FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

DEPARTMENT OF COMPUTER SCIENCE

Master's thesis

Unsupervised Clustering of Sparse Data in Futhark

Till Severin Grenzdörffer <vmt184@alumni.ku.dk>

Supervisors
Cosmin Eugen Oancea <cosmin.oancea@di.ku.dk>
Rory Mitchell <rorym@nvidia.com>

Abstract

K-means is a basic building block of modern machine learning. As such, its performance has a critical impact on workflows and explorations it is involved in. In this thesis, we focus on application domains that involve large sparse datasets and investigate the feasibility of the Futhark programming language to map k-means and its generalization, mixture models, to efficient GPU code. We propose a framework that abstracts from the (possibly sparse) representation of the data while maintaining the efficiency of sparse representations, where they are used. We demonstrate, that the implementation of k-means, spherical k-means, Gaussian mixture models and von Mises-Fisher mixture models through our framework is possible without the need to explicitly address the underlying data representation. Our k-means implementation yields performance speedups of at least factor 10 over the multicore CPU implementation of scikit-learn and our implementation of Gaussian mixture models with diagonal covariance matrices achieves a speedup of factor 1893 over a single-core CPU implementation that does not support sparse data representations.

Contents

Chapter 1

Introduction

Unsupervised clustering is an important tool in the field of modern machine learning. It is used whenever an unlabeled set of samples needs to be divided into subgroups based on the features of the samples. In a simple case, this could be the physical location of 3-dimensional points. One of the common algorithms for unsupervised clustering is k-means, which takes a set of n d-dimensional samples as well as an integer k as input. The algorithm then finds k cluster centers, such that the cumulative distances from every point to their nearest cluster are minimized. This can be achieved by iteratively executing two steps:

1. Assigning points to their nearest cluster. For this we need to compute the distances from all points to all cluster centers and select the cluster with the smallest distance for every point.

2. Calculating the centroid of each cluster. The centroid is computed by summing up the data points assigned to each cluster and then dividing this sum by the number of points assigned to the respective cluster.

Similarly, mixture models can be used for unsupervised clustering. Instead of representing clusters by their centers, here clusters are represented by probability distributions and data are classified by calculating the likelihood of the data being a sample of each distribution. This distribution could be a Gaussian distribution, but other probability distributions, for example von Mises-Fisher distributions [1] can also be used. Expressing the clusters in this way allows to describe the shapes of the clusters, whereas k-means only describes the location of the clusters.

Modifications of k-means and mixture models are used in many applications that range from data mining [2], to feature extraction for other learning algorithms [3], to classification of unseen data [4]. Since these algorithms are such widely used and versatile tools, and the datasets involved often contain many samples, accelerating them through the use of GPUs will be a benefit to their users and will enable more efficient workflows and learning pipelines.

As discussed in [1], there are several application domains of unsupervised clustering, like document classification or gene expression analysis that deal with high-dimensional and highly sparse data. In this type of data, most entries are zeroes and only relatively few entries contain nonzero values. For

example, in document clustering the bag of words model can be used to represent documents. A document is represented as a vector where each entry indicates the number of occurrences of one word from a fixed dictionary. Since most documents only use a fraction of all words in this dictionary, a lot of entries have count 0.

Implementing algorithms that work on these types of data for GPUs poses a challenge because of the highly irregular fashion in which sparse data is usually represented, since GPUs generally favor regular data layouts. This thesis explores the performance characteristics of different sparse representations and implementations on the example of the classical (unmodified) k-means algorithm on the GPU. For this purpose, the functional and data-parallel array language Futhark [5] is used, as well as the CUDA programming language for experiments involving lower level constructs that are not supported by Futhark.

Our experiments show that for the distance computation step of k-means the CSR format yields the best performance, whereas for the reduction step the COO format is better suited. For higher k values, sorting the data points by their cluster assignments yields performance benefits. Our implementation that uses a combination of the two formats achieves speedups of at least factor 10 over a multicore CPU implementation on our test datasets.

We implement an orthogonal framework that abstracts from the sparse data representation and allows the implementation of various mixture models. We choose the interface for the data representation such, that it can accommodate different sparse or dense data representations, while providing operations necessary to fit different mixture models. Since k-means and also Gaussian mixture models are semantically unsuitable for high-dimensional data [1], this generalization is important. We show, that the implementation of more suitable models like spherical k-means [6] and mixtures of von Mises-Fisher distributions [1] is possible through our framework, without the need to explicitly interact with the data representation of the input data.

Our implementation of the Gaussian mixture model[1] compiles to efficient GPU code and shows significant speedups of up to x1893 over a single-core CPU implementation that does not provide support for sparse input data.

We also provide an intuition into the mathematical background of mixture models and show how our framework is used to translate the mathematical formulations into parallelizable code.

1.1 Related work

Since both k-means and mixture models are widely known algorithms, there exist many implementations of them, both on CPU and on GPU. The well-known python library scikit-learn [7] provides CPU implementations both for k-means and Gaussian (as well as Bayesian) mixture models.

Another python library, cuML [8], provides GPU implementations for multiple clustering algorithms including k-means. Though cuML generally supports sparse inputs, its k-means implementation does not (with good reason). It also does not support mixture models as of the time of writing this thesis.

[1] With the constraint that the covariance matrix is diagonal

There are multiple implementations of mixture models that utilize the GPU hardware to accelerate the computation with great success [9] [10], however, they also do not take sparse input data into account.

Of these implementations, only the k-means implementation of scikit-learn supports input in sparse format and to my knowledge, no library exists that supports sparse data formats as input to GPU-accelerated k-means or mixture models.

However, computing sparse linear algebra operations efficiently on the GPU is not an unexplored topic. There have been numerous works concerning sparse matrix-dense vector multiplication. For example, [11] [12] explore the efficiency of different sparse formats, while [13] and [14] use autotuning and load balancing strategies to further accelerate the operations.

In a more general approach, [15] proposes a GPU-accelerated semiring primitive, which allows the efficient implementation of common distance measures for sparse data. This semiring primitive can be used for the main computation of the k-means algorithm, as well as for the evaluation of multivariate Gaussian and von Mises-Fisher distributions, and likely many more.

A distinction has to be made for the term *sparse k-means*, which refers to a modification of the k-means algorithm that selects a subset of the features (i.e. dimensions) for each sample [16]. This approach has also been implemented on the GPU [17]. In this thesis, however, sparsity refers to the input data containing many redundant entries with value 0 and the representations in which these kinds of data are typically stored.

1.2 Structure of the thesis

This thesis consists of three main parts corresponding to chapters 2-4, as well as this introduction and a conclusive section.

Chapter 2 presents background material that the other chapters of the thesis refer to. First, the classical k-means algorithm is presented. Then, general mixture models and the EM (EM) framework are introduced and a connection to k-means is established. Afterwards, there is a section covering the topic of high dimensional sparse data and its representations and finally, the Futhark programming language is briefly introduced.

In the third chapter, explorations on the problem of handling sparse data representations efficiently on the GPU are presented. These explorations focus on the classical k-means algorithm and propose an efficient implementation of this algorithm in Futhark. The different approaches are evaluated through benchmarks.

The fourth chapter covers the generalization of k-means, i.e. mixture models. A framework is presented which utilizes the findings of chapter 3 to allow efficient implementations of mixture models for sparse datasets.

Chapter 2

Background

This chapter presents concepts that are used throughout my thesis.

2.1 Futhark

Futhark is a purely functional programming language that compiles to highly optimized parallel code. It enables its users to write high-level code through array operations but still provides similar performance to low level implementations.

The language builds upon three basic building blocks: `map`, `reduce` and `scan`. Each of them are applied to arrays and execute highly parallel over the entries of the array.

`map` applies an arbitrary function to each element of an array. For example, we can square the values of an array (named `values`) with the following code snippet:

```
let values_squared = map (\x -> x*x) values
```

The `reduce` function takes an associative operator and the corresponding neutral element (i.e. a monoid) as arguments, as well as the array this operator should be applied to. This function then reduces the array by applying the operator across its element. For example, computing the sum over an array can be done in the following way:

```
let value_sum = reduce (+) 0 values
```

Here, $(+)$ is the associative operator *addition* and 0 the corresponding neutral element.

`scan` takes the same arguments as `reduce`, however, it computes the inclusive scan (also known as prefix sum) [18] of the array. This building block is commonly used in highly parallel implementations, for example in the implementation of radix sort [19]. A simple example is shown below:

```
let values = [1,4,2,3,4]
in scan (+) 0 values
```

The result of this code snippet is the array $[1, 5, 7, 10, 14]$.

All of these operations can be arbitrarily nested and Futhark applies incremental flattening [20] to produce multiple, semantically equivalent code versions, which incrementally utilize more and more levels of the application's nested parallelism. These semantically equivalent code versions are discriminated by placing them inside branches whose conditions compare (static) threshold values with the amount of parallelism exploited by that version. The threshold values are found by autotuning the program on a set of representative datasets [21], which has the potential to near-optimally map future datasets to their most suitable code version. As an example, an implementation of the dense matrix vector product can be written as:

```
let mv_prod [m][n] (mat: [m][n]f32) (vec: [n]f32): [m]f32 =
    map (\row ->
            reduce (+) 0
                (map2 (\a b -> a * b) row vec
        ) mat
```

The function `mv_prod` takes a 2d-array of 32 bit floats in the shape m x n and an array of 32 bit floats of length n as input arguments and produces an array of 32 bit floats of length m.

Applying incremental flattening can configure different levels of parallelism depending on the input data and the hardware being used. For example, it might be the case that the outer `map` already exhausts all available parallelism, and it is preferable that the code inside it is sequentialized. This configuration is tuned through the `futhark autotune` [21] feature.

Throughout this thesis another Futhark primitive, `reduce_by_index`, is used extensively. This function has the following signature:

```
val reduce_by_index 'a [m][n]:
            (dest: *[m]a) ->
            (f: a -> a -> a) ->
            (ne: a) -> --the neutral element of f
            (is: [n]i64) ->
            (as: [n]a) ->
                *[m]a
```

`reduce_by_index` implements the generalized histogram described in [22]. More specifically, this function modifies the array `dest` by updating all positions indicated by `is` with `f` applied to the current value of `dest` and the value in `as` that corresponds to the position in `is`. This means, that all values in `as` that have the same index in the corresponding `is` array are reduced with `f` into the same "bin". For simple addition, this is visualized in figure 2.1. In this figure, the colors indicate the index values in the `is` array, i.e. `as[0] = 2` and `is[0] = 1`, and `dest` is assumed to be an array of zeroes.

Futhark module system

Since we use the Futhark module system to achieve generality when implementing the mixture framework, we briefly introduce the key concepts of this system.

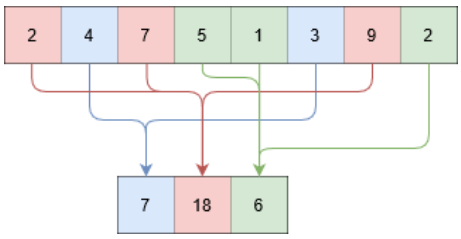

Figure 2.1: Visualization of `reduce_by_index`.

A Futhark module is a collection of types, functions, other modules and module types. It can be used to encapsulate functionality by providing type definitions and operations that work on these definitions.

For example, we can define a type to encapsulate operations on complex numbers as follows:

```
module complex = {
    type t = (f32, f32)

    let mk_complex (a: f32) (b: f32): t = (a, b)
    let get_real (number: t): f32 = number.0
    let get_imaginary (number: t): f32 = number.1
    let add (a: t) (b: t) =
            (a.0 + b.0, a.1 + b.1)
    ...
}
```

Here, we represent the complex number as a tuple of 32-bit floats. In Futhark, we can access the first element of a tuple `tup` by writing `tup.0`, the second element by writing `tup.1`, and so on. In our code we can now use this module as follows:

```
let test (x: f32)=
    let a = complex.mk_complex x 6
    let b = complex.mk_complex 3 5
    let c = complex.add a b
    in complex.get_real c
```

In this code snippet, we create two complex numbers and add them together through the `complex` module.

Now we might not always want to represent complex numbers through 32-bit floats. For example, when precision is an important factor, we might want to use 64-bit representations instead. In order to provide this abstraction without the need to manually replace all types in the `complex` modules, we can use parametric modules.

A parametric module takes another module as an argument and can then use the types defined therein. To parametrize our `complex` module with respect to the float representation, we can change the definition slightly:

```
module complex (R: real) = {
    type t = (R.t, R.t)
    let mk_complex (a: f32) (b: f32): t = (R.f32 a, R.f32 b)
    ...
    let add (a: t) (b: t) =
        (a.0 R.+ b.0, a.1 R.+ b.1)
}
```

This means, we take any module that implements the module type `real` [1] as input, and we can call functions that are defined in this module type, like `R.+` for addition and `R.f32` for conversion *from* a 32-bit float representation.

We can then instantiate the complex module with the module `f64`, which implements `real`, and use it as follows:

```
module complex64 = complex f64
let test (x: f32) =
    let a = complex64.mk_complex x 6
    ...
```

Now we might want to use this type in a function that is generic to the type of number it uses. For example, we could have a module that defines a function to add two numbers together, but does not specify whether they are real or complex numbers. We call this module `abstract_adder`. For this case, we can define a module type `addable`, that acts as an interface to abstract from the specification of the number.

```
module type addable = {
    type t
    val add: t -> t -> t
}
module abstract_adder (A: addable) =
{
    let add (a: A.t) (b: A.t) = A.add a b
}
```

`addable` requires the definition of a type `t` and a function `add`, which takes two arguments of type `t` and returns the sum. Since the `complex` module defines the necessary type and function, we can ascribe the module type `addable` to it. Finally, we can use the abstract adder on complex numbers as follows:

```
module complex32 = complex f32 -- parametrize complex module
module complex_addable = complex32 : addable -- type ascription
module complex_adder = abstract_adder complex32 -- parametrize abstract_adder
let main (x: f32) =
    let a = complex32.mk_complex x 6
    let b = complex32.mk_complex 3 5
    in complex_adder.add a b
```

[1] This module type is part of Futhark and provides typical mathematical operations on real numbers (see the documentation).

2.2 K-means

K-means is one of the simplest clustering methods in the machine learning domain. It is also a fairly old method, with sources from the 1950s and 1960s already describing the idea [23] [24] [25].

The objective of k-means is to group a set of n d-dimensional points into k clusters such that the distances of the points within each cluster to their respective cluster center are minimized. More formally, the objective of k-means is to divide the points into k subsets S_h such that the intra-cluster distance

$$\sum_{h=1}^{k} \sum_{\boldsymbol{x} \in S_h} ||\boldsymbol{x} - \boldsymbol{\mu}_h||^2$$

is minimized. The $\boldsymbol{\mu}_h$ represent the cluster centers, i.e. the means of the points assigned to each cluster.

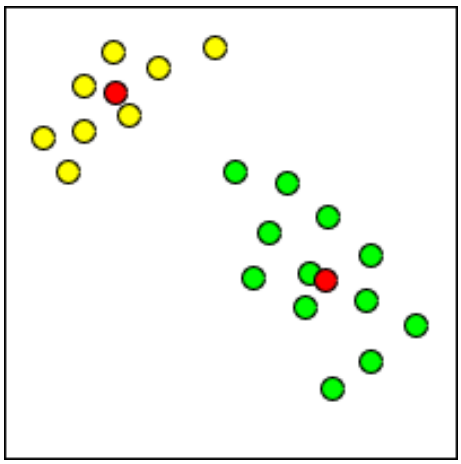

Figure 2.2: Visualization of 2-means.

Figure 2.2 shows, schematically, the result of 2-means being applied to a 2-dimensional dataset. The red dots indicate the position of cluster centers after convergence and the color of the other dots indicates which of the clusters they belong to.

The standard algorithm to solve this problem is often called Lloyd's algorithm, based on its formulation by Stuart Lloyd [26]. This algorithm is shown in Algorithm 1. After converging, a *local* minimum for the cluster centers is found. Because of this, running the algorithm several times with different initializations can have an impact on the outcome. There are initialization methods that can improve the results of k-means, like k-means++ [27]. This initialization method uniformly picks one cluster center from the set of points, calculates the distances of every point to this cluster center, and then samples more cluster centers from the points such that points are more likely to be sampled if their distance to the already picked cluster center(s) is large. This process is repeated until k cluster centers are sampled. We did not implement

this initialization method in our framework, still it is a simple but significant improvement that should be implemented in the future.

Another strategy to improve the results is running the algorithm multiple times with different random initializations and selecting the cluster centers that result in the lowest intra-cluster distance. This means, that the performance of the implementation also has an indirect impact on the quality of the results.

Algorithm 1 K-means

Input: $\boldsymbol{x} \in R^{n \times d}$ (a set of n d-dimensional points); $k \in \mathbb{Z}^+$ (the number of clusters)

Output: $\boldsymbol{\mu} \in R^{k \times d}$ (the cluster centers)

for $h = 0 \dots k$ **do**
  $\boldsymbol{\mu}_h \leftarrow init$ ▷ Some initialization for cluster centers
end for
$converged \leftarrow false$
while $!converged$ **do**
  for $i = 0 \dots n$ **do**
    for $h = 0 \dots k$ **do**
      $d_h \leftarrow ||\boldsymbol{x_i} - \boldsymbol{\mu}_h||^2$ ▷ Compute distance to all cluster centers
    end for
    $membership_i \leftarrow \text{argmin}_h(d_h)$ ▷ Assign new membership
  end for
  for $h = 0 \dots k$ **do**
    $\boldsymbol{\mu}_h \leftarrow 0$ ▷ Initialize new cluster centers
    $member\_count_h \leftarrow 0$
  end for
  for $i = 0 \dots n$ **do** ▷ Sum up values according to cluster
    $h \leftarrow membership_i$
    $\boldsymbol{\mu}_h \leftarrow \boldsymbol{\mu}_h + \boldsymbol{x_i}$
    $member\_count_h \leftarrow member\_count_h + 1$
  end for
  for $h = 0 \dots k$ **do**
    $\boldsymbol{\mu}_h \leftarrow \frac{\boldsymbol{\mu}_h}{member\_count_h}$ ▷ Compute mean
  end for
  $converged \leftarrow \dots$ ▷ Any convergence criterion
end while

An interesting property of the classical k-means algorithm is that it is a specialization of applying the EM framework to Gaussian mixture models (see section 2.4), as shown in [28] and references therein. This enables us to include k-means in the framework presented in chapter 4.

2.3 High-dimensional sparse data

The term *sparse data* refers to any type of data with highly redundant entries, for example large matrices that contain mostly zeroes and only comparably few nonzero entries. Of course, processing these data in a so-called dense

representation (i.e. all entries of that matrix are stored, even the zeroes) takes up a lot of unnecessary space and computations.

In order to avoid this problem, this type of data is often represented in sparse formats that only store the nonzero entries and some additional information about the position of these entries in the dense representation.

Sparse data representations

The following sections describe the formats investigated in this thesis. As an example, matrix

$$\boldsymbol{A} = \begin{bmatrix} 0 & 0 & 0 & 5 & 0 & 0 \\ 1 & 0 & 6 & 0 & 0 & 0 \\ 0 & 7 & 3 & 0 & 5 & 1 \\ 4 & 0 & 0 & 2 & 9 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8 \end{bmatrix}$$

is used to demonstrate the conversion into the different formats.

CSR/CSC-format

The compressed sparse row format represents a dense $n \times m$[2] matrix with nnz nonzero entries with three arrays:

1. VAL: All nonzero entries are stored here sequentially, ordered by row (column in CSC).
2. $INDICES$: In the same order as VAL, the respective column-indices (row-indices in CSC) are stored.
3. $POINTERS$: For each row (column in CSC), the first index of a nonzero element of this row (column in CSC) in the VAL and $INDICES$ array is stored. Additionally, the total number of nonzero entries is appended to this array.

Both VAL and $INDICES$ thus have a length of nnz, while $POINTERS$ has length $n + 1$ ($m + 1$ in CSC).

The memory footprint of this format can be calculated as follows, with each nonzero value using val memory and each index using ind memory:

$$total\_memory = (n + 1) \cdot ind + nnz \cdot (ind + val) \tag{2.1}$$

The matrix $\boldsymbol{A}$ can be represented in this format in the following way:

1. $VAL = \{5, 1, 6, 7, 3, 5, 1, 4, 2, 9, 8\}$
2. $INDICES = \{3, 0, 2, 1, 2, 4, 5, 0, 3, 4, 5\}$
3. $POINTERS = \{0, 1, 3, 7, 10, 11\}$

[2]Since matrices, in the context of this thesis, represent a dataset where the number of samples, or rows, is usually indicated by n, we use n to indicate the number of rows and m (or d) to indicate the number of columns (or dimensions) of matrices.

COO-format

The coordinate format (COO) represents a dense $n \times m$ matrix with nnz nonzero with three arrays:

1. VAL: Stores the values of all nonzero entries in arbitrary order.
2. ROW: Stores the row index of the nonzero entries in the VAL array in the same order.
3. COL: Stores the column index of the nonzero entries in the VAL array in the same order.

The memory footprint of this format can be calculated as follows, with each nonzero value using val memory and each index using ind memory:

$$total\_memory = 2 \cdot nnz \cdot ind + nnz \cdot val \tag{2.2}$$

In this format, $\boldsymbol{A}$ is represented as:

1. $VAL = \{5, 1, 6, 7, 3, 5, 1, 4, 2, 9, 8\}$
2. $ROW = \{0, 1, 1, 2, 2, 2, 2, 3, 3, 3, 4\}$
3. $COL = \{3, 0, 2, 1, 2, 4, 5, 0, 3, 4, 5\}$

ELLPACK-format

The ELLPACK format stores a dense $n \times m$ matrix with nnz nonzero entries using two dense matrices. The size of these matrices depends on the row of the matrix with the most nonzero entries nz_{max}.

1. $\boldsymbol{VAL}$ is an $n \times nz_{max}$ matrix. For each row of the original matrix, the nonzero elements are stored sequentially in the respective row of $\boldsymbol{VAL}$. Since not all rows might have the same number of nonzero entries, the rows of $\boldsymbol{VAL}$ are padded with zeros.
2. $\boldsymbol{COL}$ also is an $n \times nz_{max}$ matrix, however, instead of the values of nonzero entries, it stores the respective column index. Instead of padding with zeros, here we insert an indicator value that indicates whether this element is valid or part of the padding.

The advantage of this format is, that we get a regular array layout that we can map well to the GPU and apply techniques like tiling on. However, this comes at the cost of a larger memory footprint due to the padding. The memory footprint of this format is the following:
$total\_memory = n \cdot nz_{max} \cdot (val + ind)$
The matrix $\boldsymbol{A}$ is represented as:

$$\boldsymbol{VAL} = \begin{bmatrix} 5 & 0 & 0 & 0 \\ 1 & 6 & 0 & 0 \\ 7 & 3 & 5 & 1 \\ 4 & 2 & 9 & 0 \\ 8 & 0 & 0 & 0 \end{bmatrix} \qquad \boldsymbol{COL} = \begin{bmatrix} 3 & * & * & * \\ 0 & 2 & * & * \\ 1 & 2 & 4 & 5 \\ 0 & 3 & 4 & * \\ 5 & * & * & * \end{bmatrix}$$

where $*$ is a value that indicates that this entry is part of the padding.

Hybrid formats

In the ELLPACK format, a big problem is that the size of the compressed matrices depends on the maximum number of nonzeros within one row. This leads to a lot of space being wasted in cases where the number of nonzeros differ greatly between rows. In the worst case, a single row is completely dense, and we store twice as many values as in the original dense representation.

In order to mitigate this, hybrid representations are commonly used to provide a trade-off between regularity of the data representation and compression.

Here, we choose the size τ of the two matrices such that they are of size $n \times \tau$. These matrices are filled with nonzero entries as in the ELLPACK format, however, in all rows that have more than τ nonzero values, the remaining ones are stored in any of the other sparse formats, for example CSR or COO.

2.4 Mixture models and the EM framework

The framework presented in chapter 4 aims to facilitate the implementation of mixture models. These mixture models can be fitted to a given dataset through use of the Expectation-Maximization (EM) framework [29], which is based on the maximum likelihood approach. The next section describes the maximum likelihood approach on the simple case of fitting a single Gaussian distribution to data. Afterwards, it is demonstrated how the maximum likelihood approach and EM can be used to fit mixtures of probability distributions to a given dataset.

Fitting a single distribution to data

First, we look at the simple case in which we want to fit a single d-dimensional Gaussian distribution in a way that it maximizes the probability of the n d-dimensional data points ($\boldsymbol{X} \in \mathbb{R}^{n \times d}$ with $\boldsymbol{X} = \{x_0, x_1, \cdots, x_n\}$) being samples of this distribution. This maximization approach is called maximum likelihood estimation.

Given $\mu \in \mathbb{R}^d$ and $\Sigma \in \mathbb{R}^{d \times d}$ where Σ is a symmetric and positive-definite matrix, the probability for one data point can be calculated with this formula:

$$p(\boldsymbol{x_i}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{\sqrt{(2\pi)^d \det(\boldsymbol{\Sigma})}} e^{-\frac{1}{2}(\boldsymbol{x_i}-\boldsymbol{\mu})^T \cdot \boldsymbol{\Sigma}^{-1}(\boldsymbol{x_i}-\boldsymbol{\mu})}$$

The task now is to maximize the parameters $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ given the data. In order to fit the distribution to *all* the data, we need to consider the joint distribution:

$$p(\boldsymbol{X}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = \prod_{i=0}^{n} \frac{1}{\sqrt{(2\pi)^d \det(\boldsymbol{\Sigma})}} e^{-\frac{1}{2}(\boldsymbol{x_i}-\boldsymbol{\mu})^T \cdot \boldsymbol{\Sigma}^{-1}(\boldsymbol{x_i}-\boldsymbol{\mu})}$$

Instead of maximizing this quantity, we can maximize the logarithm of this formula to turn the product into a sum, which is easier to handle:

$$\ln p(\boldsymbol{X}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = \sum_{i=0}^{n} \ln\left(\frac{1}{\sqrt{(2\pi)^d \det(\boldsymbol{\Sigma})}} e^{-\frac{1}{2}(\boldsymbol{x_i}-\boldsymbol{\mu})^T \cdot \boldsymbol{\Sigma}^{-1}(\boldsymbol{x_i}-\boldsymbol{\mu})}\right)$$

$$= \sum_{i=0}^{n} \ln\left(\frac{1}{\sqrt{(2\pi)^d \det(\boldsymbol{\Sigma})}}\right) - \frac{1}{2}(\boldsymbol{x_i}-\boldsymbol{\mu})^T \cdot \boldsymbol{\Sigma}^{-1}(\boldsymbol{x_i}-\boldsymbol{\mu})$$

$$= \sum_{i=0}^{n} -\ln\left(\sqrt{(2\pi)^d \det(\boldsymbol{\Sigma})}\right) - \frac{1}{2}(\boldsymbol{x_i}-\boldsymbol{\mu})^T \cdot \boldsymbol{\Sigma}^{-1}(\boldsymbol{x_i}-\boldsymbol{\mu})$$

$$= -\sum_{i=0}^{n} \frac{1}{2}\ln(2\pi)^d + \frac{1}{2}\ln\det(\boldsymbol{\Sigma}) + \frac{1}{2}(\boldsymbol{x_i}-\boldsymbol{\mu})^T \cdot \boldsymbol{\Sigma}^{-1}(\boldsymbol{x_i}-\boldsymbol{\mu})$$

$$= -\frac{n}{2}\left(\ln(2\pi)^d - \ln\det(\boldsymbol{\Sigma})\right) - \frac{1}{2}\sum_{i=0}^{n}(\boldsymbol{x_i}-\boldsymbol{\mu})^T \cdot \boldsymbol{\Sigma}^{-1}(\boldsymbol{x_i}-\boldsymbol{\mu})$$

In order to find the parameters μ and Σ that produce the best fit for the data, we can take the gradient with respect to those parameters. For μ:

$$\frac{\partial}{\partial \boldsymbol{\mu}}\left(\ln p(\boldsymbol{X}|\boldsymbol{\mu}, \boldsymbol{\Sigma})\right) = \frac{\partial}{\partial \boldsymbol{\mu}}\left(-\frac{n}{2}\left(\ln(2\pi)^d - \det(\boldsymbol{\Sigma})\right) - \frac{1}{2}\sum_{i=0}^{n}(\boldsymbol{x_i}-\boldsymbol{\mu})^T \cdot \boldsymbol{\Sigma}^{-1}(\boldsymbol{x_i}-\boldsymbol{\mu})\right)$$

$$= \frac{\partial}{\partial \boldsymbol{\mu}}\left(-\frac{1}{2}\sum_{i=0}^{n}(\boldsymbol{x_i}-\boldsymbol{\mu})^T \cdot \boldsymbol{\Sigma}^{-1}(\boldsymbol{x_i}-\boldsymbol{\mu})\right)$$

Now we can use the assumption that $\boldsymbol{\Sigma}$ is symmetric and apply the identity $\frac{\partial}{\partial \boldsymbol{\mu}}(\boldsymbol{x}_i-\boldsymbol{\mu})^T\boldsymbol{\Sigma}(\boldsymbol{x}_i-\boldsymbol{\mu}) = -2\boldsymbol{\Sigma}(\boldsymbol{x}_i-\boldsymbol{\mu})$ [30]:

$$= -\frac{1}{2}\sum_{i=0}^{n} -2\boldsymbol{\Sigma}^{-1}(\boldsymbol{x_i}-\boldsymbol{\mu}) = \sum_{i=0}^{n}\boldsymbol{\Sigma}^{-1}(\boldsymbol{x_i}-\boldsymbol{\mu})$$

Since the negative log likelihood is convex, we can find the maximum by setting this gradient to 0:

$$0 = \sum_{i=0}^{n}\boldsymbol{\Sigma}^{-1}(\boldsymbol{x_i}-\boldsymbol{\mu})$$

$$0 = \sum_{i=0}^{n} \boldsymbol{\Sigma}^{-1} \boldsymbol{x_i} - \sum_{i=0}^{n} \boldsymbol{\Sigma}^{-1} \boldsymbol{\mu}$$

$$n\boldsymbol{\Sigma}^{-1} \boldsymbol{\mu} = \sum_{i=0}^{n} \boldsymbol{\Sigma}^{-1} \boldsymbol{x_i}$$

$$\boldsymbol{\mu} = \frac{1}{n} \sum_{i=0}^{n} \boldsymbol{\Sigma} \boldsymbol{\Sigma}^{-1} \boldsymbol{x_i} = \frac{1}{n} \sum_{i=0}^{n} \boldsymbol{x_i}$$

We can see, that $\boldsymbol{\mu}$ is maximized by taking the mean of all data points.

In order to find the optimal covariance matrix $\boldsymbol{\Sigma}$, we have to take the derivative with respect to $\boldsymbol{\Sigma}$:

$$\frac{\partial}{\partial \boldsymbol{\Sigma}} \left(\ln p(\boldsymbol{X} | \boldsymbol{\mu}, \boldsymbol{\Sigma}) \right) = \frac{\partial}{\partial \boldsymbol{\Sigma}} \left(-\frac{n}{2} \left(\ln(2\pi)^d - \det(\boldsymbol{\Sigma}) \right) - \frac{1}{2} \sum_{i=0}^{n} (\boldsymbol{x_i} - \boldsymbol{\mu})^T \cdot \boldsymbol{\Sigma}^{-1} (\boldsymbol{x_i} - \boldsymbol{\mu}) \right)$$

$$= \frac{\partial}{\partial \boldsymbol{\Sigma}} \left(-\frac{n}{2} \det(\boldsymbol{\Sigma}) \right) - \frac{\partial}{\partial \boldsymbol{\Sigma}} \left(\frac{1}{2} \sum_{i=0}^{n} (\boldsymbol{x_i} - \boldsymbol{\mu})^T \cdot \boldsymbol{\Sigma}^{-1} (\boldsymbol{x_i} - \boldsymbol{\mu}) \right)$$

Using the following identity: $\frac{\partial \ln \det(\boldsymbol{\Sigma})}{\partial \boldsymbol{\Sigma}} = \left(\boldsymbol{\Sigma}^{-1} \right)^T$ [30], omitting transpositions of $\boldsymbol{\Sigma}$ since it is symmetric.

$$= -\frac{n}{2} \boldsymbol{\Sigma}^{-1} - \frac{\partial}{\partial \boldsymbol{\Sigma}} \left(\frac{1}{2} \sum_{i=0}^{n} (\boldsymbol{x_i} - \boldsymbol{\mu})^T \cdot \boldsymbol{\Sigma}^{-1} (\boldsymbol{x_i} - \boldsymbol{\mu}) \right)$$

Using the identity $\frac{\partial \boldsymbol{a}^T \boldsymbol{\Sigma}^{-1} \boldsymbol{b}}{\partial \boldsymbol{\Sigma}} = -\boldsymbol{\Sigma}^{-1} \boldsymbol{a} \boldsymbol{b}^T \boldsymbol{\Sigma}^{-1}$ [30]:

$$= -\frac{n}{2} \boldsymbol{\Sigma}^{-1} - \frac{1}{2} \sum_{i=0}^{n} -\boldsymbol{\Sigma}^{-1} (\boldsymbol{x_i} - \boldsymbol{\mu}) (\boldsymbol{x_i} - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}$$

Setting this to zero:

$$n\boldsymbol{\Sigma}^{-1} = \sum_{i=0}^{n} \boldsymbol{\Sigma}^{-1} (\boldsymbol{x_i} - \boldsymbol{\mu}) (\boldsymbol{x_i} - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}$$

Multiply by $\boldsymbol{\Sigma\Sigma}$

$$\boldsymbol{\Sigma} = \frac{1}{n}\sum_{i=0}^{n}(\boldsymbol{x_i} - \boldsymbol{\mu})(\boldsymbol{x_i} - \boldsymbol{\mu})^T$$

In some cases (for example when solving the maximum likelihood estimation for mixture models), it can be necessary to introduce a Lagrange multiplier in order to find a solution.

We can see that fitting a single distribution to some data is relatively straightforward. However, this does not help us with our actual goal, i.e. clustering the data.

Fitting mixture models to data

Mixture models can be used to cluster a set of data points into a known number of clusters. A probability distribution that suits the application domain is chosen, for example the Gaussian distribution. Each cluster is then represented by one distribution with an individual set of parameters, often called *components*. In order to assign data points to clusters, the probability distribution is evaluated for this point and each of the k sets of parameters. The point is then assigned to the cluster that yields the highest probability. Usually, each of the k distributions is also assigned a weight α, which represents the global likelihood of each of the distributions.

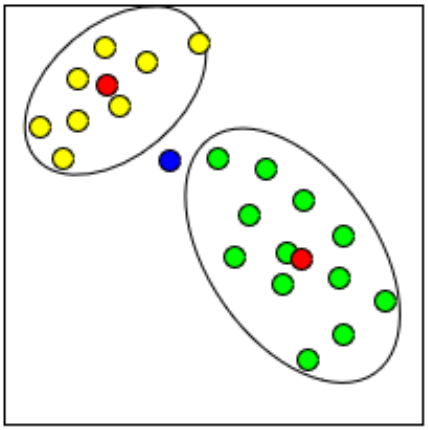

Figure 2.3: Visualization of a mixture model consisting of 2 Gaussian distributions in 2 dimensions.

Figure 2.3 shows the result of fitting a Gaussian mixture model consisting of two components in 2-dimensional space. This result is much more expressive than that of simple k-means, since it captures the shape of the clusters. For example, the blue dot between the two clusters is closer to the center of the yellow cluster, however, it fits the shape of the data in the green cluster better. The mixture model takes this into account, whereas k-means would simply assign it to the nearest cluster.

The rest of this section describes how the EM procedure can be used to fit mixture to datasets and is based on the explanations in [1] and [31].

Given a probability distribution $f(\boldsymbol{x}, \theta)$ and k sets of parameters $\theta_1, \theta_2, \cdots, \theta_k$, as well as parameters $\alpha_1, \alpha_2, \cdots, \alpha_k$ with $\sum_{h=1}^{k} \alpha_h = 1$ and $\alpha_h \geq 0$ $(1 \leq h \leq k)$ we can write the probability of our data being generated from this mixture as

$$p(\boldsymbol{x}|\Theta, A) = \sum_{h=1}^{k} \alpha_h f(\boldsymbol{x}, \theta_h) \tag{2.3}$$

where $\Theta = \{\theta_1, \theta_2, \cdots, \theta_k\}$ and $A = \{\alpha_1, \alpha_2, \cdots, \alpha_k\}$.

Intuitively, samples are generated from this distribution by first choosing parameters θ_h with probability α_h and then sampling from $f(\theta_h)$. This in turn means, that we assume our data to be generated this way. Thus, every data point x_i has a hidden variable z_i that denotes which of the distributions it was sampled from.

Given values $\boldsymbol{Z} = \{z_1, z_2, \cdots, z_n\}$ where n is the number of samples in our data, we could use the maximum likelihood approach described in the previous section to find the parameters θ for the distribution $f(\theta)$. However, since we are looking at unsupervised clustering the true values of $\boldsymbol{Z}$ are unknown and need to be deduced from the data. This is where the EM approach comes into play.

The EM framework consists of two steps, i.e. the expectation step and the maximization step, as the name suggests.

1. In the expectation step, we assume that we are given approximations for the θ parameters of all k distributions and based on this we compute the likelihood of a point belonging to a certain distribution as follows:

$$p(z_i = h|\boldsymbol{x_i}, \Theta) = \frac{\alpha_h f(\boldsymbol{x_i}, \theta_h)}{\sum_{l=1}^{k} \alpha_l f(\boldsymbol{x_i}, \theta_l)} \tag{2.4}$$

2. The maximization step then estimates new parameters θ based on these assignments. For this, we have to look at the expected value of our mixture $\mathbb{E}\left[p(\boldsymbol{X}|\Theta, A)\right]$ and use the maximum likelihood approach to find the parameters $\hat{\theta}_h$ and $\hat{\alpha}_h$ that maximize the expected value. For example, for a mixture of multivariate Gaussians these can be computed as follows:

$$\hat{\alpha}_h = \frac{1}{n} \sum_{i=1}^{n} p(z_i = h|\boldsymbol{x_i}, \Theta) \tag{2.5}$$

$$\hat{\boldsymbol{\mu}}_{\boldsymbol{h}} = \frac{\sum_{i=1}^{n} \boldsymbol{x_i} p(z_i = h|\boldsymbol{x_i}, \Theta)}{\sum_{i=1}^{n} p(z_i = h|\boldsymbol{x_i}, \Theta)} \tag{2.6}$$

$$\hat{\boldsymbol{\Sigma}}_{\boldsymbol{h}} = \frac{\sum_{i=1}^{n} (\boldsymbol{x_i} - \hat{\boldsymbol{\mu}}_{\boldsymbol{h}})(\boldsymbol{x_i} - \hat{\boldsymbol{\mu}}_{\boldsymbol{h}})^T p(z_i = h|\boldsymbol{x_i}, \Theta)}{\sum_{i=1}^{n} p(z_i = h|\boldsymbol{x_i}, \Theta)} \tag{2.7}$$

with $f(\boldsymbol{x_i}, \theta_h) = \frac{1}{\sqrt{(2\pi)^d \det(\boldsymbol{\Sigma_h})}} e^{-\frac{1}{2}(\boldsymbol{x_i} - \boldsymbol{\mu_h})^T \cdot \boldsymbol{\Sigma_h}^{-1} (\boldsymbol{x_i} - \boldsymbol{\mu_h})}$

These steps are executed repeatedly until the joint probability of the mixture over all data points changes less than a chosen amount between iterations. Since in each iteration, this probability can not decrease [32], it is likely (though not guaranteed) that we have found a local maximum after convergence.

Note that in the expectation step we are not directly computing $\boldsymbol{Z}$, since it is discrete. Instead, for each data point we have k probabilities, one for each of the distributions. For data point x_i and distribution h, we compute $p(z_i = h|\theta, \boldsymbol{x_i})$. We can then estimate the z_i by assigning the cluster with the highest probability: $\hat{z}_i = \text{argmax}_h\, p(z_i = h|\theta, \boldsymbol{x_i})$.

2.5 Soft assignment vs. hard assignment

In the following chapters of this thesis we use the distinction between hard and soft cluster assignments. This refers to the method in which data points are associated to clusters. In algorithms using hard cluster assignment, a data point is always fully assigned to one cluster. For example in classical k-means, we store a `membership` variable for each of the data points that indicates which of the clusters they are closest to.

On the other hand, algorithms that use soft cluster assignments will store k values for each data point (k being the number of clusters). These values indicate how likely it is that a data point belongs to each of the clusters. In mixture models, these values are the posterior probabilities of evaluating each component of the mixture model for each of the data points.

The characteristics of the two strategies are analyzed in [33], however, this thesis is only concerned with the implementation of the strategies and not their semantic traits.

Chapter 3

Sparse k-means explorations

When implementing an algorithm with a focus on performance, it is critical to focus on the biggest contributors to the runtime first. Since there is an inherently sequential loop in the k-means algorithm, most time will be spent inside this loop.

The operations within the loop can be split into two parts:

1. The distance computations (corresponding to the expectation step of the EM scheme). Here, we compute the distances from each data point to each of the cluster centers and assign a membership value to each point, indicating which cluster is the closest.

2. The reduction step (corresponding to the maximization step of the EM scheme). In this step the new cluster centers for the next iteration are computed. This means, that for each cluster we need to add up all the points assigned to this cluster and then divide the summed up vector by the number of assigned points.

3.1 Distance computation and optimizations

The distance computation is the main part of the k-means algorithm when it comes to computational complexity. In this step, $n \cdot k$ Euclidean distances are computed. This computation is fairly simple in the dense case. However, there are several ways in which we can transform this operation for sparse data to achieve good performance. In this section, four approaches and their suitability for the k-means algorithm are discussed.

To get an intuition for the problem, the distance computation for dense data is shown in listing 3.1 (kindly taken from the Futhark benchmarks repository [34]):

```
-- squared Euclidean distance: ||(p1 -p2)||^2
let euclidean_dist_sq p1 p2 =
                sum (map (\x->x*x) (map2 (-) p1 p2))
let compute_distances [n][d][k]
    (points: [n][d]f32)
    (cluster_centers: [k][d]f32):
        [n][k]f32 =
    map (\p ->
            map (\center ->
                euclidean_dist_sq p center
            ) cluster_centers
    ) points
```

Listing 3.1: Distance computation for dense data.

This code simply maps the squared [1] Euclidean distance function over all points and all clusters. This computation, however, is not as straightforward on sparse data as it is for the dense case. Specifically the subtraction inside the distance computation is troublesome, since for each dimension of the dense cluster center, it is not directly possible to look up whether a data point has a zero entry in this dimension, and if so, where in the sparse data representation the respective value is located.

One solution to this problem is to formulate the computation in terms of the semiring primitive presented in [15]. The next section describes this transformation and provides intuitions on how sparse data affects the asymptotic cost of the transformed metric.

Expansion of the Euclidean metric

As concluded in [15], it is beneficial for the performance to extend the squared Euclidean metric when computing distances across sparse datasets. In order to derive this form, we start with the squared Euclidean metric for d-dimensional vectors $\boldsymbol{x}$ and $\boldsymbol{y}$, written as

$$d(\boldsymbol{x}, \boldsymbol{y}) = \sum_{i=1}^{d} (x_i - y_i)^2 \tag{3.1}$$

We can extend this to

$$d(\boldsymbol{x}, \boldsymbol{y}) = \sum_{i=1}^{d} \left[x_i^2 - 2x_i y_i + y_i^2\right] \tag{3.2}$$

$$= \sum_{i=1}^{d} x_i^2 - 2\sum_{i=1}^{d} x_i y_i + \sum_{i=1}^{d} y_i^2 \tag{3.3}$$

We can observe, that now there is a dot product between $\boldsymbol{x}$ and $\boldsymbol{y}$ instead of a subtraction. In the case of either $\boldsymbol{x}$ or $\boldsymbol{y}$ being sparse, we only have to

[1] For the k-means algorithm it suffices to compute the squared Euclidean metric. This is correct since we only compare distances with each other and square root is a monotone function. This simplification is used throughout this chapter.

compute values for the columns that are nonzero in $\boldsymbol{x}$ *and* in $\boldsymbol{y}$, i.e. the union of nonzero columns.

In the case of k-means, one of the vectors $\boldsymbol{x}, \boldsymbol{y}$ represents a cluster center, stored in a dense data representation. We can use this fact and the previous observations to compute the order of operations necessary to compute the distances of n sparse d-dimensional points with a total of nnz nonzero entries to k dense d-dimensional cluster centers. In the computations $\boldsymbol{x}$ will represent a dense cluster center and $\boldsymbol{y}$ one data point.

The first term $\sum_{i=1}^{d} x_i^2$ does not involve any sparse operations, so here we have $O(d)$ operations. Since we have k cluster centers, there are $O(k \cdot d)$ computations. This computation does not depend on the dataset, so this quantity only needs to be computed once for each cluster center instead of once for each combination of data point and cluster center.

In the second term $-2\sum_{i=1}^{d} x_i y_i$, we compute the sparse dot product of $\boldsymbol{x}$ $\boldsymbol{y}$. Since only the nonzero entries from $\boldsymbol{y}$ contribute to the sum, we only have $O(nnz)$ operations for the whole dataset for a single cluster center. Thus, in total there are $O(k \cdot nnz)$ operations.

The third term $\sum_{i=1}^{d} y_i^2$, again, is only influenced by the nonzero entries of $\boldsymbol{y}$. Thus, there are $O(nnz)$ operations.

In order to avoid unnecessary accesses to the sparse dataset, we can group together the second and third term:

$$\sum_{i=1}^{d} (y_i - 2x_i) y_i \tag{3.4}$$

The following code snippet shows how these observations can be used to compute the distances for a sparse dataset in CSR format (given the `val`, `indices` and `pointers` arrays).

```
let compute_distances [nnz][np1][k][d]
    (values: [nnz]f32) --we assume CSR representation
    (col_indices: [nnz]i64)
    (pointers: [np1]i64) --pointers contains n+1 entries
    (cluster_centers: [k][d]f32): --centers stored as dense 2d array
        [n][k]f32 =
    let cluster_squared_sum = map (\c ->
                                sum (map (\x -> x*x) c)
                            ) cluster_centers
    let sparse_terms =
            map (\cluster ->
                map (\i
                    let row_start = pointers[i]
                    let row_end = pointers[i+1]
                    let nnz = row_end - row_start
                    in reduce (+) 0
                        (map (\j ->
                            let value = values[j]
                            let col = col_indices[j]
                            let cluster_val = cluster[col]
                            in (value - 2 * cluster_value)*value
                        ) (map (+row_start) (iota nnz)))
                ) (iota n)
            ) cluster_centers
    -- combining the two terms:
    in map2 (\sp cluster_term -> map (+cluster_term) sp)
                sparse_term cluster_squared_sum
```

Listing 3.2: Irregular implementation of distance computation

This code only uses the basic building blocks `map` and `reduce` of the Futhark language, which can be optimized automatically using the incremental flattening feature through the `futhark autotune` command [21].

However, the inner `reduce` and `map` are irregular, meaning that they operate on arrays of varying lengths for each value of `i`. Since Futhark cannot exploit irregular parallelism like this, we have to either use a regular sparse representation of our data like the ELLPACK format, or manually flatten the code if we wish to utilize the parallelism of the innermost map-reduce construct. The advantages and disadvantages of these approaches are discussed in the following sections.

Flat segmented scan

A common approach for solving the irregularity problem is flattening. By applying a set of flattening rules to a given program, we can get rid of nested parallelism and rewrite it with the help of segmented operations (i.e. segmented scan and segmented reduce). This enables us to extract as much parallelism as possible from the problem, as we are able to parallelize all the $O(k \cdot nnz)$ operations.

In order to flatten the code in this manner, we need to create a flag array that indicates the segments for the nonzero elements of each row from the CSR representation. This is a relatively simple transformation and can be done outside the main loop, thus the overhead is negligible.

Given this `flags` array, as well as the `col_indices` and `val` arrays from the CSR representation, we can flatten the distance computation as shown in listing 3.3.

```
let compute_sparse_terms [nnz][k][d]
    (values: [nnz]f32) --we assume CSR representation
    (col_indices: [nnz]i64)
    (flags: [nnz]bool)
    (cluster_centers: [k][d]f32): --centers stored as dense 2d array
        [n][k]f32 =
    map (\cluster->
        let dist_terms =
            map (\element_index element_value ->
                let cluster_value = cluster[element_index]
                in (element_value - 2 * cluster_value)*element_value
        ) col_indices values
        in segreduce (+) 0 flags dist_terms :> [n]f32
    ) cluster_centers
```

Listing 3.3: Flattened implementation of distance computation.

The function `segreduce` computes the segmented reduction for an array. This means, that for every segment indicated by the `flags` array, the `reduce` operation with the specified operator and neutral element is applied. The result is an array that has the number of segments as its length. This is visualized in figure 3.1.

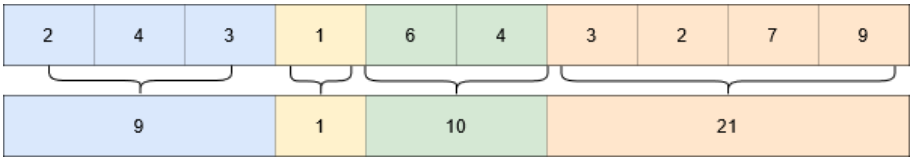

Figure 3.1: Visualization of the segmented reduction operation.

Note that we need to cast the result of the segmented reduction to the right size, in order to let the Futhark compiler know that the result has the same type across the outer map, i.e. there is no irregular nested parallelism. This also assumes that there are no empty rows in the dataset, since this would lead to empty segments and the size of the result of `segreduce` would be less than n.

This approach has the big advantage that it uses all the available parallelism for this operation. Since Futhark can flatten the outer map over the cluster centers, we are able to use $k \times nnz$ parallel threads for the distance computation. If the dataset is small enough or the GPU sufficiently powerful with enough physical threads to match this amount of parallelism, we can expect good performance.

On the other hand, flattening often causes the memory requirements of the algorithm to grow too much to be feasible (see this blog post [35]) and might also lead to worse spatial locality, as the memory accesses are spread out over more threads.

This approach is referred to as **flat** in the benchmark section.

Sequential reduction of nonzeros in each row

An alternative approach to flattening the sparse distance computation is explicitly sequentializing the innermost map-reduce construct over the nonzero entries of each row. This will, of course, not use all the available parallelism, however, if we have enough rows and cluster centers to saturate all threads of the hardware, this approach might yield a better performance than flattening.

Listing 3.4 shows the manually sequentialized version of the distance computation. Futhark will generate similar code from the irregular formulation in listing 3.2, however, explicitly sequentializing this code allows us to reason about its performance more clearly.

```
let compute_sparse_terms [nnz][np1][k][d]
        (values: [nnz]f32) --we assume CSR representation
        (col_indices: [nnz]i64)
        (pointers: [np1]i64) --pointers contains n+1 entries
        (cluster_centers: [k][d]f32): --centers stored as dense 2d array
            [n][k]f32 =
    map (\cluster ->
        map (\row ->
            let index_start = pointers[row]
            let nnz = pointers[row+1] - index_start
            let row_dist = 0
            let row_dist =
            loop row_dist for j < nnz do
                let value = values[index_start+j]
                let column = col_indices[index_start+j]
                let cluster_value = cluster[column]
                let value = (value - 2 * cluster_value)*value
                in row_dist+value
            in row_dist
        ) (iota n)
    ) cluster_centers
```

Listing 3.4: Distance computation with sequentialized inner map-reduce.

We can observe, that in this implementation neighboring threads access the `val` and `col_indices` arrays with an irregular stride of the nonzero elements of each row. This uncoalesced access pattern[2] is not optimal and likely lowers the performance of this computation.

Furthermore, the loops in successive threads do not necessarily have the same number of iterations, since the number of nonzero elements generally varies between rows. This causes thread divergence, another culprit for suboptimal performance.

Luckily, a simple loop interchange of the two outermost maps alleviates both of these issues. If the map over the cluster centers is inside the map over the rows, k successive threads will access the same data in the same order, hence benefiting from temporal locality. This improvement will be greatest once k is higher than the number of threads within a warp, as thread diver-

[2]When a warp of threads accesses consecutive memory locations, this access will be serviced by one memory transaction. However, if they do not access memory consecutively (i.e. in an *uncoalesced* fashion), there can be as many transactions as there are threads in a warp in the worst case.

gence is eliminated and enough data is loaded from global memory, such that no redundant memory transactions are necessary.

This approach is referred to as **seq** in the benchmark section.

Another way to treat the thread divergence for small k is to sort the data rows by the number of nonzero entries. This makes sure that the loops in neighboring threads (i.e. neighboring rows) run for a similar amount of iterations.

In the benchmark section, the **seq_sorted** label indicates the combination of the **seq** approach with this sorting procedure.

It is important to note that this approach only parallelizes over the data points and clusters (i.e. we can use $n \times k$ parallel threads). If there are more physical threads available on the hardware than $n \times k$ for the given dataset and k, we are not using the hardware optimally and flat implementations will likely perform better.

ELLPACK

Another way to tackle the irregularity of this problem is to introduce padding in a way that the computations of all rows have the same number of operations. This allows Futhark to apply techniques that are useful for operations with dense matrices like transpositions and tiling.

The problem with this approach, however, is that the amount of padding necessary varies a lot between datasets and the benefits of the better data layout might be neutralized by the added amount of calculations.

By transforming the input to the ELLPACK format we can theoretically implement this approach, but in practice this representation takes a lot of memory which is scarce on GPUs, so this is only feasible for small datasets.

This approach is referred to as **ellpack** in the benchmark section.

Hybrid (ELLPACK + COO)

In order to address the regularity problem in a memory efficient manner, it is possible to combine multiple of the previous approaches. Such a hybrid approach is, for example, found to be a good solution to the sparse matrix-vector multiplication problem, as shown in [14].

As described in section 2.3, the dataset is split into a regular part in the ELLPACK format, which ideally should cover most of the necessary operations, and in an irregular representation like the COO format, which ideally only contains comparably few elements.

The distance contributions of the regular section can be easily computed by using a combination of regular `map` and `reduce` operations, as shown in listing 3.5.

```
let compute_sparse_terms [n][width][k][d]
    (ellpack_mat: [n][width](f32,i64)) --we assume ELLPACK format
    [n][k]f32 =
    map (\row ->
        map (\cluster->
            reduce (+) 0
                (map (\(v,i) ->
                    if i < 0 then 0 else -- check for padding
                        let cluster_value = cluster[i]
                        in (v - 2 * cluster_value)*v
                ) row)
        ) cluster_centers
    ) ellpack_mat
```

Listing 3.5: Computation of sparse terms from data in ELLPACK format.

In this thesis, we chose the irregular part of the representation to be in COO format, with the additional assumption that successive nonzero values of a row are stored successively. We tested two ways of computing the distance contributions of the irregular part of the representation.

1. Atomic addition: Since most of the nonzero values are ideally covered by the regular ELLPACK part of this representation, there should be relatively few conflicts when adding the irregular distance contributions to the previously computed regular contributions using atomic operations.

2. Segmented reduction: Since we need to sum over the values of each row, and we stored the data in a way that these are also stored successively, we also tested a segmented reduction over the irregular representation, similar to the approach described in section 3.1.

These approaches are labeled **hybrid** and **hybrid_flat** in the benchmark section.

3.2 Reduction step and optimizations

In the reduction step the data points belonging to each cluster need to be summed up and divided by the number of data points in the respective cluster to calculate the new cluster centers.

More specifically, for each cluster center $\boldsymbol{c}_h$ ($1 \leq h \leq k$) and row i, where p_h d-dimensional data points are assigned to cluster center $\boldsymbol{c}_h$ we need to calculate

$$\boldsymbol{c}_h = \frac{1}{p_h} \sum_{\{i|z_i=h\}} \boldsymbol{x}_i \tag{3.5}$$

Listing 3.6 shows the Futhark implementation from the Futhark benchmarks repository of the reduction step of k-means. This implementation uses the `reduce_by_index` function to sum up the points of each cluster. The operator used in the reduction is the vectorized $+$. This operator takes two vectors of the same length (in our case the data points) and adds them together component-wise.

```
let reduce_cluster_centers [n][d]
            (k: i64)
            (points: [n][d]f32)
            (membership: [n]i32):
                [k][d]f32 =
    -- we first use a histogram to count
    -- how many data points are in each cluster
    let points_in_clusters =
        reduce_by_index (replicate k 0) (+) 0 membership (replicate n 1)

    let cluster_sums =
        reduce_by_index (replicate k (replicate d 0)) -- initialize k x d zeroes
                        (map2 (+)) -- the reduce operator is vectorized(!) +
                        (replicate d 0) -- neutral element is a vector of 0
                        membership -- membership used as keys
                        points

    -- divide by number of data points
    in map2 (\center count ->
        map (\x -> x / count) center
    ) new_centers points_in_clusters
```

Listing 3.6: Futhark implementation of the reduction step for dense data

Unfortunately, we cannot use this operator in the case of sparse data. Similarly to the subtraction inside the distance metric (see section 3.1), we have no easy way of looking up the shared nonzero columns of two sparse vectors. Instead, we can use the `reduce_by_index` function to sum over sparse vectors. Figure 3.2 visualizes how this can be done on the example of three 5-dimensional vectors in sparse representation where each color indicates one vector. Following the arrows from the location in which the values are stored to the location in which they end up is not straightforward and shows how irregular this problem is. We can observe, that the values processed in the same operations are not necessarily close to each other in memory. This forces us to either sort the data in a way that orders the nonzero elements accordingly, or process the data in the random input order.

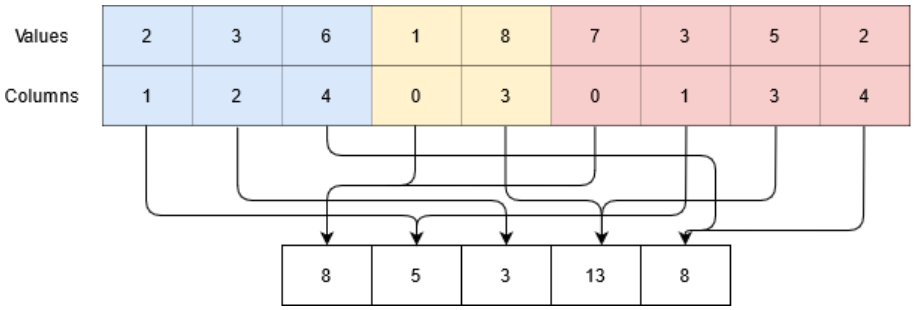

Figure 3.2: Visualization of adding 3 sparse vectors of dimensionality 5.

This is, however, only solving the summation of vectors and not the distribution of the vectors to the cluster centers they are assigned to. To achieve this distribution, we can manually flatten this construct such that the cluster centers are represented by a flat array of length $k \times d$. Then, we need to add an offset to the target columns such that only points that are assigned to the

same cluster center are summed together. For this, nonzero values from a vector assigned to cluster h have the offset $h \cdot d$ added to them.

To compute this offset efficiently, we need to associate a cluster assignment to each nonzero value. With the COO format, it is possible to directly look up the row index of a nonzero value, and by that we can also easily find the cluster assignment. In contrast, finding the row index of a nonzero entry in the CSR format is much more complex, as it requires either a linear or, more optimized, a binary search into the pointers array. As this additional complexity did not seem to be worth the smaller memory footprint, we decided to base our implementations for the reduction step on the COO format.

Listing 3.7 shows this implementation for a sparse input in COO format.

```
let reduce_cluster_centers [nnz][n]
                (k: i64)
                (vals: [nnz]f32) -- nonzero values
                (row_indices: [nnz]i64)
                (col_indices: [nnz]i64)
                (d: i64) -- number of columns in the dataset
                (membership: [n]i64): cluster membership of each point
                    [k][d]f32 -- k d-dimensional cluster centers
    -- we first use a histogram to count
    -- how many data points are in each cluster
    let points_in_clusters =
            reduce_by_index (replicate k 0) (+) 0 membership (replicate n 1)

    -- we interpret the k cluster centers as flat k x d array.
    -- the indices have to be computed accordingly
    -- by taking the membership of each point into account
    let target_indices = map2 (\row col ->
                            d*membership[row] + col
                        ) row_indices col_indices

    -- the main reduction
    let new_centers = reduce_by_index (replicate d 0)
                (+) 0 target_indices vals

    -- unflatten to return a k x d array
    let new_centers = unflatten k d new_centers

    -- divide by number of data points
    in map2 (\center count ->
            map (\x -> x / count) center
        ) new_centers points_in_clusters
```

Listing 3.7: Futhark implementation of the reduction step for data in COO format

The implementation of `reduce_by_index` uses a multi-pass strategy [22]. This strategy performs multiple passes over the input and only updates a subset of locations such that in each pass all updates fit into the L2 cache or the fast shared memory space of the GPU. However, if we order the data in such a way that successive items fall into a smaller range of target locations we can reduce the number of passes without sacrificing performance. For example, if we order all the data points by the cluster center they are assigned to, we only need to keep at most d (the dimensionality of the data points) locations in resident memory instead of $k \cdot d$. This becomes increasingly important, as we increase k.

To further optimize this reduction, for example by applying ideas from [36], we need access to atomic operations that are not directly accessible through Futhark. Hence, we decided to implement and benchmark different approaches in CUDA.

Atomic operations

The most trivial approach for the reduction step is to use atomic operations, specifically the well-optimized `atomicAdd` CUDA function to sum the nonzero elements of the data to the dense cluster centers. For this, we need to associate the column, as well as the index of the assigned cluster to every nonzero value of our data. If this data is given, a CUDA kernel for the reduction step can be implemented as shown in listing 3.8

```
__global__ void reduce_atomic(
    float* cluster_centers,
    const int* membership,
    const float* data,
    const long long* row_indices,
    const long long* column_indices,
    int n,
    int nnz,
    int columns,
)
{
    const int index = blockDim.x * blockIdx.x + threadIdx.x;
    if (index >= nnz)
    return;

    const int row = row_indices[index];
    const int col = column_indices[index];
    float val = data[index];
    const int m = membership[row];

    const int target_pos = columns*m + col;
    atomicAdd(&cluster_centers[target_pos], val);
}
```

Listing 3.8: Reduction step of sparse points in CUDA

The normalization in equation 3.5 can either be done after the sum in a second kernel or by dividing each individual element before adding it to the respective cluster center. The latter introduces more floating point rounding errors. However, it can be beneficial to the performance for higher values of k, since the operation of dividing k dense cluster centers by the counts is a memory-bound operation. Dividing the values before adding them makes sure, that we only do nnz divisions instead of $k \cdot d$ divisions.

These two code versions are referred to as **atomic** and **atomic - divide before** in the benchmarks.

Sorting by cluster assignments

Previous work investigating the implementation of k-means for dense datasets on the GPU [36] has found that sorting the data points by their assigned cluster enables an irregular reduction operation to be used for the reduction step

of k-means. The full irregular reduction is presented in section 3.2, but proved to be inappropriate for the kinds of data investigated in this thesis due to their high dimensionality.

The idea of sorting by cluster assignment is still useful for our cause, though. Spatial and temporal locality of the atomic operations can be improved by this idea. Since the cluster centers are dense and very high-dimensional, the destinations of the atomic operations are scattered across a large memory space. More specifically, for a d-dimensional dataset, there are $k \cdot d$ possible destinations for the atomic operations. For higher values of k and the datasets investigated in this thesis, this space will not fit into the caches of the GPU. Thus, arranging the data points according to cluster improves the likelihood of the atomic operations working on cached memory locations instead of global memory.

Figure 3.3 visualizes this idea for the 1-dimensional case and $k = 3$, where each color represents a cluster. We can see that after sorting, spatial locality is highly improved as items that fall into the same cluster are stored consecutively.

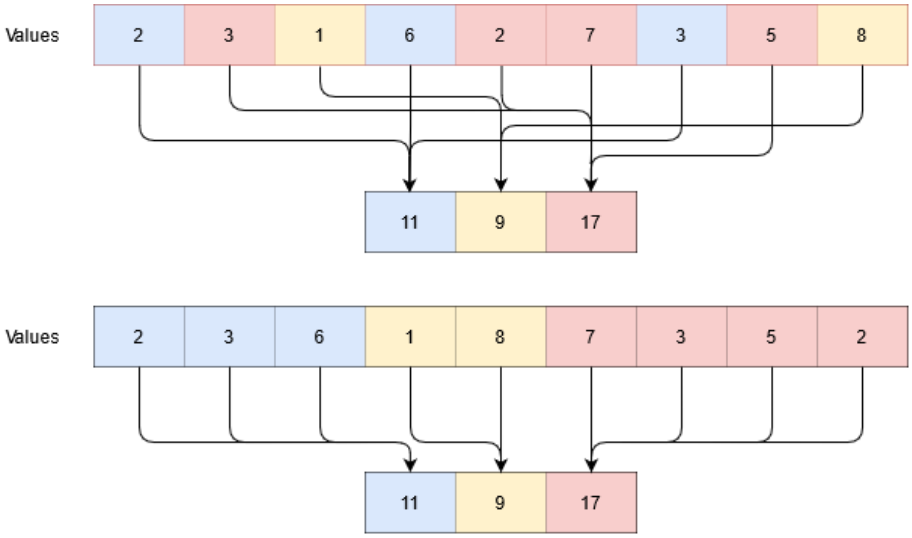

Figure 3.3: Advantage of sorting by cluster assignment visualized

For this approach to be successful, the overhead of sorting the dataset has to be small enough that it does not mitigate its beneficial effect. In [36], the counting sort algorithm is used to sort the data points by the cluster.

Counting sort is a sorting algorithm that is useful for sorting items into a relatively small number of bins. It consists of three steps:

1. **Count**: The first step counts the number of elements that fall into each of the bins (`cluster_counts` in listing 3.9). Since we also want to find a position for the elements inside the bins, we can use CUDA's `atomicAdd` function. It atomically adds a value to the specified location and returns the previous value of that location. To count the number of items in each bin, we can just `atomicAdd` the value 1 to a bin, for

each item that falls into this bin. When we add the first item to a bin, `atomicAdd` will return 0. For the second item added to this bin, it will return 1. In the sorting step we can then safely put the items in these positions of the respective bin without causing conflicts.

2. **Scan**: Since we are sorting a flat array into sections corresponding to the bins, we need to know where each of the sections start. For this purpose, we can perform an exclusive scan on the `cluster_counts`. For the first cluster this will be 0, since we want to start filling the sorted array at index 0. The second section will start at index `cluster_count[0]`, since we put all items that fall into cluster 0 before the second section start, and so on.

3. **Sort**: For each item, we now know into which section it needs to be put, based on its cluster assignment and the previously computed scan. We also know, at which position inside this section it needs to be stored because of the return values of the `atomicAdd`, which were saved in the counting step. By adding these two indices, we get the sorted position of each item and can simply copy the items to this position.

Listing 3.9 shows the implementation of the first and third step of counting sort in CUDA. Note that this implementation keeps the bins (`cluster_counts`) in global memory which is not optimal, as discussed in [36]. We also implement counting sort in a more optimized fashion, where the bins are modified in shared memory such that only threads in the same CUDA block perform atomic operations on the same memory location. This greatly reduces the number of conflicts and thus improves the performance. This approach was also used in [36] and the authors provide an analysis of the number of conflicts in their work.

For the second step of counting sort, we used the `exclusive_scan` function of the thrust [37] library.

```
__global__ void count(
        const long long* row_indices,
        int* membership,
        int* cluster_counts,
        long long* sort_offsets,
        int nnz,
        int k)
{
    const int index = blockDim.x * blockIdx.x + threadIdx.x;
    if (index >= nnz)
        return;
    int m = membership[row_indices[index]];
    int offset = atomicAdd(&cluster_counts[m], 1);
    sort_offsets[index] = offset;
}
__global__ void sort(const long long* col_indices,
        const float* data,
        long long* col_indices_sorted,
        float* data_sorted,
        const long long* row_indices,
        long long* row_indices_sorted,
        int* membership,
        int* cluster_count_scan,
        long long* sort_offsets,
        int nnz,
        int k)
{
    const int index = blockDim.x * blockIdx.x + threadIdx.x;
    if (index >= nnz)
        return;
    int m = membership[row_indices[index]];
    int cluster_offset = cluster_count_scan[m] + sort_offsets[index];
    data_sorted[cluster_offset] = data[index];
    col_indices_sorted[cluster_offset] = col_indices[index];
    row_indices_sorted[cluster_offset] = row_indices[index];
}
```

Listing 3.9: Counting sort in CUDA

When sorting, in most cases there is a choice to be made about what exactly is sorted in memory. One option is to move the actual data points in memory such that they are sorted by the assigned cluster, the other is to just sort the indices of the data into the desired order and later access the data indirectly. The latter option does not improve the spatial locality of the data, but still improves the temporal locality. However, the cost of moving the high-dimensional data points might outweigh the benefit of spatial locality.

We benchmark both approaches, with **indirect sort** indicating the indirect sort through indices in section 3.3. The combination with **divide before** is also tested.

Sorting by cluster assignment and column

In an attempt to further improve the locality of the atomic operations, we evaluate a solution that sorts the nonzero values of the dataset by their assigned cluster and by their column. Figure 3.4 shows the benefits this can provide when dealing with sparse data. In this figure, three sparse vectors (indicated by color) are summed up into the same cluster center. We can see, that sorting

by the column of the nonzero entries helps the spatial locality. Note that in figure 3.4, the previous optimization is included, as the sparse vectors all belong to the same cluster. In the case where this is not given, we can combine the column indices with the respective cluster assignment to form a new index. Sorting by this index will then sort the nonzero entries both by cluster assignment and column, combining the previous optimization from section 3.2 with this one.

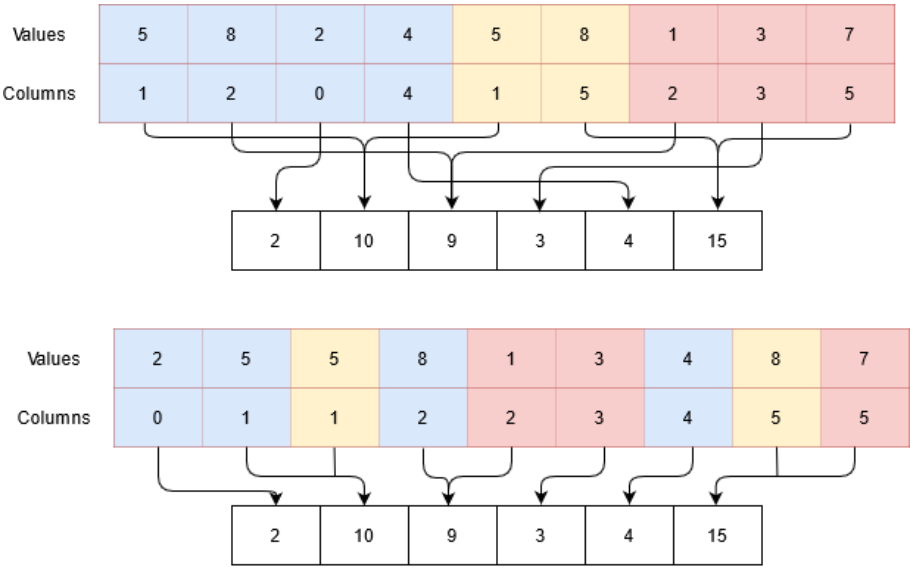

Figure 3.4: Advantage of sorting by column visualized

Unfortunately, the counting sort technique used before cannot be applied in this case, since counting the number of nonzeros for each of the $k \times d$ bins is approximately as expensive as computing the whole reduction with the naive atomic add approach. Hence, we used radix sort instead. In the benchmark section this approach is labeled **sort full atomic**.

After sorting the data in this way we can also use a segmented sum operation instead of atomic add to compute the reduction, since now values that will be summed up into the same memory location are laid out consecutively. This approach is referred to as **sort full segmented reduce**.

3.3 Benchmarks

This section presents the results of our experiments concerning the performance of our implementations. All benchmarks were performed on a system with an NVIDIA GeForce RTX 2080 Ti graphics card and an Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60GHz CPU. This CPU has 8 physical cores with 2-way multithreading.

Dataset	dimensions	nonzero entries	density
BBC	2 225 x 29 126	322 146	0.49%
Movielens	138 493 x 131 263	20 000 263	0.11%
NY Times	299 752 x 102 661	69 679 427	0.23%
SCRNA	65 662 x 26 485	126 510 394	7.27%

Table 3.1: Overview over datasets after removing empty rows

Datasets

For benchmarking purposes we use four datasets. The first and smallest, dataset is the BBC dataset presented in [38], which contains 2225 news articles from five different categories. As a preprocessing step the data is transformed using TF-IDF, which is similar to the bag of words approach described in the introduction and yields a highly sparse matrix of word frequencies.

The other three datasets are a subset of the datasets used in [15]: Movielens dataset, NY Times dataset and single-cell RNA dataset. The Movielens dataset consists of the ratings for movies given by many users. Since not every user rates each movie, this dataset is also highly sparse. Similarly to the BBC dataset, the NY Times dataset consists of news articles, however, this dataset is considerably larger than the BBC dataset. The last dataset, single-cell RNA contains gene expressions. It is the least sparse dataset in this comparison and consists of the most nonzero entries. An overview over important metrics of the datasets can be found in table 3.1. Note that the reported metrics were extracted after removing all empty rows of the datasets.

Distance

For benchmarking the distance computations, we used the `futhark bench` tool with CUDA backend. For each of the approaches, all preparations that need to be done for the given implementation (like converting to a different sparse format, sorting, etc.) are performed. The reported performances are calculated by counting the number of bytes our sequential baseline implementation accesses and then dividing the result by the average runtimes of the programs.

Unfortunately, some of our implementations were suboptimal in terms of memory consumption and ran out of memory for some datasets. These cases were excluded from the graphs to allow a better comparison of the more feasible implementations.

For each experiment, the `futhark autotune` command [20] is used to perform incremental flattening on the given algorithm and choose the version that performs best for the respective inputs.

The performance of the different implementations is displayed in figure 3.5.

We can see that for every dataset except BBC, the **flat** approach performed worst. Since it also does not seem to scale better than the other approaches with increasing k, it seems that this approach is not very suitable for the implementation of k-means. As expected, the flattening also caused the memory footprint to increase drastically, preventing us from benchmarking this approach for higher k.

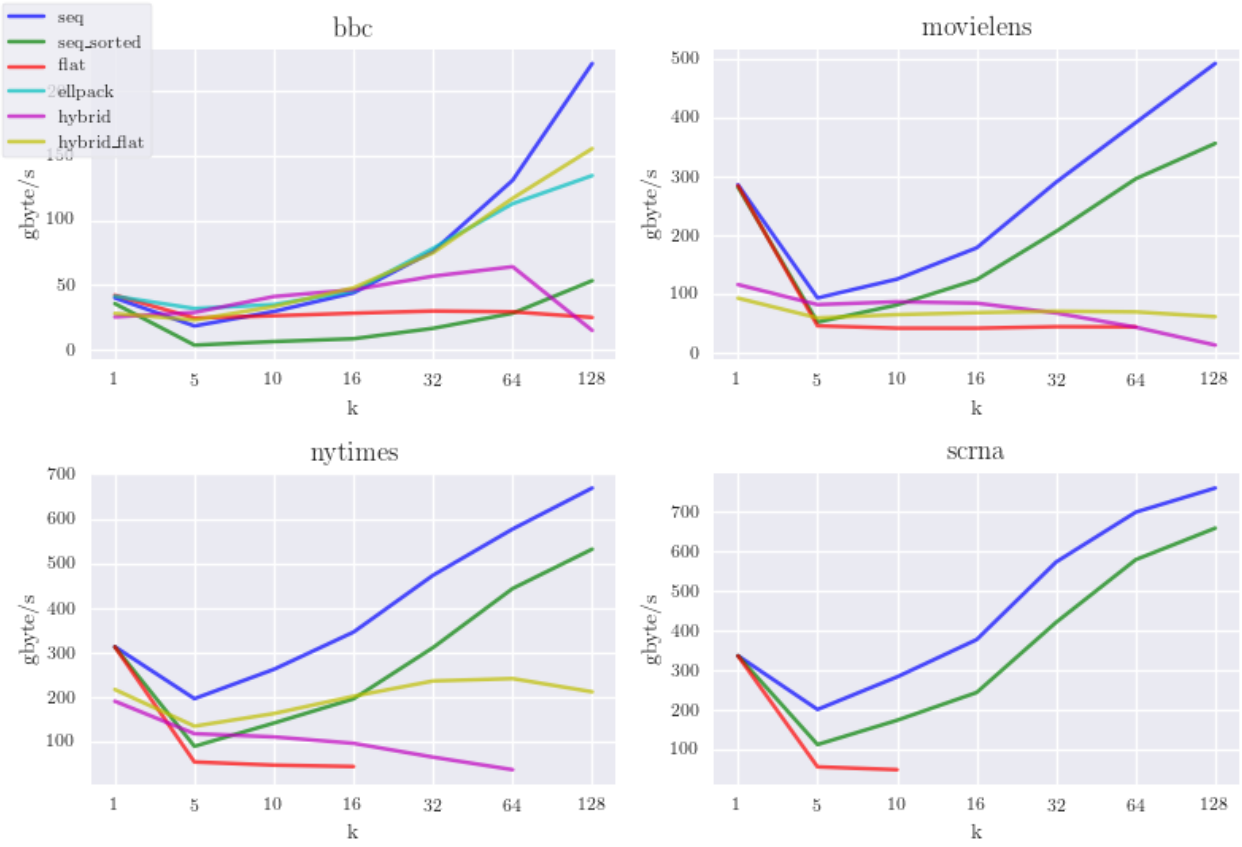

Figure 3.5: Performance of distance computations.

The **hybrid_flat** approach has a similar problem, as it also flattens the computations for the irregular part that does not fit into the regular ELLPACK matrix. However, especially for the smallest dataset (BBC) it performs better than the **flat** approach.

For smaller k the hybrid implementation that uses atomic operations for the irregular part and the **hybrid_flat** version perform similarly. However, as k increases, the locality of the atomic operations gets worse and the performance and falls behind **hybrid_flat**.

For all datasets except BBC, the **seq** and **seq_sorted** implementations are clearly the best approaches. It seems that sorting the dataset by nonzeros does not yield a significant improvement, as the overhead of sorting is very visible in the performance.

The last implementation that uses the ELLPACK format consumes most memory of all the approaches. This is expected and also the reason for using the hybrid strategies. However, for the BBC dataset the whole ELLPACK matrix fits into GPU memory, and the benchmark demonstrates how well the regular layout performs. The performance is almost identical to the sequential implementation for most k.

An interesting observation for the sequential and ELLPACK format is, that the performance significantly increases with higher k. This effect is probably caused by better memory access patterns and less thread divergence, as described in section 3.1.

Reduction

The benchmarking strategy for the reduction operation is slightly different from the distance computations. Since the cluster assignments change in every iteration, the preparations like sorting by cluster assignments also need to be executed in every iteration. Hence, we benchmark 100 iterations of the complete reduction step including sorting and report the average performance in figures 3.6 and 3.7. The former shows the higher level optimizations, while the latter focuses on the small optimizations like **indirect** and **divide before**.

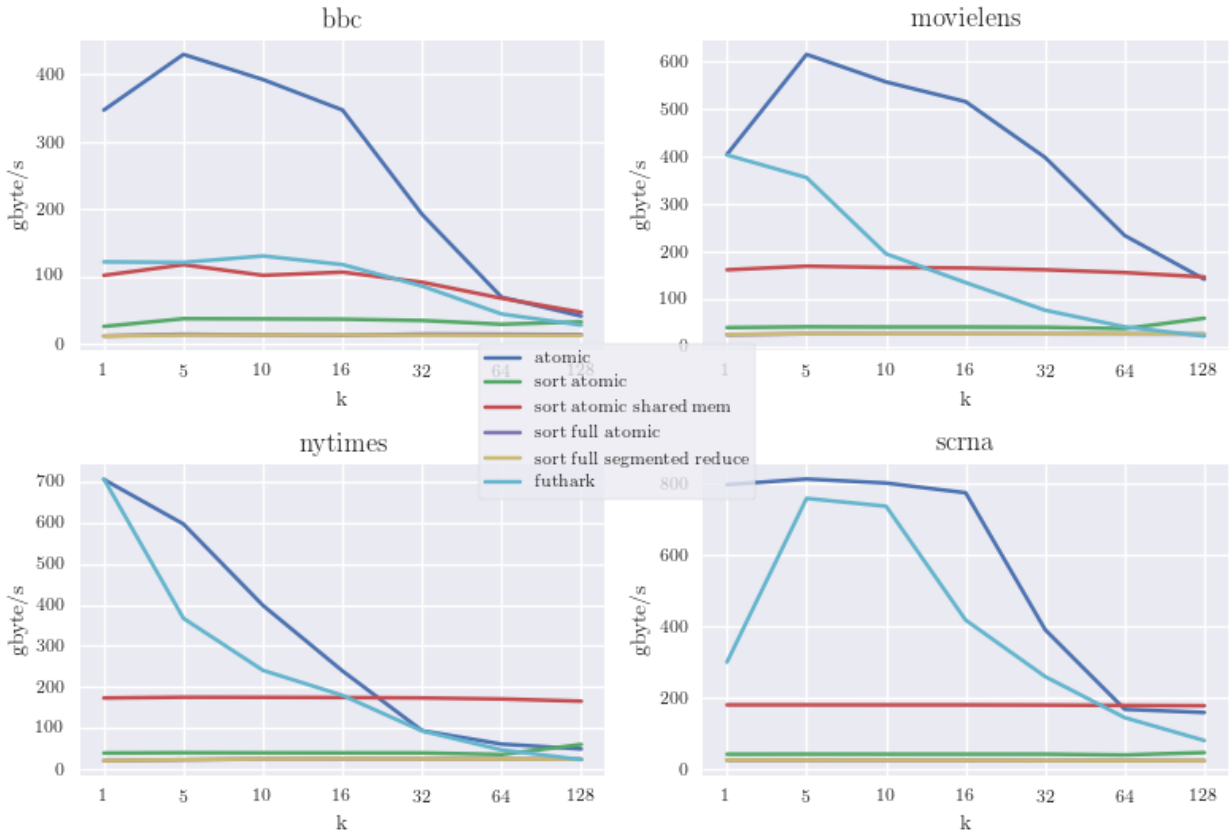

Figure 3.6: Performance of the reduce operation.

It is immediately visible that both approaches that use the full radix sort (**sort full...**) are not performing very well. The overhead of sorting all nonzero values by membership and column seems to outweigh all of its presumed

benefits. Note that this implementation uses the radix sort implementation of the thrust [37] library. The implementation of other libraries might improve the overhead of this approach, however, we decided that such an investigation is subject to future work.

On the other end, the naive baseline approach that simply uses CUDA's `atomicAdd` function seems to perform better than most approaches for small k. However, it does not scale very well as k increases, which is explained by the spatial locality getting worse as k increases. The **futhark** baseline implementation (see listing 3.7) shows similar performance characteristics, however, it performs consistently worse than the CUDA baseline.

When it comes to the implementations that sort the nonzero values by their respective cluster memberships, we observe that **sort atomic** is significantly slower than the other version (**sort atomic shared mem**). This is due to the use of atomic operations on global memory in the **Count** step of counting sort. Since the atomic operations for the sorting add nnz values into k buckets, there are many conflicts. As described in [36] and section 3.2, using shared memory greatly reduces the number of conflicts. This improves the performance, as is visible in our benchmarks.

The smaller optimizations, **divide before** and **indirect sort** do not have as much of an impact as the use of shared memory. However, **divide before** seems to scale very well with k, as the division of the dense cluster centers by the number of assigned points takes up a larger proportion of the operation. Since **divide before** does not have a large impact for small k, it is important to balance the performance benefits against the lower accuracy due to rounding errors.

Sorting only the indices of the nonzero values instead of moving the actual values seems to also benefit the performance. For all datasets, the best approach apart from the baseline implementation uses this optimization, independently of k.

There is a threshold value of k for each dataset, where the performance of the naive implementation intersects the performance of the best sorting approach. This threshold is $k = 64$ for the BBC and SCRNA datasets, $k = 32$ for the NY Times dataset and $k = 128$ for the Movielens dataset. This shows that there is no fixed threshold at which it is beneficial to switch implementation and either the threshold has to be tuned to match the given data, or further research has to be done on the underlying factors that influence this threshold.

Speedup against scikit-learn

In this section we show the results of benchmarking our full Futhark implementation of sparse k-means against the scikit-learn [7] implementation. In both cases, the input data is formatted in the CSR format.

The k-means implementation in scikit-learn uses all available CPU cores by default. Since this is representative of regular use of the library, we decided to show the speedup against this multicore version. In our case, the CPU has 8 physical cores.

Since our well-performing implementations of the reduce operation use CUDA-specific operations for the sorting step, the Futhark implementation benchmarked in this section does not include these implementations. Instead,

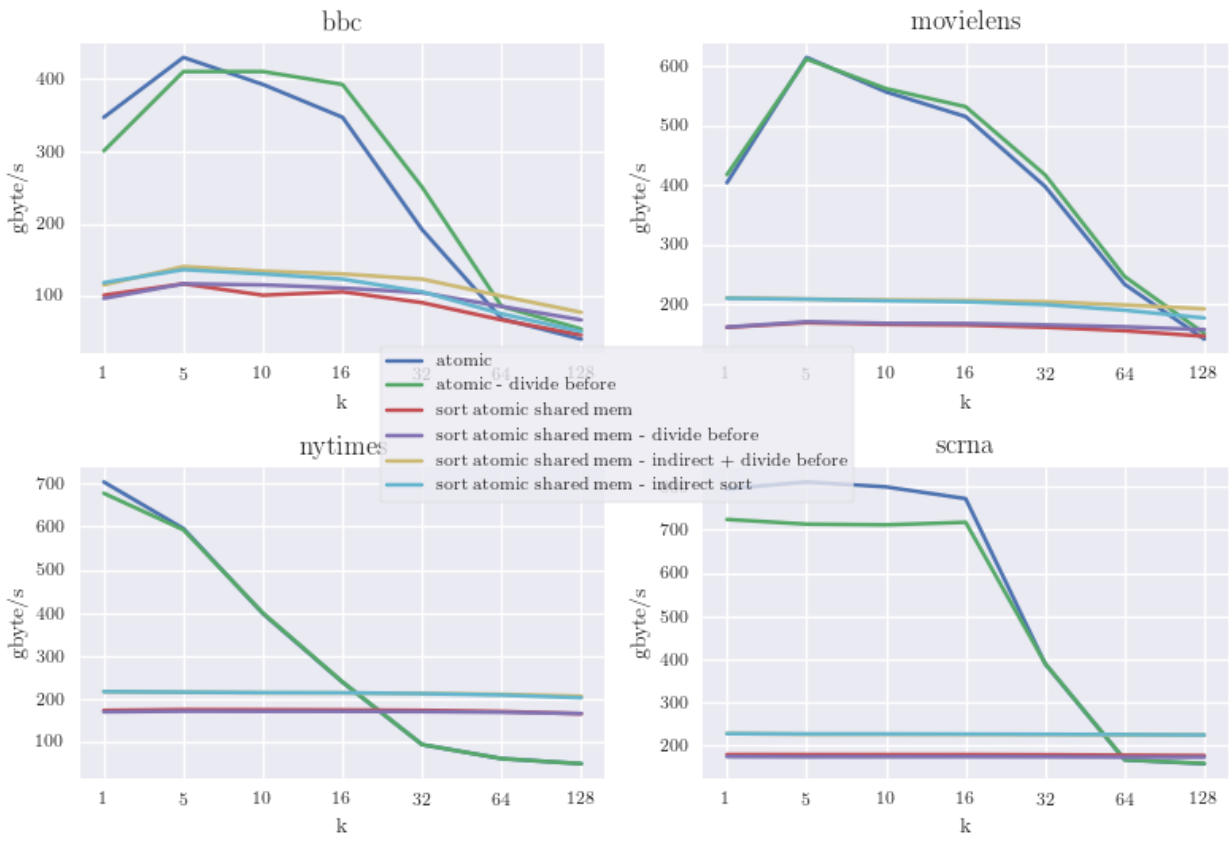

Figure 3.7: Performance of the smaller optimizations for the reduce operation.

the `reduce_by_index` construct is used, which implements efficient generalized histograms [22] and in our use case uses CUDA's `atomicAdd` internally. This implementation is not optimized for the case of sparse input data, however, it still performs well enough to produce good speedups.

In figure 3.8 the speedups of our Futhark implementations over the multicore k-means implementation of scikit-learn are shown. For all datasets, our best implementation always provides a speedup of at least factor 10 up to well over 100 for large k on the BBC dataset. Interestingly, the reduction part implementation that uses the hybrid formats seems to perform well on the BBC dataset, however, we did not investigate these implementations further since they consume too much memory to be practical. It might be possible to improve that aspect and also switch between the approaches based on the given dataset, however, that is subject to future work.

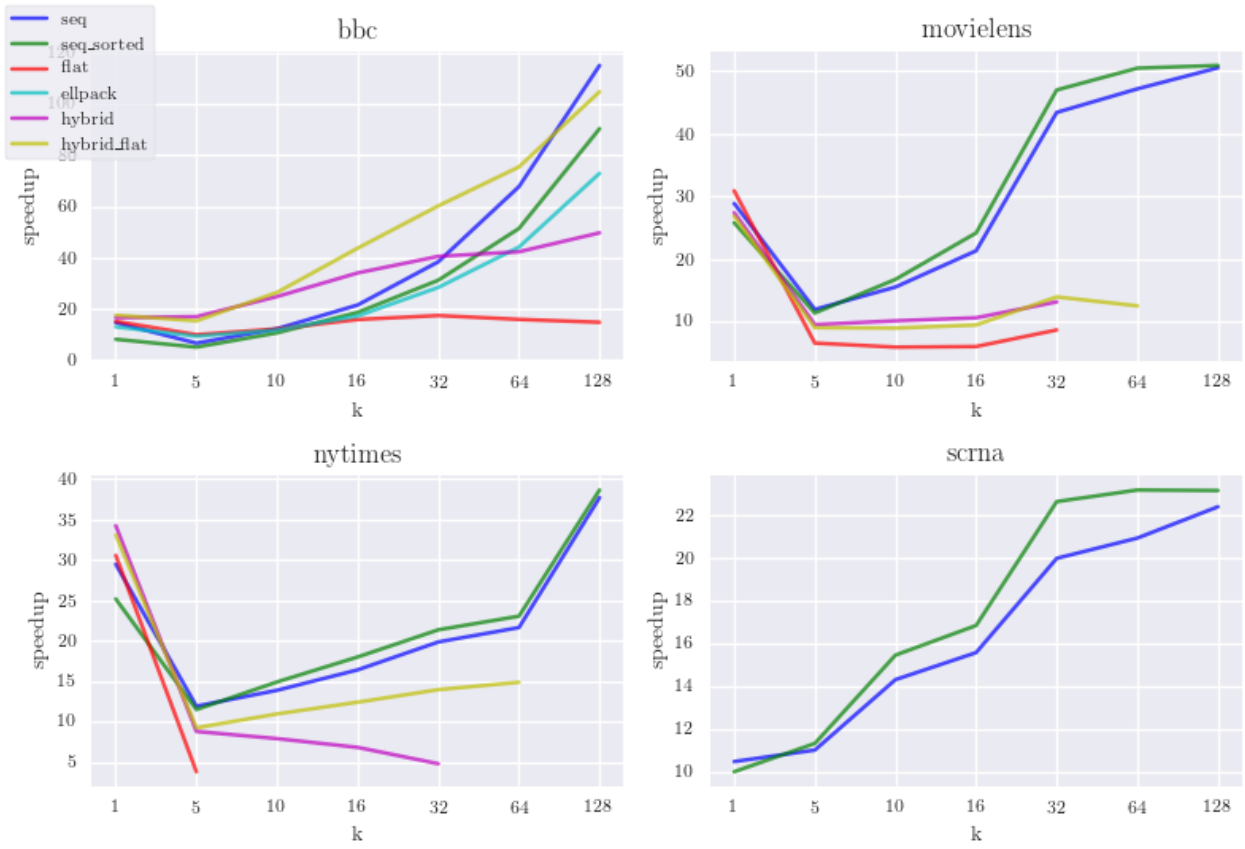

Figure 3.8: Speedups against the scikit-learn implementation

Summary of benchmarks

To summarize the results of the previous sections, we have compiled the most noteworthy observations below.

1. For the distance computation in the k-means algorithm, our implementation that sequentializes the innermost map-reduce construct yielded the best overall performance as can be seen in figure 3.5. This implementation makes use of the CSR matrix representation.

2. The best implementation for the reduction step heavily depends on the value of k. For small k, the baseline CUDA implementation is best suited, while for higher k it is beneficial to sort the data points by their cluster assignments before the reduction (see figure 3.6). All of our implementations for the reduction step favor the COO matrix representation.

3. Smaller optimizations like indirect sorting and spreading the division by n in equation 3.5 over the nonzero values can help to improve the performance slightly (see figure 3.7).

4. Our Futhark implementation achieves good speedups of at least factor 10 over the commonly used scikit-learn multicore implementation.

Chapter 4

Generalizing framework for mixtures

Since k-means in its basic form is not a suitable algorithm for clustering high-dimensional data, we investigated the possibility to transfer the learnings from the k-means implementation to more powerful models. This section presents a small framework in the Futhark language that enables the implementation of mixture models based on primitives found to be performant on high-dimensional sparse datasets. Mixture models implemented through this interface can be fitted using the EM approach.

4.1 Data representation

This framework is generic in the way it represents data. We only define certain operations on a dataset that are useful to implement mixtures, however, the implementation depends on the data representation. For example, we can implement these operations for a dense data representation, for a sparse representation that is based on the CSR format or for a sparse representation that uses any other format. The following sections specify this interface and describe the operations we found to be important for the implementation of mixtures. We also show how this interface is implemented for a sparse representation that is based on a mix of the COO and CSR formats.

The `mixture_dataset` interface

We chose to define the interface of a dataset for our framework as shown in listing 4.1.

```
module type mixture_dataset = {
    module V : real
    module I : integral
    -- the type of the nonzero values, for example f32
    -- we assume they are real values
    type val_t = V.t
    -- the type of the indices, for example i64
    type index_t = I.t

    -- the actual data representation
    type~ data_rep

    -- functions to create this representation
    -- from a dense representation or CSR format
    val mk_from_dense [n][d]: [n][d]val_t -> data_rep
    val mk_from_csr [nnz][np1]: (values: [nnz]val_t) ->
                                (col_indices: [nnz]index_t) ->
                                (pointers: [np1]index_t) ->
                                (columns: index_t)
                                -> data_rep

    val apply_k_semirings 't [k][d]: ... -- defined in the next sections
    val map_index_reduce_by_key [n]: ... -- defined in the next sections
    val map_index_reduce: ...            -- defined in the next sections

    -- normalize the rows of the dataset
    val normalize: (X: data_rep)
                           -> data_rep

    -- take the first k data points
    -- and return them in dense representation
    val take: (X: data_rep)
                -> (k: i64)
                -> [k][]val_t

    -- getters for the dimensions of the dataset
    val get_d: data_rep -> i64
    val get_n: data_rep -> i64

    -- helper function to convert the indices to i64
    val ind_to_i64: index_t -> i64
}
```

Listing 4.1: Interface of mixture_dataset

With this interface, we are free to choose the type of the data representation, as long as we can implement the defined functions on it. Of course, in the future this might have to be extended, but we show in later sections that these three functions suffice to implement at least four kinds of mixture models. These functions `apply_k_semirings`, `map_index_reduce_by_key` and `map_index_reduce` are defined and described in the next sections.

Semiring operation

In the previous explorations on the k-means algorithm, we found that the structure of our distance implementation resembles the semiring definition in [15]. The authors define the semiring in the following way:

A monoid is a semigroup consisting of an associative binary operator op and an identity element id_{op}. A semiring is a tuple containing a domain S, as well as an additive ($\oplus$) and a multiplicative ($\otimes$) monoid. The additive monoid has to be commutative, distributive and $id_{\oplus} = 0$, and the multiplicative monoid distributes over the additive monoid. Additionally, they define an annihilator as an input to the multiplicative monoid that will evaluate to 0, i.e. for all $x \in S$: $\otimes(x, 0) = \otimes(0, x) = 0$.

As an example, the standard dot product can be expressed as a semiring with the additive monoid being $(+, 0)$, and the multiplicative monoid being $(\times, 1)$. In our implementation we do not make use of most properties of this semiring definition. However, we make use of the assumption that the multiplicative operator has an annihilator of 0 for its first argument. This means, that if the first argument of the function is 0, then the result of this operation will be 0. We also assume that the additive monoid is associative, since it may be passed to Futhark's `reduce` construct.

Using this construct, we were able to implement the distance computation for k-means, the expectation step of Gaussian mixture models, and the expectation step of von Mises-Fisher mixture models. The authors of [15] show that this construct can be used to express many common distance functions, and so we hope that it is also general enough to allow for more types of mixture models to be implemented.

The signature of the function in Futhark is:

```
val apply_k_semirings 't [k][d]:
                      (X: data_rep)
                      -> (k_vectors: [k][d]t)
                      -> (mul: val_t -> t -> val_t)
                      -> (add_op: val_t -> val_t -> val_t)
                      -> (add_ne: val_t)
                      -> [][k]val_t
```

For each of the k d-dimensional dense vectors and each of the n data points in `X`, this function applies the multiplicative operation `mul` to each nonzero value and the corresponding element of the dense vectors. The results of this operation are reduced with `add_op` for each data point, resulting in a dense n x k matrix.

This operation is visualized in figure 4.1 for a single data point, $k = 3$ and $d = 4$.

This visualization also demonstrates the use of the assumption about the annihilator. Since the data point in the example has no nonzero entry for column 1 (marked in light blue), the values for this column in the dense cluster centers are not used.

Reduce operation

The reduce operation also has potential for generalization. It comes into play when summing across the rows of a dataset, for example when calculating the mean of the data. Here, a distinction has to be made between hard and

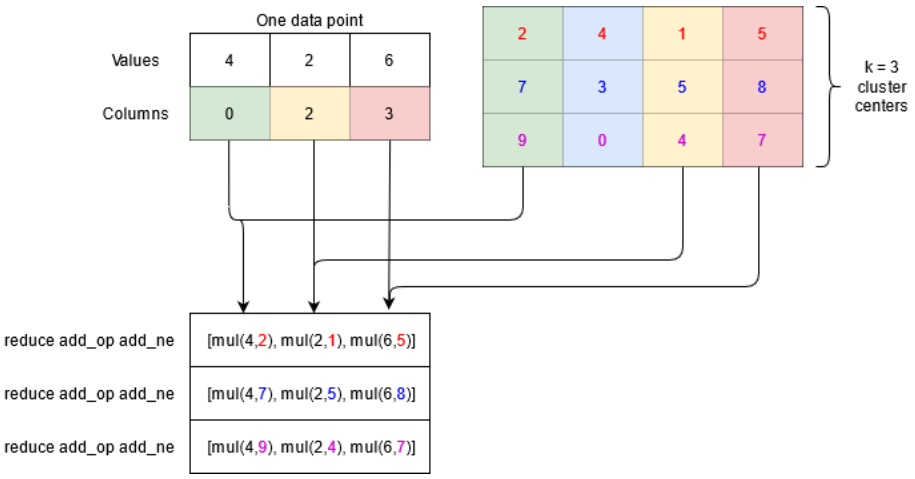

Figure 4.1: Visualization of the semiring operation.

soft clustering algorithms. In the case of hard clustering (like in the classical k-means), each data point contributes to exactly one of the k cluster centers. This means, that in the reduction step we have to take the assigned cluster into account. In contrast, in soft clustering implementations every data point contributes to each of the k cluster centers, which leads to k full reductions.

Based on this observation, our framework provides two functions with the following signatures:

```
val map_index_reduce_by_key [n]:
    (X: data_rep)
    -> (keys: [n]index_t)
    -> (n_distict_keys: i64) -- how many bins do we have for keys
    -> (map_f: val_t -> index_t -> index_t -> val_t)
    -> (red_op: val_t -> val_t -> val_t)
    -> (ne: val_t)
    ->  [n_distict_keys][]val_t

val map_index_reduce:
    (X: data_rep)
    -> (map_f: val_t -> index_t -> index_t -> val_t)
    -> (red_op: val_t -> val_t -> val_t)
    -> (ne: val_t)
    -> [d]val_t
```

The first function (`map_index_reduce_by_key`) first applies the passed map function `map_f` to each nonzero value of the dataset and then reduces the data points by the keys passed to the function. Thus, the result of this function will have `n_distinct_keys` rows and d columns for a d-dimensional dataset. The initial map step provides the opportunity to associate the nonzero values with other values based on their row and column indices. Thus, `map_f` takes one value of type `val_t` that represents the nonzero value and two

values of type `index_t` that represent the row and column indices of the nonzero value. As `map_index_reduce_by_key` function distributes the data points across the keys, it is useful in hard clustering algorithms.

The second function has similar semantics, except that it does not take any keys into account. It always reduces all data points to one dense vector of length d for a d-dimensional dataset, after applying the `map_f` as described above. This means, that for soft clustering algorithms we have to apply this function once for each of the k components to get a $k \times d$ matrix of results (see section 4.4 for an example).

Implementation for sparse dataset

To implement the `mixture_dataset` interface for the case of a sparse dataset, we first need to define the types of the interface. Listing 4.2 shows these types.

```
module sparse_dataset (R: real) (I: integral): (mixture_dataset
        with val_t = R.t
        with index_t = I.t
    ) = {
    module V = R
    module I = I
    type val_t = R.t
    type index_t = I.t
    type~ data_rep =
        {
            n: index_t,
            d: index_t,
            coo: [](val_t, index_t, index_t),
            pointers: []index_t
        }

    ...
```

Listing 4.2: Types of the sparse dataset implementation

The first few lines of listing 4.2 express that a sparse dataset is generic in terms of two other modules. One module `R` implementing the `real` interface defined by Futhark, and the other, `I`, implementing the `integral` interface. This allows us to generalize this implementation in terms of the used precision for the values and indices stored in this dataset. For example, we can pass the modules `f32` and `i64` in the instantiation of this type to create a dataset that holds 32-bit float values and uses integral indices with 64-bit precision.

We also specify, that this module implements the module type (i.e. interface) `mixture_dataset`. The notation `with val_t = R.t` denotes that the type `val_t` is exactly the type `t` from the module `R`. Analogously, we specify this relation between `index_t` and `I`.

To implement the semiring and reduce operations in a way we found to be efficient in the previous chapter, we chose our data representation `data_rep` to contain a mix of the CSR and COO format. This format simply extends the CSR format by additionally providing `row_indices`. This information is redundant, since it can be computed from the `pointers` array. But

as we use this information in multiple places, we decided that computing `row_indices` once and storing it is a better choice than computing this information multiple times.

This data representation is organized as a Futhark record, i.e. a named tuple. We can, for example, extract `n` from a tuple `X` of type `data_rep` by writing `let n = X.n`. We also grouped together the three arrays that have length nnz, i.e. `values`, `row_indices` and `col_indices` to an array of tuples. This adds the constraint to this type that the three arrays actually have the same length. In Futhark, arrays of tuples are stored as tuple of arrays internally, so we do not change the performance characteristics of the implementation by doing so. This representation is stored in the `coo` field of the `data_rep` record. We can thus extract the fifth nonzero value of our dataset along its row and column index as follows:

```
-- X is a variable of type data_rep
let (value, row_index, col_index) = X.coo[5]
```

We can also turn the `coo` field back into the three `values` `row_indices` and `col_indices` arrays with the following line of code that executes in constant time:

```
-- X is a variable of type data_rep
let (values, row_indices, col_indices) = unzip3 X.coo
```

Now we can implement the semiring and reduce functions based on these types. Listing 4.3 shows the implementation of the semiring function. It highly resembles our sequentialized implementation of the k-means distance computation (see listing 3.4), except for the introduction of the `add_op` and `pairwise_op` functions that allow to generalize this construct to be used by other kinds of mixtures, and the use of the `data_rep` type.

```
let apply_k_semirings 't [k][d] (X: data_rep)
            (mixture_vectors: [k][d]t)
            (pairwise_op: val_t -> t -> val_t)
            (add_op: val_t -> val_t -> val_t)
            (add_ne: val_t) : [][k]val_t =
    let (values, _, col_indices) = unzip3 X.coo
    in map (\row ->
        map (\vector->
            let index_start = I.to_i64 X.pointers[row]
            let nnz =  (I.to_i64 X.pointers[row+1]) - index_start
            let row_dist = add_ne
            let row_dist =
            loop row_dist for j < nnz do
                let element_value = values[index_start+j]
                let column = I.to_i64 col_indices[index_start+j]
                let vector_value = vector[column]
                let value = pairwise_op element_value vector_value
                in add_op row_dist value
            in row_dist
        ) mixture_vectors
    ) (iota (I.to_i64 X.n))
```

Listing 4.3: Implementation of semiring function for `sparse_dataset`

The implementations of `map_index_reduce_by_key` and `map_index_reduce` are presented in listings 4.4 and 4.5. The former corresponds to the reduction step investigated in chapter 3 and uses Futhark's `reduce_by_index` function in the same way as it is used in listing 3.7. The function `map_index_reduce` is even simpler, as it simply passes the nonzero values and column indices to `reduce_by_index`. Both implementations apply `map_f` to all nonzero values of the dataset before the reduction.

These implementations are not very complex, however, they use the intrinsic data representation to perform the reduction. Since we want to abstract from this representation, it is necessary to define these wrapper functions. Additionally, we found in chapter 3 that there are several optimizations for `map_index_reduce_by_key`, which we only implemented in CUDA since they rely on CUDA specific functions. In later versions of Futhark there might be a way to directly call CUDA kernels from Futhark, so eventually, faster implementations in Futhark will be possible.

```
let map_index_reduce_by_key [n]
        (X: data_rep)
        (keys: [n]index_t)
        (n_distinct_keys: i64)
        (map_f: val_t -> index_t -> index_t -> val_t)
        (red_op: val_t -> val_t -> val_t)
        (ne: val_t) : [n_distinct_keys][]val_t =
    let d = I.to_i64 X.d
    let (values, row_indices, col_indices) = unzip3 X.coo
    let map_vals = map3 map_f values row_indices col_indices
    let flat_results = reduce_by_index (replicate (n_distinct_keys*d) ne)
                        red_op ne
                        (map2 (\row col -> X.d * keys[row] + col)
                            row_indices col_indices)) map_vals
        in unflatten n_distinct_keys d flat_results
```

Listing 4.4: Implementation of the `map_index_reduce_by_key` function for `sparse_dataset`

```
let map_index_reduce
        (X: data_rep)
        (map_f: val_t -> index_t -> index_t -> val_t)
        (red_op: val_t -> val_t -> val_t)
        (ne: val_t) : [d]val_t =
    let d = I.to_i64 X.d
    let (values, row_indices, col_indices) = unzip3 X.coo
    let map_vals = map3 map_f values row_indices col_indices
        in reduce_by_index (replicate d ne)
                           red_op ne
                           col_indices
                           map_vals
```

Listing 4.5: Implementation of the `map_index_reduce` function for `sparse_dataset`

4.2 The `mixture` interface

To accommodate different mixtures while providing a clean interface to the end-user, we define a fairly general interface for the mixture implementations.

To define a mixture the following types and functions have to be specified:

```
module type mixture = {
    -- type of the parameters of this distribution
    type~ theta

    -- type of the evaluation result of the distribution
    type~ posteriors

    -- type of the data representation
    type~ data

    -- evaluates the distribution with given data and parameters
    -- corresponds to the expectation step of the EM framework
    val eval: data -> theta -> posteriors

    -- maximization step of the EM framework
    val maximize: data -> theta -> posteriors -> theta

    -- and a few more smaller functions for
    -- initializing the parameters,
    -- checking for convergence,
    -- retrieving results, etc.
    ...
}
```

Mixtures that implement this interface can then be fitted through a simple function that calls `eval` and `maximize` in a loop (see section 4.3), until convergence is reached.

4.3 The `em` module

The final building block of our framework is the `em` module. It contains the main EM loop and fits the mixture models to data through calls to the `mixture` interface. This module only has one function, `fit`, and its implementation is shown in listing 4.6. Apart from iteratively calling the mixture's `eval` and `maximize` functions until convergence, it also keeps track of the objective function value throughout the fitting process and reports the values to the user.

```
import "mixture"

-- we parametrize over modules of type mixture
module em (distribution: mixture) = {

  let fit (threshold: distribution.conv_limit_t)
          (k: i32)
          (max_iterations: i32)
          (X: distribution.data):
            (distribution.theta, i32, distribution.posteriors, []f64) =
      let k = i64.i32 k

      -- we keep track of the objective function value through iterations
      let objective_history = replicate (i64.i32 max_iterations) (f64.f32 0)

      -- initial guess of parameters based on data and k
      let params = distribution.init_theta X k

      -- Initial assignment of posteriors.
      let posterior = distribution.eval X params

      let converged = false
      let i = 0
      let (posts,params,_,i,objective_history) =
        loop (posterior, params, converged, i, objective_history)
        while !converged && i < max_iterations do

          let new_params = distribution.maximize X params posterior

          let new_posterior = distribution.eval X new_params

          let converged = distribution.check_converged
                                    posterior new_posterior threshold

          let objective_value = distribution.compute_objective_function
                                                          new_posterior
          let objective_history[i] = objective_value

          in (new_posterior, new_params, converged, i+1, objective_history)
      in (params, i, posts, (take (i64.i32 i) objective_history))
}
```

Listing 4.6: Implementation of the `em` module

4.4 Implementations of mixtures

With the three functions (`apply_k_semirings` , `map_index_reduce_by_key` and `map_index_reduce`) defined in section 4.1, it is possible to abstract from the representation of the dataset while providing good performance in the case of sparse datasets. The following sections demonstrate how the functions can be used to implement various mixture algorithms.

K-means

Since k-means is a specialization of the Gaussian mixture, we can also express it through this framework.

The parameters `theta` for k-means are the k dense mean vectors. Since k-means is a hard clustering algorithm, the evaluation step results in n labels, indicating the cluster assignment for each point. Thus, `posteriors` is an array of cluster indices. Finally, for the type of the data representation, we can use Futhark's parametric module system to generalize it for different implementations. For example, a sparse dataset can have a different data representation than a dense dataset.

In Futhark, we can thus define the types of our k-means mixture as follows:

```
module k_means_mixture (dataset: mixture_dataset): (mixture
                    with data = dataset.data_rep) = {
    type~ theta = [][]f32 -- the cluster centers
    type~ posteriors = []i64 -- the cluster assignments
    type~ data = dataset.data_rep
    ...
}
```

The first line indicates that this module is parametric with respect to the dataset (and thus, data representation) we use. We can instantiate the `k_means_mixture` module with any module that implements the `mixture_dataset` interface defined in section 4.1. We also specify that this module implements the `mixture` interface. The statement `with data = dataset.data_rep` specifies that the type `data` of the `k_means_mixture` is exactly the same type as the type `data_rep` in the `dataset` parameter.

Since the sizes of the types `theta` and `posteriors` are unknown at compile time, we have to use Futhark's size-lifted types (indicated by $\sim$). This restricts the usage of this type, such that we cannot create an array of elements of this type. I.e. Futhark does not allow the type `[]data` to be used. Without this restriction, there would be no guarantee that elements of an array of `data` have the same size, and thus we could form an irregular array, which is disallowed in Futhark.

The `eval` function of k-means computes the distances of each point to the cluster centers and then takes the minimum over it. We can use the semiring-like construct introduced above to compute these distances:

```
let eval [k] (X: data) (params: theta_sized [k][]): posteriors =
    -- sum mu^2
    let cluster_squared_sum = map (\c -> sum (map (\x -> x*x) c)) params

    -- sum x^2 - 2 * x * mu
    let mul_op =  \data_val cluster_val ->
                (data_val - (2 * cluster_val)) * data_val
    let distances = dataset.apply_k_semirings X params mul_op (+) 0

    let distances = ... -- combine the two partial distances
    -- argmin over distances
    in map argmin distances
```

The implementation of `maximize` uses the `map_index_reduce_by_key` function and is similarly short:

```
let maximize [k] (X: data) (_: theta_sized [k][]) (posts: posteriors): theta =
    -- we do not need the map functionality
    -- -> pass the identity function
    let id = (\x _ _ -> x)
    let cluster_sums =
        dataset.map_index_reduce_by_key X posts k id (+) 0

    -- count elements for each cluster
    let ones = replicate n 1
    let center_counts = reduce_by_index (replicate k 0) (+) 0 posts ones
    -- divide cluster sum by number of elements
    in map2 (\center count ->
                map (\x -> x / count) center
            ) cluster_sums center_counts
```

We can see that for this implementation, there is no need to deal with the intrinsics of the sparse data representation, and we can even generalize it such that we can interchange sparse and dense datasets as parameter to this module, as long as they implement the constructs defined before.

Spherical k-means

Spherical k-means [6] is a simple modification of k-means that aims to improve the quality of the clustering for directional data. Since the document clustering and gene expression datasets we used for benchmarking fall into this category [1], spherical k-means is a modification of k-means that is relevant for this project.

The implementation of spherical k-means is very straightforward. The only differences to the classical k-means algorithm are

1. The input data has to be normalized.
2. The cluster centers have to be normalized in each iteration.

The dataset interface of our framework provides a function for normalizing the data, which takes care of the first requirement.

The second requirement does not concern the sparse data representation but rather the dense cluster centers. This normalization can be implemented in a few lines of Futhark code:

```
let normalize_cluster_centers [k][d] (centers: [k][d]f32): [k][d]f32
    let cluster_magnitudes = map (\c ->
                                  let c_squared = map (\x -> x * x) c
                                  in sqrt (sum c_squared)
                                  ) centers
    let normalized_clusters = map2 (\center center_magnitude ->
                                  map (\x -> x / center_magnitude) center
                              ) centers cluster_magnitudes
        in normalized_clusters
```

Gaussian mixture model

We also provide an implementation of the Gaussian mixture model in our framework. For the sake of simplicity, our current implementation assumes that the covariance matrix is diagonal. The limitation is motivated by the need to invert the $d \times d$ covariance matrix, which is prohibitively expensive ($O(d^3)$ when using Gaussian elimination [39]). Especially when applying this algorithm to high-dimensional data, the simplification not only helps shorten the runtime of the algorithm, but also reduce the space requirements. For datasets with $d \approx 100000$ and $k = 5$, storing these matrices in single precision would consume around 200 gigabyte of memory, which is more than most modern GPUs can hold. In contrast, diagonal matrices are cheap to invert ($O(d)$) and only require the storage of the d diagonal entries. Storing these in single precision for $d \approx 100000$ and $k = 5$ consumes only 2 megabyte.

This simplification also improves the performance of the implementation and is usually an option in implementations of the algorithm, for example in scikit-learn [7].

Expectation

In the expectation step of the algorithm, which evaluates the probability distribution for all data points, often the logarithm of the likelihood is calculated. This improves numerical stability and readability of the code, since multiplications turn into additions and the exponential of the Gaussian function disappears.

We show the derivation of the log-likelihood for a d-dimensional multivariate Gaussian distribution with diagonal $\mathbf{\Sigma}$ starting from

$$\log p(\boldsymbol{x_i}|\boldsymbol{\mu}, \mathbf{\Sigma}) = \log \left(\frac{1}{\sqrt{(2\pi)^d \det(\mathbf{\Sigma})}} e^{-\frac{1}{2}(\boldsymbol{x_i}-\boldsymbol{\mu})^T \mathbf{\Sigma}^{-1}(\boldsymbol{x_i}-\boldsymbol{\mu})} \right) \tag{4.1}$$

Using $\log \frac{1}{\sqrt{x}} = -\frac{1}{2}\log x$

$$= -\frac{1}{2}\log\left((2\pi)^d \det(\boldsymbol{\Sigma})\right) + \log\left(e^{-\frac{1}{2}(\boldsymbol{x_i}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{x_i}-\boldsymbol{\mu})}\right) \quad (4.2)$$

For the case of $\boldsymbol{\Sigma}$ being diagonal, a few simplifications can be made.

Let $\sigma_1^2, \ldots, \sigma_d^2$ be the diagonal entries of the $d \times d$ matrix Σ.

Then $\boldsymbol{\Sigma}^{-1} = \begin{bmatrix} \frac{1}{\sigma_1^2} & 0 & \ldots & 0 \\ 0 & \frac{1}{\sigma_2^2} & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & \frac{1}{\sigma_d^2} \end{bmatrix}$.

It holds that $(\boldsymbol{x_i} - \boldsymbol{\mu})^T\boldsymbol{\Sigma^{-1}}(\boldsymbol{x_i} - \boldsymbol{\mu}) = \sum_{j=0}^{d} \frac{1}{\sigma_j^2}(x_{i_j} - \mu_j)^2$ and $\det(\boldsymbol{\Sigma}) = \prod_{j=1}^{d} \sigma_j^2$. Thus, we get

$$\log p(\boldsymbol{x_i}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = -\frac{1}{2}\left(d\log(2\pi) + \log\left(\prod_{j=1}^{d}\sigma_j^2\right) + \sum_{j=0}^{d}\frac{1}{\sigma_d^2}(x_{i_j} - \mu_j)^2\right) \quad (4.3)$$

$$= -\frac{1}{2}\left(d\log(2\pi) + \sum_{j=1}^{d}\log(\sigma_j^2) + \sum_{j=0}^{d}\frac{1}{\sigma_j^2}(x_{i_j} - \mu_j)^2\right) \quad (4.4)$$

The first term $d\log(2\pi)$ is a scalar that is trivially computed, the second term $\sum_{j=1}^{d}\log(\sigma_j^2)$ is a simple map-reduce over the diagonals of the covariance matrix and the third term $(\boldsymbol{x_i} - \boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{x_i} - \boldsymbol{\mu})$, sometimes referred to as the squared Mahalanobis distance, can be implemented through our semiring construct.

Analogously to the Euclidean distance, we can rewrite the computation of $\sum_{j=0}^{d}\frac{1}{\sigma_j^2}(x_{i_j} - \mu_j)^2$ as $\sum_{j=0}^{d}\frac{1}{\sigma_j^2}\left(x_{i_j}^2 - 2x_{i_j}\mu_j + \mu_j^2\right)$ and then distribute the divisions by σ_j^2 over it:

$$\sum_{j=0}^{d}\frac{x_{i_j}^2}{\sigma_j^2} - \frac{2x_{i_j}\mu_j}{\sigma_j^2} + \frac{\mu_j^2}{\sigma_j^2}$$

Now we can split the computations into two parts again. One part computes $\sum_{j=0}^{d}\frac{\mu_j^2}{\sigma_j^2}$, which does not depend on the data, and the other computes $\sum_{j=0}^{d}\frac{x_{i_j}^2}{\sigma_j^2} - \frac{2x_{i_j}\mu_j}{\sigma_j^2}$ through the semiring construct.

The Futhark code implementing the squared Mahalanobis distance for diagonal $\boldsymbol{\Sigma}$ is presented in listing 4.7.

```
let compute_mahalanobis_squared [n][d][k]
                    (X: data)
                    (mus: [k][d]f32)
                    (sigmas: [k][d]f32):
                        [n][k]f32
    let mu_squared_scaled_sum = map2 (\mus sigs ->
            sum (map2 (\m s -> m * m / s) mus sigs)
        ) mus sigmas

    let mul_op = \data_val theta_tup ->
            let mu = theta_tup.0
            let sigma = theta_tup.1
            in ((data_val - 2 * mu) * data_val) / sigma

    -- zip mus and sigmas to allow access from semiring function
    let mu_sigmas = map2 (\mus sigmas ->
                        zip mus sigmas
                    ) mus sigmas

    let mahalanobis_sq_partial =
            dataset.apply_k_semirings X mu_sigmas mul_op (+) 0

    -- combine the partial computations...
        in map (\row
                  map2 (+) row mu_squared_scaled_sum
                ) mahalanobis_sq_partial
```

Listing 4.7: Futhark implementation of the squared Mahalanobis distance

Maximization

In the maximization step we use the result of the expectation step to estimate the new parameters $\hat{\alpha}_h, \hat{\boldsymbol{\mu}}_h$ and $\hat{\boldsymbol{\Sigma}}_h$. For this, we need to compute the quantities

$$p(z_i = h|\boldsymbol{x_i}, \Theta) = \frac{\alpha_h f(\boldsymbol{x_i}, \theta_h)}{\sum_{l=1}^{k} \alpha_l f(\boldsymbol{x_i}, \theta_l)}$$

(see equation 2.4). Since we deal with logarithmic probabilities, the right-hand side changes to

$$\log \alpha_h + \log f(\boldsymbol{x_i}, \theta_h) - \log \sum_{l=1}^{k} \alpha_l f(\boldsymbol{x_i}, \theta_l)$$

We observe that the $\alpha_l f(\boldsymbol{x_i}, \theta_l)$ in the sum can be written as $\exp(\log \alpha_l + \log f(\boldsymbol{x_i}, \theta_l))$ and get

$$\log \alpha_h + \log f(\boldsymbol{x_i}, \theta_h) - \log \sum_{l=1}^{k} \exp(\log \alpha_l + \log f(\boldsymbol{x_i}, \theta_l))$$

Note that it is possible to reuse the terms $\log \alpha_h + \log f(\boldsymbol{x_i}, \theta_h)$ for the denominator. Listing 4.8 shows the calculation of this quantity in Futhark. The

`logsumexp` function used here first maps the exponential function over an array, sums up the result and finally computes the logarithm of the sum, i.e. $\text{logsumexp}(\boldsymbol{x}) = \log \sum_{l=1}^{k} \exp(x_l)$[1].

```
let maximize [k] (X: data) (_: theta)
                 (log_posts: posteriors): -- the results of the eval step
                    theta =
    -- log_posts is an n x k array, containing the logarithmic
    -- posterior probabilities for each data point and component
    let log_post_scaled = map (\ps
                                map (\p alpha -> (log alpha) + p) ps alphas
                              ) log_posts
    let log_prob_sums = map logsumexp log_post_scaled

    -- compute the normalization
    let normalized_log_probs = map2 (\prob logprobsum ->
                                      map (\x -> x - logprobsum) prob
                                    ) log_post_scaled log_prob_sums
    ...
```

Listing 4.8: Futhark implementation of the scaled, normalized posterior probabilities in the maximize function of the Gaussian mixture.

Before finally using this quantity in the estimation of the new parameters, we need to convert it back from the logarithmic scale by applying the exponential function. We also need to compute the sum of these probabilities over all data points:

```
    ...
let posteriors = map (\row ->
    map (exp) row
) normalized_log_probs
let posterior_sum = map (\col ->
    sum col
) (transpose posts)
    ...
```

Now we have calculated all required quantities to compute the new parameters:

$$\hat{\alpha}_h = \frac{1}{n} \sum_{i=1}^{n} p(z_i = h | \boldsymbol{x_i}, \Theta)$$

turns into

```
-- n is the number of data points
let new_alphas = map (\alpha -> alpha / n) posterior_sum
```

[1] `logsumexp` is not only a convenience function, but also helps the numerical stability of our implementation. Consider $\exp(100) \simeq 1.97 \times 10^{434}$. This value is already outside the limits of a 64-bit float representation. However, we can use the identity $\log \sum_{l=1}^{k} \exp(x_l) = x_{max} + \log \sum_{l=1}^{k} \exp(x_l - x_{max})$, where $x_{max} = \max_i x_i$ to shift the arguments of the exponential to be at most 0 and make overflows impossible.

The computation of $\hat{\boldsymbol{\mu}}_{\boldsymbol{h}}$ is a bit more complicated and requires the use of `map_index_reduce` since we compute the sum over $\boldsymbol{x}_i$, i.e. all of our data points:

$$\hat{\boldsymbol{\mu}}_{\boldsymbol{h}} = \frac{\sum_{i=1}^{n} \boldsymbol{x_i} p(z_i = h|\boldsymbol{x_i}, \Theta)}{\sum_{i=1}^{n} p(z_i = h|\boldsymbol{x_i}, \Theta)}$$

This quantity can be expressed through the `map_index_reduce` construct as follows:

```
    ...
-- d is the number of columns of our data
let new_mus = map2 (\h post_sum ->
    let mu_unscaled = dataset.map_index_reduce X d
                        (\v row _ -> v * posteriors[row, h]) (+) 0
        in map (\x -> x / post_sum) mu_unscaled
) (iota k) posterior_sum

let new_sigmas = ... -- omitted
    in (new_alphas, new_mus, new_sigmas)
```

Note that we use the `map_f` argument of `map_index_reduce` to access the posterior probabilities based on the row index of each nonzero value in the dataset.

The new $\hat{\boldsymbol{\Sigma}}_h$ also make use of the `map_index_reduce` construct. Since our implementation for the computation of the new $\hat{\boldsymbol{\Sigma}}_h$ uses this construct similarly as the computation of $\hat{\boldsymbol{\mu}}_{\boldsymbol{h}}$ above, the details are omitted for the sake of brevity. One notable difference is, though, that for computing $\hat{\boldsymbol{\Sigma}}_h$, `map_f` uses both its row *and* column arguments to access the posterior probabilities and the previously computed $\hat{\boldsymbol{\mu}}_{\boldsymbol{h}}$ depending on the indices of the nonzero values:

```
    ...
let map_f = (\v row col ->
    let post_fac = posteriors[(row), i]
    in (v - 2 * new_mus[i, col]) * v * post_fac
)
    ...
```

Von Mises-Fisher mixture model

The von Mises-Fisher mixture model, as taken from [1], uses the multivariate von Mises-Fisher distribution in a mixture model. The d-variate von Mises-Fisher distribution is defined in [1] as:

$$f(\boldsymbol{x}|\boldsymbol{\mu}, \kappa) = \frac{\kappa^{d/2-1}}{(2\pi)^{d/2-1} I_{d/2-1}(\kappa)} e^{\kappa \boldsymbol{\mu}^T \boldsymbol{x}}$$

Under the constraints that $||\boldsymbol{x}|| = ||\boldsymbol{\mu}|| = 1$, $\kappa \geq 0$ and $d \geq 2$.
$I_r(\kappa)$ represents the modified Bessel function of the first kind and order r, applied to κ.

As in the case of Gaussian mixture models, a von Mises-Fisher mixture model consists of k distributions, each with their own α_h, $\boldsymbol{\mu}_h$ and κ_h.

Expectation

Similarly to evaluating the Gaussian mixture model, we have to compute some normalizing scalar $\left(\frac{\kappa^{d/2-1}}{(2\pi)^{d/2-1} I_{d/2-1}(\kappa)}\right)$, as well as the expression inside the exponential function $\left(\kappa \boldsymbol{\mu}^T \boldsymbol{x}\right)$. The normalizing scalar does not depend on the dataset and is challenging to implement due to the computation of the modified Bessel function. CUDA's math library supplies this function, however, it is not (yet) accessible through Futhark. Therefore, the implementation of this computation is incomplete at the time of writing this thesis.

The exponential part computes a simple dot product with a scaling factor κ applied afterwards. Expressing this through the semiring construct is trivial, as the dot product can be expressed as a semiring with $(+, 0)$ as additive monoid and $(\times, 1)$ as multiplicative monoid. Listing 4.9 shows this implementation in Futhark.

```
let compute_vmf_log_prob [n][d][k] (X: data)
                                   (mus: [k][d]f32)
                                   (kappas: [k]f32):
                                       [n][k]f32
    let unscaled_log_probs = dataset.apply_k_semirings X mus (*) + 0
    let scaled_log_probs = map (\row_probs ->
                                map2 (\p k -> k * p ) row_probs kappas
                            ) unscaled_log_probs
        in scaled_log_probs
```

Listing 4.9: Evaluating the exponent of von Mises-Fisher distributions

Maximization

For the maximization, new $\boldsymbol{\mu_h}$ need to be estimated. As shown in [1], this calculation is almost identical to the computation of the new means in the Gaussian case, except for a normalization that is applied afterwards to satisfy the constraint that $||\mu|| = 1$.

In contrast, the estimation of the new κ_h is not straightforward. The authors of [1] suggest an approximation to the updated κ:

$$\hat{\kappa}_h = \frac{\bar{r}_h d - \bar{r}_h^3}{1 - \bar{r}_h^2}$$

with $\bar{r}_h = ||\boldsymbol{\mu}_h||/(n\alpha_h)$, n being the number of data points in the dataset and $||\boldsymbol{\mu}_h||$ being calculated before the normalization of $\boldsymbol{\mu}_h$ described before.

However, there exist numerous approximations to this problem as shown in [40], and it is subject to future work to decide which approximation suits our problem domain best, or provide implementations of multiple approximations to the user.

4.5 Instantiating a mixture model

For a user of our framework, it is important to know how they can apply our library to their dataset. Listing 4.10 shows how a user can instantiate a mixture model of their liking and fit it to their data. First, the dataset has to be instantiated with the desired floating point precision. In our example, the data are stored as 64-bit floats and 64-bit integers are used for indexing the data. Then the desired mixture model is instantiated for the previously defined dataset. Our example uses the Gaussian mixture model with diagonal covariance matrix. Finally, the `em` module is instantiated with this mixture and the resulting module can be used later in the code.

The function `fit_gaussian` assumes that the user inputs their data in CSR format. However, it can be the case that they use a different precision than the one specified for our mixture dataset. In our example, the data of the user represents the nonzero values of the dataset as 32-bit floats instead of 64-bit floats. Thus, the data needs to be converted to the right type. The code line `let values = map dataset.V.f32 values` indicates, that all entries in the array `values` are converted from `f32` to the value type used in the dataset.

Finally, a data representation that the mixture will be able to use is created through the `mk_from_csr` function. This representation is passed to `gaussian_em.fit` along the convergence threshold, the desired number of components and the maximum number of iterations. Now the user can wait (hopefully not too long) for their results and retrieve the converged means or the assigned memberships for the points.

k = 1	k = 5	k = 10	k = 16	k = 32	k = 64	k = 128
x2100	x1893	x1480	x1231	x664	x321	x152

Table 4.1: Speedups of mixture implementation over scikit-learn.

```
import "em"
import "sparse_dataset"
import "gaussian_diag"

-- specify precision for the dataset
module my_sparse_dataset = sparse_dataset f64 i64

-- instantiate the mixture with this type of dataset
module my_gaussian_mixture = gaussian_diag my_sparse_dataset

-- we want to use the em algorithm on this mixture
module my_gaussian_em = em my_gaussian_mixture

-- we assume the user stored their dataset in CSR format
entry fit_gaussian [nnz][np1] (values: [nnz]f32)
                              (col_indices: [nnz]i64)
                              (pointers: [np1]i64)
                              (columns: i64) --number of columns in the data
    let values = map dataset.V.f32 values
    let col_indices = map dataset.I.i64 col_indices
    let pointers = map dataset.I.i64 pointers
    let data = dataset.mk_from_csr values col_indices pointers columns
    let (t, i, _ ,_) = my_gaussian_em.fit (mixture.C.f32 0.001f32) k 2 data

    let means = mixture.get_means t
    let weights = mixture.get_weights t
    let membership = mixture.predict_labels data t
        in (membership, means, weights, i)
```

Listing 4.10: Usage of our framework

4.6 Benchmark

To show the importance of providing an implementation of mixture models that allows for sparse data representations and executes on the GPU, we compared our implementation of the Gaussian mixture model to the implementation in scikit-learn[2]. For benchmarking purposes, we ran both implementations for 30 iterations, regardless of convergence. Table 4.1 shows the speedup of our version on the BBC dataset.

Note that we disregard the speedup for $k = 1$ in our contributions, since this case is of no practical relevance.

Since scikit-learn only supports input in dense representation, we converted the BBC dataset to this representation for the benchmarks. The other datasets were too big for this to be feasible, which shows the importance of providing support for sparse representations. Of course, the speedups reported are a result of both the smaller amount of computations necessary when dealing with sparse data, and the highly parallel execution on the GPU.

[2]Since our implementation restricts the covariance matrix to be diagonal, we chose this option for the scikit-learn implementation accordingly.

In order to quantify the impact of each of these effects, further baseline implementations have to be considered.

We can see that the speedup diminishes for higher values of k. This might be a result of us using the unoptimized Futhark implementation of the reduction step (see figure 3.6) and might be alleviated in the future by including the optimizations discussed in chapter 3.

Chapter 5

Conclusion

In this thesis we have investigated the implementation of selected unsupervised clustering methods in a way that handles large sparse datasets efficiently on the GPU.

In chapter 3 we explore different ways of implementing computations that work on sparse data representations, taking the classical k-means algorithm as example. We focus on the two most costly computations inside the main loop: the distance computations and the computation of the new cluster centers.

For the distance computation we found that extending the Euclidean metric helps to avoid unnecessary computations in the case of sparse data, and we explored the performance characteristics of different sparse representations. We conclude that for our datasets, sequentializing the inner reduction of the computation is the best approach.

For the computation of new cluster centers, a reduction step is needed. Since our ideas for optimizing this step relied on the semantics of the CUDA-specific atomic functions, we investigated these ideas in CUDA. We found that there is a certain threshold of k from which sorting the dataset by their cluster membership improves the performance, however, we were unable to identify any rules for choosing this threshold depending on the dataset.

We show that our best implementations on the GPU provide speedups of at least factor 10 over a multicore CPU implementation for our datasets.

Chapter 4 builds on the learnings from chapter 3, by generalizing the optimized operations used in k-means such, that they can be used to implement mixture models on sparse data efficiently. We present a framework in the Futhark programming language and show how k-means, spherical k-means, Gaussian mixture models and von Mises-Fisher mixture models can be implemented through it. Unfortunately, we were not able to implement all of our identified optimizations in this framework due to some limitations of the Futhark language. However, as the language is still in ongoing development we hope that in the future we will be able to access the necessary tools for our optimizations through Futhark.

Appendix A

Source code repositories

The library created and described as part of this thesis can be found in the github repository https://github.com/Sefrin/futhark-mixtures. The repository also contains a small example and instructions on how to run them.

The code for benchmarks and different code versions of the k-means explorations are located in the repository https://github.com/Sefrin/master_thesis. The Futhark code for the different distance computations can be found in the folder `code/skmeans/futhark` and the experiments for the reduction involving CUDA can be found in `code/skmeans/cuda`.

A python notebook used for preprocessing datasets and profiling the scikit-learn implementation is located in `code/skmeans/python`.

Bibliography

[1] A. Banerjee, I. S. Dhillon, J. Ghosh, S. Sra, and G. Ridgeway, "Clustering on the unit hypersphere using von mises-fisher distributions.," *Journal of Machine Learning Research*, vol. 6, no. 9, 2005.

[2] M. Z. Hossain, M. N. Akhtar, R. B. Ahmad, and M. Rahman, "A dynamic k-means clustering for data mining," *Indonesian Journal of Electrical engineering and computer science*, vol. 13, no. 2, pp. 521–526, 2019.

[3] B. Zheng, S. W. Yoon, and S. S. Lam, "Breast cancer diagnosis based on feature extraction using a hybrid of k-means and support vector machine algorithms," *Expert Systems with Applications*, vol. 41, no. 4, pp. 1476–1482, 2014.

[4] P. A. Burrough, P. F. van Gaans, and R. MacMillan, "High-resolution landform classification using fuzzy k-means," *Fuzzy sets and systems*, vol. 113, no. 1, pp. 37–52, 2000.

[5] T. Henriksen, N. G. W. Serup, M. Elsman, F. Henglein, and C. E. Oancea, "Futhark: Purely functional gpu-programming with nested parallelism and in-place array updates," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, (New York, NY, USA), pp. 556–571, ACM, 2017.

[6] I. S. Dhillon and D. S. Modha, "Concept decompositions for large sparse text data using clustering," *Machine learning*, vol. 42, no. 1, pp. 143–175, 2001.

[7] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[8] S. Raschka, J. Patterson, and C. Nolet, "Machine learning in python: Main developments and technology trends in data science, machine learning, and artificial intelligence," *arXiv preprint arXiv:2002.04803*, 2020.

[9] J. Vaněk, J. Trmal, J. V. Psutka, and J. Psutka, "Full covariance gaussian mixture models evaluation on gpu," in *2012 IEEE International Symposium on Signal Processing and Information Technology (ISSPIT)*, pp. 000203–000207, IEEE, 2012.

[10] N. P. Kumar, S. Satoor, and I. Buck, "Fast parallel expectation maximization for gaussian mixture models on gpus using cuda," in *2009 11th IEEE International Conference on High Performance Computing and Communications*, pp. 103–109, IEEE, 2009.

[11] Y. Liu and B. Schmidt, "Lightspmv: Faster csr-based sparse matrix-vector multiplication on cuda-enabled gpus," in *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pp. 82–89, IEEE, 2015.

[12] W. Cao, L. Yao, Z. Li, Y. Wang, and Z. Wang, "Implementing sparse matrix-vector multiplication using cuda based on a hybrid sparse matrix format," in *2010 International Conference on Computer Application and System Modeling (ICCASM 2010)*, vol. 11, pp. V11–161, IEEE, 2010.

[13] P. Guo and L. Wang, "Auto-tuning cuda parameters for sparse matrix-vector multiplication on gpus," in *2010 International Conference on Computational and Information Sciences*, pp. 1154–1157, IEEE, 2010.

[14] H. Anzt, T. Cojean, C. Yen-Chen, J. Dongarra, G. Flegar, P. Nayak, S. Tomov, Y. M. Tsai, and W. Wang, "Load-balancing sparse matrix vector product kernels on gpus," *ACM Transactions on Parallel Computing (TOPC)*, vol. 7, no. 1, pp. 1–26, 2020.

[15] C. J. Nolet, D. Gala, E. Raff, J. Eaton, B. Rees, J. Zedlewski, and T. Oates, "Semiring primitives for sparse neighborhood methods on the gpu," *arXiv preprint arXiv:2104.06357*, 2021.

[16] D. M. Witten and R. Tibshirani, "A framework for feature selection in clustering," *Journal of the American Statistical Association*, vol. 105, no. 490, pp. 713–726, 2010.

[17] C. Scully-Allison, R. Wu, S. Dascalu, L. Barford, and F. C. HarrisJr., "Data imputation with an improved robust and sparse fuzzy k-means algorithm," 2019.

[18] G. E. Blelloch, *Vector models for data-parallel computing*, vol. 2. MIT press Cambridge, 1990.

[19] N. Satish, M. Harris, and M. Garland, "Designing efficient sorting algorithms for manycore gpus," in *2009 IEEE International Symposium on Parallel & Distributed Processing*, pp. 1–10, IEEE, 2009.

[20] T. Henriksen, F. Thorøe, M. Elsman, and C. Oancea, "Incremental flattening for nested data parallelism," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, (New York, NY, USA), pp. 53–67, ACM, 2019.

[21] P. Munksgaard, S. L. Breddam, T. Henriksen, F. C. Gieseke, and C. Oancea, "Dataset sensitive autotuning of multi-versioned code based on monotonic properties," in *Proceedings of the International Symposium on Trends in Functional Programming*, TFP 2021, Springer, 2021.

[22] T. Henriksen, S. Hellfritzsch, P. Sadayappan, and C. Oancea, "Compiling generalized histograms for gpu," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20, IEEE Press, 2020.

[23] H. Steinhaus *et al.*, "Sur la division des corps matériels en parties," *Bull. Acad. Polon. Sci*, vol. 1, no. 804, p. 801, 1956.

[24] J. MacQueen *et al.*, "Some methods for classification and analysis of multivariate observations," in *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, vol. 1, pp. 281–297, Oakland, CA, USA, 1967.

[25] E. Forgy, "Cluster analysis of multivariate data : efficiency versus interpretability of classifications," *Biometrics*, vol. 21, pp. 768–769, 1965.

[26] S. Lloyd, "Least squares quantization in pcm," *IEEE transactions on information theory*, vol. 28, no. 2, pp. 129–137, 1982.

[27] D. Arthur and S. Vassilvitskii, "k-means++: The advantages of careful seeding," tech. rep., Stanford, 2006.

[28] J. Lücke and D. Forster, "k-means as a variational em approximation of gaussian mixture models," *Pattern Recognition Letters*, vol. 125, pp. 349–356, 2019.

[29] T. K. Moon, "The expectation-maximization algorithm," *IEEE Signal processing magazine*, vol. 13, no. 6, pp. 47–60, 1996.

[30] K. B. Petersen and M. S. Pedersen, "The matrix cookbook," nov 2012. Version 20121115.

[31] J. H. Plasse, *The EM algorithm in multivariate Gaussian mixture models using Anderson acceleration*. PhD thesis, Worcester Polytechnic Institute, 2013.

[32] S. Borman, "The expectation maximization algorithm-a short tutorial," *Submitted for publication*, vol. 41, 2004.

[33] M. Kearns, Y. Mansour, and A. Y. Ng, "An information-theoretic analysis of hard and soft assignment methods for clustering," in *Learning in graphical models*, pp. 495–520, Springer, 1998.

[34] Available at https://github.com/diku-dk/futhark-benchmarks [Accessed: 13-Aug-2021].

[35] T. Henriksen, "Incremental flattening for nested data parallelism on the gpu." Available at https://futhark-lang.org/blog/2019-02-18-futhark-at-ppopp.html [Accessed: 13-Aug-2021], Feb 2019.

[36] B. Dhanasekaran and N. Rubin, "A new method for gpu based irregular reductions and its application to k-means clustering," in *Proceedings of the fourth workshop on general purpose processing on graphics processing units*, pp. 1–8, 2011.

[37] N. Bell and J. Hoberock, "Thrust: A productivity-oriented library for cuda," in *GPU computing gems Jade edition*, pp. 359–371, Elsevier, 2012.

[38] D. Greene and P. Cunningham, "Producing accurate interpretable clusters from high-dimensional data," in *European conference on principles of data mining and knowledge discovery*, pp. 486–494, Springer, 2005.

[39] G. Sharma, A. Agarwala, and B. Bhattacharya, "A fast parallel gauss jordan algorithm for matrix inversion using cuda," *Computers & Structures*, vol. 128, pp. 31–37, 2013.

[40] K. Hornik and B. Grün, "movmf: an r package for fitting mixtures of von mises-fisher distributions," *Journal of Statistical Software*, vol. 58, no. 10, pp. 1–31, 2014.