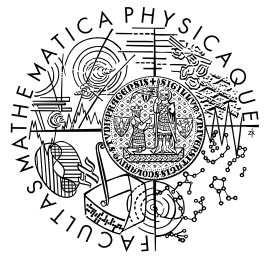

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Bc. Adam Šmelko

Mahalanobis based hierarchical clustering accelerated on GPU

Department of Software Engineering

Supervisor of the master thesis: RNDr. Miroslav Kratochvíl
Study programme: Computer Science
Study branch: Software and Data Engineering

Prague 2020

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I dedicate this to my parents, for their kindness and support.
I would like to thank my supervisor Mirek Kratochvíl for his expertise, very fast replies, time and patience.
I thank GPULAB cluster for the performed experiments.

Title: Mahalanobis based hierarchical clustering accelerated on GPU

Author: Bc. Adam Šmelko

Department: Department of Software Engineering

Supervisor: RNDr. Miroslav Kratochvíl, Department of Software Engineering

Abstract: Hierarchical clustering algorithms are common tools for simplifying, exploring and analyzing datasets in many areas of research. For flow cytometry, a specific variant of agglomerative clustering has been proposed, that uses cluster linkage based on Mahalanobis distance to produce results better suited for the domain. Applicability of this clustering algorithm is currently limited by its relatively high computational complexity, which does not allow it to scale to common cytometry datasets. This thesis describes a specialized, GPU-accelerated version of the Mahalanobis-average linked hierarchical clustering, which improves the algorithm performance by several orders of magnitude, thus allowing it to scale to much larger datasets. The thesis provides an overview of current hierarchical clustering algorithms, and details the construction of the variant used on GPU. The result is benchmarked on publicly available high-dimensional data from mass cytometry.

Keywords: clustering high-dimensional data GPU

Contents

Introduction

Clustering is a commonly used technique for simplifying the work with complex datasets. The main principle is to represent a dataset composed of a huge number of elements by a simplified (and significantly smaller) set of element groups which are commonly called clusters. There are many various ways of constructing the clustering algorithms because each discipline typically requires a different kind of cluster similarity, it manipulates completely different data (texts, sequences, vectors) and it has different expectations about the organizing of the resulting data.

Clustering is commonly used to analyze datasets originating in single-cell cytometry [22], where grouping the cells by similar measured features often corresponds to creating clusters of all individual biological types of the cells. This greatly simplifies a data analysis and a distinction of different cell types. Many customized approaches for clustering cytometry data have been developed, including FlowSOM [24], PhenoGraph [15], SPADE [19] or FlowGrid [25]. This thesis focuses on agglomerative hierarchical clustering with the Mahalanobis metric, originally developed by Fišer et al. [8] for the purpose of monitoring minimal residual decease in patients with leukemia.

The Mahalanobis clustering is, however, seriously restricted by the performance and complexity of the current implementation, which can cluster tens of thousands of cells on a common hardware. The computation of these clusters is time demanding and consumes a lot of computer memory. Specifically for the Mahalanobis clustering, the maximum size of a dataset that can be processed on common hardware varies at about 100,000 cells. As a result, the performance of the Mahalanobis clustering is unsuitable for processing data from modern cytometers used in current experiments, which often produce datasets of more than several million cells.

The primary goal of this thesis is to research possibilities of accelerating the Mahalanobis clustering on a GPU, develop the implementation of the clustering accelerated on a GPU and measure its results. Many clustering algorithms have been already ported to a GPU device, eg. DBSCAN [3], UPGMA [11], etc.. The Mahalanobis clustering is extremely suitable for cytometry because it naturally forms elliptical clusters. A GPU-accelerated implementation that could improve the performance on large datasets has not been developed yet.

The thesis first discusses the hierarchical clustering (section 1.1.2) and variations of algorithms that can be used (section 1.3). Then, an overview of the programming for the GPU is introduced (section 2.1). Using this information, the thesis designs a parallelized algorithm of MHCA which can be run on a GPU; therefore, it is significantly faster. The combination of simple metric spaces with

Minkowski distance specialization and enormous GPU throughput allows for a major memory requirements reduction. The total acceleration gain of the implementation varies from 20 times to 5000 times. The quality and speed of the result is demonstrated on flow and mass cytometry datasets. We hope that the results of the thesis will be possible to package and use for biologically relevant purposes.

1. Agglomerative clustering algorithms

Clustering analysis — or clustering — is a reduction of a complex object group into several small, less complex disjunctive subgroups. The reduction is performed in such way that objects from one subgroup share a common property (e.g. they are mutually compatible, or similar). Hence, clustering is used to identify and extract significant partitions from the underlying data.

In the field of clustering analysis, there is no strict definition for a cluster itself. That may be one reason why there is such a vast amount of clustering algorithms; many authors such as Estivill-Castro [7] discuss this topic. Despite the lack of the definition, the one common property that we can find among all the algorithms is the presence of a group of data objects.

Depending on a field of use, the data objects are represented variously (as graphs, text sequences, etc.). The current thesis will focus on a clustering of objects represented by a vector of real numbers.

Suppose a dataset $\mathcal{D}$ given as a n d-dimensional vectors $(x_1, \ldots, x_d) \subset \mathbb{R}^d$ — objects; each element of a vector describes a specific object property. Two objects are similar if values of their respective properties are alike. Then, a clustering analysis can be defined as a form of an object grouping into subsets of $\mathcal{D}$ that maximizes the inter-set object similarity and minimizes the intra-set object similarity.

1.1 Clustering models

Specific variations of clustering analysis are defined by a clustering model. There is a great amount of them, since their field of use varies. For the purpose of the following chapters, we first describe the centroid-based model. Then, we fully focus on the hierarchical model.

Since the current thesis focuses on the clustering of vectors of real numbers, we define the following terms and establish the terminology:

Definition 1 (Vector-space dataset)**.** *Given the real vector space $\mathbb{R}^d$, an input of clustering analysis $\mathcal{D} \subset \mathbb{R}^d$ is called the vector-space dataset.*

Definition 2 (Cluster)**.** *Given a vector-space dataset $\mathcal{D}$, we define the cluster C of $\mathcal{D}$ as any subset of $\mathcal{D}$.*

Definition 3 (Centroid)**.** *Given a vector-space dataset $\mathcal{D} \subset \mathbb{R}^d$ and its cluster C, we define the centroid of C as a point in $\mathbb{R}^d$ such that its i-th element is equal to*

Algorithm 1 k-means clustering

procedure k-MEANS($k \in \mathbb{R}$, $\mathcal{D} \subset \mathbb{R}^d$, $d \in \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$)
  $C \leftarrow$ first k objects from $\mathcal{D}$ ▷ select initial centroids
  repeat
    $\forall i \in \{1 \dots k\} : K_i \leftarrow \{\}$ ▷ create empty clusters
    $\forall o \in \mathcal{D} : K_j \leftarrow K_j \cup o$ for such j that $d(C_j, o)$ is minimal ▷ assign objects to clusters
    $C' \leftarrow \{\text{mean}(K_1), \dots, \text{mean}(K_k)\}$ ▷ compute new centroids
    swap C and C'
  until $C = C'$
  return C
end procedure

the arithmetic mean of the i-th elements of all points $o \in C$. We will denote it by $\text{mean}(C)$.

1.1.1 Centroid-based model

The centroid-based clustering model represents clusters only by a central vector — a centroid — which is not necessarily a member of a dataset. Further on in the thesis, we will refer to the cluster as to a subset

Many implementations of this model need the number of required centroids in advance (denoted by k). We define the following optimization problem for this kinds of algorithms:

Problem (Centroid-based clustering)**.** *Having a distance function d, find k centroids $c_1, \dots, c_k$ from the domain of the dataset $\mathcal{D}$ such that the sum 1.1 is minimized.*

$$\sum_{o}^{\mathcal{D}} \min_{i=1\dots k} d(o, c_i) \tag{1.1}$$

This problem is difficult to solve; in the euclidean space, the problem is NP-hard [2]. Hence, many approximation algorithms have emerged.

k-means

The most common implementation of a centroid-based clustering is k-means. Its algorithm can be expressed in a few simple steps (see alg. 1).

The algorithm divides data into k clusters in an iterative manner. Before the first iteration, initial k central vectors are selected from the dataset (we chose

the first k objects but the way of selecting k initial vectors varies between implementations). In the iteration loop, dataset objects are grouped into clusters according to the closest centroid (also called the cluster mean; hence, k-means). After that, new centroids are computed from new clusters. Next iteration follows until centroids does not change or a predefined number of iterations is reached.

Since the only performance demanding parts of the k-means algorithm are the assignment of points to clusters and the centroid computation, the algorithm is simple and fast. As a result of the simplicity, it is unable to deal with a noise in a dataset and clusters of a non-convex shape [23].

1.1.2 Hierarchical model

In the hierarchical clustering model, objects are connected together forming a tree-like structure. In contrast with the aim of a centroid-based model that returns only k centroids, hierarchical clustering analysis (HCA) captures the whole connecting process. HCA algorithms start with all objects from a dataset as initial clusters. Each iteration, two clusters are connected creating a new one, finishing with one all-inclusive cluster. Commonly, the algorithms represent the connecting process as an ordered list of pairs — a list of connected clusters [12].

The result of a hierarchical clustering can be viewed as a *dendrogram* (see fig. 1.1). The y-axis states the measure of similarity between connected clusters. The x-axis shows labels of the objects from a dataset. Hence, the clusters that are connected in the higher part of the dendrogram are considered less similar, and the clusters connected at its bottom are more similar.

Hierarchical algorithms use two approaches on how to create a dendrogram; agglomerative and divisive. An agglomerative hierarchical algorithm starts with each object representing a cluster on its own. Then, in a bottom-up fashion, clusters are successively connected into the only cluster. The divisive algorithm begins with a single all-inclusive cluster which is divided into sub-clusters until single objects remain [20].

To know which two clusters are connected (or respectively, how a cluster is divided into two), algorithms use a dissimilarity measure between clusters.

Definition 4 (Dissimilarity measure)**.** *Given a vector-space dataset $\mathcal{D} \subset \mathbb{R}^d$, we define the dissimilarity measure as the pair (M, L). M is a metric over $\mathbb{R}^d$ and L is a linkage criterion; a function that can measure dissimilarity of subsets of $\mathbb{R}^d$ using M. They are used to measure the dissimilarity of clusters during a $\mathcal{D}$ clustering.*

A hierarchical clustering model distinguishes various kinds of algorithms based on the choice of a metric and a linkage criterion. A distance function can serve as a metric in a dissimilarity measure:

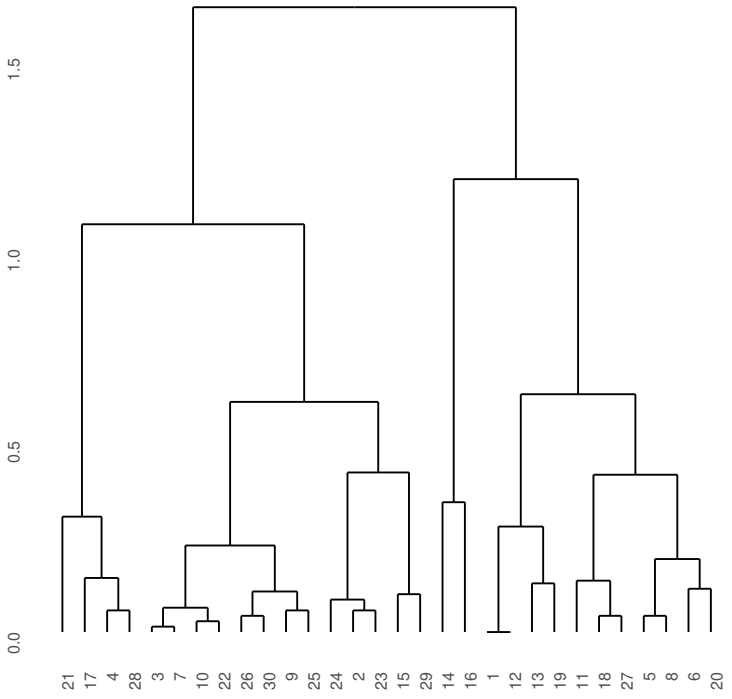

Figure 1.1: An example of a dendrogram; a plant growth under different treatment conditions (R dataset `PlantGrowth`).

Distance functions

A distance function is used on objects of a dataset to measure how far they are from each other in the observed domain. For objects from a vector-space dataset, variations of *Minkowski distance formula* (see eq. 1.2) can be used to easily create the metric for a dissimilarity measure. They are *Manhattan distance* ($p = 1$), *Euclidean distance* ($p = 2$) and *Chebyshev distance* ($p \to \infty$) (see tab. 1.1). The other possible metric is the *cosine similarity* (see eq. 1.3).

$$||a - b||_p = (\sum_{i=1...d} |a_i - b_i|^p)^{\frac{1}{p}} \tag{1.2}$$

$$cos(a, b) = \frac{a \cdot b}{||a||||b||} \tag{1.3}$$

As the choice of a distance function influences the result of a clustering, it should be chosen with respect to the properties of a provided dataset. Aggarwal, Hinneburg, and Keim [1] show the qualitative behavior of different distance functions in the k-means algorithm.

Distance measure	Formula
Manhattan	$\|a-b\|_1 = \sum_i \|a_i - b_i\|$
Euclidean	$\|a-b\|_2 = \sqrt{\sum_i (a_i - b_i)^2}$
Chebyshev	$\|a-b\|_\infty = \max_i \|a_i - b_i\|$

Table 1.1: Variations of the Minkowski distance formula.

Linkage criteria

A hierarchical algorithm can not compute the dissimilarity of two clusters only by a distance function; it is a function of dataset objects. To completely define the dissimilarity measure, we need a function of sets of object — a linkage criterion. It describes any process of measuring dissimilarity between two groups of objects. In a hierarchical algorithm, it measures clusters to determine which two will be linked together. Given clusters A and B of a vector-space dataset and a distance function d, we define the following linkage criteria [26] (see fig. 1.2):

Single linkage – The single linkage criterion computes the distance between A and B as the minimum distance between all pairs $(a, b) \in A \times B$:

$$\min\{d(a,b) : a \in A, b \in B\}.$$

The major drawback this criterion suffers is the cluster chaining. It occurs when connected clusters do not share any other pair of close objects than the one that determined the connection. This produces long thin clusters with a big distance between some objects.

Complete linkage – The complete linkage criterion is similar to the single linkage criterion. But as opposed to finding the minimum, this criterion uses the maximum of object pairs for the computation of a cluster dissimilarity:

$$\max\{d(a,b) : a \in A, b \in B\}.$$

The criterion suffers from its simplicity as well as the single linkage. But instead of naively connecting dissimilar clusters, here similar clusters are not connected in some cases. Having all object pairs in a close proximity to each other but one object being rather far from the others, the criterion will not link the clusters as the maximum distance deteriorates the rest.

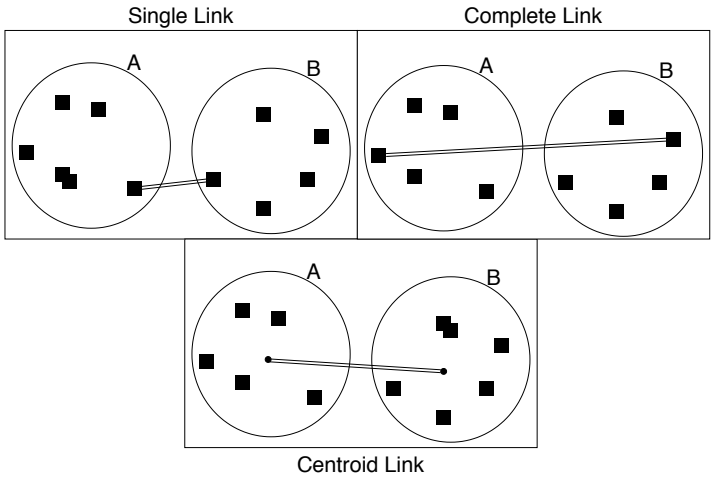

Figure 1.2: An example of three linkage criteria. The double line represents the distance between clusters A and B according to the respective criterion.

Centroid linkage – The centroid linkage criterion tries to solve the problems of the aforementioned criteria by measuring the distance between the centroids of clusters. It introduces a form of an average into the computation; the think that the criteria above lack.

The choice of a linkage criterion in hierarchical clustering algorithm is vital. As stated above, it can change the course of clustering in a great manner. Chosen improperly, it can have a disastrous effect on the final result.

1.2 Hierarchical clustering with the Mahalanobis linkage

Commonly, HCA algorithms branch into different variations to become more suitable for a specific dataset type [17, 18, 27]. A common cause for such customization is a shape of clusters, i.e. an elliptical or gaussian shape (see fig. 1.3). A general HCA is unable to properly cluster such dataset. The *Mahalanobis-average hierarchical clustering analysis* (MHCA) focuses on the datasets that create clusters of ellipsoid shapes. Such datasets commonly originate in the measurements of cell cytometry data in bioinformatics, which was the original purpose for the design of MHCA.

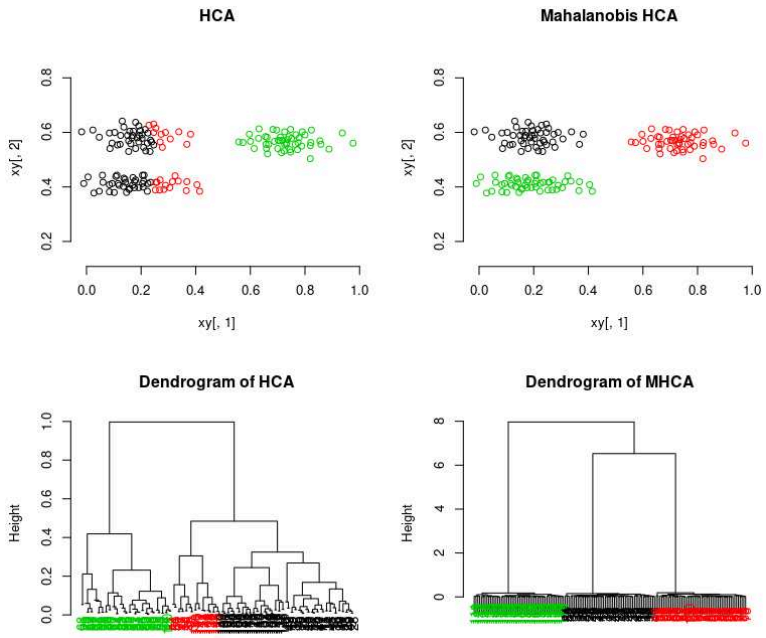

Figure 1.3: A comparison of HCA and MHCA clustering on a dataset that contains elliptic groups of points (taken from the `tsieger/mhca` Github repository). The upper two figures show the state of clustering when both algorithms cluster the dataset into three clusters. MHCA properly recognizes the elliptic clusters.

1.2.1 Mahalanobis distance

An important part of MHCA is the dissimilarity measure based on the *Mahalanobis distance* [16], a distance between a point and a set of points (in our case a cluster). If points in the set are strongly correlated in some axis, then points laying on this axis are closer to the set than points that are not; even if their euclidean distance to the center of the set is closer.

To measure the Mahalanobis distance, a covariance matrix of a participating cluster needs to be computed. We compute the covariance matrix using the random vector of a cluster.

Definition 5 (Random vector of a cluster)**.** *Given a cluster C of a vector-space dataset, we define its random vector v as a vector of d discrete random variables, where the random variable v_i takes values from $\{x_i : x \in C\}$ with equal probability.*

Definition 6 (Covariance matrix)**.** *Given a random vector v of length n, we define the covariance matrix $\mathrm{cov}(v)$ as a $n \times n$ matrix that holds covariances of each pair*

of vector element:

$$\operatorname{cov}(v)_{i,j} = \operatorname{cov}(v_i, v_j)$$

Using the random vector of a cluster we can compute the covariance matrix of a cluster and properly define the Mahalanobis distance:

Definition 7 (Mahalanobis distance)**.** *Suppose a cluster C of a vector-space dataset $\mathcal{D} \in \mathbb{R}^d$ and the random vector v of C. If the $\operatorname{cov}(v)$ is regular, we define the* Mahalanobis distance *between $u \in \mathbb{R}^d$ and C as*

$$d_{Maha}(u, C) = \sqrt{(u - \operatorname{mean}(C))^T \operatorname{cov}(v)^{-1} (u - \operatorname{mean}(C))}. \tag{1.4}$$

Note that this equation computes a distance between a point and a cluster. To fully incorporate the Mahalanobis distance in a hierarchical algorithm, we need to construct a method to measure a distance between two clusters.

Two usable methods naturally arise. We compute either the arithmetic mean of the distances between all points and a cluster or just the distance between a centroid and a cluster. We respectively call these methods the Full Mahalanobis distance (FMD) and the Centroid Mahalanobis distance (CMD).

Definition 8 (Full Mahalanobis distance)**.** *Having clusters C and C', we define the Full Mahalanobis distance between clusters C and C' as the arithmetic mean of the Mahalanobis distances between each object $o \in C$ and the cluster C':*

$$d_{MahaFull}(C, C') = \frac{1}{|C|} \sum_{o \in C} d_{Maha}(o, C').$$

Definition 9 (Centroid Mahalanobis distance)**.** *Having a cluster C with its centroid c and a cluster C', we define the Centroid Mahalanobis distance between clusters C and C' as the Mahalanobis distance between c and C':*

$$d_{MahaCentroid}(C, C') = d_{Maha}(c, C').$$

To illustrate the measure of the Mahalanobis distance, let us suppose we have two elliptic clusters. In the means of the proximity, the measure favors such clusters that their ellipsis are alongside rather than in a prolongation of one another [5] (see fig. 1.4). Only when the objects of a cluster form a spherical shape, this measure of dissimilarity is proportional to the euclidean distance with a corresponding linkage.

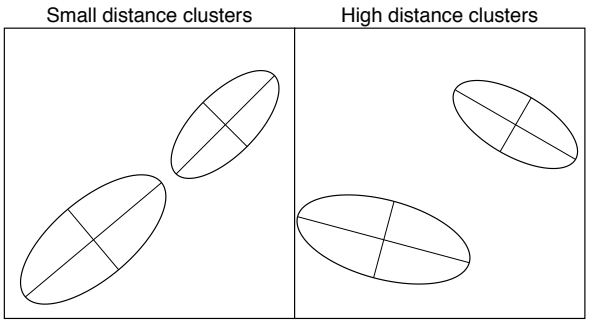

Figure 1.4: An example of the cluster dissimilarity in the Mahalanobis-average hierarchical clustering.

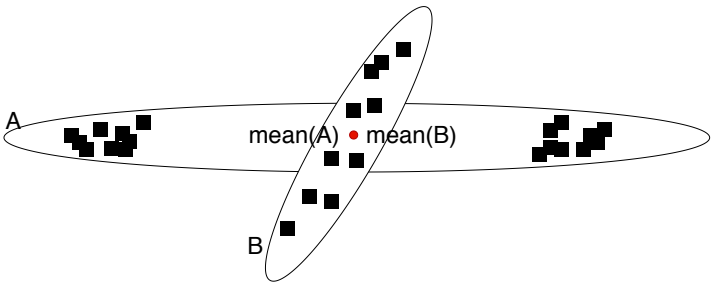

Figure 1.5: An example of clusters with a big difference between FMD and CMD dissimilarity measures.

CMD can be interpreted as an approximation substitute for FMD variant. A case when CMD variant fails to compute good approximate measure happens when centroids of measured clusters are very close to each other while their points do not lay on each others major axis (see fig. 1.5). Such case is rather unrealistic; therefore, we assume it is unlikely to happen in cell cytometry datasets.

To properly utilize the Mahalanobis distance variants as dissimilarity metrics in a MHCA implementation, we need the distance function to be symmetric. Originally, the Mahalanobis distance is not a symmetric function. Fortunately, we easily fix the aforementioned distance functions with the Generalized distance.

Definition 10 (Generalized distance)**.** *Given clusters C, C' and a variant of the Mahalanobis distance $d_{Variant}$ (either FMD or CMD), we define the* Generalized distance *between clusters C and C' as the arithmetic mean of distances between C, C'*

and C', C:

$$d_{General}(C, C') = \frac{d_{Variant}(C, C') + d_{Variant}(C', C)}{2}.$$

1.2.2 Singularity of cluster covariance matrix

In early stages of MHCA — when clusters consist of fewer points — the covariance matrix of a cluster is often singular. As a result, we can not compute its inverse and perform the distance measure. Furthermore, the matrix can happen to be close to singular and — when inverted — can produce false results such as negative distances or infinity floating points.

To solve this problem, Fišer et al. [8] specified a threshold of a cluster size; if the size of a cluster is lower than the threshold, then the cluster dissimilarity measurements are computed using euclidean distance. The value of the threshold is proportional to the size of a dataset. We denote this threshold as the *Mahalanobis threshold*. To implement the singularity workaround we defined the Non-singular Mahalanobis Distance.

Definition 11 (Non-singular Mahalanobis Distance)**.** *Given a threshold T_M, a cluster C and its centroid c, a cluster C' and its centroid c', a variant of the Mahalanobis distance $d_{Variant}$ (either FMD or CMD) and the Generalized distance $d_{General}$, we define the Non-singular Mahalanobis Distance as*

$$d_{NSing}(C, C') = \begin{cases} d_{General}(C, C'), & \text{if } |C| \geq T_M \text{ and } |C'| \geq T_M, \\ \dfrac{d_{Variant}(C, C') + ||c - c'||_2}{2}, & \text{if } |C| < T_M \text{ and } |C'| \geq T_M, \\ \dfrac{||c - c'||_2 + d_{Variant}(C', C)}{2}, & \text{if } |C| \geq T_M \text{ and } |C'| < T_M, \\ ||c - c'||_2, & \text{if } |C| < T_M \text{ and } |C'| < T_M. \end{cases}$$

1.3 Computational complexity of the hierarchical clustering algorithms

The common problem in clustering algorithms is their time complexity. In general, the time complexity of an agglomerative HCA is $\mathcal{O}(n^3)$, which restricts its use for large datasets ($\geq 10^5$ points when using the contemporary hardware) [21]. Day and Edelsbrunner [6] propose three different HCA variants based on the data structures they utilize:

- HCA with the dissimilarity matrix,
- HCA with the nearest neighbor array,
- HCA with priority queues.

These variants are restricted to metrics from the Minkowski distance family and a specified list of linkage criteria. To measure a dissimilarity between two clusters, they use the Lance and Williams recurring formula [13, 14].

The formula can be used only with specific linkage criteria. For a dissimilarity measure d, it specifies a measurement between a new cluster c merged from clusters (i, j) and any other cluster k as a recurring formula

$$d(c, k) = \alpha_i \cdot d(i, k) + \alpha_j \cdot d(j, k) + \beta \cdot d(i, j) + \gamma |d(i, k) - d(j, k)|.$$

For each linkage criterion, specific values for constants α_i, α_j, β and γ are defined.

1.3.1 HCA with the dissimilarity matrix

The first variant caches the dissimilarity measurements in the *dissimilarity matrix* M.

Definition 12 (Dissimilarity matrix)**.** *Having a vector-space dataset $\mathcal{D}$ divided into clusters $C_1, \ldots, C_m$ and a function d as a measure of dissimilarity, we define the* dissimilarity matrix M *as a $m \times m$ matrix where $M_{ij} = d(C_i, C_j)$.*

Unless there remains only one cluster, this algorithm searches M for the most similar pair of clusters (represented by the minimum of matrix elements), stores the pair and updates M (see alg. 2). The initialization on line 2 computes the Minkowski distance between all n points. As the time complexity of the distance is proportional to the point dimensionality, the initialization time is $\mathcal{O}(d \cdot n^2)$.

The main cycle on line 3 repeats $n = |\mathcal{D}|$ times; in each iteration, the number of clusters is reduced by 1. The search on line 4 is bounded by $\mathcal{O}(n^2)$ time, with maximal complexity in the first iteration when $k = n$. The update in line 6 reflects the deletion of clusters i and j and the addition of the new one. Hence, it needs to perform k new dissimilarity measurements for the new cluster. Using the recurring formula and cached measurements in M, this step can be performed in $\mathcal{O}(k)$ time (bounded by $\mathcal{O}(n)$ during the first iteration as well). As there are total n iterations performed, this results in the overall time complexity of $\mathcal{O}(n^3 + d \cdot n^2)$.

The space required to store M is $\mathcal{O}(n^2)$. As there is no other non-trivial requirement, the overall space complexity is $\mathcal{O}(n^2)$ as well.

Algorithm 2 HCA with dissimilarity matrix

```
1: procedure DISMAT(D ⊂ ℝ^d)
2:     initialize M                                   ▷ time: O(d · |D|^2)
3:     for k = |D| ... 1 do
4:         search M for the closest pair (i, j)       ▷ time: O(k^2)
5:         store cluster pair (i, j) into the merge list   ▷ time: O(1)
6:         update M                                   ▷ time: O(k)
7:     end for
8:     return list of merged clusters
9: end procedure
```

1.3.2 HCA with the nearest neighbor array

In addition to the dissimilarity matrix, this algorithm introduces the array of the nearest neighbors.

Definition 13 (Nearest neighbor array)**.** *Suppose a vector-space dataset $\mathcal{D}$ divided into clusters $C_1, \ldots, C_m$ and a function d as a measure of dissimilarity. Then, the* nearest neighbor array N *is a m-element array of indices $\{1, \ldots, m\}$ such that for each element N_i holds*

$$d(C_i, C_{N_i}) = \min\{d(C_i, C_j) : j \in \{1, \ldots, m\} \setminus \{i\}\}$$

Each cluster is assigned the index pointing to its closest neighboring cluster in the dissimilarity matrix. Compared with the previous algorithm, this algorithm trades the expensive *closest pair search* with the expensive *structure update* (see alg. 3). On line 2 and 3, M and N are initialized. To initialize N, whole M is searched for the closest neighbor of each point.

On line 5, each closest pair search can be performed in $\mathcal{O}(n)$ time as the array length does not exceeds n elements. Line 7 does dot differ from the previous variant. Line 8 updates N. The worst case update of N happens when each cluster resides in the closest neighborhood with the clusters that are being merged (clusters i and j). In that case, whole dissimilarity matrix has to be searched. Hence, the time complexity for this step is $\mathcal{O}(n^2)$. To sum up, the overall time complexity is $\mathcal{O}(n^3 + d \cdot n^2)$ and the space complexity is $\mathcal{O}(n^2)$ because we add N that does not have more than linear space requirements.

Despite the equal time complexities, this algorithm may outperform the previous one because in the majority of situations the update step on line 8 does not require the whole array to be recomputed. Moreover, if the number of elements to be updated remains constant each iteration, the overall algorithm time complexity may be $\mathcal{O}(d \cdot n^2)$.

Algorithm 3 HCA with the nearest neighbor array

```
1:  procedure NEIGHBOR(D ⊂ ℝ^d)
2:      initialize M                                  ▷ time: O(d · |D|^2)
3:      initialize N                                  ▷ time: O(|D|^2)
4:      for k = |D| ... 1 do
5:          search N for the closest pair (i, j)      ▷ time: O(k)
6:          store cluster pair (i, j) into the merge list  ▷ time: O(1)
7:          update M                                  ▷ time: O(k)
8:          update N                                  ▷ time: O(k^2)
9:      end for
10:     return list of merged clusters
11: end procedure
```

Algorithm 4 HCA with priority queues

```
1:  procedure QUEUES(D ⊂ ℝ^d)
2:      initialize M                                  ▷ time: O(d · |D|^2)
3:      for all o ∈ D do
4:          initialize a priority queue from D \ {o}  ▷ time: O(|D|)
5:      end for
6:      for k = |D| ... 1 do
7:          search k queues for the closest pair (i, j)  ▷ time: O(k)
8:          store cluster pair (i, j) into the merge list  ▷ time: O(1)
9:          update M                                  ▷ time: O(k)
10:         update k − 1 priority queues              ▷ time: O(k · log k)
11:     end for
12:     return list of merged clusters
13: end procedure
```

1.3.3 HCA with priority queues

This algorithm takes the advantage of the previous one employing the fast search and combines it with the fast update using *priority queues*. Each object from a dataset is assigned a priority queue constructed from the remainder of the dataset. The priority label of a queued element is a dissimilarity measure between the object and the queued element.

When a priority queue is implemented as a heap, its time complexity can be $\mathcal{O}(1)$ for the minimum retrieval, $\mathcal{O}(n)$ for initialization and $\mathcal{O}(\log n)$ for the insertion and deletion of an element [10].

Therefore, in alg. 4, the search step on line 7 takes $\mathcal{O}(k)$ time (bounded by $\mathcal{O}(n)$ during the first iteration). Next on line 10, we need two delete and one

insert operations (corresponding to deleting two merged clusters and inserting one new). We can perform such update on n queues in $\mathcal{O}(n \cdot \log n)$ time.

To sum up, the overall time complexity is $\mathcal{O}(n^2 \cdot \log n + d \cdot n^2)$. The space complexity is $\mathcal{O}(n^2)$ because all n queues have the linear space requirements.

1.3.4 In-place HCA

For some of the linkage criteria above, we can remove the dissimilarity matrix from the aforementioned HCA variants to decrease the space complexity while preserving the same asymptotic time complexity.

We propose two variations for centroid linkage with Minkowski distance:

In-place HCA removes the matrix from the HCA with dissimilarity matrix. To find the most similar cluster pair, it computes the dissimilarities in-place.

To speed the dissimilarity measurements, an array of centroids is maintained. With the centroids precomputed, the dissimilarity measure complexity is dependent only on the Minkowski distance; hence, it can be performed in $\mathcal{O}(d)$.

As the search step now takes $\mathcal{O}(d \cdot n^2)$ time, the overall time complexity is $\mathcal{O}(d \cdot n^3)$. The space complexity is $\mathcal{O}(n)$ because the centroid array requires only linear space.

In-place HCA with the nearest neighbor array adds the neighbor array to the in-place HCA.

As the array has linear space complexity, the overall space complexity is $\mathcal{O}(n)$. The time complexity for N search is the same as the original variant and the N update complexity increases by the factor of d.

Therefore, the time complexity is $\mathcal{O}(d \cdot n^3)$. Same as in the original variant, this variant promises the time complexity $\mathcal{O}(d \cdot n^2)$ if the number of the nearest neighbors to update remains constant.

We did not include the in-place HCA with priority queues as we can not reduce the space complexity to subquadratic without removing the queue structures.

We summarize the time and space complexity of the mentioned algorithms in the table 1.2.

To show an example of the big algorithm complexity, we tested the limits of a dissimilarity matrix variant implementation. We measured the R language

HCA variant	Time complexity	Space complexity
HCA with dissimilarity matrix	$\mathcal{O}(n^3 + d \cdot n^2)$	$\mathcal{O}(n^2)$
HCA with the nearest neighbor array	$\mathcal{O}(n^3 + d \cdot n^2)$	$\mathcal{O}(n^2)$
HCA with priority queues	$\mathcal{O}(n^2 \log(n) + d \cdot n^2)$	$\mathcal{O}(n^2)$
in-place HCA	$\mathcal{O}(d \cdot n^3)$	$\mathcal{O}(n)$
in-place HCA with the neighbor array	$\mathcal{O}(d \cdot n^3)$	$\mathcal{O}(n)$

Table 1.2: The summary of time and space complexity for the HCA variants.

library function `hclust`; a frequently used implementation of HCA. It computes the single linkage with the euclidean distance and uses the dissimilarity matrix.

Fig. 1.6 supports the above stated polynomial time complexity of the implementation. Moreover, it stopped at a dataset size of 47K because the testing machine [1] ran out of memory.

In conclusion, the space complexity of HCA can be even more restrictive in the means of the overall algorithm usability than its time complexity. The performance and usability of the HCA variants depends on many factors and the programmer needs to find a balance between their tradeoffs. The metric function is one of them. For example, when the Minkowski distance is used, we can prefer the in-place HCA over the HCA with dissimilarity matrix for a lower space complexity while retaining the same asymptotic time complexity. However, if the distance metric is more complex, one may prefer to store the measures in the memory.

[1] Intel Core i9-8950HK, 32GB RAM

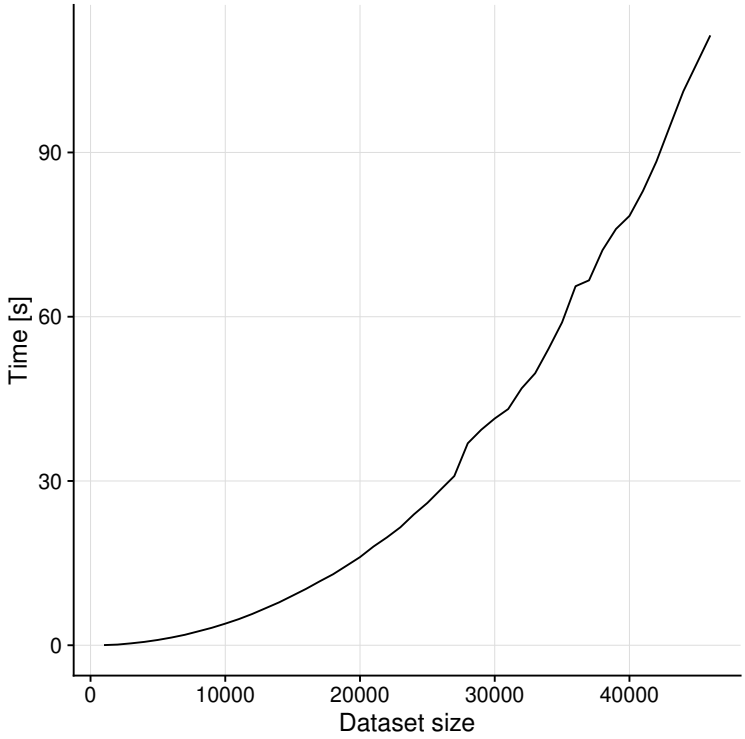

Figure 1.6: Time complexity of `hclust` with respect to the size of a dataset.

2. GPU implementation

This chapter focuses on the implementation of the Mahalanobis-average hierarchical clustering analysis. First, we summarize the most important parts of the CUDA parallel platform. Next, we describe the algorithm and the high level look on the implementation. Last, we thoroughly describe the most important functions.

2.1 CUDA programming model overview

The problem this thesis aims to solve is to implement an algorithm that is able to provide a hierarchical clustering of very large datasets while retaining a reasonable computation time — the time required to compute a magnitude smaller datasets with current HCA algorithms. In an effort of achieving this performance, we used a combination of C++ programming language and *CUDA*[1] API.

CUDA is a parallel platform and API allowing a programmer to use GPU for general purpose programming. This API exposes a computational power of hundreds (even thousands) cores of CUDA-enabled GPUs [4].

2.1.1 Terminology

The starting point for running any code on GPU using CUDA is a *kernel*. A kernel is a function that is executed on GPU; we say it contains *device code*. Complementary to a device code, a *host code* is a phrase for code executed on a CPU. Hence, a common CUDA application runs host code that determines which device code to run next.

A kernel is run n times in parallel by n threads each having its unique *thread ID*. IDs of threads can be identified by *one-dimensional*, *two-dimensional* or *three-dimensional* indices forming a block of threads, called *thread block*. This property reflects shapes of common structures such as vectors and matrices resulting in more natural programming work.

Since all block threads reside on the same processor and share common resources, the block size is limited[2]. However, a kernel can be launched with multiple equally shaped blocks to increase the number of running threads. They can be organized into up to three-dimensional structure called *grid*. This naturally implies unique block ID. A grid can be of an arbitrary size; usually dictated

[1] Compute Unified Device Architecture

[2] Currently up to 1024 threads

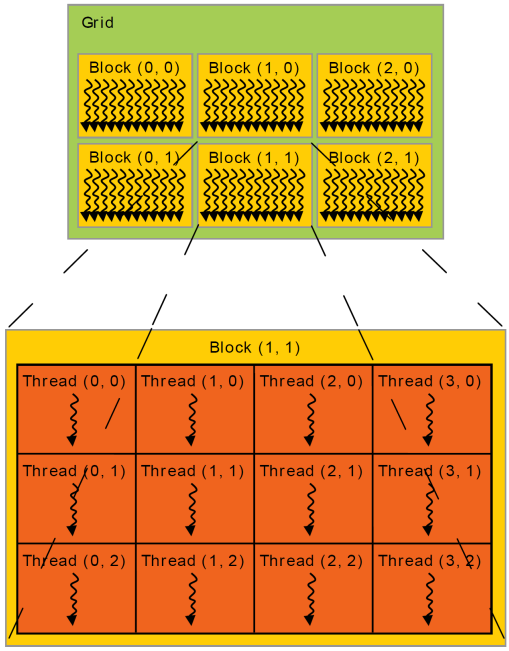

Figure 2.1: An example of 2D grid of size (3,2) with 2D blocks of size (4,3) (taken from the CUDA C++ Programming Guide).

by the computed data (see fig. 2.1). It is a common practice that grid size surpasses the number of GPU processors.

The device memory is hierarchically divided into parts each being accessible by a different set of threads:

Global memory – This memory is accessible by any thread. Any memory request is transferred via transactions; hence, to avoid decrease of the data throughput, memory accesses should be coalesced.

Local memory – Each thread has exclusive access to its local memory. As it resides in a global memory, the local memory is a thread private global memory.

Shared memory – The shared memory is assigned to a block. It means that threads from the same block have access to the same shared memory. It is placed on-chip so it has much lower latency and higher bandwidth than the global or local memory.

Constant memory – The constant memory is a read-only memory accessible by any thread. Due to its read-only property, it can be heavily cached and

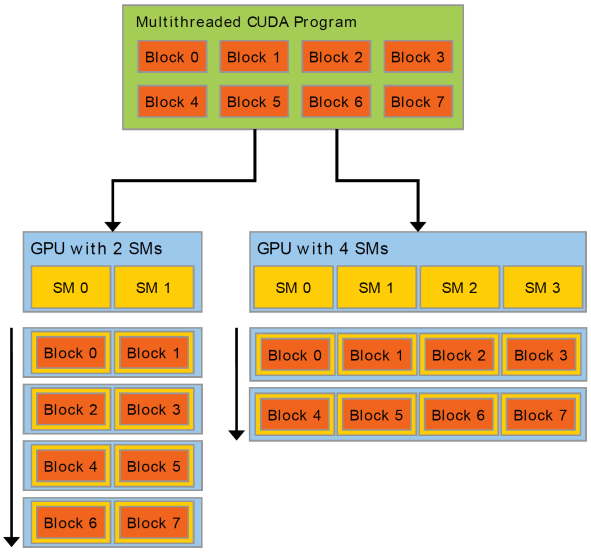

Figure 2.2: An example of different grid block distributions among SMs (taken from the CUDA C++ Programming Guide).

may perform better than global memory. Together with global memory, it persists across different kernel launches.

The building block of the CUDA-enabled GPU hardware architecture is multithreaded *streaming multiprocessor (SM)*. A GPU contains an array of multiprocessors whose number varies between different GPUs. When a kernel launches, its grid of blocks is distributed among available multiprocessors and execute in parallel (see fig. 2.2). On the selected multiprocessor, block threads are executed in parallel as well. Moreover, multiple blocks can run concurrently on one multiprocessor.

To achieve this grade of parallelism, a GPU employs the *SIMT architecture* (Single Instruction - Multiple Threads). A multiprocessor partitions an assigned block to chunks of 32 threads called *warps*. Each warp thread is executed concurrently. During the execution, threads start with the the same program address but have own registers and instruction pointers so they can branch independently. However, a warp executes one common instruction at the time. Hence, to achieve the greatest performance all warp threads must agree on the program path.

Moreover, as warp threads execute concurrently, CUDA offers *warp shuffle instructions*. All threads in a warp are able to distribute their data to another thread within just one instruction. This comes to a great use for synchronization but can also be used to achieve high throughput reduction operations as well.

2.1.2 GPU performance and optimization concerns

To greatly utilize a CUDA-enabled GPU and achieve a good performance, a program code must be written with respect to the GPU architecture. These guidelines help us in this task:

A GPU task has to be split to many small data dependent sub-tasks (e.g. a computation of one element in a dissimilarity matrix). Threads must not perform many different tasks. Rather, they must perform the same task with a different data. Satisfying this condition, we follow the model of the SIMT architecture.

The next step is to maximize the memory throughput. The programmer should minimize data transfers with low bandwidth such as copying data between CPU and GPU. This can be accomplished by creating and operating with structures directly in a GPU without mapping it to the CPU memory.

On-chip shared memory should be utilized. It is equivalent to the user-managed cache and provides much higher data throughput than the global memory. Heavy use task related data should be moved there.

Last, the application should maximize the instruction throughput. It is accomplished by using lower precision data types that does not affect the result or by using high throughput instructions like warp shuffle. The most importantly, the programmer should minimize the use of control flow instructions (if, switch, while, etc.) that cause divergent warps.

2.2 Implementation strategy

Section 1.3 introduced three variants of a hierarchical clustering analysis. To decide which one to implement, it must satisfy the memory usage and the level of parallelism. Specifically, the algorithm must have subquadratic space requirements as it must be able to process rather large datasets. Next, the algorithm should exhibit some parallelism opportunities; otherwise massive GPU parallel properties will be to no use. We compare the mentioned algorithms according to these conditions:

In-place HCA satisfies the memory usage with its linear space complexity. In this variant, finding the most similar cluster pair is equivalent to a computation of the whole dissimilarity matrix. The parallelism requirement is satisfied because each measure of dissimilarity can be computed independently, exposing great parallelism opportunity.

In-place HCA with the nearest neighbor array satisfies the required memory usage as well. On the other hand, it provides less space for a parallelism because the algorithm does not compute whole dissimilarity matrix as in

the previous algorithm. Large dataset sizes may compensate for this disadvantage.

HCA with priority queues is unsuitable as it does not satisfy the memory usage. Moreover, operation over priority queues (insert, delete and min) are not of a parallel nature and may create a bottleneck in the computation.

For this thesis we chose to implement HCA with the nearest neighbor array. It promises better time complexity than the HCA with dissimilarity matrix which can come to a great use when clustering big datasets. In addition, we utilized the advantages of priority queues by defining a constant that declares the number of closest neighbors assigned to each cluster (so there can be also the second closest, the third closest, etc.).

Additionally, we decided to implement the CMD variant of the Mahalanobis distance (see def. 9) as a measure of dissimilarity. It is simpler to implement than FMD and it may provide further knowledge for implementation of the FMD variant.

2.2.1 Apriori clustering optimization

As already mentioned, the present HCA algorithms can hardly cluster datasets of big sizes. This can be improved by implementing the apriori clustering method.

Definition 14 (Apriori clustering)**.** *Given a dataset $\mathcal{D}$ and a HCA algorithm $\mathcal{A}$, we define the apriori clusters of $\mathcal{A}$ as a partitioning of $\mathcal{D}$ into non-empty subsets $\mathcal{D}_1, \ldots, \mathcal{D}_k$ if these conditions hold:*

1. *$\mathcal{A}$ performs clustering of each subset separately; meaning that clusters from different subsets can not be merged.*

2. *When each subset is clustered into the only cluster, $\mathcal{A}$ clusters the resulting clusters as if they were in one subset.*

The intended usage of this method is that the apriori clusters are created from a dataset using non-hierarchical clustering algorithm such as *k-means* — an algorithm that is capable of creating a dataset partitioning fast. With this feature, a HCA algorithm is guided by the apriori clusters to create the corresponding merged clusters. They do not cluster the whole dataset at once. Rather, each apriori cluster is clustered separately. This results in a much faster computation (see section 3.2.2) and increased size of computable datasets.

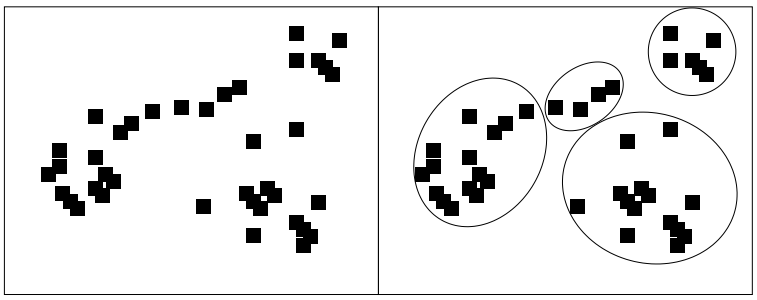

Figure 2.3: An example dataset with (right) and without (left) visualized precomputed apriori clusters (each apriori cluster is visualized as an ellipse — dataset points contained in an ellipse represent a subset of the dataset).

2.3 Benchmarking framework

We implemented the selected algorithm with the apriori clustering method in C++ language. Its implementation resides in the main class of the program — `gmhc`[3]. It inherits from an abstract template class `hierarchical_clustering`. This template provides key data fields and methods for a hierarchical clustering algorithm (see list. 2.1):

`initialize()` sets the fields of the class. A templated field `points` is an array of dataset objects — points. They are expected to be of a floating-point numeric type. Fields `point_dim` and `points_size` state a point dimensionality and a total number of points in the array. Hence, each `point_dim` consecutive array elements represent one point and the total array size is `point_dim` · `points_size` elements.

`run()` initiates the clustering. It returns a vector of `pasgnd_t` structures — pairs that contain *assignments* and a *distance* — the IDs of merged clusters and the distance between them. Each point from the input array is assigned an unique consecutive ID starting from 0. When two clusters are merged, the new cluster is assigned the next available unique ID. The process of merging is then stored in the returning vector. Hence, the vector completely describes the whole clustering process.

`free()` deallocates all acquired resources.

[3] An abbreviation for GPU Mahalanobis-average Hierarchical Clustering

```
using csize_t = uint32_t;
using asgn_t = csize_t;
using pasgn_t = std::pair<asgn_t, asgn_t>;
template <typename T>
using pasgnd_t = std::pair<pasgn_t, T>;

template <typename T>
class hierarchical_clustering
{
protected:
        const T* points;
        csize_t points_size;
        csize_t point_dim;
public:
        virtual void initialize(...);
        virtual std::vector<pasgnd_t<T>> run() = 0;
        virtual void free() = 0;
};
```

Listing 2.1: A summary of `hierarchical_clustering` header file.

2.4 Mahalanobis clustering implementation

The class `gmhc` is the entry point of the whole gpu-accelerated MHCA algorithm. It inherits from `hierarchical_clustering<float>` template specialization, which means that the algorithm expects dataset objects to be single-precision points of a specified dimension. The class communicates with a GPU; hence, it holds structures used by the device.

2.4.1 Initialization

The initialization happens in the `initialize` method. Due to the initialization of device and host structures, this stage must be treated with importance. The class overloads `initialize` method so that it sets three additional fields:

- `mahalanobis_threshold` indicates the threshold of the Non-singular Mahalanobis distance (see def. 11).
- `apriori_assignments` is an array of assignments that splits `points` into *apriori clusters*. This is an optional field of the `initialize` method.
- `validator` is an optional field used for testing purposes.

Next follows the main responsibility of the `initialize` method — device and host data allocation and initialization. They are *point*, *assignment*, *centroid*, *inverse covariance*, *neighbor* and *status arrays* $\mathcal{P}$, $\mathcal{A}$, $\mathcal{C}$, $\mathcal{I}$, $\mathcal{N}$, $\mathcal{S}$.

Definition 15 (Point array)**.** *Suppose a dataset $\mathcal{D}$ that contains objects represented by k-dimensional points. We define point array $\mathcal{P}$ as an array that contains consecutive sequence of all objects in $\mathcal{D}$.*

Definition 16 (Assignment array)**.** *Suppose a partitioning of a point array $\mathcal{P}$ into clusters $c_1, \ldots, c_k$ where each cluster c_i is defined by its unique ID id_i and a set of indices I_i that state which points from $\mathcal{P}$ belong to the cluster. The array of length $|\mathcal{P}|$ is the assignments array $\mathcal{A}$ when for each cluster c_i the following equation holds:*

$$\forall j \in I_i : \mathcal{A}_j = id_i$$

Definition 17 (Cluster-related arrays)**.** *Given a dataset $\mathcal{D}$, its clusters $c_1, \ldots, c_k$, a Mahalanobis threshold T_M and a positive number $m \leq |\mathcal{D}|$, we define the following cluster-related arrays:*

- *the centroid array $\mathcal{C}$ as an array that contains centroids of all k clusters,*
- *the inverse covariance array $\mathcal{I}$ as an array that contains inverse covariance matrices of all clusters that reached T_M,*
- *the neighbor array $\mathcal{N}$ as an array that contains m indices and distances to the m closest clusters for each of k clusters,*
- *the status array $\mathcal{S}$ as an array that contains sizes and unique ids of all k clusters.*

All but the status array are device arrays. Device arrays are used within kernels while the host array controls the flow of run kernels.

2.4.2 Clustering

The clustering algorithm resides in the method `run`. This method utilizes the array of `clustering_context_t` objects — instances of a *clustering context* — that divide the dataset according to the provided apriori clusters (see fig. 2.4).

Definition 18 (Clustering context)**.** *Given a dataset $\mathcal{D}$, its point array $\mathcal{P}$, assignment array $\mathcal{A}$ and cluster-related array tuple $(\mathcal{C}, \mathcal{I}, \mathcal{N}, \mathcal{S})$, we define the clustering context of $\mathcal{D}' \subset \mathcal{D}$ as a tuple $(\mathcal{P}', \mathcal{A}', \mathcal{C}', \mathcal{I}', \mathcal{N}', \mathcal{S}')$. In the tuple, each array is a sub-array of the corresponding array of $\mathcal{D}$. Each sub-array contains data of points and clusters created within $\mathcal{D}'$.*

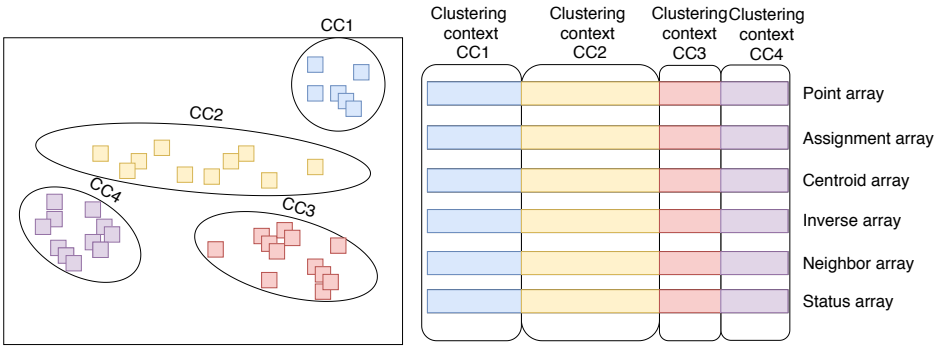

Figure 2.4: An example of a dataset division into subsets and creation of clustering contexts (each context is assigned the sub-arrays of the main arrays with data relevant only to the subset).

The algorithm proceeds as follows. If `apriori_assignments` are present, the clustering context array is initialized and filled with clustering contexts. Each of them describes the data structure of the corresponding apriori cluster and holds methods for its further clustering. It will be further called the *context array*.

In addition to that, the method `run` uses one special clustering context called the *final context*. If apriori clusters are present (and the context array is non-empty), the final context is unset and serves as a context that will host the apriori cluster results for the final clustering. If there are no apriori clusters, the final context is set and contains the whole dataset.

We divide the whole clustering algorithm into the top level *apriori clustering layer* and the low level *context clustering layer*.

Apriori clustering layer handles a separate clustering of apriori clusters, merges their results into one clustering context and performing the final clustering (see alg. 5).

In this layer, the algorithm iterates over each apriori cluster from the context array. It enters the context clustering layer to perform the clustering. It retrieve the result that is then inserted into the final context. Either when all apriori clusters are merged or when there are none at all, the final context with the remaining clusters is clustered and the list of assignment pairs is returned.

Before any context begins the clustering, its closest neighbor array has to be initialized. It is done by the kernels `neighbors`.

Context clustering layer performs one iteration of clustering within the spec-

Algorithm 5 Apriori clustering

1: **procedure** APRIORI(context array $\mathcal{T}$, the final context ctx_f)
2: **for all** $ctx \in \mathcal{T} \cup ctx_f$ **do** ▷ iterate over apriori clusters and the final context
3: initialize the closest neighbor array of ctx
4: **while** ctx contains more than one cluster **do**
5: perform context clustering of cxt for the closest pair (i, j) and their distance d
6: store $((i, j), d)$ into the returning list
7: **end while**
8: $ctx_f \leftarrow ctx_f \cup ctx$ ▷ assign merged cluster into the final context
9: **end for**
10: **return** list of merged clusters
11: **end procedure**

ified context (see alg. 6). First, it goes through the neighbor array of the context and finds the closest pair of clusters (kernel `neighbor_min`). At that moment, there are two clusters to be removed and one new cluster to be created. Hence, the cluster-related arrays of the context need to be updated accordingly.

The remainder of this method sets data for the new cluster. First, $\mathcal{A}$ is recomputed to correctly show that points of the former two clusters now belong to the new one (kernel `merge_clusters`). Then, the new centroid is computed and stored into $\mathcal{C}$ (kernel `centroid`) and status array is updated as well (in the host code). Last, if the new cluster reaches the Mahalanobis threshold, its inverse covariance matrix is computed and stored into $\mathcal{I}$ (kernels `compute_icov`).

The final statement of the context clustering method checks the whole $\mathcal{N}$ according to the one created and two deleted clusters and updates it when needed (kernels `update_neighbors`).

2.4.3 Data order

This section describes the actual order of data in the cluster-related arrays and describesthe process of reorganizing the cluster-related arrays provided that a cluster pair (i, j) is to be merged.

This is achieved with a help of the `cluster_bound_t` structure called the *indexing structure*. It is a rather simple structure that describes how the cluster-related arrays organize their data. The structure distinguishes two parts of the

Algorithm 6 Context clustering

procedure CONTEXT($\mathcal{A}, \mathcal{C}, \mathcal{I}, \mathcal{N}, \mathcal{S}, \mathcal{Q}$)
    retrieve the closest pair (i, j) and distance d from $\mathcal{N}$
    initialize an unique id of the new cluster c
    update $\mathcal{Q}$ and retrieve new index idx for c
    reorder data in $(\mathcal{C}, \mathcal{I}, \mathcal{N}, \mathcal{S})$
    update $\mathcal{A}$
    $\mathcal{C}_{idx} \leftarrow$ compute the centroid of c
    $\mathcal{S}_{idx} \leftarrow \{|c|, id\}$
    if $|c| \geq$ `mahalanobis_threshold` **then**
        $\mathcal{I}_{idx} \leftarrow$ compute the inverse covariance matrix of c
    end if
    update $\mathcal{N}$
    return $((i, j), d)$
end procedure

arrays: The first part contains the data of clusters whose size have not reached the Mahalanobis threshold while the second part stores the data of the remaining clusters. Hence, it contains a pair of indices indicating beginnings of the two parts and another pair describing the sizes of the parts. We will respectively call them *Euclidean* and *Mahalanobis array parts*. Note, the structure is uniform for each cluster-related array; meaning that a different array data of the same cluster resides on the same index.

When a cluster pair (i, j) is to be merged, the indexing structure first recognizes the parts of an array where the indices i, j belong. Hence, it identifies the places of arrays that are no longer valid. It gets rid of the places in the two particular ways:

- assigns the invalid place as the place for the new cluster,
- moves the end data of the corresponding part to the invalid place.

The reason why we organize the data in such way is to maintain continuously distributed data (see fig. 2.5). This data distribution serves a great help for running device kernels.

Remark. Naturally, there are other ways to maintain continuously distributed data within arrays. We can move the data in such way that the new cluster will always reside at the beginning of the corresponding part of an array. This approach could decrease the complexity of several kernels. On the other hand, it requires at most a linear time to move the data instead of a constant time.

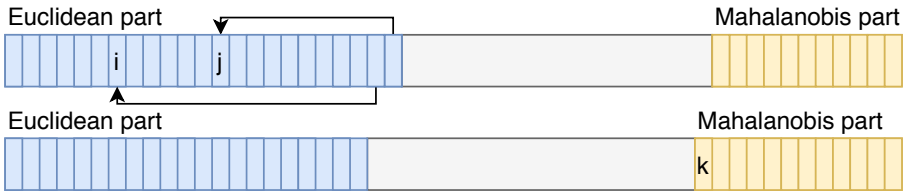

Figure 2.5: An example of the data distribution in a cluster-related array. Two clusters of non-Mahalanobis size with their indices i, j are being merged into a Mahalanobis-sized cluster on the index k.

2.5 CUDA Kernels

Kernels handle the computation critical parts of the algorithm. They are divided into several groups:

- *Clusters merging* updates the assignment array so it reflects the merging of two clusters.
- *Centroid computation* goes through the assignment array and computes the centroid of the newly formed cluster.
- *Inverse covariance matrix computation* is a set of kernels which compute the inverse of the covariance matrix of newly formed clusters that reached the Mahalanobis threshold.
- *Minimum retrieval* iterates over the neighbor array to find the closest two clusters.
- *Neighbor array update* updates the neighbor array structure with respect to the newly created cluster.

Each kernel group reads and modifies a different data array and has its own time complexity. Hence, to complete the kernel summary, we provide this information in the table 2.1. The following sections will describe each kernel group in more detail.

2.5.1 Clusters merging

The kernel `merge_clusters` is responsible for a merging of clusters. As the input, it takes the assignment array $\mathcal{A}$, a pair of old cluster IDs (of merged clusters)

Kernel group	Input arrays	Output arrays	Complexity
Clusters merging	$\mathcal{A}$	$\mathcal{A}$	$O(n)$
Centroid computation	$\mathcal{A}, \mathcal{P}$	$\mathcal{C}$	$O(n)$
Inverse covariance matrix computation	$\mathcal{A}, \mathcal{P}, \mathcal{C}$	$\mathcal{I}$	$O(n)$
Minimum retrieval	$\mathcal{N}$	–	$O(c)$
Neighbor array update	$\mathcal{C}, \mathcal{I}, \mathcal{N}$	$\mathcal{N}$	$O(c^2)$

Table 2.1: A summary of the kernel groups (n is a number of points in a clustering context; c is a number of clusters in a clustering context).

and a new ID (of a created cluster). It goes through the array in a *grid-stride loop* and replaces any of the old IDs with the new ID.

Definition 19. *A grid-stride loop iterates over all threads in the grid multiple times until the desired number of iterations is hit. The following code shows an example of a 1D grid stride loop. The first assignment creates an unique thread index across a grid. The index is added a grid size until the condition fails.*

```
for (auto idx = threadIdx.x + blockIdx.x * blockDim.x;
     idx < iters;
     idx += gridDim.x * blockDim.x);
```

2.5.2 Centroid computation

The kernel `centroid` computes the centroid of a cluster with a specified id. According to the id, the kernel filters the point array $\mathcal{P}$ using the assignment array $\mathcal{A}$ to find the involved points.

Each thread maintains its local array that stores the sum of the points that belong to the cluster.

First, a grid-stride loop is performed over $\mathcal{P}$. If a point belongs to the cluster (determined using $\mathcal{A}$ and id), the thread aggregates it in the sum array.

After the loop ends, the *shared memory* and *warp shuffle instructions* are utilized to reduce the sum arrays of all block threads into one array (we will call this the *block reduction*).

Each block reduced array is then divided by the cluster size and stored into $\mathcal{C}$ using the *atomic add instruction*.

2.5.3 Inverse covariance matrix computation

To compute the inverse covariance matrix of a cluster with an id, multiple kernels are run from the host method `compute_icov`.

First, the covariance matrix of the cluster has to be computed via kernels `covariance` and `finish_covariance`.

Then, the *CUBLAS*[4] library is used to invert the covariance matrix. If the matrix can not be inverted (due to the singular property) the inverse is set to the default value — identity.

Last, a special transformation is performed on the inverse covariance matrix and the result is stored to $\mathcal{I}$ via the `store_icovariance` kernel.

Covariance matrix computation

The first kernel `covariance` computes the upper triangular covariance matrix of a cluster multiplied by the cluster size.

To achieve a linear complexity in the number of points, we use the centroid C of the cluster (it is already computed by the previous kernel). Before the kernel starts, C is copied to the *constant memory* so all threads can use it.

At the kernel start, each thread uniformly selects a part of the *shared memory* as its intermediate upper-triangulate covariance matrix. As more threads can share the same part, the atomic access is necessary.

Then, a grid-stride loop over $\mathcal{P}$ with an id filtering follows. If a point belongs to the cluster, a thread subtracts it by C; we will denote the subtraction result with S. After that, each element i of S is separately multiplied with the element $j \geq i$ and atomically added to the corresponding place of the intermediate covariance matrix of the thread. The equation 2.1 describes the process. There, $\mathbf{X}$ denotes the discrete random variable of the x-coordinate of all cluster points, C_x denotes the x-coordinate of C and $\mathbf{S_x}$ denotes discrete random variable $\mathbf{X} - C_x$.

$$\begin{aligned} n\,\mathrm{cov}(\mathbf{X},\mathbf{Y}) &= n\mathbf{E}[(\mathbf{X}-\mathbf{E}(\mathbf{X}))(\mathbf{Y}-\mathbf{E}(\mathbf{Y}))] = n\mathbf{E}[(\mathbf{X}-C_x)(\mathbf{Y}-C_y)] \\ &= n\mathbf{E}[(\mathbf{S_x})(\mathbf{S_y})] = n\frac{1}{n}\sum_{i=1}^{n}\mathbf{S}_\mathbf{x}^i\mathbf{S}_\mathbf{y}^i = \sum_{i=1}^{n}\mathbf{S}_\mathbf{x}^i\mathbf{S}_\mathbf{y}^i \end{aligned} \tag{2.1}$$

After the loop, the *block sum reduction* is performed and the resulting covariances are stored to the *global memory* via the *atomic add instruction.*

[4]CUDA implementation of BLAS library

Finishing covariance matrix

Immediately next follows the kernel `finish_covariance`. It takes the result of the previous kernel as the input, it divides the matrix by the cluster size and fills the lower triangular part for the inversion operation.

We could incorporate the division operation into the previous kernel. However, it would be needed to be done by each thread performing the atomic operation. Here, each element is divided just once.

Storing inverse covariance matrix

After the inversion done by the CUBLAS subroutine `cublasSmatinvBatched`, the kernel `store_icovariance` performs the following actions with its result:

1. Transforms the matrix to the upper triangulate variant.
2. Multiplies the non-diagonal elements of the matrix by a factor 2.
3. Stores the transformed matrix into $\mathcal{I}$.

The first step is performed to get rid of redundant information and to save space as the $\mathcal{I}$ array has the greatest space complexity among the cluster-related arrays.

The second step is an optimization for the computation of the Mahalanobis distance equation (see eq. 7). We can allow this optimization as the distance is a *quadratic form* that can be simplified as shown in the equation 2.2.

$$\mathbf{x}^\top \mathbf{M}\mathbf{x} = \sum_{i=1}^{n}\sum_{j=1}^{n} m_{ij}x_i x_j = \sum_{i=1}^{n} m_{ii}x_i^2 + \sum_{i=1}^{n}\sum_{j>i}^{n} 2m_{ij}x_i x_j \tag{2.2}$$

Remark. The overall time complexity of the `compute_icov` function is $O(n)$ as each kernel needs to iterate over all points in a clustering context. We omitted complexity of computing a point covariance ($O(dim^2)$) and a matrix inversion ($O(dim^3)$) as the size of the point dimension is negligible compared to the size of the clustering context.

2.5.4 Minimum retrieval

The kernel `neighbor_min` finds the closest neighbor among all the neighbors in the neighbor array $\mathcal{N}$.

The loop over $\mathcal{N}$ is performed block-stride as the grid must consist of just one block. That is because there is no grid synchronization in CUDA; meaning

that threads can be synchronized only within the same block. In this task, we need to synchronize all involved threads.

Each thread iterates through the loop and stores its local closest neighbors (their distances and indices).

After the loop ends, the threads of the only block perform the block reduction achieving the globally closest neighbors.

2.5.5 Neighbor array update

The host method `update_neighbors` is a set of five kernels where each of them takes an important part in updating the neighbor array $\mathcal{N}$.

First, the kernel `update` collects invalidated indices. Next, two distinct kernels `neighbors_u` and `neighbors_mat_u` are run to update the array — the first one updates the Euclidean part while the second kernel updates the Mahalanobis part. Then, the reduction kernel `reduce_u` is called that reduces intermediate results generated by the previous two kernels. Last, the kernel `neighbors_new` performs a special update of $\mathcal{N}$ for the newly created clusters.

Apriori clustering layer uses the kernel set `neighbors`. This set is used for the initialization of $\mathcal{N}$; it is a modification of the currently described kernels. There, the `update` and `neighbors_new` kernels are ignored as all elements of the array have to be updated and none of them is created. The notation of kernel names suggests that kernels suffixed with `_u` have a suffix-less variant that processes all elements.

Update detection kernel

The kernel `update` checks the validity of the neighbors of each cluster using a grid-stride loop. The check consists of these steps:

- If the index of the closest neighbor belongs to a cluster that was merged, remove the neighbor.
- If the neighbor index belongs to a cluster that was moved during an array reorder, change the index accordingly.

When no neighbors are present, the particular cluster is marked for the update. The index of this cluster is stored in the *update array* $\mathcal{U}$. It is divided into two parts — same as for the other cluster-related arrays.

$\mathcal{U}$ to contain continuous indices of update-ready clusters, threads synchronize over the *global memory* variable indicating the next available location of $\mathcal{U}$. The variable is manipuled by threads using the *atomic add instruction*.

Euclidean update kernel

The `neighbor_u` kernel takes care of updating clusters whose size have not reached the Mahalanobis threshold (we call them *Euclidean clusters* for simplicity). For that reason, the input of the kernel is $\mathcal{C}$ (to compute distances) and $\mathcal{U}$ (to target clusters for the computation).

The first loop iterates through $\mathcal{U}$. All threads iterate the whole loop. Therefore, when a cluster c is selected from $\mathcal{U}$, all threads contribute to the computation of its closest neighbors.

For each $c \in \mathcal{U}$, threads perform a grid-stride loop over Euclidean clusters. Each thread takes care of one cluster at a time, computes its distance with c and updates its local closest neighbors of c.

Then, the threads perform the block reduction and create the block-local closest neighbors of c. As there is no grid synchronization, each block stores its result in the intermediate neighbor array $\mathcal{N}'$ ready for a further reduction.

The kernel ends when all clusters from $\mathcal{U}$ are visited.

Remark. As the Non-singular Mahalanobis distance function is symmetric, there is no need to compute a distance between all clusters and a cluster c (the one that needs to be updated). We only need to compute distances of clusters with their indices greater than the index of c. If we follow this invariant throughout the algorithm, we can safely ignore the clusters of lower index in the computation as during *theirs* computation the cluster c was already involved. This does not hold for newly created clusters. They need to be measured against all the remaining clusters to hold the invariant (see the kernel `neighbors_new`).

Mahalanobis update kernel

The `neighbor_mat_u` kernel is similar to the previous one. The main distinction is that this kernel updates clusters that reached the Mahalanobis threshold (*Mahalanobis clusters*, for simplicity) instead of those that did not. We created the separate kernel to tailor the thread utilization for the demanding Mahalanobis distance computation.

In addition to $\mathcal{C}$ and $\mathcal{U}$, the kernel takes $\mathcal{I}$ as the input. It is important to access the inverse covariance matrices because they are involved in a distance computation of Mahalanobis clusters.

Same as in the previous kernel, the first loop is performed by all threads to cooperatively compute the closest neighbors for each element of $\mathcal{U}$.

In contrast to the previous kernel, each warp takes care of one cluster at a time in the grid-stride loop instead of a thread. Warp threads cooperate together in the matrix-vector multiplications as the Mahalanobis distance function is now involved. The previous kernel does not perform a matrix-vector multiplication

for computing the distance so only one thread computes one distance measurement.

Finally, the kernel performs the same reduction as in the previous kernel and stores the block-local intermediate results into $\mathcal{N}'$.

Remark. The invariant of distance computation holds in this kernel as well. However, when a Mahalanobis cluster c needs to be updated, distances between c and *all* Euclidean clusters must be also computed regardless the index. This is a trade-off for having Euclidean and Mahalanobis clusters divided from themselves. However, the majority of Euclidean clusters tend to merge themselves early in the clustering.

Reduction kernel

The kernel `reduce_u` reduces the intermediate results of the updated clusters in $\mathcal{N}'$ into the neighbor array $\mathcal{N}$. For each updated cluster, there is as much closest neighbor entries as there were blocks in a grid of the previous kernels. As this may be a big number, one *warp* cooperates in the reduction of the entries for one updated cluster.

New cluster update kernel

The `neighbor_new` kernel updates $\mathcal{N}$ in the means of the newly created cluster c. It computes the remainder of the distances (between c and the clusters of lower index) to satisfy the mentioned invariant. Hence, this kernel moves that responsibility from the kernels `neighbor_u` and `neighbor_mat_u` to simplify their code and process the special measures faster. The kernel is called after the reduction kernel as it operates directly with $\mathcal{N}$.

Remark. The time complexity of the `update_neighbors` kernels is $O(c^2)$, where c denotes the number of clusters in a clustering context. This is due to the kernels that compute the closest neighbors. In the worst case, neighbors of each cluster need to be recomputed resulting in the mentioned time complexity. However, this is rarely the case. More often, the number of update-needed clusters resides in a yet unpredictable but reasonable scope. Hence, when we denote the number of clusters needed to be updated as u, we can start reasoning about the more precise time complexity $O(uc)$.

3. Results

The last chapter of the thesis describes the experiments that were performed to provide a clear result of the MHCA implementation performance. The aim of the thesis was to create the MHCA implementation that is capable of clustering datasets of hundred thousands and millions of points in a reasonable amount of time.

The results were compared with the existing implementation of Mahalanobis-average hierarchical clustering [8]. The implementation is written in R language with performance critical parts written in C. For linear algebra computation, it uses a BLAS library. It provides both FMD and CMD measures of dissimilarity and the apriori clusters computation as well. It performs a serial CPU computation. For brevity, we denote the thesis GPU implementation by `gmhclust` and the serial CPU version by `mhclust`.

The testing datasets were divided into these two categories:

Single-point datasets These datasets are clustered without a knowledge of any apriori clusters. Hence, the computing starts from a single point. The datasets are purely of a high-dimensional flow and mass cytometry data [9] (see tab. 3.1).

Apriori datasets These datasets are clustered with the apriori clustering method. They do not represent any measured data (alternatively, we could use single-point datasets pre-clustered with k-means). The apriori datasets contain small gaussian clusters; each of them is marked as an apriori cluster. The datasets contain lot of these clusters located on a different place.

The describing parameters of apriori datasets comprise *apriori cluster size*, the *apriori cluster count* and the *point dimensionality* (see tab. 3.2).

Dataset name	Size	Dimension
Niellson_rare	44K	14
Levine_13dim	167K	13
Levine_32dim	265K	32
Mosmann_rare	396K	15
Samusik_all	841K	39

Table 3.1: Flow and mass cytometry datasets selected for experiments.

Dataset name	Cluster size	Cluster count	Size	Dimension
apriori_1M	1000	1000	1M	20
apriori_2M	2000	1000	2M	20

Table 3.2: Apriori datasets generated for experiments.

3.0.1 Benchmarking setup

All experiments were run on the same machine (Intel Xeon Silver 4110, 256 GB RAM, NVIDIA Tesla V100 PCIe 16 GB). The performance-related experiments were performed five times to detect possible anomalies. As there were negligible performance differences in the separate runs, all the following plots depicting performance show their arithmetic means.

3.1 The closest neighbor parameter

To determine which clusters to merge, the GPU implementation uses the closest neighbor array. The number of the closest neighbors assigned to each cluster in the array is variable. We experimented on single-point and apriori datasets with this parameter to determine which number performs best.

We assumed that *two* closest neighbors will achieve the best performance as it will result in the reduced number of cluster to be updated each iteration. *Three* and more neighbors will not provide such reduction that would compensate the diminishing returns caused by the longer time for a cluster update.

As seen from the figure 3.1, the variant of 1 closest neighbor achieved the best performance. This may be caused by the fact that the lower number of cluster updates did not overcome the increased time for each update. Alternatively, it may be caused by the composition of tested datasets; datasets that naturally form cluster of other than elliptical shape may benefit better from another closest neighbor number variant.

The forthcoming experiments will be performed with the 1 closest neighbor variant.

3.2 Clustering speedup

The following set of experiments determines the performance speedup of the GPU implementation over the CPU implementation. The performance comparison was performed on single-point and apriori datasets separately.

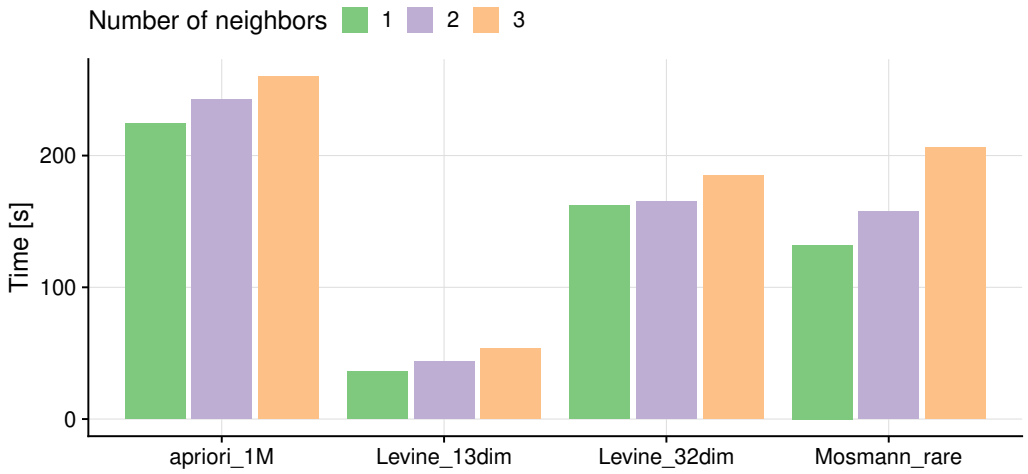

Figure 3.1: A comparison of `gmhclust` with various numbers of neighbors assigned to clusters.

Unfortunately, during the single-point testing, we reached the computational limits of the CPU implementation. Even for the smallest dataset, the computation took nearly 14 hours. Due to the polynomial time complexity and extremely increasing memory requirements (193GB for Levine_32), we decided that in order to test the datasets, we need to down-sample them. We uniformly randomly selected a fraction of the dataset points to create a smaller dataset. Note that these consequences are necessary (yet unforeseen) as the larger datasets require such big amount of time and space resources that they are by no means practically computable (see fig. 3.2).

3.2.1 Speedup on single-point datasets

To determine the performance increase, we decided to down-sample Nillson_rare dataset (as it is the smallest one). We created 10 samples with their sizes corresponding to 10% of the dataset up to 100%. As seen from the figure 3.3, the performance difference increases as the dataset size increases. It is a common consequence of the fact that there is less work to parallelize in smaller datasets than in bigger datasets. In the dataset of size around 4K, the performance of `gmhclust` is 60-times greater. This scales up to 5000-times faster time for the full Nilsson_rare dataset.

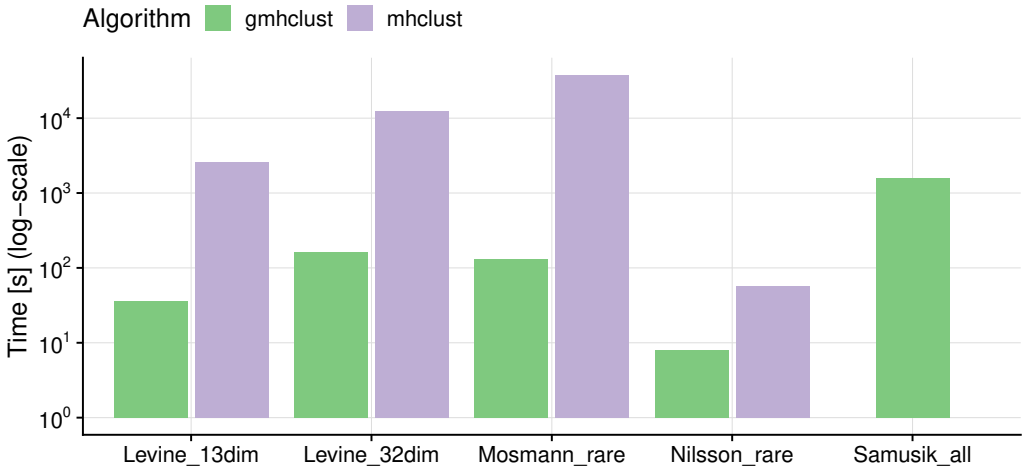

Figure 3.2: An example of better performance of `gmhclust` compared with `mhclust` on single-point datasets. Datasets of `mhclust` were down-sampled to the fraction of $\frac{1}{10}$ while `gmhclust` datasets were not changed. For Samusik_all dataset, neither the fraction of $\frac{1}{10}$ did help to finish the computation within a 24 hours mark.

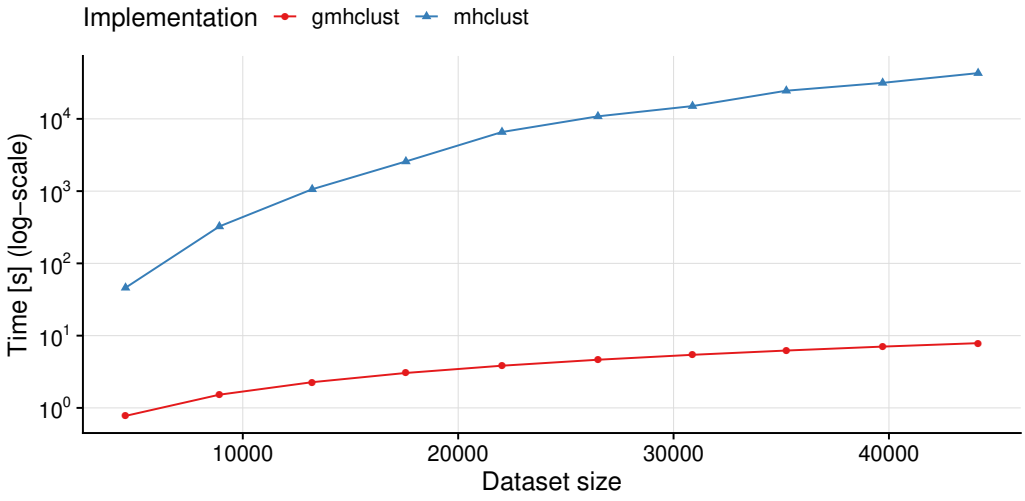

Figure 3.3: A comparison of `gmhclust` and `mhclust` performance on various sizes of Nilsson_rare single-point dataset.

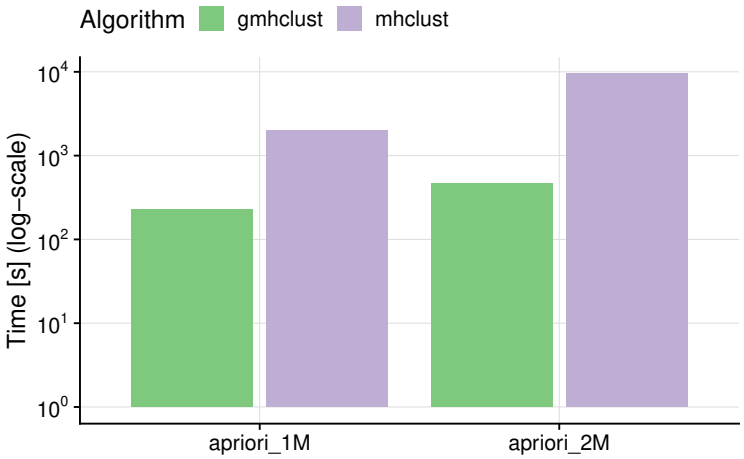

Figure 3.4: A comparison of `gmhclust` and `mhclust` on apriori datasets.

3.2.2 Speedup on apriori datasets

In contrast to the single-point datasets, the CPU implementation managed to compute apriori datasets of big sizes. It is a consequence of the wisely chosen apriori clusters.

When the algorithm is inputed with apriori clusters, it does not compute the whole dataset at once — it clusters each apriori cluster first. This fact reduces the overall time complexity significantly, allowing the CPU algorithm to compute big datasets in a short amount of time.

For example, suppose the input dataset apriori_2M and a function f that takes a dataset size and outputs the required time for its clustering. Then, the algorithm performance is proportional to

$$1000f(2000) + f(1000)$$

that is — deducing from the previous experiment — far less than $f(1000 \cdot 2000)$.

We measured the performance of the CPU and GPU implementation on apriori_1M and apriori_2M (see fig. 3.4). The results of the performance increase are 8-times and 20-times respectively. As we expected, it is not as measurable as in single-point datasets even though it has much more points.

It is caused by computing a lot of small apriori clusters. From the previous experiment we observed that small-sized datasets do not exhibit such big increase in performance. Therefore, the overall performance increase in apriori datasets is not expected to be drastically better.

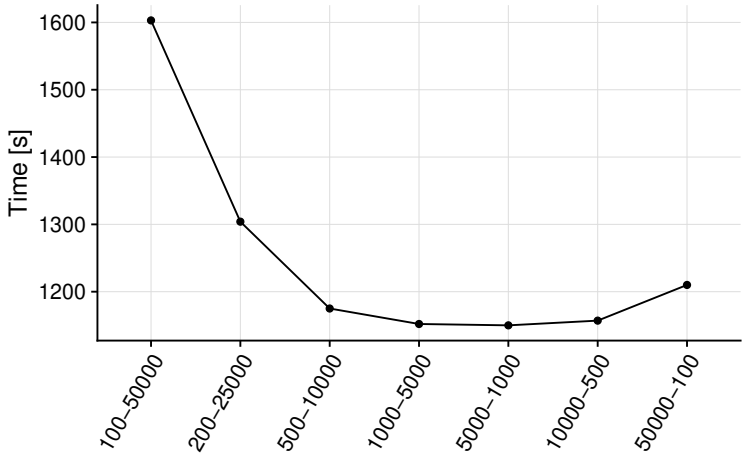

Figure 3.5: `gmhclust` performance for different ratios of apriori cluster size and count (depicted as $size$-$count$) with Mahalanobis threshold set to $size$.

Apriori dataset parameters

The next experiment tests how the apriori dataset parameters *size* and *count* influence the overall algorithm time. We tested 7 different size–count ratios on a dataset of 5M 20-dimensional points (see fig. 3.5). The Mahalanobis threshold of each run was set to be equal to the apriori cluster size.

The runs are ordered in such way that the apriori cluster size is increasing (and the apriori cluster count is decreasing). The plotted line first rapidly decreases, then is close to constant and starts to increase in the end. This may be caused by two factors:

First, according to the previous experiment, we expected the following relation: the bigger is the maximum of the size and count parameters, the bigger is the overall computational time. Hence, big measured values are seen on the edges of the plot.

The second factor may be the Mahalanobis threshold. As the threshold increases with the increasing cluster size, more clustering work is performed using the less demanding euclidean distance. The threshold was set to be equal to the apriori cluster size; hence, the Mahalanobis distance is used after all apriori clusters are clustered. As the consequence, the right edge of the plot is not as steep as the left edge.

3.3 Results comparison

Next, we will compare the clustering results of the GPU and CPU implementation to see if they are similar. Comparing results for equality would put big restrains on the implementation. Rather, we want to allow algorithms to have slight differences during their clustering and compare the whole clustering process. First, we need to define the *point height*.

Definition 20 (Point height)**.** *Suppose a dataset $\mathcal{D} \subset \mathbb{R}^d$ and an ordered sequence of cluster pairs $p_1, \ldots, p_{n-1}$ that defines a result $\mathcal{R}$ of a hierarchical clustering on $\mathcal{D}$. We define the* height of point $o \in \mathcal{D}$ *in $\mathcal{R}$ as the sum of distances between all cluster pairs p_i such that o belongs to their union.*

We can easily imagine point height using a dendrogram. The point height represents the distance from the bottom of the dendrogram (starting in the corresponding point) to the top.

To compare two clusterings, we compute the height of each point in both clusterings. Then we take the two distances of each point as its coordinates and plot a scatterplot. When the correlation between point heights of two clusterings is equal to 1, clusterings formed very similar clusters. The scatterplot shows this as a straight increasing line. When different clusters were created, points are scattered among the whole plot.

An alternative approach for comparing hierarchical clusterings would be to compare a selected parts of their dendrograms. Dendrograms are *cut* to expose a small amount of clusters (i.e. 10). Then, they are directly compared using similarity measures like the F-score.

However, the MHCA algorithm tends to create small clusters that are propagated to the very top of the dendrogram. They usually consist of noise because Mahalanobis distance makes noise clusters dissimilar to the rest and they are merged in the late clustering stages. This can be a problem for the direct cluster-by-cluster comparison as these noise clusters can easily skew the results. In compared clusterings, they may be composed of different points or form different amount of cluster; all this can worsen the result of potentially similar clusterings.

3.3.1 Single-point dataset results comparison

We down-sampled four selected single-point datasets to the fraction of $\frac{1}{10}$. Then, we computed heights of each point and plotted them as discussed above (see fig. 3.6). The figure shows, that that GPU implementation chose different paths during its clustering process creating a different result. However, we can see that

the points reside mainly on the diagonal; hence, the compared clusterings may created a lot of clusters that shared the majority of their points.

The possible reason of this behavior may be a slight difference in the measure of dissimilarity of the CPU implementation. There, small clusters are assigned an artificial covariance matrix so Mahalanobis distance can be applied on them. However, the GPU implementation computes only the euclidean distance. This can be a reason for branching of the algorithms in theirs early stages. As a consequence, the GPU and CPU clustering results differ slightly.

3.3.2 Apriori dataset results comparison

We did not down-sample the selected apriori datasets and proceeded the same way as in the previous experiment (see fig. 3.7). Here, the experiments showed the almost straight line. It was expected, as the presence of apriori clusters effectively removed a possible error created during the early algorithm stages where the implementations differed. More specifically, the propagation of possible clustering differences was stopped when the apriori clusters were completely clustered (as the clusterings aligned at that point).

When all apriori clusters were processed, the GPU and CPU measures of dissimilarity applied on the remaining clusters did not differ as the clusters were big enough. Hence, no other opportunity for possible branching occurred. This experiment also shows the importance of clustering early stages and how it can affect the whole result.

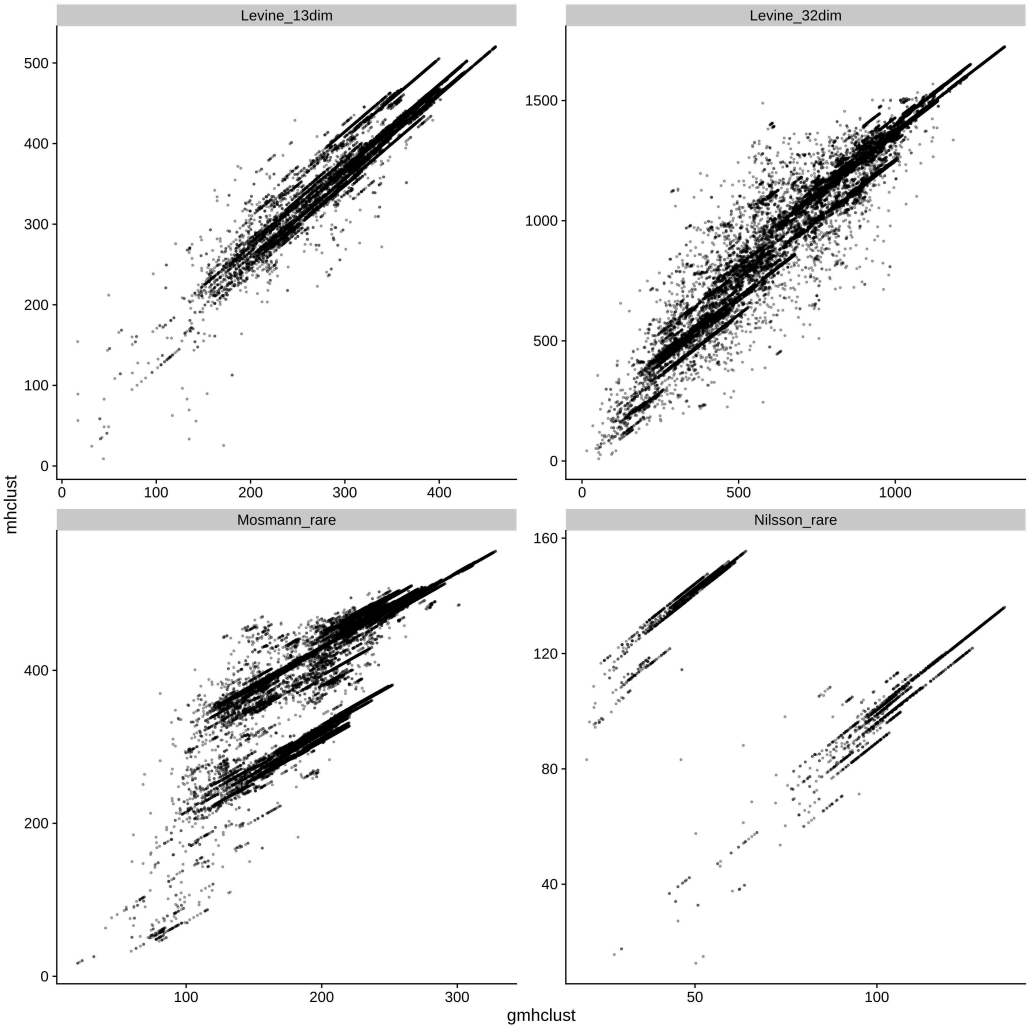

Figure 3.6: Plotted point heights of `gmhclust` and `mhclust` clusterings of the selected single-point datasets. The Mahalanobis threshold was set to default ($\frac{1}{2}$).

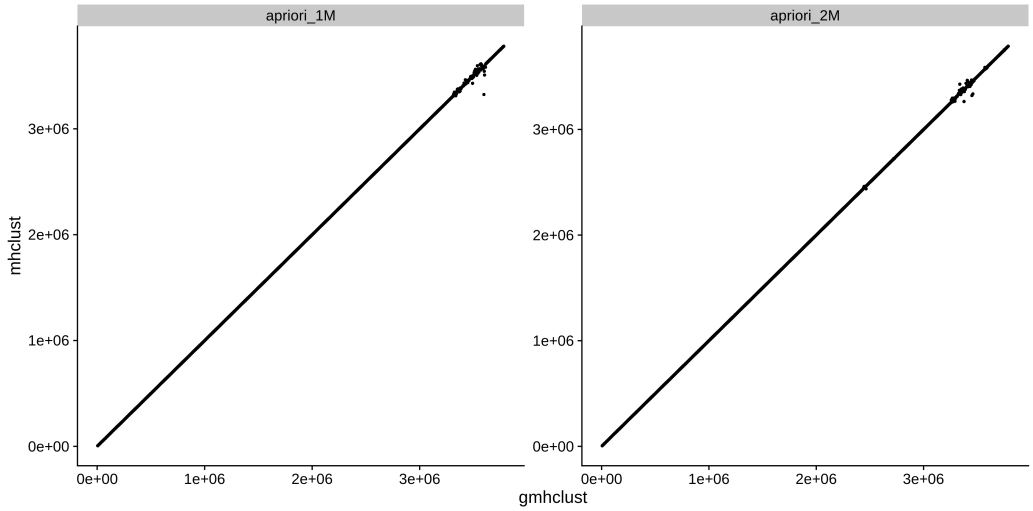

Figure 3.7: Plotted point heights of `gmhclust` and `mhclust` clusterings of the selected apriori datasets. The Mahalanobis threshold was set to apriori cluster size. The results correlate almost perfectly.

Conclusion

In this thesis we have researched possible variations of agglomerative hierarchical clustering algorithms (section 1.3) and selected the variant that promised high GPU utilization and low memory requirements (section 2.2). Then, we have implemented the Mahalanobis-average hierarchical clustering accelerated on a single GPU (section 2.4). Finally, we have performed several experiments to measure the properties of the resulting implementation (chapter 3).

The resulting application is implemented within CUDA framework, which should make it useful for scientific computing where CUDA is a de-facto standard. The implementation uses the Centroid Mahalanobis distance (see def. 9) as the dissimilarity measure, and it can cluster datasets from single-points or with a help of apriori clusters. We have tested the implementation on single-point datasets with sizes from 40K to 800K points and apriori datasets with sizes from 1M to 5M points. Due to the big time and memory requirements of the CPU implementation, datasets were down-sampled. For single-point datasets, the speedup was 60-times for 4K points and 5000-times for 40K. For apriori datasets, the speedup was 8-times for 1M points and 20-times for 2M points. We have also tested the clustering performance of different apriori cluster size and count ratios. We have discovered that clustering time increases faster with higher apriori cluster count than with higher apriori cluster size.

Regarding the comparison of clustering results, the clusterings differed for single-point datasets due to the slightly different dissimilarity measures. On the other hand, in case of more frequently used apriori clusters, the clusterings were practically the same.

Future work

Although the implementation in this thesis is useful and provides good results, there are many minor improvements that can be added. The main concern of the future work is to implement the remainder of the functionalities of the current MHCA algorithm.

Most importantly, the dissimilarity measure that can provide better clustering of small clusters can be implemented. We can transform a covariance matrix of these clusters to make it regular. Then it can be properly inverted and used in the Mahalanobis distance formula.

Next, a covariance matrix can be normalized by dividing each element by its discriminant. When used in the distance formula, this transformation outputs the combination of the Mahalanobis and Euclidean distance. The determinant can be computed as a side product of the Cholesky decomposition that can be

used to compute the inverse of a covariance matrix.

Although we do not expect it to provide any substantial improvement on reasonable datasets, the work can be further expanded by implementing the Full Mahalanobis distance (see def. 8). It can cover the cases when CMD does not provide precise dissimilarity measure which can lead to a better clustering results. Naturally, it would come for the price of a greater overall time complexity.

Bibliography

[1] Charu C Aggarwal, Alexander Hinneburg, and Daniel A Keim. "On the surprising behavior of distance metrics in high dimensional space". In: *International conference on database theory*. Springer. 2001, pp. 420–434.

[2] Daniel Aloise et al. "NP-hardness of Euclidean sum-of-squares clustering". In: *Machine learning* 75.2 (2009), pp. 245–248.

[3] Guilherme Andrade et al. "G-dbscan: A gpu accelerated algorithm for density-based clustering". In: *Procedia Computer Science* 18 (2013), pp. 369–378.

[4] *CUDA C++ Programming Guide*. URL: `https://docs.nvidia.com/cuda/cuda-c-programming-guide` (visited on 04/17/2020).

[5] P Dagnelie and A Merckx. "Using generalized distances in classification of groups". In: *Biometrical journal* 33.6 (1991), pp. 683–695.

[6] William HE Day and Herbert Edelsbrunner. "Efficient algorithms for agglomerative hierarchical clustering methods". In: *Journal of classification* 1.1 (1984), pp. 7–24.

[7] Vladimir Estivill-Castro. "Why so many clustering algorithms: a position paper". In: *ACM SIGKDD explorations newsletter* 4.1 (2002).

[8] Karel Fišer et al. "Detection and monitoring of normal and leukemic cell populations with hierarchical clustering of flow cytometry data". In: *Cytometry Part A* 81.1 (2012), pp. 25–34.

[9] *Flow Repository*. May 3, 2020. URL: `https://flowrepository.org/id/FR-FCM-ZZPH`.

[10] Michael L Fredman and Robert Endre Tarjan. "Fibonacci heaps and their uses in improved network optimization algorithms". In: *Journal of the ACM (JACM)* 34.3 (1987), pp. 596–615.

[11] Guan-Jie Hua et al. "MGUPGMA: A Fast UPGMA Algorithm With Multiple Graphics Processing Units Using NCCL". In: *Evolutionary Bioinformatics* 13 (2017).

[12] George Karypis, Eui-Hong Han, and Vipin Kumar. "Chameleon: Hierarchical clustering using dynamic modeling". In: *Computer* 32.8 (1999), pp. 68–75.

[13] Godfrey N Lance and William Thomas Williams. "A general theory of classificatory sorting strategies: 1. Hierarchical systems". In: *The computer journal* 9.4 (1967), pp. 373–380.

[14] Godfrey N Lance and William Thomas Williams. "A general theory of classificatory sorting strategies: II. Clustering systems". In: *The computer journal* 10.3 (1967), pp. 271–277.

[15] Jacob H Levine et al. "Data-driven phenotypic dissection of AML reveals progenitor-like cells that correlate with prognosis". In: *Cell* 162.1 (2015), pp. 184–197.

[16] Prasanta Chandra Mahalanobis. "On the generalized distance in statistics". In: National Institute of Science of India. 1936.

[17] Fionn Murtagh, Geoff Downs, and Pedro Contreras. "Hierarchical clustering of massive, high dimensional data sets by exploiting ultrametric embedding". In: *SIAM Journal on Scientific Computing* 30.2 (2008), pp. 707–730.

[18] Seung-Joon Oh and Jae-Yearn Kim. "A hierarchical clustering algorithm for categorical sequence data". In: *Information processing letters* 91.3 (2004), pp. 135–140.

[19] Peng Qiu et al. "Extracting a cellular hierarchy from high-dimensional cytometry data with SPADE". In: *Nature biotechnology* 29.10 (2011), p. 886.

[20] Lior Rokach and Oded Maimon. "Clustering methods". In: *Data mining and knowledge discovery handbook*. Springer, 2005, pp. 321–352.

[21] K Sasirekha and P Baby. "Agglomerative hierarchical clustering algorithm-a". In: *International Journal of Scientific and Research Publications* 83 (2013), p. 83.

[22] Howard M Shapiro. *Practical flow cytometry*. John Wiley & Sons, 2005.

[23] Santosh Kumar Uppada. "Centroid based clustering algorithms—A clarion study". In: *Int. J. Comput. Sci. Inf. Technol* 5.6 (2014), pp. 7309–7313.

[24] Sofie Van Gassen et al. "FlowSOM: Using self-organizing maps for visualization and interpretation of cytometry data". In: *Cytometry Part A* 87.7 (2015), pp. 636–645.

[25] Xiaoxin Ye and Joshua WK Ho. "Ultrafast clustering of single-cell flow cytometry data using FlowGrid". In: *BMC systems biology* 13.2 (2019), p. 35.

[26] Odilia Yim and Kylee T Ramdeen. "Hierarchical cluster analysis: comparison of three linkage measures and application to psychological data". In: *The quantitative methods for psychology* 11.1 (2015), pp. 8–21.

[27] Ying Zhao, George Karypis, and Usama Fayyad. "Hierarchical clustering algorithms for document datasets". In: *Data mining and knowledge discovery* 10.2 (2005), pp. 141–168.

A. User guide

A.1 Build guide

The Mahalanobis-average hierarchical clustering project was developed with the CMake build tool. To build the executable, use CMake configure and build commands in a build directory. Then, the directory `para` will contain `gmhclust` executable. The only dependency is the CUDA compiler (`nvcc`). The executable should be portable to all platforms supporting `nvcc`; it was successfully tested on Ubuntu 18.04 and Windows 10. See the following steps:

```
cd gmhc
mkdir build && cd build
cmake ..
cmake --build .
ls para/gmhclust
```

A.2 Running the program

The `gmhclust` executable has three command line parameters:

1. *Dataset file path* – The mandatory parameter with a path to a dataset file. The file is binary and has structure as follows:
   (a) 4B unsigned integer D – point *dimension*
   (b) 4B unsigned integer N – *number* of points
   (c) $N \cdot D$ 4B floats – N single-precision D-dimensional points stored one after another

2. *Mahalanobis threshold* – An absolute positive number that states the Mahalanobis threshold. It is the mandatory parameter.

3. *Apriori assignments file path* – An optional path to an apriori assignments file — a file with space separated 4B unsigned integers (assignment numbers). The number of integers is the same as the number of points in the dataset; it sequentially assigns each point in the dataset file an assignment number. Then simply, if the i-th and the j-th assignment numbers are equal, then the i-th and j-th points are assigned the same apriori cluster.

The executable writes the clustering process to the standard output. Each line contains an ID pair of merged clusters with their merge distance as well.

The command, that executes the program `gmhclust` to cluster `data` dataset with the apriori assignment file `asgns` and the threshold 100 is

```
./gmhclust data 100 asgns
```

A more complex examples can be seen in the `benchmark` directory of the enclosed CD. See `README.txt` for the complete guide.

B. Enclosed CD

The enclosed CD contais three folders:

- `gmhc` contains the source code of the implementation.
- `benchmark` contains scripts and a guide how to reproduce some of the performed experiments.
- `docs` contains the program documentation.