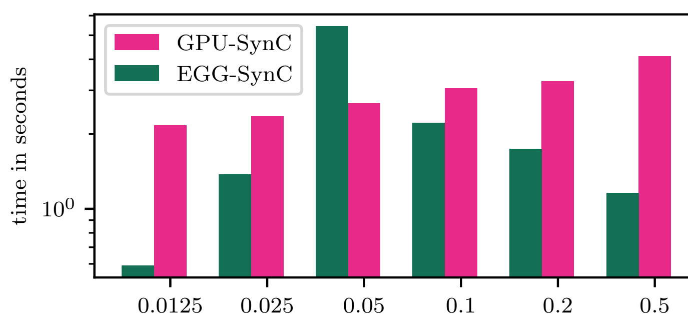


Figure 11.5: Changing ε for the Skin dataset

for a correct result. If speed-up should be preferred to accuracy, a termination threshold as in SynC or a maximum number of iterations could be used.

Hyperparameters

We study the sensitivity of the runtime of the algorithms w.r.t different settings of hyperparameters. The only hyperparameter to set for the three algorithms SynC, FSynC, GPU-SynC, and EGG-SynC alike is the neighborhood radius ε. Intuitively, a lower ε implies fewer points in the neighborhood and, therefore, a lower runtime, especially for the algorithms that utilize a data structure to find the neighborhood without looking at all points. For EGG-SynC, a lower ε implies that it has to iterate over fewer points, but it also implies that each cell becomes smaller and, therefore, in the beginning, the data points are spread across more non-empty cells. In Figure 11.3f, EGG-SynC still provides substantial speed-up for all values of ε compared to SynC and FSynC. At very low values, the speedup for FSynC compared to SynC increases slightly and the speedup of EGG-SynC decreases slightly. However, for all other values, the speedup of EGG-SynC compared to SynC and FSynC remains several orders of magnitude.

size of dataset	Method	Allocating	Build structure	Update	Extra check	Clustering	Free Memory
256000	GPU-SynC	0.003808	0.000000	1.123219	0.000000	0.228805	0.000000
	EGG-SynC	0.000977	0.006878	0.316819	0.007083	0.000461	0.000000
512000	GPU-SynC	0.002676	0.000000	4.663134	0.000000	0.869475	0.000000
	EGG-SynC	0.001469	0.022316	1.088505	0.000403	0.001016	0.000000
1024000	GPU-SynC	0.004343	0.000000	14.145755	0.000000	3.361058	0.000000
	EGG-SynC	0.002172	0.026141	2.723254	0.000763	0.002334	0.000000

Table 11.1: Break down of stages.

Stage and iteration breakdown

To study how different stages of EGG-SynC contribute to the overall runtime, we provide a breakdown of the runtime of each stage of GPU-SynC and EGG-SynC, see Table 11.1. We see that as the data size increases, the construction time of the grid structure becomes minuscule, and the update of points strongly dominates the runtime. More importantly, we see that compared to GPU-SynC, both the update and the gathering of clusters are reduced dramatically as an effect of using the efficiently constructed grid structure. Thus, spending relatively little time on the construction of the grid structure to speed up the update of points and the gathering of clusters is clearly worth the effort.

In Figure 11.3g, we see that GPU-SynC's iteration becomes slightly more expensive as the iterations increase, and our EGG-SynC spends less time. This is because our summarized statistics provide more benefits for dense data. Furthermore, thanks to our effective statistics and data structure, our approach is much faster than GPU-SynC, even though it terminates later when all points have been correctly clustered according to clustering by synchronization, Definition 11.4.1.

Space usage

In Section 11.4, we state that the space usage is linear in the size of the dataset $O(n \times d)$. We validate this in Figure 11.3h, where we see that, indeed, the space increases linearly as the number of data points increases. As expected, EGG-SynC uses a constant factor of more space on the grid structure, which GPU-SynC does not use, but the memory consumption is reasonable and provides a clear runtime benefit.

11.6 Conclusion

In this paper, we propose a novel GPU-parallelized approach to clustering by synchronization, named EGG-SynC. EGG-SynC introduces the first exact termination criterion that guarantees that the synchronization process is finished when the algorithm terminates. EGG-SynC presents a strategy for summarizing the data in a GPU-friendly grid structure and proposes a highly efficient GPU-parallel algorithm for exact clustering by synchronization.

The experimental evaluation on synthetic and real-world data shows substantial, 2 to 3 orders of magnitude, speedup over existing algorithms for varying data and

problem sizes as well as hyperparameter settings.

11.7 Acknowledgments

This work is supported by Independent Research Fund Denmark.

11.8 Proof of Lemma 11.4.3

Proof. We prove by contradiction. Assuming that there exists a point $a \in N_\varepsilon(p)$ where $b \in N_\varepsilon(a)$ but $b \notin N_\varepsilon(p)$. First notice that $\forall p \in D, \nexists q \in D : \varepsilon/2 \leq ||p-q|| \leq \varepsilon$ implies that for all points p $N_\varepsilon(p) = N_{\varepsilon/2}(p)$. Since $a \in N_{\varepsilon/2}(p) \Rightarrow ||p-a|| \leq \varepsilon/2$ and $b \in N_{\varepsilon/2}(a) \Rightarrow ||a-b|| \leq \varepsilon/2$ and using the triangle inequality, we get $||p-b|| \leq ||p-a|| + ||a-b|| \leq \varepsilon$. However then it cannot be true that $b \notin N_\varepsilon(p)$ since it would imply that $||p-b|| > \varepsilon$. Therefore, all points in $N_\varepsilon(p)$ must have the same neighbors. □

11.9 Proof of Lemma 11.4.4

Proof. Given Equation 11.5, we have:

$$\begin{aligned}
|p_i^{t+1} - q_i^{t+1}| &= \Big| p_i^t + \frac{1}{|N_\varepsilon(p^t)|}\left(\cos(p_i^t)s(p,i) - \sin(p_i^t)c(p,i)\right) \\
&\quad - \Big(q_i^t + \frac{1}{|N_\varepsilon(q^t)|}(\cos(q_i^t)s(p,i) - \sin(q_i^t)c(p,i)\Big)\Big)\Big| \\
&= \Big| p_i^t - q_i^t + \frac{1}{|N_\varepsilon(p^t)|} \times \left(\left(\cos(p_i^t) - \cos(q_i^t)\right)s(p,i)\right. \\
&\quad \left. - \left(\sin(p_i^t) - \sin(q_i^t)\right)c(p,i)\right)\Big|,
\end{aligned}$$

where $s(p,i) = \left(\sum_{y\in N_\varepsilon(p)} \sin(y_i^t)\right)$, $c(p,i) = \left(\sum_{y\in N_\varepsilon(p)} \cos(y_i^t)\right)$. For the SynC algorithm to work, Shao et al. [72] require that the data is normalized between $[0,1]$. This implies that $\sum_{y\in N_\varepsilon(p)} \sin(y_i^t)$ and $\sum_{y\in N_\varepsilon(p)} \cos(y_i^t)$ are both always positive. We split the proof into two cases, for $p_i^t - q_i^t \geq 0$ and $p_i^t - q_i^t < 0$. If $p_i^t - q_i^t \geq 0$ then $\cos(p_i^t) - \cos(q_i^t) \leq 0$ and $\sin(p_i^t) - \sin(q_i^t) \geq 0$ in the interval between $[0,1]$, implying that $|p_i^{t+1} - q_i^{t+1}| \leq |p_i^t - q_i^t|$. Similarly, if $p_i^t - q_i^t < 0$ then $\cos(p_i^t) - \cos(q_i^t) > 0$ and $\sin(p_i^t) - \sin(q_i^t) < 0$ in the interval between $[0,1]$, implying that $|p_i^{t+1} - q_i^{t+1}| < |p_i^t - q_i^t|$. We can therefore conclude that $|p_i^{t+1} - q_i^{t+1}| \leq |p_i^t - q_i^t|$. □

11.10 Proof of Lemma 11.4.5

Proof. Since $\frac{\sin(y-\sin(y))}{\sin(x-\sin(x))} > \frac{\sin(y)}{\sin(x)}$ implies $\frac{\sin(x)}{\sin(x-\sin(x))} > \frac{\sin(y)}{\sin(y-\sin(y))}$, and for $0 < x \leq 1$, $\frac{d}{dx}\left(\frac{\sin(x)}{\sin(x-\sin(x))}\right) = \csc(x-\sin(x))(\cos(x) + \sin(x)(\cos(x)-1)\cot(x-\sin(x))) < 0$,

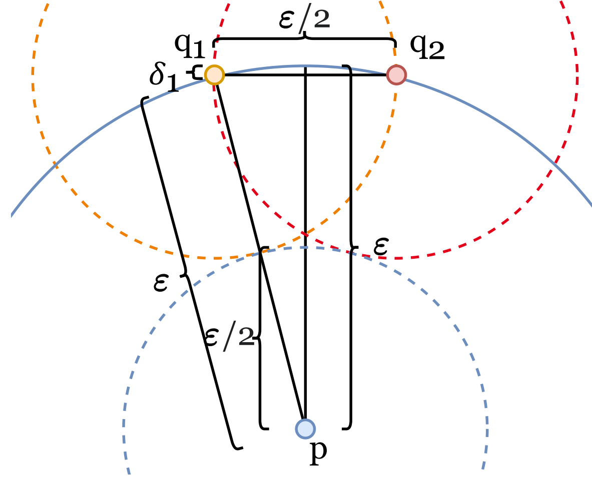


Figure 11.6: Example of two points q_1, q_2 dragging each other into a third point's p neighborhood.

which implies that $\frac{\sin(x)}{\sin(x-\sin(x))}$ is decreasing as x increases within the interval of concern, and, furthermore, that $\frac{\sin(y-\sin(y))}{\sin(x-\sin(x))} > \frac{\sin(y)}{\sin(x)}$. □

11.11 Proof of Lemma 11.4.6

Proof. By Lemma 11.4.3 all neighbors share a neighborhood when the synchronization criterion is met, and by Lemma 11.4.4 all points in a shared neighborhood move closer to each other, implying that they stay in the neighborhood. A neighborhood can then only expand, if points drag each other into another neighborhood using Equation 11.1. When all points are within $\varepsilon/2$ distance of their neighbors, this can only happen when two or more points $q_1, q_2, \ldots$ are outside ε radius of a point p, but within each other $\varepsilon/2$ neighborhoods, as illustrated in Figure 11.6. This implies that there is a distance from the neighborhood where there can exist points $q_1, q_2, \ldots$ that can potentially be dragged into the neighborhood. If the location of the points was updated in a straight line, we could compute the extra distance δ_1 as:

$$\begin{aligned}
\varepsilon^2 =& (\varepsilon - \delta_1)^2 + (\varepsilon/4)^2 \\
\implies (\varepsilon - \delta_1) =& \sqrt{\varepsilon^2 - (\varepsilon/4)^2} = \varepsilon\sqrt{15/16} \\
\implies \delta_1 =& \varepsilon - \varepsilon\sqrt{15/16}.
\end{aligned} \tag{11.7}$$

However, since the update function, Equation 11.1, does not update the location of points in a straight line, we need to check a slightly larger extra distance $\delta = \delta_1 + \delta_2$. Since, for each dimension, the points are updated with the average sin of the difference to all other points in the neighborhood, and since sin is not a linear function, the update deviates δ_2 from a straight line. We overestimate the deviation δ_2 by

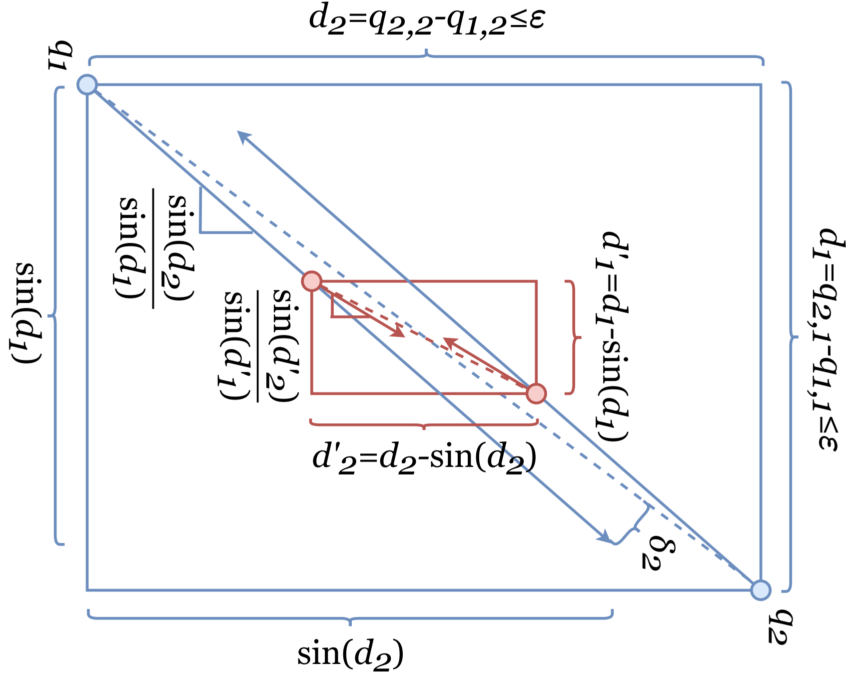


Figure 11.7: Deviation from straight line when updating points q_1, q_2 for two consecutive iterations.

considering the worst-case location and infinitely many points located at this location, i.e., when the points are the furthest apart, the distance along one dimension is as short as possible, and the other is as long as possible. When all points are within $\varepsilon/2$ distances of their neighbors, a point can at most be updated with $\sin(\varepsilon/2)$ along each dimension. This implies that the deviation δ_2 cannot exceed $\delta_2 = \varepsilon/2 - \sin(\varepsilon/2)$, due to Pythagorean theorem, illustrated in Figure 11.7. Furthermore, since the deviation from a straight line changes between iterations, in a subsequent iteration, the deviation could potentially intersect the ε neighborhood of point p if the slope tilts more towards p. Figure 11.7 illustrates updating such points at locations q_1, q_2, in two iterations. All information related to the first iteration is colored blue, and the second is colored red. In the first iteration, the point locations differ by $d_1 = q_{2,1} - q_{1,1}$ and $d_2 = q_{2,2} - q_{1,2}$ and the points at location q_1 are updated along the slope $\frac{\sin(d_2)}{\sin(d_1)}$ diverging slightly from the straight dashed line between the two locations. How far the points are updated along this slope depends on the fraction of points α that are located at q_1 compared to q_2. If there is only one point at each location, then $\alpha = 0.5$ and the points will end up close to the middle as in the illustration; else, they will end up close to the location with the most points. After the update, the points at q_1 move to $q'_{1,j} = q_{1,j} + \alpha \sin(q_{2,j} - q_{1,j}) \forall i \in [0, d-1]$ and the points at q_2 move to $q_{2,j} = q_{2,j} + (1-\alpha)\sin(q_{1,j} - q_{2,j}) \forall i \in [0, d-1]$. Making the different between the points at the two locations $d'_1 = (q_{1,1} + \alpha \sin(q_{2,1} - q_{1,1})) - (q_{2,1} + (1-\alpha)\sin(q_{1,1} -$

$q_{2,1})), d'_2 = (q_{1,2} + \alpha \sin(q_{2,2} - q_{1,2})) - (q_{2,2} + (1-\alpha)\sin(q_{1,2} - q_{2,2}))$ along the two dimensions. The slope the update follow for the subsequent iteration is therefore $\frac{\sin(d'_2)}{\sin(d'_1)}$. Notice that the behavior is mirrored on the straight dashed line, and it does not matter which side of the line point p is located. We prove this in the 2-dimensional case, but it can similarly be expanded to higher-dimensions. Furthermore, how the points are located in relation to each other can be mirrored such that $0 < d_1 \leq 1$, $0 < d_2 \leq 1$, $d_1 < d_2$, and the points p is below the dashed line; the other cases can be proven analogously. To prove that the updates in the subsequent iterations do not bring the points at q_1, q_2 close to p we must show that the slope increases in the subsequent iterations $\frac{\sin(d_2)}{\sin(d_1)} < \frac{\sin(d'_2)}{\sin(d'_1)}$. We first simplify $d'_j = (q_{2,j} + \alpha \sin(q_{1,j} - q_{2,j})) - (q_{1,j} + (1-\alpha)\sin(q_{2,j} - q_{1,j})) = d_j - \alpha \sin(d_j) - (1-\alpha)\sin(d_j)) = d_j - \sin(d_j)$, implying that $\frac{\sin(d_2)}{\sin(d_1)} < \frac{\sin(d'_2)}{\sin(d'_1)} = \frac{\sin(d_2 - \sin(d_2))}{\sin(d_1 - \sin(d_1))}$. By Lemma 11.4.5, this is true, and the points can, therefore, not move into the ε neighborhood in subsequent iterations either. This makes the total extra distance $\delta = \delta_1 + \delta_2 = \varepsilon - \varepsilon\sqrt{15/16} + \varepsilon/2 - \sin(\varepsilon/2)$. Furthermore, if there exists a point q within this border, but the minimum bounding rectangle of the points in its neighborhood does not intersect the ε neighborhood of p, then there cannot exist two points in $N_\varepsilon(q)$ that could drag each other into the neighborhood of p. $\square$

Bibliography

[1] Andrew Adinetz, Jiri Kraus, Jan Meinke, and Dirk Pleiter. Gpumafia: Efficient subspace clustering with mafia on gpus. In *European Conference on Parallel Processing*, pages 838–849. Springer, 2013. 47, 54, 55, 76, 103

[2] Yaw Adu-Gyamfi. Gpu-enabled visual analytics framework for big transportation datasets. *Journal of Big Data Analytics in Transportation*, 1(2-3):147–159, 2019. doi: 10.1007/s42421-019-00010-y. 106

[3] Charu C Aggarwal and Philip S Yu. Finding generalized projected clusters in high dimensional spaces. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 70–81, 2000. 39, 55, 103

[4] Charu C Aggarwal, Joel L Wolf, Philip S Yu, Cecilia Procopiuc, and Jong Soo Park. Fast algorithms for projected clustering. *ACM SIGMoD Record*, 28(2): 61–72, 1999. 4, 39, 41, 54, 55, 84, 85, 92, 103, 106

[5] Rakesh Agrawal, Heikki Mannila, Ramakrishnan Srikant, Hannu Toivonen, A Inkeri Verkamo, et al. Fast discovery of association rules. *Advances in knowledge discovery and data mining*, 12(1):307–328, 1996. 23

[6] Rakesh Agrawal, Johannes Gehrke, Dimitrios Gunopulos, and Prabhakar Raghavan. Automatic subspace clustering of high dimensional data for data mining applications. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 94–105, 1998. 23, 54, 55, 84, 102, 103

[7] Guilherme Andrade, Gabriel Ramos, Daniel Madeira, Rafael Sachetto, Renato Ferreira, and Leonardo Rocha. G-dbscan: A gpu accelerated algorithm for density-based clustering. *Procedia Computer Science*, 18:369–378, 2013. 4, 13, 16, 47, 54, 55, 74, 115

[8] Mihael Ankerst, Markus M Breunig, Hans-Peter Kriegel, and Jörg Sander. Optics: Ordering points to identify the clustering structure. *ACM Sigmod record*, 28(2):49–60, 1999. 115

[9] Ira Assent, Ralph Krieger, Emmanuel Müller, and Thomas Seidl. Dusc: Dimensionality unbiased subspace clustering. In *seventh IEEE international conference on data mining (ICDM 2007)*, pages 409–414. IEEE, 2007. 24, 25, 26, 54, 55, 56, 57

[10] Ira Assent, Ralph Krieger, Emmanuel Müller, and Thomas Seidl. Edsc: efficient density-based subspace clustering. In *Proceedings of the 17th ACM conference on Information and knowledge management*, pages 1093–1102, 2008. 24, 26, 76

[11] Ira Assent, Ralph Krieger, Emmanuel Müller, and Thomas Seidl. Inscy: Indexing subspace clusters with in-process-removal of redundancy. In *2008 Eighth IEEE International Conference on Data Mining*, pages 719–724. IEEE, 2008. 4, 24, 25, 27, 29, 54, 55, 56, 59, 76, 103

[12] Anna Beer, Nadine Sarah Schüler, and Thomas Seidl. A generator for subspace clusters. In *LWDA*, pages 69–73, 2019. 79, 97, 132

[13] Anna Beer, Ekaterina Allerborn, Valentin Hartmann, and Thomas Seidl. Kiss-a fast knn-based importance score for subspaces. In *EDBT*, pages 391–396, 2021. 116

[14] Kevin Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. When is "nearest neighbor" meaningful? In *International conference on database theory*, pages 217–235. Springer, 1999. 3, 83, 134

[15] James C Bezdek, Robert Ehrlich, and William Full. Fcm: The fuzzy c-means clustering algorithm. *Computers & geosciences*, 10(2-3):191–203, 1984. 39

[16] Janki Bhimani, Miriam Leeser, and Ningfang Mi. Accelerating k-means clustering with parallel implementations and gpu computing. In *2015 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2015. 47

[17] Christian Böhm, Robert Noll, Claudia Plant, and Bianca Wackersreuther. Density-based clustering using graphics processors. In *Proceedings of the 18th ACM conference on Information and knowledge management*, pages 661–670, 2009. 47, 54, 55

[18] Christian Böhm, Claudia Plant, Junming Shao, and Qinli Yang. Clustering by synchronization. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 583–592, 2010. 4, 13, 17, 18, 102, 113, 114, 115, 116, 117, 118, 119, 132

[19] Stuart K Card, George G Robertson, and Jock D Mackinlay. The information visualizer, an information workspace. In *Proceedings of the SIGCHI Conference on Human factors in computing systems*, pages 181–186, 1991. 84

[20] Elaine Y Chan, Wai Ki Ching, Michael K Ng, and Joshua Z Huang. An optimization algorithm for clustering using weighted dissimilarity measures. *Pattern recognition*, 37(5):943–952, 2004. 103

[21] Lei Chen, Jing Zhang, Li-Jun Cai, Ting-Qin He, and Tao Meng. Parallel synchronization-inspired partitioning clustering. *Journal of Computational and Theoretical Nanoscience*, 13(11):8709–8729, 2016. 114, 116, 119

[22] Xinquan Chen. Fast synchronization clustering algorithms based on spatial index structures. *Expert Systems with Applications*, 94:276–290, 2018. 17, 18, 113, 114, 115, 118, 132

[23] Yewang Chen, Xiaoliang Hu, Wentao Fan, Lianlian Shen, Zheng Zhang, Xin Liu, Jixiang Du, Haibo Li, Yi Chen, and Hailin Li. Fast density peak clustering for large scale data based on knn. *Knowledge-Based Systems*, 187:104824, 2020. 116

[24] Chun-Hung Cheng, Ada Waichee Fu, and Yi Zhang. Entropy-based subspace clustering for mining numerical data. In *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 84–93, 1999. 23, 55, 102, 103

[25] Amitava Datta, Amardeep Kaur, Tobias Lauer, and Sami Chabbouh. Exploiting multi–core and many–core parallelism for subspace clustering. *International Journal of Applied Mathematics and Computer Science*, 29(1):81–91, 2019. 103

[26] Zhaohong Deng, Kup-Sze Choi, Fu-Lai Chung, and Shitong Wang. Enhanced soft subspace clustering integrating within-cluster and between-cluster information. *Pattern recognition*, 43(3):767–781, 2010. 103

[27] Niklas Elmqvist, Andrew Vande Moere, Hans-Christian Jetter, Daniel Cernea, Harald Reiterer, and TJ Jankun-Kelly. Fluid interaction for information visualization. *Information Visualization*, 10(4):327–340, 2011. doi: 10.1177/1473871611413180. 106

[28] Alex Endert, M. Shahriar Hossain, Naren Ramakrishnan, Chris North, Patrick Fiaux, and Christopher Andrews. The human is the loop: New directions for visual analytics. *Journal of Intelligent Information Systems*, 43(3):411–435, 2014. doi: 10.1007/s10844-014-0304-9. 105

[29] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd*, volume 96, pages 226–231, 1996. 4, 13, 14, 36, 54, 55, 57, 62, 73, 102, 114, 115

[30] Reza Farivar, Daniel Rebolledo, Ellick Chan, and Roy H Campbell. A parallel implementation of k-means clustering on gpus. In *Pdpta*, volume 13, pages 212–312, 2008. 55, 103, 114

[31] Jean-Daniel Fekete and Catherine Plaisant. Interactive information visualization of a million items. In *Proc. of the IEEE Symposium on Information Visualization (InfoVis)*, pages 117–124. IEEE, 2002. doi: 10.1109/INFVIS.2002.1173156. 106

[32] Marta Fort, J Antoni Sellarès, and Nacho Valladares. A parallel gpu-based approach for reporting flock patterns. *International Journal of Geographical Information Science*, 28(9):1877–1903, 2014. 47

[33] Sanjay Goil, Harsha Nagesh, and Alok Choudhary. 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*, volume 443, page 452. ACM, 1999. 23, 55, 102, 103

[34] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, pages 47–57, 1984. 18

[35] Robert B Haber and David A McNabb. Visualization idioms: A conceptual model for scientific visualization systems. *Visualization in scientific computing*, 74:93, 1990. 44

[36] Xiaofei He and Partha Niyogi. Locality preserving projections. *Advances in neural information processing systems*, 16, 2003. 39

[37] Alexander Hinneburg, Daniel A Keim, et al. An efficient approach to clustering in large multimedia databases with noise. In *KDD*, volume 98, pages 58–65, 1998. 115

[38] Bai Hong-Tao, He Li-li, Ouyang Dan-tong, Li Zhan-shan, and Li He. K-means on commodity gpus with cuda. In *2009 WRI World Congress on Computer Science and Information Engineering*, volume 3, pages 651–655. IEEE, 2009. 103

[39] Jakob Rødsgaard Jørgensen, Katrine Scheel, and Ira Assent. Gpu-inscy: A gpu-parallel algorithm and tree structure for efficient density-based subspace clustering. In *EDBT*, pages 25–36, 2021. 4, 28, 29, 30, 32, 33, 36, 47, 97, 103, 115

[40] Jakob Rødsgaard Jørgensen, Ira Assent, and Hans-Jörg Schulz. Avid: Gpu-enabled visual analytics with gpu-fast-proclus. In *EDBT*, pages 2–562, 2022. 45

[41] Jakob Rødsgaard Jørgensen, Katrine Scheel, Ira Assent, Ajeet Ram Pathak, and Anne C Elster. Gpu-fast-proclus: A fast gpu-parallelized approach to projected clustering. In *EDBT*, 2022. 4, 18, 19, 39, 41, 47, 106, 107

[42] Jakob Rødsgaard Jørgensen, Scheel, and Ira Assent. Egg-sync: Exact gpu-parallelized grid-based clustering by synchronization. In *EDBT*, 2023. 4, 13, 17, 18, 47

[43] Karin Kailing, Hans-Peter Kriegel, and Peer Kröger. Density-connected subspace clustering for high-dimensional data. In *Proceedings of the 2004 SIAM international conference on data mining*, pages 246–256. SIAM, 2004. 23, 29, 54, 55, 57, 84, 102, 103

[44] Leonard Kaufman and Peter J Rousseeuw. Partitioning around medoids (program pam). *Finding groups in data: an introduction to cluster analysis*, 344:68–125, 1990. 13, 14

[45] Amardeep Kaur and Amitava Datta. Subscale: Fast and scalable subspace clustering for high dimensional data. In *2014 IEEE International Conference on Data Mining Workshop*, pages 621–628. IEEE, 2014. 103

[46] Kai J Kohlhoff, Marc H Sosnick, William T Hsu, Vijay S Pande, and Russ B Altman. Campaign: an open-source library of gpu-accelerated data clustering algorithms. *Bioinformatics*, 27(16):2321–2322, 2011. 103

[47] Hans-Peter Kriegel, Peer Kröger, and Arthur Zimek. 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):1–58, 2009. 55

[48] Mi Li, Jie Huang, and Jingpeng Wang. Paralleled fast search and find of density peaks clustering algorithm on gpus with cuda. In *2016 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, pages 313–318. IEEE, 2016. 47

[49] You Li, Kaiyong Zhao, Xiaowen Chu, and Jiming Liu. Speeding up k-means algorithm by gpus. *Journal of Computer and System Sciences*, 79(2):216–229, 2013. 55, 103

[50] Guimei Liu, Jinyan Li, Kelvin Sim, and Limsoon Wong. Distance based subspace clustering with flexible dimension partitioning. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 1250–1254. IEEE, 2007. 103

[51] Stuart Lloyd. Least squares quantization in pcm. *IEEE transactions on information theory*, 28(2):129–137, 1982. 13, 102

[52] Woong-Kee Loh and Hwanjo Yu. Fast density-based clustering through dataset partition using graphics processing units. *Information Sciences*, 308:94–112, 2015. 16, 47, 55

[53] Woong-Kee Loh, Yang-Sae Moon, and Young-Ho Park. 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):1947–1951, 2014. 16, 47, 55

[54] James MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297. Oakland, CA, USA, 1967. 13

[55] Danilo Melo, Sávyo Toledo, Fernando Mourão, Rafael Sachetto, Guilherme Andrade, Renato Ferreira, Srinivasan Parthasarathy, and Leonardo Rocha. Hierarchical density-based clustering based on gpu accelerated data indexing strategy. *Procedia computer science*, 80:951–961, 2016. 47, 115

[56] Gabriela Moise and Jörg Sander. 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*, pages 533–541, 2008. 55, 103

[57] Todd Mostak. Using gpus to accelerate data discovery and visual analytics. In *Proc. of the Future Technologies Conference (FTC)*, pages 1310–1313. IEEE, 2016. doi: 10.1109/FTC.2016.7821771. 106

[58] Emmanuel Müller, Stephan Günnemann, Ira Assent, and Thomas Seidl. Evaluating clustering in subspace projections of high dimensional data. *Proceedings of the VLDB Endowment*, 2(1):1270–1281, 2009. 41, 55, 84

[59] Hamza Mustafa, Eleazar Leal, and Le Gruenwald. An experimental comparison of gpu techniques for dbscan clustering. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 3701–3710. IEEE, 2019. 55

[60] David J Newman, SCLB Hettich, Cason L Blake, and Christopher J Merz. Uci repository of machine learning databases, 1998, 1998. 76, 98, 132

[61] Raymond T. Ng and Jiawei Han. Clarans: A method for clustering objects for spatial data mining. *IEEE transactions on knowledge and data engineering*, 14 (5):1003–1016, 2002. 13, 14, 84, 85, 102, 103, 106

[62] Sandra Obermeier, Anna Beer, Florian Wahl, and Thomas Seidl. Cluster flow-an advanced concept for ensemble-enabling, interactive clustering. *BTW 2021*, 2021. 116

[63] Eric Papenhausen, Bing Wang, Sungsoo Ha, Alla Zelenyuk, Dan Imre, and Klaus Mueller. Gpu-accelerated incremental correlation clustering of large data with visual feedback. In *Proc. of the IEEE International Conference on Big Data*, pages 63–70. IEEE, 2013. doi: 10.1109/BigData.2013.6691716. 106

[64] Lance Parsons, Ehtesham Haque, and Huan Liu. Subspace clustering for high dimensional data: a review. *Acm Sigkdd Explorations Newsletter*, 6(1):90–105, 2004. 55, 102

[65] Adhi Prahara, Dewi Pramudi Ismi, and Ahmad Azhari. Parallelization of partitioning around medoids (pam) in k-medoids clustering on gpu. *Knowledge Engineering and Data Science*, 3(1):40–49, 2020. 103

[66] Donghao Ren, Bongshin Lee, and Tobias Höllerer. Stardust: Accessible and transparent gpu support for information visualization rendering. *Computer Graphics Forum*, 36(3):179–188, 2017. doi: 10.1111/cgf.13178. 106

[67] Alex Rodriguez and Alessandro Laio. Clustering by fast search and find of density peaks. *science*, 344(6191):1492–1496, 2014. 13, 16, 102, 115

[68] Enrique H Ruspini. A new approach to clustering. *Information and control*, 15 (1):22–32, 1969. 39

[69] Karlton Sequeira and Mohammed Zaki. Schism: a new approach to interesting subspace mining. *International Journal of Business Intelligence and Data Mining*, 1(2):137–160, 2005. 55

[70] Junming Shao, Christian Böhm, Qinli Yang, and Claudia Plant. Synchronization based outlier detection. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 245–260. Springer, 2010. 114, 115

[71] Junming Shao, Xiao He, Christian Böhm, Qinli Yang, and Claudia Plant. Synchronization-inspired partitioning and hierarchical clustering. *IEEE Transactions on Knowledge and Data Engineering*, 25(4):893–905, 2012. 114, 115

[72] Junming Shao, Xinzuo Wang, Qinli Yang, Claudia Plant, and Christian Böhm. Synchronization-based scalable subspace clustering of high-dimensional data. *Knowledge and information systems*, 52(1):83–111, 2017. 114, 115, 137

[73] Junming Shao, Yue Tan, Lianli Gao, Qinli Yang, Claudia Plant, and Ira Assent. Synchronization-based clustering on evolving data stream. *Information Sciences*, 501:573–587, 2019. 114, 115

[74] Ben Shneiderman. Dynamic queries for visual information seeking. *IEEE software*, 11(6):70–77, 1994. 44, 84, 98, 106, 107

[75] Kelvin Sim, Vivekanand Gopalkrishnan, Arthur Zimek, and Gao Cong. A survey on enhanced subspace clustering. *Data mining and knowledge discovery*, 26(2): 332–397, 2013. 55

[76] Alexander S Szalay, Jim Gray, Ani R Thakar, Peter Z Kunszt, Tanu Malik, Jordan Raddick, Christopher Stoughton, and Jan vandenBerg. 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, pages 570–581, 2002. 77, 98

[77] Andrada Tatu, Fabian Maaß, Ines Färber, Enrico Bertini, Tobias Schreck, Thomas Seidl, and Daniel Keim. Subspace search and visualization to make sense of alternative clusterings in high-dimensional data. In *Proc. of the IEEE Conference on Visual Analytics Science and Technology (VAST)*, pages 63–72. IEEE, 2012. doi: 10.1109/VAST.2012.6400488. 106

[78] Rajeev J Thapa, Christian Trefftz, and Greg Wolffe. Memory-efficient implementation of a graphics processor-based cluster detection algorithm for large spatial databases. In *2010 IEEE International Conference on Electro/Information Technology*, pages 1–5. IEEE, 2010. 16, 55

[79] Christian Tominski and Heidrun Schumann. *Interactive Visual Data Analysis*. A K Peters/CRC Press, 2020. doi: 10.1201/9781315152707. 106

[80] Daniel Weiskopf. *GPU-based interactive visualization techniques*. Springer, 2007. doi: 10.1007/978-3-540-33263-3. 106

[81] Kyoung-Gu Woo, Jeong-Hoon Lee, Myoung-Ho Kim, and Yoon-Joon Lee. Findit: a fast and intelligent subspace clustering algorithm using dimension voting. *Information and Software Technology*, 46(4):255–271, 2004. 55, 103

[82] Dongkuan Xu and Yingjie Tian. A comprehensive survey of clustering algorithms. *Annals of Data Science*, 2(2):165–193, 2015. 102

[83] Bo Yan, Ye Zhang, Zijiang Yang, Hongyi Su, and Hong Zheng. Dvt-pkm: An improved gpu based parallel k-means algorithm. In *International Conference on Intelligent Computing*, pages 591–601. Springer, 2014. 47

[84] Wenhao Ying, Fu-Lai Chung, and Shitong Wang. Scaling up synchronization-inspired partitioning clustering. *IEEE Transactions on Knowledge and Data Engineering*, 26(8):2045–2057, 2013. 116

[85] Xiaoru Yuan, Donghao Ren, Zuchao Wang, and Cong Guo. Dimension projection matrix/tree: Interactive subspace visual exploration and analysis of high dimensional data. *IEEE Transactions on Visualization and Computer Graphics*, 19(12):2625–2633, 2013. doi: 10.1109/TVCG.2013.150. 106

[86] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. Birch: A new data clustering algorithm and its applications. *Data mining and knowledge discovery*, 1(2): 141–182, 1997. 116

[87] Mengyang Zhao, Aadarsh Jha, Quan Liu, Bryan A Millis, Anita Mahadevan-Jansen, Le Lu, Bennett A Landman, Matthew J Tyska, and Yuankai Huo. Faster mean-shift: Gpu-accelerated clustering for cosine embedding-based cell segmentation and tracking. *Medical Image Analysis*, 71:102048, 2021. 47