UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE

Project Outside Course Scope

Kasper Unn Weihe (PXH755), Kristian Quirin Hansen (HGJ828),
Peter Kanstrup Larsen (ZLC797)

Linear Algebra in Futhark

Supervisor: Troels Henriksen

January 28, 2021

Abstract

This paper describes the implementation and design of multiple linear algebra algorithms in the data-parallel purely functional array language Futhark. Linear algebra algorithms are oftentimes used on large datasets and are usually very time consuming due to a large number of matrices and vector operations. This project aimed to explore and work with data-parallel functional programming to see how well Futhark handles some of the most used linear algebra functions, including NMF, Cholesky Decomposition, QR Decomposition, and Solving of linear systems. All the implementations of the linear algebra algorithms in Futhark are compared in performance to the Python libraries - NumPy and scikit-learn. Our benchmark results show that our implemented linear algebra programs in Futhark could not always keep up with the Python libraries. The reason behind this could be that the Python libraries are highly hand-optimized. However, the reason is likely concerning us not implementing very optimized versions of these algorithms and the compiler not generating optimal code in some cases. Nonetheless, from our project results, we still think Futhark and linear algebra suits each other well.

Contents

Chapter 1

Introduction

Linear algebra is an important branch of mathematics used in various fields to model and compute data. These algorithms are typically used on large datasets and are often very time-consuming due to a large number of matrix and vector operations. Some of the most well-known linear algebra topics are decomposition algorithms, solving equations, inverting matrices, and computation of determinants, to name a few.

This motivates the use of the data-parallel functional language Futhark, which is designed with an optimizing compiler able to generate parallel code for massively parallel hardware like GPUs. Programming in such a language often brings up new design considerations and challenges. In this paper, we present several implementations in Futhark of some of the most used algorithms in Linear algebra such as QR, NMF, LU, Cholesky, etc.

1.1 Project objective

This project's main objective is to explore and work with the data-parallel functional Futhark and examine whether it is suited for linear algebra programming. We want to find out if three third-year computer science students with no former experience of data-parallel computing, let alone purely functional programming, can implement and express some of the most used linear algebra functions in Futhark. We want to analyze how well these algorithms perform in Futhark compared to libraries such as NumPy and scikit-learn based on benchmarks. These Python libraries primarily use subroutines from C and Fortran libraries, such as OpenBLAS[1] and LAPACK[2]. OpenBLAS and LAPACK are the 'de facto' standard of high-performance scientific computing.

[1] `https://github.com/xianyi/OpenBLAS`
[2] `http://www.netlib.org/lapack/`

1.2 Report structure

This project report consists of different chapters concerning the implemented linear algebraic primitives in Futhark. Each chapter will contain an introduction to the presented algorithm and the theory behind the algorithm. We then follow this up by discussing our specific implementation and design of the algorithm in Futhark, followed up by benchmarks comparing to the chosen Python libraries.

The first chapter will introduce several implementations of small basic linear algebra operations, which we have used throughout the different algorithms. The reader is not expected to have prior knowledge of the presented algorithms but should have some familiarity with introductory linear algebra. Chapter 8 presents future work regarding the project and a comprehensive conclusion on the results of the project. The report is best read in chronological order. However, we have attempted to make each individual chapter self-explanatory.

1.3 Futhark

Futhark is a small, purely functional array-based programming language in the ML family, designed with an optimizing compiler that can generate code for the GPU through the OpenCL or CUDA frameworks. Futhark programs mostly consist of Second-Order Array Combinators (SOACs) (like **`map`**, **`reduce`**, **`filter`**, and so forth), which can be compiled to parallel code [EHO18b]. However, it is worth noting that Futhark is not a parallel language per default; it is the programmer's job to identify parallelism and use SOACs to implement efficient GPU code.

The Futhark programming language feels similar to other functional programming languages, and as mentioned, the most essential part of programming in Futhark is array programming with SOACs. As a small example of a Futhark program, we can compute the dot product $\sum_i xs_i \cdot ys_i$ of two vectors with the SOACs: **`map2`** and **`reduce`**:

```
1 let dotprod [n] (xs: [n]f32) (ys: [n]f32): f32 =
2   reduce (+) 0.0 (map2 (*) xs ys)
```

`map2` multiplies each element in the two vectors `xs` and `ys` together which produces a single array that is reduced with the operator '+' and 0.0 as the neutral element. The notation of `[n]f32` defines an array of length n as type `f32`.

Futhark is, as formerly mentioned, a purely functional language but still allows in-place array updates. This is essential for linear algebra programming since in-place updates are used in most of the algorithms. A straightforward program introducing an in-place update of an array is shown in the example below:

```
1 let modify [n] (a: *[n]i64) (i: i64) (x: i64): *[n]i64 =
2   let a[i] = a[i] + x
3   in a
```

This simple function increases the index i of array a with the value x. Here we also have the uniqueness type `*[n]i32` which we in Futhark use to guarantee that an in-place operation can be safely computed without requiring additional memory allocation or copying the array to another variable. If we pass an argument as being unique, it will be consumed in the function call, meaning it can not be used in the function later again.

Nonetheless, Futhark also comes with limitations. For instance, only regular arrays are allowed, and there is no direct support of recursive functions. However, Futhark still supports sequential for- and while-loops using the **`loop`** construct. The following small example of a for-loop computes the Fibonacci numbers:

```
1 let fib(n: i32): i32 =
2   let (x, _) = loop (x, y) = (1,1) for i < n do (y, x+y)
3   in x
```

The use of loops is also an essential functionality for linear algebraic primitives since many of the presented algorithms can not be defined exclusively by SOACs. Furthermore, many of our implemented algorithms depend on the input matrix's size, making it difficult to predict numbers in advance.

The Futhark language uses a parallel cost model, which we have used throughout the project report to define and analyze our programs' efficiency. This cost model is defined by two main concepts: **work** and **span**. Work is the total amount of work from operations in a program, while span defines the program's depth in computation where we assume infinite parallelism in SOAC computations [EHO18a].

Chapter 2

Basic Linear Algebra Operations

The following chapter will describe and present several different linear algebra operations used throughout the implementation of our presented algorithms. In linear algebra, the most common operations consist of simple matrix and vector operations; hence this section will not go into a very detailed explanation of the theory behind the operations but briefly describe the trivial matrix and vector operations.

2.1 Implementation of vector operations

The most basic and essential vector operation, which is also used in a lot of the matrix operations, is the dot product: $\sum_i xs_i \cdot ys_i$ of two vectors, which we also presented briefly in the introduction to Futhark section.

```
1 -- Work: O(n)
2 -- Span: O(log(n))
3 let dotprod [n] (xs: [n]f32) (ys: [n]f32): f32 =
4   reduce (+) 0.0 (map2 (*) xs ys)
```

The dot product of the two input arrays xs and ys with the same length n is computed with the use of **map2**, and hereby summing the numbers together with **reduce**. Another trivial but also useful operation is vector multiplication scalar, which is easily computed by mapping through the vector and applying a scalar:

```
1 -- Work: O(n)
2 -- Span: O(1)
3 let vecmul_scalar [n] (xs: [n]f32) (k: f32): [n]f32 =
4   map (*k) xs
```

The functions take an array xs of length n and the scalar k, and returns the array after applying the scalar. This function can easily be modified with another operator, e.g., vector division scalar.

The last vector operation to be presented is the outer product of two vectors $xs \otimes ys = \mathbf{A}$ which produces an $m \times n$ matrix $\mathbf{A}$. The matrix is obtained by multiplying each element of vector xs by each element of ys:

```
1 -- Work: O(n^2)
2 -- Span: O(1)
3 let outer [n][m] (xs: [n]f32) (ys: [m]f32): *[n][m]f32 =
4   map (\x -> map (\y -> x * y) ys) xs
```

The outer function takes two vectors: xs of length n and ys of length m. The nested **map** multiplies the first element of xs with each element in vector ys to get the first row in the matrix. We do this for every element in the two arrays xs and ys which will give us a matrix of size $m \times n$ which is equal to the outer product of $xs \otimes ys$.

2.2 Implementation of matrix operations

Matrix operations are the most used operations in linear algebra and can be computed very efficiently on GPU due to many parallel computations. We can write matrix multiplication as the dot product of rows and columns in the matrix as: $xs \cdot ys^T$ where ys^T is the transpose of ys.

```
1 -- Work: O(n^3)
2 -- Span: O(log(n))
3 let matmul [n][p][m] (xss: [n][p]f32) (yss: [p][m]f32): *[n][m]f32 =
4   map (\xs -> map (dotprod xs) (transpose yss)) xss
```

We use the inner **map** to multiply a row from **xss** with a column from **yss** by transposing the matrix, the outer **map** is then used to go through every column and row for the matrices to obtain the final result.

Matrix multiplication scalar is just an extension of vector scalar; all we need is to add an extra **map** to go through both the columns and the rows:

```
1 -- Work: O(n^2)
2 -- Span: O(1)
3 let matmul_scalar [m][n] (xss: [m][n]f32) (k: f32): *[m][n]f32 =
4   map (map (*k)) xss
```

This implementation can also be easily modified to another operator if we would like to subtract the matrix with a scalar factor instead of multiplying.

The last function that is important to explain before presenting the different linear algebraic algorithms is the identity function that constructs an $n \times n$ identity matrix. An identity matrix is a square matrix with ones on the diagonal and zeros everywhere else.

```
1 -- Work: O(n^2)
2 -- Span: O(1)
3 let identity (n: i64): [n][n]f32 =
4   tabulate_2d n n (\i j -> if j == i then 1f32 else 0f32)
```

Here we use `tabulate_2d n n` to construct a $n \times n$ matrix from a function, which sets the entry to 1 if we are on the diagonal ($i == j$), otherwise it is 0. `tabulate_2d n n` is equivalent to two nested maps, both with (iota n) as argument.

All the operations presented above, and small modifications will be used throughout the next chapters when we present the implementation and design of our linear algebra programs. More specific subroutines used in the algorithms will be presented in the chapters.

Chapter 3

QR Decomposition

QR Decomposition is the decomposition of an $m \times n$ matrix, with $m \geq n$, that factors a matrix $\mathbf{A}$ into a product of $\mathbf{A} = \mathbf{QR}$. Here $\mathbf{Q}$ is an $m \times m$ orthogonal matrix, and $\mathbf{R}$ is an $m \times n$ upper triangular matrix.

$$\underbrace{\begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \end{bmatrix}}_{\mathbf{A}} = \underbrace{\begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \end{bmatrix}}_{\mathbf{Q}} \underbrace{\begin{bmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \end{bmatrix}}_{\mathbf{R}} \tag{3.1}$$

QR Decomposition is often used for solving linear least-squares problems and as an efficient tool to compute eigenvalues and eigenvectors. There are numerous methods to compute the QR Decomposition, and the most famous ones consist of the Gram-Schmidt process, Householder transformations, and Givens rotations. We implemented both the Gram-Schmidt process and the Householder transformations algorithms for this project. We decided to omit Givens rotation since we discovered that it is not well suited for GPU optimization and does not scale well with large matrix sizes [AR09].

Our primary focus was mainly on the Householder transformation method since we discovered that it was possible to improve it into a blocked version. The blocked version should work well on GPUs to improve the performance. The Gram-Schmidt process is also quite numerical unstable since the resulting $\mathbf{Q}$ can be far from orthogonal, which does not make it as attractive. We will describe this more in detail later on. However, Gram-Schmidt should still yield faster results than the Householder reflection method and can still be useful for some applications [GL13a][Ste98a]. The following chapter will describe the theory and the implementation of the three QR Decomposition algorithms: Gram-Schmidt, standard Householder QR, and blocked Householder QR. In the end, we compare them to each other and the Python libraries: NumPy and scikit-learn.

3.1 The Gram-Schmidt Process Algorithm

The Gram-Schmidt process computes the orthogonal $m \times m$ matrix $\mathbf{Q}$, which we can use to compute the upper triangular matrix $\mathbf{R}$. We do this from the simple matrix multiplication: $\mathbf{R} = \mathbf{Q}^T\mathbf{A}$, where $\mathbf{Q}^T$ denotes the transpose of $\mathbf{Q}$.

The Gram-Schmidt process consists of reconstructing the column vectors of the input matrix $\mathbf{A}$ into orthonormal vectors that we can use to define our orthogonal matrix $\mathbf{Q}$. We do this by transforming the columns into orthogonal vectors and then transforming the orthogonal vectors into orthonormal vectors to get our final orthogonal matrix $\mathbf{Q}$. We have a set of linearly independent column vectors $\{\vec{a}_1, \vec{a}_2, ..., \vec{a}_k\}$ that is a basis for our input matrix $\mathbf{A}$. As mentioned, the goal is then to use the Gram-Schmidt process to construct an orthonormal basis of $\mathbf{A}$ consisting of the column vectors $\{\vec{q}_1, \vec{q}_2, ..., \vec{q}_k\}$ corresponding to the orthogonal matrix $\mathbf{Q}$ [Ste98b]. To construct the first orthonormal column vector $\vec{q_1}$, we make the first column of $\mathbf{A}$ to a unit vector:

$$\vec{q_1} = \frac{\vec{a_1}}{||\vec{a_1}||} \tag{3.2}$$

For the next column vector $\vec{q}_2$, we have to calculate the projection of $\vec{a_2}$ onto $\vec{q_1}$ and subtract it from projection $\vec{a_2}$, which gives us the orthogonal vector. We can then make this vector orthonormal by making it a unit vector. The projection of $\vec{a}_2$ onto $\vec{q_1}$ is equal to:

$$\text{proj}_{\vec{q_1}}(\vec{a}_2) = \frac{\vec{a}_2 \cdot \vec{q}_1}{\vec{q}_1 \cdot \vec{q}_1}\vec{q}_1 = (\vec{q_1}^T\vec{a}_2)\vec{q_1} \tag{3.3}$$

We can now compute the column vector $\vec{q}_2$ to:

$$\vec{q'}_2 = \vec{a}_2 - \text{proj}_{\vec{q_1}}(\vec{a}_2) \qquad \vec{q}_2 = \frac{\vec{q'}_2}{||\vec{q'}_2||} \tag{3.4}$$

For the next column vector $\vec{q}_3$, we need to calculate the projection of $\vec{a}_3$ onto $\vec{q_1}$ and $\vec{q}_2$, and then subtract it from $\vec{a_3}$:

$$\vec{q'}_3 = \vec{a}_3 - (\text{proj}_{\vec{q_1}}(\vec{a}_3) + \text{proj}_{\vec{q}_2}(\vec{a}_3)) \qquad \vec{q}_3 = \frac{\vec{q'}_3}{||\vec{q'}_3||} \tag{3.5}$$

We need to do this for every column vector in matrix $\mathbf{A}$ which gives us the following formula to compute every orthonormal vector q_k:

$$\vec{q'}_k = \vec{a}_k - \sum_{j=1}^{k-1} \text{proj}_{\vec{q_j}}(\vec{a}_k) \qquad \vec{q}_k = \frac{\vec{q'}_k}{||\vec{q'}_k||} \tag{3.6}$$

The orthogonal matrix $\mathbf{Q}$ is then equal to $\{\vec{q_1}, \vec{q}_2, ..., \vec{q}_k\}$

Design and implementation

As formerly mentioned, the Gram-Schmidt process is numerically unstable due to rounding errors when calculating the projection, which means vectors often do not get entirely orthogonal. However, it is essential to mention a modified Gram-Schmidt process exists that uses an extra for-loop to stabilize the algorithm and produce smaller errors. This modified algorithm is still not as stable as the Householder method, so we decided to implement the classical Gram-Schmidt and focus on the blocked version of Householder transformations.

As shown in the theory section above, we need to reconstruct the column vectors of $\mathbf{A}$ into being orthonormal to define our orthogonal matrix $\mathbf{Q}$. As shown in (3.2), we simply need to calculate the unit vector for the first column. After this, we loop through the rest of the columns in $\mathbf{A}$ and compute the projection. From the projection, we then calculate the reciprocal unit vector as shown in (3.6).

The most trivial part of the implementation was to slice the columns of the matrices correctly. Due to convenience, we decided to slice the column vectors a_k of $\mathbf{A}$ as 1D arrays, which also meant we needed to create small operations to multiply a vector and a matrix element-wise to compute the vector projections. The entire implementation takes an input array $\mathbf{A}$ of size $m \times n$ where $m \geq n$, and uses Gram-Schmidt to construct $\mathbf{Q}$ and $\mathbf{R}$:

```
1  -- Work: O(n^3)
2  -- Span: O(n · log(n))
3  let gram_schmidt [m][n] (A: *[m][n]f32): (*[m][n]f32, *[n][n]f32) =
4    let Q = replicate m (replicate n 0f32)
5    let Q[:,0] = vecdiv_scalar A[:,0] (vector_length A[:,0])
6    let Q =
7      loop Q for i in 1..<n do
8        let q = Q[:,:i]
9        let sum_qA = sum_row (matvecmul_col (transpose q) A[:, i])
10       let sum_qAq = sum_row (matvecmul_col q sum_qA)
11       let q' = vecmin A[:,i] sum_qAq
12       let Q[:,i] = q'
13       let Q[:,i] = vecdiv_scalar Q[:,i] (vector_length Q[:,i])
14     in Q
15   let R = matmul (transpose Q) A
16   in (Q, R)
```

First, we allocate $\mathbf{Q}$ and calculate the first unit vector (3.2); we then loop through the rest of the columns in $\mathbf{A}$ and compute the orthogonal vector in lines 6-9 from the projections. In the end, we calculate the orthonormal vector in line 11 and put it in the corresponding column for $\mathbf{Q}$. The function then returns the constructed orthogonal matrix $\mathbf{Q}$ and the triangular matrix $\mathbf{R}$.

The `vector_length` function computes $||\vec{v}|| = \sqrt{a^2 + b^2}$, which we can do efficiently with **`reduce`** and **`map`**:

```
1 -- Work: O(n)
2 -- Span: O(log(n))
3 let vector_length [n] (xs: [n]f32): f32 =
4   reduce (+) 0f32 (map(\x -> f32.abs(x)**2) xs) |> f32.sqrt
```

The rest of the functionalities are basic linear algebraic operations, which were all covered in chapter 2.

3.2 Householder Algorithm

QR Decomposition with householder transformations computes the two matrices $\mathbf{Q}$ and $\mathbf{R}$ from a $m \times n$ input-matrix $\mathbf{A}$ [AR09]. We do this by applying Householder transformations in-place to our input-matrix $\mathbf{A}$. The following will show how to compute $\mathbf{Q}$ and $\mathbf{R}$ for the non-blocked Householder transformation algorithm; later on, we will present the blocked version.

The orthogonal matrix $\mathbf{Q}$ is equitable to the dot product of a sequence of Householder transformation matrices:

$$\mathbf{Q} = (\mathbf{H}_1 \cdot \mathbf{H}_2 ... \mathbf{H}_{n-1}) \tag{3.7}$$

We can define the first Householder transformation as the matrix $\mathbf{H}_1$ from the first Householder vector v_1:

$$\mathbf{H}_1 = \mathbf{I} - 2 \cdot \frac{v_1 \cdot v_1^T}{v_1^T \cdot v_1} \tag{3.8}$$

v_1 is the first Householder vector for the first column vector a_1 from our input matrix $\mathbf{A}$, and $\mathbf{I}$ is our $m \times m$ identity matrix equal to $\mathbf{Q}^T\mathbf{Q}$. We can calculate v_1 from:

$$v_1 = a_1 + \text{sign}(a_{11})||a_1||e_1 \tag{3.9}$$

Here $||a_1||$ is the vector length of the first matrix column in $\mathbf{A}$, the `sign` function is simply an extraction of the sign of a number, and e_1 is the first column of the identity matrix. With the different Householder transformation matrices: $(\mathbf{H}_1, \mathbf{H}_2, ...\mathbf{H}_{n-1})$, we can compute our upper triangular matrix $\mathbf{R}$ from:

$$\mathbf{R} = (\mathbf{H}_{n-1} ... \mathbf{H}_2 \mathbf{H}_1 \mathbf{A}) \tag{3.10}$$

After overwriting the input matrix with the first Householder transformation: $\mathbf{H}_1 \cdot \mathbf{A}$, the first column will consist of zeros below the diagonal. We then move to the next column and one row down to repeat the process and compute the next Householder transformation $\mathbf{H}_2$. We do this process for all columns repeatedly on the updated input matrix corresponding to $\mathbf{R}$, for the number of columns in our input matrix until $\mathbf{A}$ is triangularized.

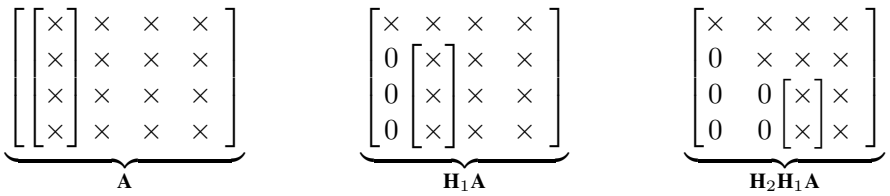

Figure 3.1: In-place update of $\mathbf{A}$ with Householder transformations to compute $\mathbf{R}$

Implementation

The standard algorithm to compute QR decomposition with Householder transformations was reasonably straightforward to implement. We needed to consistently update our input matrix $\mathbf{A}$ with our new Householder matrices to get the upper triangular matrix $\mathbf{R}$. The orthogonal matrix $\mathbf{Q}$ is equal to the product of the computed Householder transformation matrices as in (3.7). The main design problem for the implementation was to make in-place updates without consuming variables. We achieved this safely with the uniqueness type to ensure functions and in-place updates would not consume values instead of using computational time copying matrix slices. When computing the Householder vectors, we could not circumvent that we needed to use a copy of the corresponding column in $\mathbf{A}$ to calculate the final householder matrix.

For the implementation, we needed a function to compute the Householder vectors v, where we could simultaneously compute a part of the Householder transformation $\beta = \frac{2}{v^T v}$:

```
1 -- Work: O(n)
2 -- Span: O(log(n))
3 let house [n] (x: [n]f32): (*[n]f32, f32) =
4   let dot = dotprod x x
5   let v = copy x with [0] = x[0] - (f32.sqrt dot)
6   let dot' = dot - x[0]**2 + v[0]**2
7   let beta = if dot' != 0 then 2.0/dot' else 0
8   in (v, beta)
```

As seen in the implementation of the `house` function, we could not omit the copy of column vector x, since v is the exact same vector. From equation 3.9, we see that e_1 is the first column of the identity matrix. This column will always consist of a vector with the first element as 1 and 0's below. Hence, the Householder vector v will only differ from the input vector x for the first element, which means we only need to compute the first element for the Householder vector. We can construct the Householder vector by updating the first element from the input vector's length. β is then computed and divided by two if it is not equal to 0. In the end, we return both the Householder vector v and β as a tuple.

For the entire algorithm's implementation, as seen below, we compute the Householder transformation matrix from the Householder vector v and β for the corresponding column in matrix $\mathbf{A}$

as shown in figure 3.1. We go through every column in $\mathbf{A}$ and make in-place updates for $\mathbf{Q}$ and $\mathbf{A}$ with a for-loop. In the end, we return the matrices in a tuple, where $\mathbf{A}$ is the upper triangular matrix $\mathbf{R}$.

```
1  -- Work: O(n^3)
2  -- Span: O(n · log(n))
3  let householder [m][n] (A: *[m][n]f32): ([m][m]f32, [m][n]f32) =
4    let Q = identity m
5    let (Q,A) =
6      loop (Q,A) for k in 0..<(n) do
7        let (v, B) = house A[k:m,k]
8        let vvT = outer v v
9        let BvvT = matmul_scalar vvT B
10       let BvvTA = matmul BvvT A[k:m,k:n]
11       let BQ = matmul_scalar Q[0:m,k:m] B
12       let BQvvT = matmul BQ vvT
13       let A[k:m,k:n] = matsub A[k:m,k:n] BvvTA
14       let Q[0:m,k:m] = matsub Q[0:m,k:m] BQvvT
15   in (Q, A)
16 in (Q,A)
```

The for-loop in lines 4-13 runs for the number of columns n in the input matrix $\mathbf{A}$. As formerly described, we need to go the next column and one row down for every iteration; we do this with the slice in line 5. Lines 6-12 is the computation of the Householder transformation matrix and in-place updates of both $\mathbf{Q}$ and $\mathbf{A}$. We only update the required parts of the matrices to make as few computations as possible.

Overall the implementation is not very parallel but mostly consists of parallel matrix and vector operations, which were described briefly in chapter 2. We could probably have implemented this standard Householder algorithm with more parallelism, but it was not our focus since we mainly used it as a stepping stone for the blocked version.

3.3 Blocked Householder Algorithm

The solution to QR-Decomposition presented above is simple and mainly consists of matrix-vector multiplications, which suits parallel functional programming well. However, it is not well optimized, and the amount of computation per memory element from global memory is relatively low [AR09]. It is possible to improve the performance by applying more than one Householder transformation matrix at a time. This algorithm is called blocked Householder QR Decomposition, which the next section will describe.

Instead of applying Householder transformations as single column updates to the identity matrix $\mathbf{I}$ as in the non-blocked version above, it is possible to partition the input matrix $\mathbf{A}$ into $m \times r$ blocks $\mathbf{A} = \begin{bmatrix}\mathbf{A}_1 & \mathbf{A}_2 & \mathbf{A}_3\end{bmatrix}$ [GL13a]. The block-size r is chosen based on the problem size that conducts the best performance for the given input matrix.

$$\mathbf{A} = \begin{bmatrix} \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \end{bmatrix} = \underbrace{\left[\begin{bmatrix} \times & \times \\ \times & \times \\ \times & \times \\ \times & \times \\ \times & \times \end{bmatrix} \begin{bmatrix} \times & \times \\ \times & \times \\ \times & \times \\ \times & \times \\ \times & \times \end{bmatrix} \begin{bmatrix} \times & \times \\ \times & \times \\ \times & \times \\ \times & \times \\ \times & \times \end{bmatrix} \right]}_{\mathbf{A} = \begin{bmatrix} \mathbf{A}_1 & \mathbf{A}_2 & \mathbf{A}_3 \end{bmatrix}} \tag{3.11}$$

Figure 3.2: Matrix is partitioned into blocks, i.e. here block-size r is 2

The goal is then to apply Householder transformations to the first block. Instead of moving on to apply more Householder reflections to the columns of the remaining blocks as before, we construct two new matrices. Two $m \times r$ matrices $\mathbf{Y}$ and $\mathbf{W}$ that we can use to represent our Householder matrices as:

$$\mathbf{H}_{wy} = \mathbf{H}_1\mathbf{H}_2...\mathbf{H}_r \tag{3.12}$$

$$= \mathbf{H} + \mathbf{W}\mathbf{Y}^T \tag{3.13}$$

Both $\mathbf{W}$ and $\mathbf{Y}$ are computed from our Householder vector v and β from the `house` function as presented above. With $\mathbf{W}$ and $\mathbf{Y}$, we can now compute the in-place updates of $\mathbf{Q}$ and $\mathbf{A}$ as:

$$\mathbf{A} = \mathbf{A} + \mathbf{Y}\mathbf{W}^T\mathbf{A} \tag{3.14}$$

$$\mathbf{Q} = \mathbf{Q} + \mathbf{Q}\mathbf{W}\mathbf{Y}^T \tag{3.15}$$

To summarize: for each block k, we need to compute a total amount of r Householder vectors and the corresponding β's and update the current block we are working with to triangularize the columns with Householder transformation matrices. The $\mathbf{W}$ and $\mathbf{Y}$ matrices are then computed from the results to apply $\mathbf{H}_{wy} = \mathbf{I} + \mathbf{W}_k\mathbf{Y}_k^T$ to the remaining blocks of $\mathbf{A}$ and to $\mathbf{Q}$ as described above. Overall this approach will yield an algorithm with more matrix multiplications and fewer vector multiplications, achieving a better GPU performance.

Design and implementation

The design and implementation of the Block Householder QR in Futhark were inspired by a research paper on QR Decomposition on GPUs using the CUBLAS library in C++ [AR09]. The entire implementation in Futhark with all the steps to update $\mathbf{Q}$ and $\mathbf{A}$ as described above is shown below:

```
1  -- Work: O(n^3)
2  -- Span: O(n · log(n))
3  let blocked_householder [m][n] (A: *[m][n]f32) (r: i64):
4                                 ([m][m]f32, [m][n]f32) =
5    let Q = identity m
6    let (Q,A) =
7      loop (Q,A) for k in 0..<(n/r) do
8        let s = k * r
9        let V = replicate m (replicate r 0f32)
10       let Bs = replicate r 0f32
11       -- Compute Householder vectors, betas and
12       -- tranquilize the block
13       let (Bs, V, A) = loop (Bs, V, A) for j in 0..<r do
14         let u = s + j
15         let (v0, B) = house A[u:,u]
16         let V[u+1:,j] = A[u+1:,u]
17         let V[k * r + j:u + 1, j] = [v0]
18         let v = V[k * r + j:, j]
19         let l1 = m-u
20         let l2 = r-j
21         let BvvT =
22           matmul_scalar (outer v v) B :> [l1][l1]f32
23         let BvvTAk =
24           matmul BvvT (A[u:, u:s+r] :> [l1][l2]f32) :> [l1][l2]f32
25         let A[u:, u:s+r] =
26           matsub (A[u:, u:s+r] :> [l1][l2]f32) BvvTAk
27         let Bs[j] = B
28         in (Bs, V, A)
29
30       -- Initialize and compute columns for the Y and W matrices
31       let Y = replicate r (replicate m 0f32)
32       let W = replicate r (replicate m 0f32)
33       let Y[0] = V[:, 0]
34       let W[0] = vecmul_scalar Y[0] (-Bs[0])
35
36       let (Y, W) = loop (Y, W) for j in 1..<r do
37         let v = V[:, j]
38         let WYTv =
39           matvecmul_row (matmul (transpose W[0:j]) Y[0:j]) v
40         let BWYTv = vecmul_scalar WYTv Bs[j]
41         let mbj = vecmul_scalar v (-Bs[j])
42         let z = map2 (-) mbj BWYTv
43         let Y[j] = v
44         let W[j] = z
45         in (Y, W)
46
47       -- In-place update of Q and A
48       let l = m - s
```

```
49        let YWTA = matmul (matmul (transpose Y)  W) A[:, s+r:n]
50        let A[:, s+r:n] = matadd A[:, s+r:n] YWTA
51        let WY = (matmul (transpose W[:, s:]) Y[:, s:]) :> [l][l]f32
52        let Q_block = (Q[:, s:]) :> *[m][l]f32
53        let QWY = matmul Q_block WY
54        let Q[:, s:] = matadd Q_block QWY
55      in (Q,A)
56  in (Q,A)
```

Since the blocked QR Decomposition algorithm works with different matrix sizes depending on the loop, Futhark required us to manually coerce matrix blocks to the specified size to perform matrix operations without getting errors. Our initial approach was to use the same `house` function for the first loop, as shown for the standard Householder above. However, as we were already allocating space for the matrix $\mathbf{V}$ on line 7, which should contain the Householder vectors to calculate $\mathbf{W}$ and $\mathbf{Y}$, we only needed to compute the first element of the Householder vector and place it in the matrix with the rest of the column as seen in line 15-16. The modified `house` function, which only returns two floats, is shown below:

```
1  let house [d] (x: [d]f32): (f32, f32) =
2    let dot = dotprod x x
3    let v0 = x[0] - (f32.sqrt dot)
4    let dot' = dot - x[0]**2 + v0**2
5    let beta = if dot' != 0 then 2.0/dot' else 0
6    in (v0, beta)
```

To compute the matrices $\mathbf{Y}$ and $\mathbf{W}$ in lines 29-43, we decided upon allocating space for them before computing them. This solution meant we did not need to use **`concat`** to add the vectors z and v to the matrices, which gave a more elegant and faster solution. Our biggest speed-up came from optimizing our update of the orthogonal matrix $\mathbf{Q}$; with our initial approach, we simply updated it with the full-size matrices, as described earlier: $\mathbf{Q} = \mathbf{Q} + \mathbf{IWY}^T$. By recognizing that we only needed to compute parts of the matrix for each iteration, we could save computation time by only updating the block size instead of the whole matrix.

In this implementation, the block-size r should be chosen based on the problem size that conducts the best performance for the given input matrix. This solution is not very flexible and requires either knowledge about the matrix and vector computation or much testing with benchmarking. A more flexible approach to matrix blocking exists, which uses a divide-and-conquer approach that should be effective for parallel computation [GL13a]. However, we did not implement this approach since it is recursive, making it complicated to implement in Futhark.

We can easily implement a batched version of the blocked Householder transformations algorithms with the the use of **`map`**. We map over a 3D array containing our 2D input arrays for the algorithm. We can use the same approach as shown below for both the Gram-Schmidt process and the standard Householder algorithm:

```
1 -- Work: O(m · n^3)
2 -- Span: O(n · log(n))
3 let qr_batched [m][n] (xsss: [m][n][n]f32) (r:i67)
4                        : *[m]([n][n]f32, [n][n]f32) =
5   map (\xss -> block_householder (copy xss) r) xsss
```

Batched algorithms for linear algebra consists of computing a large number of small independent matrices. Batched linear algebra algorithms are not practical for this project but can be very useful for various important computer science fields. Some of these include, e.g., deep learning, data mining, and image processing [Azz17]. To name some: Batched versions of QR Decomposition is, for example, used in radar processing [MK12], and a batched version of Cholesky, as we will present in chapter 5, can be used in computer vision and hyperspectral anomaly detection [Jos14].

3.4 Benchmarks

This section will compare the performance of our three implemented QR algorithms: The Gram-Schmidt process, Householder transformations, and the blocked Householder transformations with QR Decomposition functions from NumPy and scikit-learn. Both NumPy and scikit-learn make use of the blocked Householder version from LAPACK, which is fully parallel and makes use of all CPU cores while running. The first table only shows the blocked version of Householder to demonstrate how the block-size r affects performance.

All the Futhark benchmarks were conducted using `futhark bench`[1] with CUDA as backend on Futhark version 0.19, on datasets generated with `futhark dataset`[2]. The Python benchmarks were conducted 10 times in a for-loop where the average run time is shown in the tables below. The computer used has an an Intel(R) Core(TM) i5-8600K CPU @ 3.60GHz, and an NVIDIA GeForce GTX 1070 Ti graphics card with 8GB VRAM.

[1] `https://futhark.readthedocs.io/en/latest/man/futhark-bench.html`
[2] `https://futhark.readthedocs.io/en/latest/man/futhark-dataset.html`

Matrix size \ Library	Futhark Blocked Householder
2000x2000, r=20	3659.1ms
2000x2000, r=50	3242,3ms
3000x1500, r=30	6150.9ms
3000x1500, r=60	5732.8ms
3000x3000, r=60	10325.5ms
3000x3000, r=20	12201ms
4000x4000, r=50	25058.6ms
4000x4000, r=100	28720.4ms
6000x6000, r=200	157632.4ms

Table 3.1: Benchmarks of blocked Householder, showing how block size r affects performance

Matrix size \ Library	Futhark HH	Futhark B-HH	**Futhark GS**	NumPy	SciPy	Speedup
1000x1000	1642.36ms	492.3	**129.2ms**	38.8ms	29.7ms	0.23x
3000x1500	78175.92ms	5732.8	**638.5ms**	342.6ms	389.6ms	0.61x
3000x3000	120826.238ms	12201.6ms	**2354.3ms**	1173.2ms	591.1ms	0.25x
4000x2000	-	15865.1ms	**5292.7ms**	2722.6ms	1497.6ms	0.28x
5000x5000	-	59938.8ms	**11602.5ms**	4968.2ms	2707.1ms	0.23x
6000x6000	-	157632.4ms	**18756.8ms**	8627.7ms	4377.3ms	0.23x

Table 3.2: Benchmark results of QR algorithms. Best Futhark performance is marked with **bold**
HH = Standard Householder, B-HH = Blocked Householder, GS = Gram-Schmidt process
Speedup is relative, calculated by $\frac{SciPy\ time}{Futhark\ GS\ time}$

Matrix size \ Library	Futhark HH	Futhark B-HH	**Futhark GS**	NumPy	SciPy	Speedup
10k, 10x10 arrays	9.2ms	8.6ms	**3.4ms**	190.2ms	189.7ms	55.79x
100k, 10x10 arrays	67.2ms	72.5ms	**32.8ms**	1127.6ms	1098.6ms	33.5x
500k, 10x10 arrays	328.9ms	358.9ms	**160.9ms**	5266.7ms	5061.3ms	31.46x
1000k, 10x10 arrays	663.7ms	719.0ms	**321.9ms**	10603.8ms	10167.3ms	31.59x

Table 3.3: Benchmark results of batched algorithms. Best Futhark performance is marked with **bold**
HH = Standard Householder, B-HH = Blocked Householder, GS = Gram-Schmidt process
Speedup is relative, calculated by $\frac{SciPy\ time}{Futhark\ GS\ time}$

From the results, we sadly see that the different Futhark implementations of QR decompositions are relatively slow compared to the Python libraries. The relatively unstable Gram-Schmidt implementation yields the best performance as expected due to a high level of parallelism. The standard Householder version is very slow, and due to the long run times, we did not benchmark it on arrays larger than 3000×3000. We clearly see that the Blocked version of Householder improved the performance a lot. However, we still expected much better performance from the blocked version after looking at the results from a similar implementation in C++ with the

CUBLAS library [AR09]. The results might be related to our implementation not being as optimized as it could possibly be. However, we think the slow results of the blocked Householder version in Futhark could be related to Futhark compiler optimizations.

The batched versions of the algorithms that calculate the QR Decomposition for many small matrices of equal size are all much faster than the Python libraries. This is due to the power of the heavily optimizing Futhark compiler that can compile all of the QR function calls to be computed in parallel and it will take the contents of the QR-function into account when optimizing. Python, on the other hand, can only do naive parallelization of the QR-function from LAPACK, which is hand-optimized for non-batched programs. We implemented the batched-QR in Python with `pool.map` from `multiprocessing`[3] to make sure we were using all CPU cores to make as fair of a comparison as possible.

[3] `https://docs.python.org/3/library/multiprocessing.html`

Chapter 4

Non-negative matrix factorization

Non-Negative matrix factorization (NMF), also often called non-negative matrix approximation, is an algorithm that projects data into lower-dimensional spaces. It does this by effectively reducing the number of features while retaining the basis information necessary to reconstruct original data. It decomposes an input matrix with non-negative components into the product of two non-negative matrices with lower dimensions. NMF is commonly used in image processing, text mining, bioinformatics, physics, recommendation systems for movies or online shopping, and much more.

4.1 The algorithm

Given the input matrix $\mathbf{A} \in \mathbb{R}_+^{m \times n}$ containing only non-negative coefficients, and a specified positive integer $1 \leq k \leq \texttt{min}(m, n)$, NMF produces two matrices $\mathbf{W} \in \mathbb{R}_+^{m \times k}$ and $\mathbf{H} \in \mathbb{R}_+^{k \times n}$ which also consist of non-negative coefficients. The product of $\mathbf{W}$ and $\mathbf{H}$ approximates $\mathbf{A}$:

$$\mathbf{A} \approx \mathbf{WH}$$

$$\underbrace{\begin{bmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \end{bmatrix}}_{\mathbf{A}} \approx \underbrace{\begin{bmatrix} \times & \times \\ \times & \times \\ \times & \times \end{bmatrix}}_{\mathbf{W}} \times \underbrace{\begin{bmatrix} \times & \times & \times \\ \times & \times & \times \end{bmatrix}}_{\mathbf{H}} \tag{4.1}$$

We chose the specified integer k so the approximation $\mathbf{WH}$ is a compressed form of the original data. Each column in $\mathbf{A}$ consists of n samples with m features, as an example, if we have m rows representing pixels and n columns each representing images of faces. NMF will then produce matrices the $\mathbf{W}$ and $\mathbf{H}$, where columns of $\mathbf{W}$ are images and $\mathbf{H}$ consists of weights in order to reconstruct an approximation of a given face [Dav02].

There exist many different algorithm variants to compute NMF to get a good approximation of $\mathbf{A}$. One variant is *Multiplicative Update Rules* (MU), which is simple, and the most commonly used algorithm for NMF [LR14]. The algorithm consists of updating the matrices $\mathbf{W}$ and $\mathbf{H}$

multiple times till either a threshold or a max number of iterations has been reached.

First of all, we need to initialize $\mathbf{W}$ and $\mathbf{H}$ as being non-negative and update them for each iteration:

$$\mathbf{H}_{ij} \leftarrow \mathbf{H}_{ij} \frac{(\mathbf{W}^T\mathbf{A})_{ij}}{(\mathbf{W}^T\mathbf{W}\mathbf{H})_{ij}} \tag{4.2}$$

$$\mathbf{W}_{ij} \leftarrow \mathbf{W}_{ij} \frac{(\mathbf{A}\mathbf{H})^T_{ij}}{(\mathbf{W}\mathbf{H}\mathbf{H}^T)_{ij}} \tag{4.3}$$

The algorithm computes the updates iteratively till the product of $\mathbf{W}$ and $\mathbf{H}$ yields a stable approximation of $\mathbf{A}$ or till it reaches the user-specified number of max iterations. To maximize efficiency for the algorithm, we can use divergence to measure the approximation quality. We do this with the Frobenius norm, which is the square root of the sum of the absolute squares of all matrix elements:

$$||\mathbf{A} - \mathbf{W}\mathbf{H}||^2 = \sum_{i,j}((\mathbf{A})_{ij} - (\mathbf{W}\mathbf{H})_{ij})^2 \tag{4.4}$$

By exploring the NMF source code from the well-known Python library Scikit-learn, we saw that they used the same divergence method. i.e., checking the Frobenius norm every tenth iteration of the loop, which we could use as inspiration for our implementation in Futhark.

4.2 Design and implementation

We decided to implement NMF with the multiplicative update rule and Frobenius norm due to its simplicity and speed of computational cost per iteration. The multiplicative update rule also relies heavily on matrix multiplication, which is a significant advantage when we want to utilize parallel programming speed in Futhark.

To initialize both $\mathbf{W}$ and $\mathbf{H}$, we made use of the random number generation library `cpprandom`[1] for Futhark to generate random integers from a normal distribution. With the usage of the normal distribution, several RNG states are computed and split into different states to generate a random 2D array. This gives us the following implementation:

[1] `https://github.com/diku-dk/cpprandom`

```
1  -- Work: O(n)
2  -- Span: O(1)
3  let stream (n: i64) (low: f32) (high: f32) =
4    let rng_state = rng_engine.rng_from_seed [123]
5    let rng_states = rng_engine.split_rng n rng_state
6    let (_,rng) = unzip (map (norm_dist.rand {mean = low,
7                                              stddev = high})
8                                              rng_states)
9    in rng
10
11 -- Work: O(n · m)
12 -- Span: O(1)
13 let stream2d (m: i64) (n: i64) (low: f32) (high: f32) =
14   unflatten m n (stream (n * m) low high)
15
16 -- Work: O(m · k + n · k)
17 -- Span: O(1)
18 let random_init (m: i64) (n: i64) (k: i64)
19                : (*[m][k]f32, *[k][n]f32) =
20   let W = stream2d m k 1 2
21   let H = stream2d k n 1 2
22   in (W, H)
```

In lines 5-9, we generate a random state from the seed `[123]` and then use `split_rng` to split it into n states to generate a random 1D array. We generate a random number from a normal distribution with **`map`** for each of these rng states and then return the random 1D array in the end. For each of these rng states, we generate a random number from a normal distribution with **`map`** and then return the random 1D array in the end. We then transform the 1D array into a 2D array with the function `stream2d` which splits it into a $m \times n$ array with the use of **`unflatten`**.

To check for divergence, we needed to efficiently compute the Frobenius norm since we had to compute it for every tenth iteration. As seen in (4.4) we can do this by looping through the matrix and calculating $||a_{ij}||^2$ for every element. Afterward, we add every element together using **`reduce`**, and then return the square-root from the sum.

In our implementation of the Frobenius norm, we can take advantage of the efficiency of a nested **`map`** in Futhark. We use it here to calculate the absolute value for every element in the matrix and then use **`reduce`** to sum the elements together.

```
1  -- Work: O(n · m)
2  -- Span: O(log(n) · O(log(m))
3  let frob_norm [m][n] (xss: [m][n]f32): f32 =
4    let abs_matrix = map(map (\x -> f32.abs(x)**2)) xss
5    let matrix_sum =
6      reduce (+) 0f32 (map(\xs -> reduce (+) 0f32 xs) abs_matrix)
7    in matrix_sum |> f32.sqrt
```

As described above, we compute the absolute square matrix in line 2 by a nested **map** going through each row in the matrix, with this summing the matrix row-wise; afterward, we compute the square root in line 2-3.

Initially, we implemented NMF without divergence, which was a straightforward solution, computed by updating $\mathbf{W}$ and $\mathbf{H}$ from (4.2) and (4.3) in a for-loop running until we reached the specified max number of iterations. However, we added divergence for a more tenable solution, inspired by the NMF source-code from scikit-learn[2]. $\mathbf{W}$ and $\mathbf{H}$ have been initialized randomly from a normal distribution, and for better divergence, we scale by the average mean of the input matrix.

```
1  -- Work: O(m · n)
2  -- Span: O(log(m) · log(n))
3  let random_init [m][n] (A: [m][n]f32) (k: i64)
4                         : (*[m][k]f32, *[k][n]f32) =
5    let avg = f32.sqrt (matmean A / f32.i64(k))
6    let W = matmul_scalar (stream2d m k 0 1) avg
7    let H = matmul_scalar (stream2d k n 0 1) avg
8    let W_abs = map(map (\x -> f32.abs(x))) W
9    let H_abs = map(map (\x -> f32.abs(x))) H
10   in (W_abs, H_abs)
```

Here `stream2d` is the same function as shown earlier, and the function `matmean` in line 2 is a simple calculation of the matrix mean, implemented as shown below:

```
1  -- Work: O(m · n)
2  -- Span: O(log(m) · log(n))
3  let matmean [m][n] (xss: [m][n]f32): f32 =
4    reduce (+) 0 (map (\xs -> reduce (+) 0 xs) xss) / f32.i64(m*n)
```

First, we calculate the absolute value for each element in the matrices before we return them to make sure they are non-negative as required. For divergence, we now needed to check if either the maximum number of iterations has been reached or if the matrices have diverged. We achieved this using a while-loop and a boolean indicating whether the while-loop should stop due to divergence or keep iterating. The most critical part of the development was to ensure divergence was done at the correct time and still be efficient. Only checking divergence for every tenth iteration granted substantially fewer computations. We could possibly scale this up even more due to working with large matrices - which does not diverge as fast as smaller matrices. The entire implementation of the NMF algorithm with the multiplicative update rule and Frobenius norm as divergence method is shown below:

[2] https://github.com/scikit-learn/scikit-learn/blob/42aff4e2e/scikit-learn/decomposition/_nmf.py#L1096

```
1  -- Work: O(l · n^3) where l is loop iterations
2  -- Span: O(l · log(n))
3  let nmf [m][n] (A: [m][n]f32)
4                 (k: i64)
5                 (max_iter: i64)
6                 (tol: f32)
7                 : ([m][k]f32, [k][n]f32, i64) =
8    let (W, H) = random_init A k
9    let init_norm = frob_norm(matsub A (matmul W H))
10   let prev_norm = init_norm
11   let n_iter = 0
12   let diverged = false
13   let curr_norm = 0.0
14   let (W, H, n_iter, _, _, _) =
15     loop (W, H, n_iter, prev_norm, curr_norm, diverged)
16          while ((n_iter < max_iter) && !diverged) do
17
18     -- Update H
19     let W_TA = matmul (transpose W) A
20     let W_TWH = matmul (transpose W) (matmul W H)
21     let H_update = matdiv_entrywise W_TA W_TWH
22     let H = matmul_entrywise H H_update
23
24     -- Update W
25     let AH_T = matmul A (transpose H)
26     let WHH_T = matmul W (matmul H (transpose H))
27     let W_update = matdiv_entrywise AH_T WHH_T
28     let W = matmul_entrywise W W_update
29
30     -- Check for divergence
31     in if tol > 0 && n_iter % 10 == 0 then
32         let curr_norm = frob_norm(matsub A (matmul W H))
33         in if (prev_norm - curr_norm) / init_norm < tol then
34           let diverged = true
35           in (W, H, n_iter, prev_norm, curr_norm, diverged)
36         else
37           let prev_norm = curr_norm
38           in (W, H, n_iter+1, prev_norm, curr_norm, diverged)
39       else
40         (W, H, n_iter+1, prev_norm, curr_norm, diverged)
41 in (W, H, n_iter)
```

The implementation consists of two parts, updating $\mathbf{H}$ and $\mathbf{W}$ (line 16-26) and checking for divergence every tenth iteration (line 28-37). The functions takes a total of four parameters: the input matrix $\mathbf{A}$, number of components k, maximum number of iterations before stopping the loop `max_iter`, and the tolerance for divergence `tol`.

The update of $\mathbf{H}$ and $\mathbf{W}$ is based on the equations (4.3) and (4.2), which is efficient since

it mostly consists of matrix multiplication, and simple matrix operations. For divergence we initialize the Frobenius norm with the `frob-norm` function from the input matrix and the randomly initialized matrices $\mathbf{W}$ and $\mathbf{H}$ from the function `random_init` as shown and described above.

With the variable `prev-norm` we check whether the current Frobenius Norm of the matrices has diverged, i.e., if it is below the specified tolerance as seen in line 31. If the matrices have not diverged, we update the previous matrix norm as the current norm for the next divergence check. The function returns the matrices $\mathbf{W}$, $\mathbf{H}$ and the integer `n_iter`. `n_iter` contains useful information regarding when the solution was accepted.

Implementing the Non-Negative Matrix Factorization algorithm without divergence was straightforward to implement in the Futhark programming model. However, it did not feel natural to implement a divergence check in the Futhark language. This was mainly due to the loop syntax requiring loop parameters to be returned in a tuple for every **`if`**, **`else`** statement. Futhark also does not support breaking a loop, which would have been convenient for divergence. As a result of this, we needed to use a while-loop and a boolean to identify when the algorithm diverged as explained above.

Finally a batched version of NMF can easily be achieved in Futhark with the use of **`map`** as seen below:

```
1 -- Work: O(l · m · n³)
2 -- Span: O(l · log(n))
3 let nmf_batched [m][n] (xsss: [m][n][n]f32) (k: i64)
4                        (max_iter: i64) (tol: f32)
5                   : *[m]([n][k]f32, *[k][n]f32, i64) =
6   map (\xss -> nmf xss k max_iter tol) xsss
```

4.3 Benchmarks

This section compares the performance of the implemented Futhark NMF function's performance with the NMF function from scikit-learn. We compared the two algorithms with different input parameters, testing both large arrays with many components and smaller arrays with a smaller number of components. Both algorithms are benchmarked with and without divergence to check if this had any significant performance difference.

The scikit-learn NMF function is designed with different solvers, divergence, and initialization methods. We used both the same solver, initialization, and divergence method for both Futhark and scikit-learn to make the comparison fair and viable for the following benchmarks. Most of the Futhark implementation of NMF consists of matrix multiplication, so we expect it to outperform scikit-learn due to the parallelism in Futhark.

All the Futhark benchmarks were conducted using `futhark bench`[3] with CUDA as backend on Futhark version 0.19, on datasets generated with `futhark dataset`[4]. The Python benchmarks were conducted 10 times in a for-loop where the average run time is shown in the tables below. The computer used has an an Intel(R) Core(TM) i5-8600K CPU @ 3.60GHz, and an NVIDIA GeForce GTX 1070 Ti graphics card with 8GB VRAM.

10 components, 0 tolerance, 500 max iterations			
Library / Matrix size	Futhark	scikit-learn	Speedup
1000x1000	502.3ms	265.8ms	0.53x
3000x3000	2800.6ms	2942.3ms	1.05x
5000x5000	4869.5ms	8364.5ms	1.71x
7000x7000	5951.02ms	16620.7ms	2.79x

Table 4.1: NMF Results with 10 components, no divergence and 500 max iterations
Speedup is relative, calculated by: $\frac{scikit-learn\ time}{Futhark\ time}$

40 components, 0.001 tolerance, 500 max iterations			
Library / Matrix size	Futhark	scikit-learn	Speedup
1000x1000	93.8ms	182.9ms	1.95x
3000x3000	299.9ms	692.4ms	2.31x
5000x5000	599.1ms	1442.2ms	2.41x
7000x7000	961.4ms	2305.4ms	2.40x
10000x10000	1066.2ms	4805.2ms	4.51x

Table 4.2: NMF Results with 40 components, 0.001 divergence tolerance and 500 max iterations
Speedup is relative, calculated by: $\frac{scikit-learn\ time}{Futhark\ time}$

500 components, 0.001 tolerance, 500 max iterations			
Library / Matrix size	Futhark	scikit-learn	Speedup
1000x1000	1778.8ms	2859.4ms	1.61x
3000x3000	11784.7ms	13798.4ms	1.17x
5000x5000	30418.1ms	32709.2ms	1.08x
7000x7000	56262.6ms	66353.8ms	1.18x

Table 4.3: NMF Results with 500 components, 0.001 tolerance and 500 max iterations
Speedup is relative, calculated by: $\frac{scikit-learn\ time}{Futhark\ time}$

[3] `https://futhark.readthedocs.io/en/latest/man/futhark-bench.html`
[4] `https://futhark.readthedocs.io/en/latest/man/futhark-dataset.html`

2 components, 0.001 tolerance, 300 max iterations			
Library / Matrix size	Futhark	scikit-learn	Speedup
10k, 10x10 arrays	43.7ms	3433.1ms	78.56x
100k, 10x10 arrays	359.1ms	33682.4ms	95.7x
100k, 10x10 arrays	1686.0ms	-	-
1000k, 10x10 arrays	3353.1ms	-	-

Table 4.4: Batched NMF results with 2 components, 0.001 tolerance and 300 max iterations
Speedup is relative, calculated by: $\frac{scikit-learn\ time}{Futhark\ time}$

The benchmarks results show that scikit-learn is a little faster for smaller size arrays. However, for 3000×3000 matrices and above, Futhark is significantly faster than scikit-learn. We expected this due to parallelism and matrix multiplication. Futhark is also way faster for the batched NMF than a multi-threaded batched NMF function in scikit-learn; we also discussed the reasons behind this batched QR function in section 3.4. The benchmarks for scikit-learn with more than 100k, 10x10 arrays have been omitted since they would take a lot of time to run.

Chapter 5

Cholesky Decomposition

A Cholesky decomposition is the decomposition of a Hermitian positive-definite matrix $\mathbf{A}$ in the form of:

$$\mathbf{A} = \mathbf{L}\mathbf{L}^* \tag{5.1}$$

$\mathbf{L}$ is a lower triangular matrix, and $\mathbf{A}$ is a Hermitian matrix meaning it is equal to its own conjugate transpose. Positive-definite denotes that the scalar $z^T\mathbf{A}z$ is positive for any non-zero column vector z. If $\mathbf{A}$ only contains real numbers, which is most common, the decomposition can be written as:

$$\mathbf{A} = \mathbf{L}\mathbf{L}^T = \begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \begin{bmatrix} l_{11} & l_{21} & l_{31} \\ 0 & l_{22} & l_{32} \\ 0 & 0 & l_{33} \end{bmatrix} \tag{5.2}$$

From a first glance, one may think that these Hermitian positive-definite matrices rarely emerge in real-world uses, but this is not the case. It can be used in linear least squares, Kalman filters, and for matrix inversion. Cholesky has a lot in common with LU-decomposition, but Cholesky is said to be roughly two times faster.[Pre92]

5.1 The algorithm

There are two variations of this algorithm named Cholesky–Banachiewicz, and Cholesky–Crout. Cholesky–Banachiewicz proceeds row by row, whereas Cholesky–Crout does it column by column. We chose Cholesky–Crout for our implementation since its access pattern seemed better suited for parallel computing.[Rus14] The goal of the algorithm is to achieve:

$$\mathbf{L} = \begin{bmatrix} \sqrt{a_{11}} & 0 & 0 \\ a_{21}/l_{11} & \sqrt{a_{22} - l_{21}^2} & 0 \\ a_{31}/l_{11} & (a_{32} - l_{31}l_{21})/l_{22} & \sqrt{a_{33} - l_{31}^2 - l_{32}^2} \end{bmatrix} \tag{5.3}$$

We can generally express this with the following formulas:

$$l_{jj} = \sqrt{a_{jj} - \sum_{k=1}^{j-1} l_{jk}^2} \tag{5.4}$$

$$l_{ij} = \frac{1}{l_{jj}}(a_{ij} \sum_{k=1}^{j\ 1} l_{ik} l_{jk}) \quad \text{for } i > j \tag{5.5}$$

5.2 Design and implementation

For a simple implementation of Cholesky, we need a function that zeros out the elements above the diagonal:

```
1  -- Work: O(n^2)
2  -- Span: O(1)
3  let tril [n] (A: [n][n]f32): *[n][n]f32 =
4    tabulate_2d n n (\i j -> if j <= i then A[i,j] else 0.0)
```

As formerly mentioned, we chose to implement the variant of Cholesky that proceeds column by column. For every iteration of the for-loop below, the diagonal element becomes its square root. Then the rest of the column is divided by this number. Note that this can not result in division by zero if the input matrix is Hermitian and positive definite. Then the outer product of the last part of the column is calculated and subtracted from the remaining rows, as seen in the implementation below:

```
1  -- Work: O(n^3)
2  -- Span: O(n)
3  let cho [n] (A: *[n][n]f32): [n][n]f32 =
4    tril <| loop A for j in 0..<n do
5      let k = n - j - 1
6      let A[j, j] = f32.sqrt A[j,j]
7      let A[j+1:,j] = map (/A[j,j]) A[j+1:n,j]
8      let v = A[j+1:n,j] :> *[k]f32
9      let op = outer v v :> [k][k]f32
10     let A[j+1:n,j+1:n] = matsub (A[j+1:n,j+1:n] :> [k][k]f32) op
11     in A
```

It was immediately apparent that the result of this algorithm only depends on the lower triangular part of the matrix, so with the implementation above, we are essentially doing double the work. We had the idea to convert the algorithm into a version that does all the computation on a 1D array only containing the lower triangular elements of the matrix.

We use the following function to find the indices of the lower triangular elements of the matrix with dimensions $n \times n$. Since our algorithm mostly operates on the columns, this function re-

turns the indices in column order starting from the left. As an example, if $n = 2$, this function returns "[(0,0), (1,0), (1,1)]". These indices are of type i16 since it offered a very good improvement in terms of memory usage as well as run-time as opposed to using i64. However, this also limits the size of matrices in our Cholesky implementation to be 32767×32767 at max. This is fine for most modern GPU's because you will run out of memory way before reaching this threshold. This function's span can be improved to $O(1)$ with the use of floating-point math, but this was not needed since this function is only called once in our flat Cholesky implementation.

```
-- Work: O(n)
-- Span: O(log(n))
let tril_indices (n: i64): *[](i16, i16) =
  tabulate (n * n) (\x ->
    let i = i16.i64 (x / n)
    let j = i16.i64 (x % n)
    in if j >= i then (j, i) else (-1i16, -1i16)
  ) |> filter (\ (x, _) -> x != -1)
```

We can use the indices returned from the above function to compute the lower triangular of the outer product. This almost halves the constant factors of computing the outer product, resulting in much better performance for larger matrices.

```
-- Work: O(n^2) where n is the length of v
-- Span: O(1)
let low_outer (v: []f32) (ti: [](i16, i16)): *[]f32 =
  map (\ x ->
    let (i, j) = x
    in #[unsafe] v[i] * v[j]
  ) ti
```

With the functions above, we could implement flat Cholesky. `tri_num n` simply returns the n'th triangle number, and `flat_cho` returns the lower triangular as a 1D-array.

```
-- Work: O(n^3)
-- Span: O(n)
let flat_cho [n] (A: [n][n]f32): []f32 =
  -- Convert A to 1D array of lower triangular elements
  let tril_is = tril_indices n
  let s = length tril_is
  let A = map (\(i, j) -> A[i, j]) tril_is
  let m = tri_num n
  let A = loop A for j in 0..<n-1 do
    -- Take the square root of the diagonal element
    let di = (tri_num n) - (tri_num (n - j))
    let sdi = f32.sqrt A[di]
    let A[di] = sdi

    -- Division step
```

```
16      let steps = n - j - 1
17      let start = di + 1
18      let stop = start + steps
19      let A[start:stop] = map (/sdi) A[start:stop]
20
21      -- Elimination step
22      let skip_cols = j + 1
23      let rest = n - skip_cols
24      let k = tri_num rest
25      let outer_is = tril_is[s-k:]
26      let outer_is = map (\(a, b) ->
27        let (c, d) = outer_is[0]
28        in (a - c, b - d)
29      ) outer_is
30      let col = A[start:stop] :> *[steps]f32
31      let outer = (low_outer col outer_is) :> [k]f32
32      let A[m-k:] = map2 (\x y -> x - y) (A[m-k:] :> [k]f32) outer
33      in A
34
35    -- Take the square root of the last element
36    let di = m - 1
37    let A[di] = f32.sqrt A[di]
38    in A
```

Most people would probably prefer to have the lower triangular returned as a matrix instead of a 1D array. Therefore, we have a function that extracts the rows from the array returned from `flat_cho` and appends zeros:

```
1  -- Work: O(n^3)
2  -- Span: O(n)
3  let cholesky [n] (A:[n][n]f32): [n][n]f32 =
4    let L = flat_cho A
5    in tabulate n (\x ->
6      let tn = tri_num n
7      let start = tn - (tri_num (n - x))
8      let stop = tn - (tri_num (n - x - 1))
9      in replicate n 0 with [x:] = L[start:stop]
10   ) |> transpose
```

We can easily achieve a batched variant by wrapping the function with a **map**.

```
1  -- Work: O(m · n^3)
2  -- Span: O(n)
3  let flat_cho_batched [m][n] (xsss: [m][n][n]f32): [m]([]f32) =
4    let tn = tri_num n
5    in map (\xss -> (flat_cho xss) :> [tn]f32) xsss
```

5.3 Benchmarks

In this section, we compare the performance of our Cholesky function `flat_cho` with Cholesky from NumPy and scikit-learn. NumPy and scikit-learn both use the `dpotrf` subroutine from LAPACK to compute the Cholesky factorization.

All the Futhark benchmarks were conducted using `futhark bench`[1] with CUDA as backend on Futhark version 0.19, on datasets generated with `futhark dataset`[2]. The Python benchmarks were conducted 10 times in a for-loop where the average run time is shown in the tables below. The computer used has an an Intel(R) Core(TM) i5-8600K CPU @ 3.60GHz, and an NVIDIA GeForce GTX 1070 Ti graphics card with 8GB VRAM.

Library / Matrix size	Futhark Standard Cho	**Futhark Flat Cho**	NumPy	SciPy	Speedup
1000x1000	57.3ms	**40.98 ms**	8.9 ms	6.0 ms	0.15x
3000x3000	996.9ms	**539.04 ms**	191.5 ms	149.2 ms	0.27x
5000x5000	4486.9ms	**2255.17 ms**	463.6 ms	409.9 ms	0.18x
7000x7000	11884.7ms	**5951.02 ms**	1097.3 ms	943.3 ms	0.16x
10000x10000	35393.2ms	**17031.33 ms**	2586.2 ms	2413.7 ms	0.14x
12000x12000	66157.9ms	**29294.83 ms**	4185.8 ms	3912.4 ms	0.13x

Table 5.1: Benchmark of Cholesky. Best Futhark performance is marked with **bold**
Speedup is relative, calculated by: $\frac{SciPy\ time}{Futhark\ flat-cho\ time}$

Unfortunately, from the results, the Futhark implementation is quite a bit slower than the Python libraries. It is worth mentioning that `dpotrf` is fully parallel and makes use of all CPU cores while running. `dpotrf` uses a highly optimized blocked variant of the algorithm. We wanted to implement a blocked-version ourselves, but we eventually had to give up. Most, if not all, implementations of blocked Cholesky use recursion and are far from trivial to implement in Futhark or any other language for that matter. We believe that the reason our implementation is not as fast or faster than the Python libraries is likely because we do not apply the same number of highly sophisticated optimizations as LAPACK. We are unsure why the relative difference gets larger for larger matrices; we think this could be related to compiler optimizations.

One huge benefit of having a Cholesky implementation in Futhark is that we can easily make a batched version that calculates the Cholesky factorization of many small matrices of equal size efficiently. Despite utilizing `pool.map` from `multiprocessing`[3] for a batched Python implementation, Futhark is much faster, as seen from the results in the table below:

[1] `https://futhark.readthedocs.io/en/latest/man/futhark-bench.html`
[2] `https://futhark.readthedocs.io/en/latest/man/futhark-dataset.html`
[3] `https://docs.python.org/3/library/multiprocessing.html`

Library / Matrix size	Futhark	NumPy	SciPy	Speedup
10k, 10x10 arrays	0.7ms	92.5ms	94.1ms	134.43x
100k, 10x10 arrays	7.0ms	656.4ms	635.5ms	90.81x
500k, 10x10 arrays	37.4ms	3200.0ms	3121.3ms	83.45x
1000k, 10x10 arrays	68.8ms	6485.5ms	7034.1ms	102.2x

Table 5.2: Benchmark of batched flat Cholesky.
Speedup is relative, calculated by: $\frac{SciPy\,time}{Futhark\,time}$

Chapter 6

Matrix Determinant

The determinant is a scalar computed by a square $n \times n$ matrix. It is commonly used in calculus, calculating the inverse of the matrix or solving systems of linear equations. There are many equivalent ways to compute the determinant, but we focused on three methods: The Cholesky decomposition method, the Doolittle LU decomposition method, and the Dodgson condensation method. We decided to implement all three methods since they are all commonly used. Furthermore, we found it interesting not only to analyze and find the quickest method theoretically but also practically.

6.1 Dodgson condensation method

The Dodgson condensation method is a method to compute the determinants of a square matrix. Given an $n \times n$ matrix, the method computes an $(n-1) \times (n-1)$ matrix, an $(n-2) \times (n-2)$ matrix, and so forth. When the algorithm arrives at a 1×1 matrix, it terminates, with the only entry in the matrix being the determinant of the original matrix.

Using the Dodgson condensation algorithm to determine the determinant involves three separate steps:[Abe07]

1. From the input matrix $\mathbf{A}$, create an $(n-1) \times (n-1)$ matrix $\mathbf{B}$, which consists of the determinants of every 2×2 sub-matrix formed from consecutive rows and columns in $\mathbf{A}$.
2. Perform step 1 on the matrix $\mathbf{B}$ to make the $(n-2) \times (n-2)$ matrix $\mathbf{C}$, and divide each term in $\mathbf{C}$ by the corresponding term in the interior of $\mathbf{A}$
3. We let $\mathbf{A} = \mathbf{B}$, and $\mathbf{B} = \mathbf{C}$. If we have not found a 1×1 matrix, repeat step 2 until this matrix occurs. The only entry in the 1×1 matrix is the determinant of matrix $\mathbf{A}$.

In step 2, we use the following definition to compute the determinants of the 2×2 submatrices:

$$|\mathbf{A}| = \begin{bmatrix} a & b \\ c & d \end{bmatrix} = ad - bc \tag{6.1}$$

Although this algorithm seems simple and can compute the determinant of any matrix with the same procedure, it has a significant flaw - the determinant of any interior matrix cannot be 0 because this would resolve in a division with zero. Sometimes, we can solve this problem by adding an extra step to the algorithm, including row and column exchanges of the input matrix. Unfortunately, this solution may not always work [LH17].

It is important to mention an variant of the Dodgson algorithm which always yield the correct solution does exist [LH17]. However, this is a symbolic algorithm that uses symbols as representatives for the elements that cause entries in the interior of A to be zeros, and we assessed this to be outside our project scope.

The Dodgson algorithm still has a remarkable advantage because it gives the opportunity never to compute a determinant of order greater than two. We decided to implement this algorithm to compute the determinant since the computation of the 2×2 can be executed in parallel and were therefore well suited for our project [HM12].

Design and implementation

Our implementation of the Dodgson condensation algorithm is straightforward. Since the size of the input matrix $\mathbf{M}$ decreases for each iteration, we made the function loop between the steps, $m - 1$ times. Afterward, the function terminates, returning a single number. The following shows the implementation of the algorithm:

```
1  -- Work: O(n^3)
2  -- Span: O(n)
3  let det [m] (A: [m][m]f32): f32 =
4    let (A, _) = loop (A, interior) = (A, A) for i < m-1 do
5      let n = length A
6      let s = n - 1
7      let ass = A[:n-1,:n-1] :> [s][s]f32
8      let bss = A[1:n,:n-1]  :> [s][s]f32
9      let css = A[:n-1,1:n]  :> [s][s]f32
10     let dss = A[1:n,1:n]   :> [s][s]f32
11     in if i != 0 then
12       let ess = interior :> [s][s]f32
13       let mat =
14         map5 (map5 (\a b c d e ->
15                 (a * d - b * c) / e)) ass bss css dss ess
16       in (mat, A[1:n-1, 1:n-1])
17     else
18       let mat = map4 (map4 (\a b c d ->
19                 a * d - b * c)) ass bss css dss
20       in (mat, A[1:n-1, 1:n-1])
21   in A[0, 0]
```

Since most of the calculations happen in the second step of the algorithm (finding the determinant of the submatrices and dividing the elements), we wanted to implement this step as parallel as possible. In lines 6-9, we split the matrix into 4 submatrices; Figure 1 shows how the submatrices to a 3×3 matrix would be.

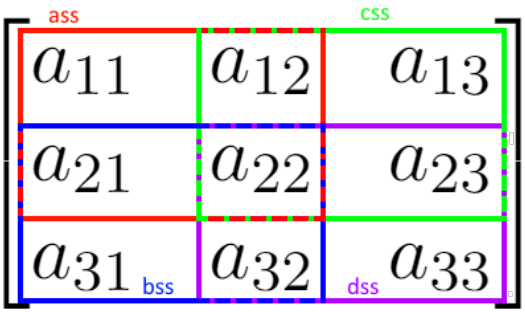

Figure 6.1: Visualization of the ass, bss, css, dss submatrices in a 3×3 matrix

This fragmentation made it possible to calculate every determinant parallel to each other using nested higher-order function **`map4`** or **`map5`** if a division with the interior is needed. We do this in lines 10-16.

Be aware that our implementation does not handle the situation where the corresponding term in the interior of $\mathbf{A}$ is zero. We do not have any division with zero handlings, which results in our program returning the not a number value: `f32.nan`, if we run into this problem.

Because of this, our primitive version only works if no zeros occur in the interior. We did this to estimate the lower-bound runtime we could expect if we were to implement a stable version. Since the primitive version's benchmarks already had poor benchmarking results (See section 6.4), and the full version only would be worse, we decided to stop further work with this algorithm and continued with another determinant algorithm.

6.2 Doolittle LU Decomposition method

This method consists of two steps: Computing the lower triangular matrix and the upper triangular matrix using the Doolittle LU Decomposition algorithm and finding the determinant from these matrices.

The algorithm

The LU Decomposition or *Lower-Upper Decomposition* factors the matrix $\mathbf{A}$ as the product of a lower triangular matrix $\mathbf{L}$ and an upper triangular matrix $\mathbf{U}$, i.e., $\mathbf{A} = \mathbf{LU}$. The method is very similar to Gaussian elimination and often used to repeatedly solve several equations with the same left-hand side, inverting a matrix or computing the matrix's determinant. We decided

to focus on the *Doolittle LU Decomposition* since this is a simple algorithm with the same procedure for any matrix size.

To find the $\mathbf{LU}$ matrices using the Doolittle Algorithm, we first take an $n \times n$ matrix $\mathbf{A}$ and assume that an LU decomposition exists. We now create the $\mathbf{LU}$ matrices where the diagonal elements of $\mathbf{L}$ are 1. As an example if $\mathbf{A}$ is a 3×3 matrix, we would have the following matrices:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{bmatrix} \cdot \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix} \tag{6.2}$$

To find the values of the elements, we form equations for each of the unknowns in $\mathbf{L}$ and $\mathbf{U}$ defined by multiplying the rows in $\mathbf{L}$ with the columns in $\mathbf{U}$ and equating them to the corresponding element in $\mathbf{A}$.

When we find the $\mathbf{LU}$ matrices, the determinant is equal to the $\mathbf{U}$ matrix's diagonal elements multiplied together.

Keep in mind that LU decomposition is closely related to Gaussian elimination, which is unstable in its pure form[Rei14]. Computing the LU matrices may, therefore, not always be possible. To solve this problem, we can compute permutation matrices that meet the following:

$$\mathbf{PA} = \mathbf{LU} \tag{6.3}$$

We can decompose every square matrix in this form, and it makes the decomposition numerically stable [TB97][Tab17]. However, since our project's scope did not include the development of algorithms to be used in practice; we decided only to implement the Doolittle LU decomposition without pivoting.

Implementation and design

The following shows the implementation of the Doolittle LU Decomposition algorithm:

```
1 -- Work: O(n^3)
2 -- Span: O(n)
3 let lu [n] (A: *[n][n]f32) : ([n][n]f32, [n][n]f32) =
4   let L = identity n
5   let (L, A) = loop (L, A) for k in 0..<n do
6     let L[k+1:n, k] = vecdiv_scalar A[k+1:n,k] A[k,k]
7     let A[k+1:n, :] = matsub A[k+1:n, :] (outer upper A[k,:])
8     in(L,A)
9   in (L,A)
```

In the algorithm's implementation, we take advantage of the first row in the $\mathbf{U}$ matrix being equivalent to the first row in $\mathbf{A}$; this means that we do not create $\mathbf{U}$ from scratch but use $\mathbf{A}$

instead. $\mathbf{L}$ has ones in its diagonal, so we initialize this to an identity matrix with the size equal to $\mathbf{A}$ (line 2).

The function is primarily a loop since the unknowns in the $\mathbf{LU}$ matrices are dependent on each other. We loop k times, where each iteration involves solving the equations to one column in $\mathbf{L}$ and one row in $\mathbf{U}$, starting from the first column in $\mathbf{L}$ and the second row in $\mathbf{U}$ (Since the first row in $\mathbf{U}$ is equivalent to the first row in $\mathbf{A}$).

The first column in $\mathbf{L}$ is only dependent on the first row in $\mathbf{U}$, making it possible to find the unknowns in this column in the first iteration. This also allows us to find the unknowns in the second row of $\mathbf{U}$. We now start the next iteration, finding the unknowns in the second column in $\mathbf{L}$, and so on.

Another benefit of using $\mathbf{U} = \mathbf{A}$ is that we can gradually remove terms in the equations once we find them and update $\mathbf{A}$. This makes it possible to calculate unknowns with the same procedure for each iteration and removes recalculation of terms.

The unknowns in the $\mathbf{L}$ matrix is computed column-wise in lines 4-5. Here we divide each element below the diagonal in column $\mathbf{k}$ with the diagonal element $\mathbf{A}[k, k]$.

Finding the unknowns in the $\mathbf{U}$ matrix is a bit more complicated. The vector calculated as `upper` is multiplied with the k'th in $\mathbf{U}$ row to make a new matrix. We now subtract this matrix from the submatrix $\mathbf{A}[k + 1 : n, :]$ and replace it in $\mathbf{A}$.

When the $\mathbf{LU}$ is found, the diagonal is extracted from $\mathbf{U}$ with **`map`** and multiplied together with **`reduce`** to compute the determinant:

```
1 -- Work: O(n^3)
2 -- Span: O(n)
3 let det [n] (A: *[n][n]f32) : f32 =
4   let (_, U) = lu A
5   let diag = map(\i -> U[i,i]) (iota n)
6   in reduce (*) 1.0 diag
```

Since we do not use LU decomposition with pivoting, nor any element evaluation, we do not detect any zero division before it happens. Because of this, our function requires specialized inputs in the form of a positive definite symmetric matrix or a diagonally dominant matrix, to compute a usable determinant[RLH04]. However, this gave us an idea of how well the version with pivoting would perform in benchmarks.

6.3 Cholesky Decomposition determinant

This method uses the same two-step procedure as Doolittle LU: Computing the decomposition matrices using Cholesky decomposition and computing the determinant from these matrices.

After the Cholesky decomposition has been made and, the $\mathbf{L}$ is found, which is included in chapter 5, the determinant is just the square of the product of the diagonal elements of the matrix $\mathbf{L}$.

Implementation and design

Since the `flat_cho` function was the fastest implemented Cholesky Decomposition, we decided to use it to compute the L matrix for the determinant. The following shows our implementation for computing the Cholesky Decomposition determinant.

```
1 -- Work: O(n^3)
2 -- Span: O(n)
3 let det [n] (A: [n][n]f32) =
4   let C = flat_cho A
5   let i = tail (iota(n+1))
6   let rev_i = i[::-1]
7   let index = scan (+) 0 rev_i
8   let di = map (\x -> if x == 0 then 0 else index[x-1]) (iota(n))
9   in reduce (*) 1 (map (\x -> C[x]**2) di)
```

Since the output matrix of `flat_cho` is a flat $1 \times n$ matrix, we needed some way to extract the diagonal elements. To do this, we create an array with values equal to the diagonal elements' indexes (line 3-6). We now extract these elements from the flat L matrix, square them, and multiply them together (line 7).

We can easily achieve a batched variant of the determinant function by wrapping it with a **`map`**:

```
1 -- Work: O(m · n^3)
2 -- Span: O(n)
3 let det_cho_batched [m][n] (xsss: [m][n][n]f32): *[m]f32 =
4   map (\xss -> det_cho xss) xsss
```

6.4 Benchmarks

The table below shows the performance of our three implemented determinant algorithms: The Dodgson condensation, the Doolittle LU decomposition determinant, and the Cholesky decomposition determinant. These are all compared to the NumPy `slogdet` function, which computes the sign and (natural) logarithm an array's determinant.

All the Futhark benchmarks were conducted using `futhark bench`[1] with CUDA as backend

[1] `https://futhark.readthedocs.io/en/latest/man/futhark-bench.html`

on Futhark version 0.19, on datasets generated with `futhark dataset`[2]. The Python benchmarks were conducted 10 times in a for-loop where the average run time is shown in the tables below. The computer used has an an Intel(R) Core(TM) i5-8600K CPU @ 3.60GHz, and an NVIDIA GeForce GTX 1070 Ti graphics card with 8GB VRAM.

Library / Matrix size	Futhark Dodgson-det	Futhark Doolittle LU-det	**Futhark Cholesky-det**	NumPy	Speedup
1000x1000	35.6ms	35.7ms	**44.5ms**	11.2ms	0.25x
3000x3000	904.4ms	679.9ms	**532.2ms**	152.9ms	0.28x
5000x5000	4150.8ms	3035.5ms	**2227.1ms**	556.8ms	0.25x
7000x7000	11424.9ms	8241.6ms	**5921.3ms**	1337.0ms	0.23x
10000x10000	33183.0ms	23358.6ms	**16962.7ms**	3799.3ms	0.22x

Table 6.1: Benchmark results of determinant algorithms. Best Futhark performance is marked with **bold**
Speedup is relative, calculated by $\frac{NumPy\ time}{Futhark\ Cholesky-det\ time}$

The above shows that the Futhark implementations are relatively slow compared to the NumPy version. However, here it is essential to consider the implementation of the NumPy function. `slogdet` uses the LAPACK `dgetrf` implementation of the LU Decomposition, which is an optimized blocked version of the LU decomposition. The difference between blocked and unblocked algorithms is very significant when comparing large matrices' runtime, as seen in Householder QR. This meant that the `slogdet` function already had a notable advantage against our Doolittle- and Cholesky decomposition determinant versions since computing the decomposition matrices includes most of the calculations. Unfortunately, we did not have time to work on blocked versions of the determinant algorithms.

We also implemented a batched Cholesky decomposition determinant version, which gave us much better benchmarking results. Compared to a batched `slogdet` with the use of `pool.map`, it was possible to reach above 60 times the speedup on every tested input size:

Library / Matrix size	Futhark Batched Cholesky-det	NumPy	Speedup
10k, 10x10 arrays	0.8ms	214.2ms	267.75x
100k, 10x10 arrays	7.0ms	542.4ms	77.5x
500k, 10x10 arrays	32.1ms	2295.0ms	71.5x
1000k, 10x10 arrays	64.4ms	4288.0ms	66.5x

Table 6.2: Benchmark results of batched determinant with Cholesky.
Best Futhark performance is marked with **bold**.
Speedup is relative, calculated by $\frac{NumPy\ time}{Futhark\ time}$

[2] `https://futhark.readthedocs.io/en/latest/man/futhark-dataset.html`

Chapter 7

Solving Linear Systems

One way of expressing linear systems is $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A}$ is a matrix of the coefficients, $\mathbf{x}$ is a vector of the unknowns, and $\mathbf{b}$ is a vector of the right-hand side. We can compute the solution to a linear system in numerous different ways. It is possible to compute it by $\mathbf{LU}$ decomposition, Gauss-Jordan elimination, as well as Cholesky if $\mathbf{A}$ is positive definite Hermitian. [GL13b]

Solving equations with Gauss-Jordan
Gauss-Jordan simply does a series of elementary row operations to produce a matrix that is in reduced row-echelon form. This can be performed by ensuring that each pivot has the value 1 and eliminating values above and below the pivots. In our implementation, we perform partial pivoting. This is the act of exchanging rows to obtain the largest possible pivot - this makes it less susceptible to round-off errors. This can be expanded to full pivoting, where both rows and columns are exchanged to obtain the largest possible pivot, but partial pivoting should be sufficient for our project's purpose.

Solving equations with LU- and Cholesky factorization
Once we have computed $\mathbf{L}$ and $\mathbf{U}$, we can solve for the right-hand side vector $\mathbf{b}$ by a two-step process. First, we solve $\mathbf{Ly} = \mathbf{b}$ by forward substitution. Finally, we can compute $\mathbf{Ux} = \mathbf{y}$ with back substitution. If we compute $\mathbf{L}$ has with Cholesky, we can simply transpose it to get $\mathbf{U}$ and follow the same process to solve the linear system.

Matrix inverse
A $n \times n$ matrix $\mathbf{A}$ is invertible if there exists a matrix $\mathbf{B}$ such that:

$$\mathbf{AB} = \mathbf{BA} = \mathbf{I}_n \tag{7.1}$$

$\mathbf{B}$ is the inverse of $\mathbf{A}$, which is commonly denoted as $\mathbf{A}^{-1}$. Matrix inverses have many applications least squares regression, 3D transformations, etc. We can compute the matrix inverse by solving the linear system $\mathbf{AX} = \mathbf{I}$, for example:

$$\left[\begin{array}{ccc|ccc} a_{11} & a_{12} & a_{13} & 1 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 & 1 & 0 \\ a_{31} & a_{32} & a_{33} & 0 & 0 & 1 \end{array}\right] \longrightarrow \left[\begin{array}{ccc|ccc} 1 & 0 & 0 & x_{11} & x_{12} & x_{13} \\ 0 & 1 & 0 & x_{21} & x_{22} & x_{23} \\ 0 & 0 & 1 & x_{31} & x_{32} & x_{33} \end{array}\right] \tag{7.2}$$

Here $\mathbf{X}$ is the inverse. Back-substitution is unnecessary when computing the inverse with Cholesky because $\mathbf{A}^{-1} = (\mathbf{L}\mathbf{L}^T)^{-1} = (\mathbf{L}^{-1})^T\mathbf{L}^{-1}$ when $\mathbf{A}$ is a positive definite Hermitian matrix. Therefore, we only need to compute the inverse of the lower triangular, which is done by forward-substitution.

7.1 Gauss-jordan Elimination

```
for i = 1 to n
  Find largest pivot p in the i'th column
  if i ≠ p then swap row i and p
  for j = 1 to n
    A_i = 1/a_ij A_i
  for j = 1 to n
    if i ≠ j then A_j = A_j − a_ij A_i
```

7.2 Forward- and Back-substitution

Forward substitution is performed by:

$$y_i = \frac{\left(b_i - \sum_{j=1}^{i-1} l_{ij} y_j\right)}{l_{ii}} \quad \text{for i = 1 to n} \tag{7.3}$$

Similarly, back substitution is performed by:

$$x_i = \frac{\left(b_i - \sum_{j=1}^{i-1} u_{ij} y_j\right)}{u_{ii}} \quad \text{for i = n to 1} \tag{7.4}$$

7.3 Design and implementation

In order to find the index of the maximum pivot, we needed an `argmax`-function. This can be fairly efficiently expressed in Futhark with the use of `reduce_comm` which is span $O(log(n))$.

```
-- Work: O(n))
-- Span: O(log(n))
let argmax (arr: []f32) =
  reduce_comm (\(a, i) (b, j) ->
    if a < b
    then (b, j)
```

```
    else if b < a then (a, i)
    else if j < i then (b, j)
    else (a, i)
  ) (0, 0) (zip arr (indices arr))
```

Below is the implementation of partial Gauss-Jordan elimination. The span of this function can be reduced to $O(m)$ if we used no pivoting since finding the max pivot with `argmax` is what makes the iterations $O(log(m))$ and not $O(1)$. However, with no pivoting, it would be highly unstable.

```
-- Work: O(min(m, n) · m · n))
-- Span: O(min(m, n) · log(m))
 let gauss_jordan [m][n] (A:[m][n]f32) =
    loop A = copy A for i < i64.min m n do
    -- Find largest pivot
    let p = A[i:,i] |> map f32.abs |> argmax |> (.1) |> (+i)
    let A = if p != i then swap i p A else A
    let irow = map (/A[i,i]) A[i]
    -- Eliminate entries above and below the pivot
    in tabulate m (\j ->
      let scale = A[j,i]
      in map2 (\x y ->
        if j != i then y - scale * x else x
      ) irow A[j]
    )

let gauss_solveAB [m][n] (A:[m][m]f32) (B:[m][n]f32) : [m][n]f32 =
    let AB = gauss_jordan (hStack A B)
    in AB[:m, m:] :> [m][n]f32

let gauss_solveAb [m] (A:[m][m]f32) (b:[m]f32) =
  unflatten m 1 b |> gauss_solveAB A |> flatten_to m
```

Forward substitution is called for lower triangular matrices $\mathbf{L}$; first, it computes y_1, then substitutes that forward into the next equation to solve for y_2, and repeats through to y_n. For upper triangular matrices, we work backward, first computing x_n, then substituting that back into the previous equation to solve for x_{n-1}, and repeating through to x_1.

```
-- Work: O(n^2)
-- Span: O(n · log(n))
let forward_substitution [n] (L: [n][n]f32) (b: [n]f32): [n]f32 =
  let y = replicate n 0.0f32
  in loop y for i in 0..<n do
    let sumy = dotprod L[i,:i] y[:i]
    let y[i] = (b[i] - sumy) / L[i,i]
    in y

```

```
-- Work: O(n^2)
-- Span: O(n · log(n))
let back_substitution [n] (U: [n][n]f32) (y: [n]f32): [n]f32 =
  let x = replicate n 0.0f32
  in loop (x) for j in 0..<n do
    let i = n - j - 1
    let sumx = dotprod U[i,i+1:n] x[i+1:n]
    let x[i] = (y[i] - sumx) / U[i,i]
    in x
```

Now we can solve linear systems by back-substitution of $\mathbf{U}$ and forward-substitution of $\mathbf{L}$ returned from the $\mathbf{LU}$-function. With Cholesky, we obtain $\mathbf{U}$ by transposing $\mathbf{L}$. We can solve for multiple right-hand side vectors $\mathbf{B}$ by solving for every $b_1 ... b_m$ with the use of a **map**.

```
let solveLUb [n] (L: [n][n]f32) (U: [n][n]f32) (b: [n]f32) =
  forward_substitution L b |> back_substitution U

let lu_solveAb [n] (A: [n][n]f32) (b: [n]f32)  =
  let (L, U) = lu A
  in solveLUb L U b

let lu_solveAB [n][m] (A: [n][n]f32) (B: [m][n]f32) =
  let (L, U) = lu A
  in map (solveLUb L U) B |> transpose

let cho_solveAb [n] (A: [n][n]f32) (b: [n]f32)  =
  let L = cholesky A
  let U = transpose L
  in solveLUb L U b

let cho_solveAB [n][m] (A: [n][n]f32) (B: [m][n]f32) =
  let L = cholesky A
  let U = transpose L
  in map (solveLUb L U) B |> transpose
```

Obtaining the inverse of $\mathbf{A}$ (given that $\mathbf{A}$ is invertible) is now just a matter of solving the linear system $\mathbf{AX} = \mathbf{I}$ using the function above and the identity function. However, when computing the inverse with Cholesky, we do not need to perform forward-substitution, so we do not use `solveLUb` here. Recall that $\mathbf{A}^{-1} = (\mathbf{L}^{-1})^T\mathbf{L}^{-1}$ for positive definite Hermitian matrices.

```
let gauss_inv [n] (A: [n][n]f32): [n][n]f32 =
  gauss_solveAB A (identity n)

let lu_inv [n] (A: [n][n]f32): [n][n]f32 =
  lu_solveAB A (identity n)

let cho_inv [n] (A: [n][n]f32): [n][n]f32 =
```

```
8   let L = cholesky A
9   let Linv = map (forward_substitution L) (identity n) |> transpose
10  in matmul (transpose Linv) Linv
```

7.4 Benchmarks

In this section, we will compare the performance of our algorithms finding matrix inverse and solving linear system to `np.linalg.inv` and `np.linalg.solve`, respectively. When comparing the functions for solving linear systems, we have benchmarked for one right-hand side vector b as well as multiple right-hand side vectors $\mathbf{B}$.

All the Futhark benchmarks were conducted using `futhark bench`[1] with CUDA as backend on Futhark version 0.19, on datasets generated with `futhark dataset`[2]. The Python benchmarks were conducted 10 times in a for-loop where the average run time is shown in the tables below. The computer used has an an Intel(R) Core(TM) i5-8600K CPU @ 3.60GHz, and an NVIDIA GeForce GTX 1070 Ti graphics card with 8GB VRAM.

Library / Matrix size	**Futhark Cho-inv**	Futhark LU-inv	Futhark GJ-inv	NumPy	Speedup
1000x1000	**87.72ms**	123.7ms	104.1ms	29.4ms	0.34x
3000x3000	**1042.9ms**	1515.2ms	2270.1ms	437.6ms	0.41x
5000x5000	**4110.9ms**	5987.6ms	10419.0ms	1733.6ms	0.42x
7000x7000	**10647.5ms**	15507.7ms	28395.0ms	4410.2ms	0.41x
10000x10000	**29795.6ms**	43063.3ms	82766.6ms	12397.8ms	0.41x

Table 7.1: Benchmark results of inverse matrix algorithms.
GJ = Gauss Jordan. Best Futhark performance is marked with **bold**
Speedup is relative, calculated by $\frac{NumPy\ time}{Futhark\ cho-inv\ time}$

Library / Matrix size	**Futhark Cho-Ab**	Futhark LU-Ab	Futhark GJ-Ab	NumPy	Speedup
$\mathbf{A}$ = 1000x1000, $\mathbf{b}$ = 1000x1	**90.6ms**	82.7ms	73.3ms	11.4ms	0.13x
$\mathbf{A}$ = 3000x3000, $\mathbf{b}$ = 3000x1	**704.4ms**	781.0ms	1188.2ms	147.3ms	0.21x
$\mathbf{A}$ = 5000x5000, $\mathbf{b}$ = 5000x1	**2537.9ms**	3056.3ms	5232.5ms	532.2ms	0.21x
$\mathbf{A}$ = 7000x7000, $\mathbf{b}$ = 7000x1	**6325.9ms**	7976.0ms	14182.6ms	1325.8ms	0.21x
$\mathbf{A}$ = 10000x10000, $\mathbf{b}$ = 10000x1	**17546.2ms**	22546.5ms	41261.4ms	4143.6ms	0.24x

Table 7.2: Benchmark results of solving linear systems $\mathbf{Ax} = \mathbf{b}$ algorithms.
GJ = Gauss Jordan. Best Futhark performance is marked with **bold**
Speedup is relative, calculated by $\frac{NumPy\ time}{Futhark\ cho-Ab\ time}$

[1] `https://futhark.readthedocs.io/en/latest/man/futhark-bench.html`
[2] `https://futhark.readthedocs.io/en/latest/man/futhark-dataset.html`

Library / Matrix size	Futhark Cho-AB	**Futhark LU-AB**	Futhark GJ-AB	NumPy	Speedup
$\mathbf{A}$ = 1000x1000, $\mathbf{B}$ = 1000x1000	136.3ms	**123.7ms**	111.8ms	31.5ms	0.25x
$\mathbf{A}$ = 3000x3000, $\mathbf{B}$ = 3000x3000	1683.0ms	**1561.9ms**	2309.5ms	483.8ms	0.31x
$\mathbf{A}$ = 5000x5000, $\mathbf{B}$ = 5000x5000	6793.5ms	**6105.1ms**	10313.6ms	1905.4ms	0.31x
$\mathbf{A}$ = 7000x7000, $\mathbf{B}$ = 7000x7000	17671.0ms	**15756.0ms**	28077.6ms	4785.3ms	0.30x
$\mathbf{A}$ = 10000x10000, $\mathbf{B}$ = 10000x10000	49870.9ms	**43796.9ms**	82399.9ms	14083.5ms	0.32x

Table 7.3: Benchmark results of solving linear systems $\mathbf{AX} = \mathbf{B}$ algorithms.
GJ = Gauss Jordan. Best Futhark performance is marked with **bold**
Speedup is relative, calculated by $\frac{NumPy\ time}{Futhark\ cho-AB\ time}$

Unfortunately, our implementations are slower across-the-board. `np.linalg.solve` uses LU decomposition from LAPACK to solve linear systems. Since our LU and $Cholesky$ implementation are slower than LAPACK, solving linear systems with these functions is inherently slower. However, it is worth noting that the difference seems to shrink as we solve for more right-hand side vectors. This is likely because solving for many right-hand side vectors has the same span as solving for one - we compute all of them in parallel.

Chapter 8

Conclusion and Future Work

This project's objective was to explore and work with the data-parallel functional programming language Futhark and examine whether it is suited for programming of linear algebra. During the project, we have worked on implementations of several different algorithms, including NMF, Cholesky Decomposition, and QR Decomposition.

We started this project with no knowledge of data-parallel computing and no experience of pure functional programming, so learning Futhark was exciting but also intimidating at times because it was so different from what we were accustomed to. Once we understood the basics of the language, it did not take long before we could implement some of the functions we needed. The parallel cost model for Futhark was remarkably helpful in gaining an understanding of how parallel our programs were/are. It is very intuitive, and once you know the cost of the different constructs in the language, you start thinking differently, which was quite rewarding. However, we missed print statements for debugging, which are naturally not in the language because of it being purely functional. That being said, `futhark repl` was a great help when debugging as well as `futhark test`. The profiler (hidden behind the -P flag) was also quite helpful in finding the most time-consuming parts of our programs. However, it is a bit daunting that you have to look in the intermediate representation to decipher the results from the profiler. Luckily, our supervisor, Troels, was super helpful.

On the contrary, all that glitters is not gold. Most of our Futhark programs were not able to beat respective NumPy and scikit-learn routines. It was not an objective of the project, but we were optimistic. We see a couple of reasons why we are not beating these libraries. The Python libraries heavily use LAPACK, a software library for numerical linear algebra, which has been worked on and optimized for 29 years by some of the world's leading experts in this field. Literature[Moa18] suggests that we should be able to beat LAPACK with GPUs. We think that this would be possible to do with the use of Futhark for some of our algorithms such as (LU and Cholesky) without further improvements to the optimizing compiler. However, it would require highly optimized blocked variants of these decomposition-algorithms, and it would be far from trivial to implement. These blocked-algorithms are often recursively defined, so implementing them in a language like Futhark that has no recursion would be complicated.

So do we think that Futhark is suited for linear algebra programming? - Absolutely. We are convinced that the performance we achieved is a lot better than what we would have achieved with any other high-level programming language in the same time frame. For the most part, we could express what we wanted in Futhark, except for recursive blocked algorithms. Futhark was not only a great language to write these algorithms in, but it was also an excellent learning tool for data-parallel programming. We beat scikit-learn with the performance of our NMF, and all of our batched functions were faster than NumPy and scikit-learn by quite a lot. It is clear that our implementations have the potential to become faster, which could be accomplished through compiler optimizations and optimizations of the implementations themselves.

8.1 Future work

In terms of future work for this project, there are still many linear algebra algorithms and optimizations that could be explored and implemented in Futhark. Among these are finding eigenvalues and vectors efficiently with QR-Decomposition and explore other blocked algorithm variants considering that blocked algorithms seem to be most commonly used by other linear algebra libraries. Further optimizations could be achieved by implementing other solvers, algorithm variants and examining other divergence methods, such as the gradient descent method for non-negative matrix factorization.

As mentioned in the report, batched versions of the linear algebra algorithms can be used in various applications, which could be interesting to explore and possibly implement in Futhark. As discussed and shown from our benchmarks, we see that this could lead to very fast algorithms.

Bibliography

[Pre92] William H.; Saul A. Teukolsky; William T. Vetterling; Brian P. Flannery Press. *Numerical Recipes in C: The Art of Scientific Computing*. Press Syndicate of the University of Cambridge, 1992, p. 994. ISBN: 0-521-43108-5.

[TB97] Lloyd N. Trefethen and David Bau. *Numerical linear algebra*. Society for Industrial and Applied Mathematics (SIAM), 1997, p. 166. ISBN: 978-0-89871-361-9).

[Ste98a] G. W. Stewart. *Matrix Algorithms*. Society for Industrial and Applied Mathematics (SIAM), 1998, pp. 277–288. ISBN: 0-89871-414-1 (v.1).

[Ste98b] G. W. Stewart. *Matrix Algorithms*. Society for Industrial and Applied Mathematics (SIAM), 1998, pp. 57–60. ISBN: 0-89871-414-1 (v.1).

[Dav02] Jordi Vitrià David Guillamet. *Classifying Faces with Non-negative Matrix Factorization*. 2002. URL: http://www.cvc.uab.cat/~jordi/FinalCCIA2002.pdf.

[RLH04] Mymathlib - RLH. *Doolittle LU Decomposition*. 2004. URL: http://www.mymathlib.com/matrices/linearsystems/doolittle.html.

[Abe07] Francine F. Abeles. *Dodgson condensation: The historical and mathematical development of an experimental method*. 2007. URL: https://www.sciencedirect.com/science/article/pii/S0024379507005290.

[AR09] Dan Campbell Andrew Kerr and Mark Richards. *QR Decomposition on GPUs*. 2009. URL: https://cpb-us-w2.wpmucdn.com/sites.gatech.edu/dist/5/462/files/2016/08/Kerr_Campbell_Richards_QRD_on_GPUs.pdf.

[HM12] Sardar Anisul Haque and Marc Moreno Maza. *Determinant Computation on the GPU using the Condensation Method*. 2012. URL: https://iopscience.iop.org/article/10.1088/1742-6596/341/1/012031/pdf.

[MK12] David Sheffield Michael J. Anderson and Kurt Keutzer. *Efficient Implementation of Hyperspectral Anomaly Detection Techniques on GPUs and Multicore Processors*. 2012. URL: https://parlab.eecs.berkeley.edu/sites/all/parlab/files/A%20Predictive%20Model%20for%20Solving%20Small%20Linear%20Problems.pdf.

[GL13a] Gene H. Golub and Charles F. Van Loan. *Matrix Computations (4th ed.)* Johns Hopkins, 2013, pp. 246–256. ISBN: ISBN 1-4214-0794-9.

[GL13b] Gene H. Golub and Charles F. Van Loan. *Matrix Computations (4th ed.)* Johnhopkins, 2013, pp. 105–144. ISBN: ISBN 1-4214-0794-9.

[Jos14] et al. José M. Molero. *Efficient Implementation of Hyperspectral Anomaly Detection Techniques on GPUs and Multicore Processors*. 2014. URL: https://www.umbc.edu/rssipl/people/aplaza/Papers/Journals/2014.JSTARS.Multicore.pdf.

[LR14] Noel Lopes and Bernardete Ribeiro. *Machine Learning for Adaptive Many-Core Machines- A Pratical Approach*. Springer, 2014, pp. 127–154. ISBN: 978-3-319-06937-1.

[Rei14] Matthew W. Reid. *Pivoting for LU Factorization*. 2014. URL: http://buzzard.ups.edu/courses/2014spring/420projects/math420-UPS-spring-2014-reid-LU-pivoting.pdf.

[Rus14] João Paulo Tarasconi Ruschel. *Parallel Implementations of the Cholesky Decomposition on CPUs and GPUs*. 2014. URL: https://www.lume.ufrgs.br/bitstream/handle/10183/151001/001009773.pdf.

[Azz17] et al. Azzam Haidar. *A Guide For Achieving High Performance With Very Small Matrices on GPU*. 2017. URL: https://www.research.manchester.ac.uk/portal/files/62970458/08214236.pdf.

[LH17] Hou-biao Li and Ting-Zhu Huang. *A note on Dodgson's determinant condensation algorithm*. 2017. URL: https://www.researchgate.net/publication/334760352_A_note_on_Dodgson's_determinant_condensation_algorithm.

[Tab17] Marco Taboga. *LU decomposition*. 2017. URL: https://www.statlect.com/matrix-algebra/lu-decomposition.

[EHO18a] Martin Elsman, Troels Henriksen, and Cosmin E. Oancea. *A Parallel Cost Model for Futhark Programs*. 2018. URL: https://futhark-book.readthedocs.io/en/latest/parallel-cost-model.html.

[EHO18b] Martin Elsman, Troels Henriksen, and Cosmin E. Oancea. *Parallel Programming in Futhark*. 2018. URL: https://futhark-book.readthedocs.io.

[Moa18] Assaad Moawad. *Benchmarking BLAS libraries*. 2018. URL: https://medium.com/datathings/benchmarking-blas-libraries-b57fb1c6dc7.