UiT
THE ARCTIC
UNIVERSITY
OF NORWAY

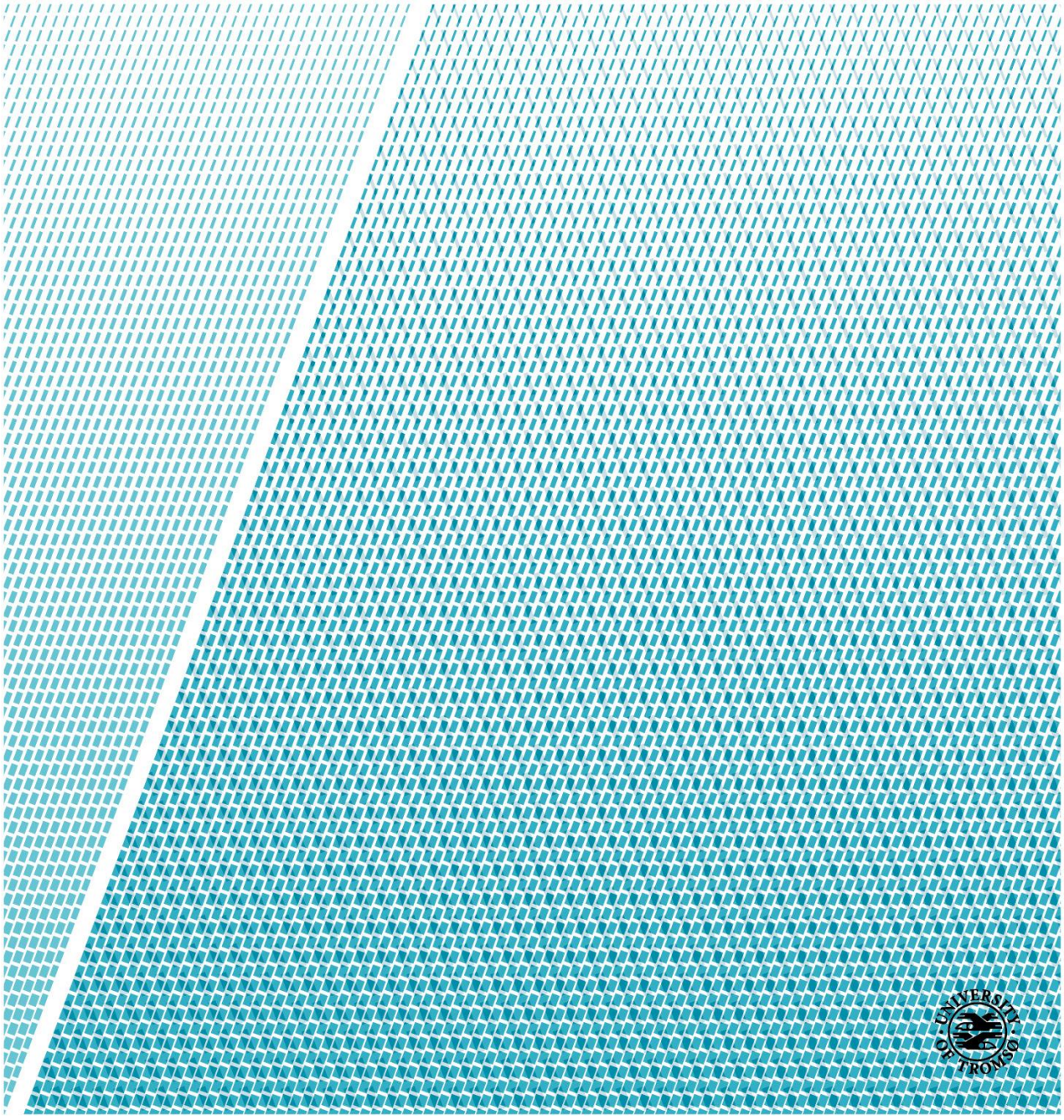
Department of physics and technology

On the improvement and acceleration of eigenvalue decomposition in spectral methods using GPUs

—

Thomas A. Haugland Johansen

FYS-3941 Master's thesis in applied physics and mathematics 30 SP
December 2016



Contents

List of Figures

Acknowledgments

First of all, I would like to thank my supervisors, Robert Jenssen, Michael C. Kampffmeyer, Filippo M. Bianchi, and Arnt-Børre Salberg. Their tireless effort in providing guidance, and motivation in the face of great difficulties, is in the end what made this thesis possible. Next, I would like to give thanks to my parents, who have inspired me, and supported all my endeavors, my entire life. Thanks to all my friends, and family whom have helped me in countless ways, and for never giving up on me.

Last, but not least, I would like to send a special thanks to my office mates, Sara Björk, Rolf Ole Jenssen, and Torgeir Brenn. Without your support and help, the process of writing this thesis would have been a lot more difficult in so many ways. I will forever cherish the thousands of hours we have spent together in our little office, supporting each other, and eating lots of cheese.

Special mention and thanks to UiT student Andreas S. Strauman who helped find the example matrix used throughout the thesis.

Abstract

The key objectives in this thesis are; the study of GPU-accelerated eigenvalue decomposition in an effort to uncover both benefits and pitfalls, and then to investigate and facilitate a future GPU implementation of the symmetric QR algorithm with permutations. With the current trend of having ever larger datasets both in terms of features and observations, we propose that GPU computation can help ameliorate the temporal penalties incurred by eigendecomposing large matrices. We successfully show the benefits of performing eigendecomposition on GPUs, and also highlight some problems with current GPU implementations. While implementing the QR algorithm on GPU, we discovered that the GPU-based QR decomposition does not explicitly form the orthogonal matrix needed as part of the QR algorithm. Therefore, we propose a novel GPU algorithm for "implicitly" computing the orthogonal matrix $\mathbf{Q}$ from the Householder vectors given by the QR decomposition. To illustrate the benefits of our methods, we show that the *kernel entropy component analysis* algorithm on GPU is two orders of magnitude faster than an equivalent CPU implementation.

Mathematical notation

$\mathbf{A}$	Matrix.
$\boldsymbol{v}$	Vector.
$\boldsymbol{a}_i$	The i-th column vector of the matrix $\mathbf{A}$.
a_{ij}	The element on the i-th row and j-th column of the matrix $\mathbf{A}$.
v_i	The i-th component of the vector $\boldsymbol{v}$.
$\langle \boldsymbol{u} \vert \boldsymbol{v} \rangle$	The inner product of the vectors $\boldsymbol{u}$ and $\boldsymbol{v}$.
$\mathbf{A} \circ \mathbf{B}$	The Hadamard product of the matrices $\mathbf{A}$ and $\mathbf{B}$.
$\mathrm{diag}\,(\mathbf{A})$	Diagonal matrix with elements corresponding to the diagonal of $\mathbf{A}$.
$\mathbf{A}(p{:}q, r{:}s)$	The sub-matrix of $\mathbf{A}$ for rows p to q, and columns r to s.

Part I.

Introduction

Introduction

Many important methods in scientific computing, including machine learning applications and related techniques, are based on solving eigensystems, which means finding eigenvalues and eigenvectors of matrices. One of the key problems with these approaches is that there does not exist a closed form solution to identify eigenvalues and eigenvectors. We must instead rely on algorithms, most of which end up exploiting numerical approximations through iterative schemes. Since there is a trend in the field of machine learning for working with ever larger datasets, characterized by a high number of features, many iterative eigenvalue algorithms become victims of numerical instabilities. Unstable numerical calculations can lead to slower convergence rates, which can mean the iterative procedure could take more time to converge to an optimal numerical solution, worst case however, such instability yields a solution which is simply incorrect. Another key problem with solving large eigensystems, which is inherently a problem of working with big matrices, is that the required computational resources scale exponentially with the dimensions of the matrices. In other words, the larger the matrix is, the higher the resulting spatial and temporal complexity will be.

In recent years, since the advent of the graphics processing unit (GPU), there has been an emergence of general-purpose computing on GPU. This has led to an increase in affordable, off-the-shelf computational power. GPUs were originally designed to render computer graphics at high framerates, which ultimately meant they were optimized for maximum parallel throughput. This turns out to make them very useful for computing algorithms that are built on e.g. algebraic operations such as matrix and vector products, which can be computed very efficiently in a parallel scheme. Importantly, many eigenvalue algorithms are based on computing such matrix and vector products. However, implementing algorithms on a GPU is a much more involved and difficult process when compared to working with a normal CPU. This added difficulty and complexity is rooted both in the differences at both the hardware and software level. Iterative algorithms that can be trivial to implement on a CPU, can turn out to be very difficult on a GPU, but perhaps also end up being slower.

Our key aim in this thesis is two-fold; the first objective will be a study of solving eigensystems on a GPU, in order to discover both benefits and pitfalls. We propose the use of GPU-based computations as one solution to the scaling problem related to working with large matrices. To illustrate our results, we will apply GPU-based computations to a machine learning method called *kernel entropy component analysis* (KECA) [1].

The second objective will be to investigate and facilitate a GPU implementation of a specific eigenvalue algorithm recently presented in Krishnamoorthy [2], the symmetric QR algorithm with permutations. It addresses the key problem related to numerical instability by optimally reordering rows and columns of matrices at each iteration of the classic QR algorithm. Because the proposed QR algorithm is based on the use of permutation matrices to perform the reordering, we conjecture it is vital to retain full control over most of the implementation details, e.g. if one wants to avoid the explicit multiplication of permutation matrices by employing a more computationally efficient, implicit reordering.

The structure of this thesis

In order to better understand how we can alleviate, or perhaps entirely resolve, the aforementioned computational difficulties and improve the convergence, we will in Chapter 1 review the linear algebra fundamentals that are required to understand the methods proposed to estimate eigendecomposition. First we will briefly discuss what eigenvalues and eigenvectors represent, and how they can be found analytically. Then, in order to understand numerical approximation methods used to solve large eigensystem problems we will need to get an understanding of some matrix decompositions.

With the linear algebra foundation covered, Chapter 2 will focus on a small, but highly relevant, subset of algorithms that can be used to find eigenvalues and eigenvectors of large systems of equations. During the course of this endeavor we will explain how these techniques may experience problems such as slow convergence and numerical instability.

This leads us to Chapter 3 in which a study of various optimization techniques that can be exploited to alleviate the computational issues. Perhaps more importantly, we will show how we can improve the convergence rate, which can lead to improved overall temporal performance. This will include showing experiments in which we partially reproduce results found in Krishnamoorthy [2]. In the article the authors claim to increase the convergence rate of the QR algorithm by nearly a factor of two, when applied to symmetric positive semi-definite matrices. The improved convergence is attributed to the use of permutation matrices. Our experiments based on the article were CPU-based.

Some spectral methods used in machine learning, such as PCA, kernel PCA, and KECA will be outlined in Chapter 4. The emphasis will be on the basic underlying concepts of each method, in conjunction with simple examples to help explain and showcase the differences between them. Covering the basics of the methods will be important in order to understand how, and to what extent, they rely on eigenvalues and eigenvectors.

With the mathematical foundation covered, we will in Chapter 5 give a basic introduction to GPU computation. We will start by explaining the differences between the approaches adopted in CPU and GPU computations, by starting with the differences in hardware architecture. Then we will consider the conceptual differences in terms of how we approach and implement algorithms on GPUs. Because optimizing GPU computations is a very difficult topic, we will only briefly cover this with a few selected benchmarks that will demonstrate the effect of choosing good and bad execution parameters.

In Chapter 6 we will experiment with accelerating spectral methods on GPU, and we will start by comparing eigenvalue decomposition on CPU versus GPU. To achieve this we will leverage the Eigen library for the CPU side, and the MAGMA [3, 4, 5, 6, 7, 8, 9, 10, 11] project for our GPU implementations. Having employed and benchmarked an eigenvalue solver, we will then implement the KECA algorithm, and attempt to improve its temporal performance by computing the eigenvalues and eigenvectors using a GPU. This chapter addresses our first objective.

Once we get to Chapter 7, we will address our second objective, which is to investigate and facilitate the implementation of the symmetric QR algorithm with permutations. By first implementing the traditional QR algorithm without permutations on GPU, we will discover what difficulties and challenges one might expect when attempting implement the permuted QR algorithm on a GPU in some future endeavor. Implementing a sequential algorithm such as the QR algorithm in a semi-parallel manner is non-trivial, and unlikely to be fast if approached naively.

For most of our experiments we will rely on the *Frey face*[1]dataset, since it is has both the appropriate dimensionality to suit our needs, but is also a frequently used benchmark dataset in machine learning. An added benefit is that it consists of small, grayscale images, which can make studying results visually intuitive and straight-forward.

[1]The *Frey face* dataset consists of 1965 grayscale images with dimensions 20×28, and was acquired from `http://www.cs.nyu.edu/~roweis/data.html`.

Part II.

Background theory

Chapter 1.

Linear algebra

1.1. Eigenvalues and eigenvectors

One approach to spectral analysis, or spectral theory, is that of eigenvalues and eigenvectors of matrices, which are special classes of scalars and vectors. The term "eigen" is German and translates directly to "own", but can also be interpreted as e.g. "characteristic" [12]. Perhaps the earliest introduction of the concept of an eigenvector, was made by Euler in 1751. He proved that any body, regardless of its shape, can be assigned an axis of rotation around which the body can rotate freely and with uniform motion [13]. This has since been further studied and enhanced by a multitude of mathematicians, notably Cauchy and Lagrange, and later Hilbert who coined the term eigenvector [14].

Today eigenvalues and eigenvectors are used in many applications beyond their initial use as principal axes of rotation of rigid bodies. Some of these applications include quantum mechanics, data transformation and reduction [15], image segmentation [16], ranking [17], economics, gene-expression, neuroscience, etc. The interpretation of the eigenvalues and eigenvectors depend upon the application or context.

Before we continue, let us define eigenvectors and eigenvalues mathematically.

Definition 1.1

Let V be any arbitrary vector space, and $T : V \to V$ be some linear operator. Then a non-zero vector $\boldsymbol{v}$ is an eigenvector of T if and only if

$$T(\boldsymbol{v}) = \lambda \boldsymbol{v} \tag{1.1}$$

for some scalar λ which is also referred to as the corresponding eigenvalue. Note that a linear operator in a vector space is a matrix, in which case (1.1) can be expressed as

$$\mathbf{A}\boldsymbol{v} = \lambda \boldsymbol{v}$$

In essence, if $\boldsymbol{v}$ is an eigenvector of the linear operator T, the transformation resulting from applying T on $\boldsymbol{v}$, consists of a rescaling of $\boldsymbol{v}$ by λ. In fact, the operator T is equivalent to the identity matrix $\mathbf{I}$ (multiplied by some constant λ) in the vector space where $\boldsymbol{v}$ is one of the basis vectors.

Remark. Geometrically, the direction will be reversed if λ is negative, but the overall orientation of the vector will remain unchanged.

As mentioned earlier, eigenvalues and eigenvectors are applied in many contexts, like graph theory, clustering, principle component analysis, and Google PageRank. The general approach to finding eigenvalues of a matrix analytically is via the so-called *characteristic equation* [12].

Definition 1.2: Characteristic equation

Let $\mathbf{A}$ be a $n \times n$ matrix. Then λ is an eigenvalue of $\mathbf{A}$ if and only if

$$\det(\mathbf{A} - \lambda \mathbf{I}) = 0 \tag{1.2}$$

where $\mathbf{I}$ is the $n \times n$ *identity matrix*.

Before going further it might be helpful to look at a simple example of using (1.2) to find the eigenvalues of a small matrix — assume we have a matrix

$$\mathbf{M} = \begin{bmatrix} 5 & -2 & -1 & 0 \\ -2 & 5 & 0 & 1 \\ -1 & 0 & 5 & 2 \\ 0 & 1 & 2 & 5 \end{bmatrix}.$$

Using (1.2) together with this matrix yields

$$\det(\mathbf{M} - \lambda\mathbf{I}) = \det \begin{bmatrix} 5-\lambda & -2 & -1 & 0 \\ -2 & 5-\lambda & 0 & 1 \\ -1 & 0 & 5-\lambda & 2 \\ 0 & 1 & 2 & 5-\lambda \end{bmatrix} = 0$$

and when we evaluate the matrix determinant we find that

$$(\lambda - 2)(\lambda - 4)(\lambda - 6)(\lambda - 8) = 0.$$

Hence the matrix $\mathbf{M}$ has eigenvalues $\lambda = \{2, 4, 6, 8\}$.

In order to find the eigenvectors corresponding to a particular eigenvalue, we need to solve the system of equations

$$(\mathbf{A} - \lambda\mathbf{I})\,\boldsymbol{v} = \mathbf{0}, \tag{1.3}$$

where we want the non-trivial solutions. We call this the null space for the *eigenspace* of the matrix $\mathbf{A}$ that corresponds to the eigenvalue λ. Using our earlier example, we want to find corresponding eigenvectors for the two eigenvalues. Starting with $\lambda = 2$ and using (1.3),

$$(\mathbf{M} - 2\mathbf{I})\,\boldsymbol{v}_1 = \mathbf{0},$$

we get the following system of equations

$$\begin{aligned} 3v_{11} - 2v_{12} - 1v_{13} &= 0 \\ -2v_{11} + 3v_{12} + 1v_{14} &= 0 \\ -1v_{11} + 3v_{13} + 2v_{14} &= 0 \\ 1v_{12} + 2v_{13} + 3v_{14} &= 0. \end{aligned}$$

The solution of the system of equations, up to a constant t, can be obtained through e.g. *Gauss-Jordan elimination* [18] and is

$$v_{11} = -t, \quad v_{12} = -t, \quad v_{13} = -t, \quad v_{14} = t,$$

and equivalently in vector notation,

$$\boldsymbol{v}_1 = \begin{bmatrix} -1 & -1 & -1 & 1 \end{bmatrix}^\mathsf{T},$$

where we have dropped the scaling factor t since that only affects the length of the vector. Repeating the process for $\lambda = \{4, 6, 8\}$ yields the corresponding eigenvectors

$$\boldsymbol{v}_2 = \begin{bmatrix} 1 & 1 & -1 & 1 \end{bmatrix}^\mathsf{T}$$

$$\boldsymbol{v}_3 = \begin{bmatrix} 1 & -1 & 1 & 1 \end{bmatrix}^\mathsf{T}$$
$$\boldsymbol{v}_4 = \begin{bmatrix} -1 & 1 & 1 & 1 \end{bmatrix}^\mathsf{T}.$$

As we have seen, for every distinct eigenvalue there exists a corresponding eigenvector. This means if we have an $n \times n$ matrix with n distinct eigenvalues, we will have to repeat the process of finding the corresponding eigenvectors n times. If we have repeated eigenvalues, additional methods will have to be employed in order to find the eigenvectors.

The algorithm that we have used so far to find eigenvectors, does not lend itself very well to being solved programatically. We need to find alternate algorithms that scale better and that can be implemented efficiently. Since there exists no closed-form algebraic solution to polynomials higher than four degrees [19, 20, 21], the eigenvalues algorithms naturally also have no closed-form, but are rather expressed as iterative schemes, and achieve efficiency by using numerical approximations. Moreover, since many of these algorithms are based on matrix decompositions, we will have to introduce further notions of linear algebra by looking closer at a few, relevant matrix decompositions.

1.2. LU decomposition

The LU decomposition was formalized by Alan Turing in his pioneering paper on the rounding-off errors in matrix calculations [22] in 1948. In the paper Turing proves, under certain restrictions, that any direct method for solving linear systems of equations of the form $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$ can be written as matrix decompositions [23].

There are several different variations of the LU decomposition, but the basic underlying principle remains the same; if the LU decomposition exists for some non-singular matrix $\mathbf{A}$, then the matrix can be factorized as

$$\mathbf{A} = \mathbf{LU},$$

where $\mathbf{L}$ is a lower triangular matrix and $\mathbf{U}$ an upper triangular matrix.

Definition 1.3: LU decomposition

Let $\mathbf{A}$ be a $n \times n$ non-singular matrix that can be reduced to row echelon form $\mathbf{U}$ by Gaussian elimination without row pivoting,

$$\mathbf{E}_k \cdots \mathbf{E}_2 \mathbf{E}_1 \mathbf{A} = \mathbf{U},$$

where $\mathbf{E}_1, \mathbf{E}_2, \ldots, \mathbf{E}_k$ are the elementary matrices corresponding to the elementary operations used to reduce $\mathbf{A}$ to row echelon form [12]. Then $\mathbf{L}$ is a lower triangular matrix formed by the matrix product of the inverted elementary matrices used to produce the upper triangular matrix $\mathbf{U}$;

$$\mathbf{L} = \mathbf{E}_1^{-1} \mathbf{E}_2^{-1} \cdots \mathbf{E}_k^{-1}. \tag{1.4}$$

Remark. Using this general definition, the LU decomposition only exists for matrices that can be reduced to row echelon form without row permutations.

Using this definition of the LU decomposition, let us find the decomposition of the example matrix $\mathbf{M}$ we used previously.

$$\mathbf{M} = \begin{bmatrix} 5 & -2 & -1 & 0 \\ -2 & 5 & 0 & 1 \\ -1 & 0 & 5 & 2 \\ 0 & 1 & 2 & 5 \end{bmatrix}$$

$$\mathbf{E}_1\mathbf{M} = \begin{bmatrix} 1 & -\frac{2}{5} & -\frac{1}{5} & 0 \\ -2 & 5 & 0 & 1 \\ -1 & 0 & 5 & 2 \\ 0 & 1 & 2 & 5 \end{bmatrix} \qquad \mathbf{E}_1 = \begin{bmatrix} \frac{1}{5} & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad R_1 \times \frac{1}{5}$$

$$\mathbf{E}_2\mathbf{E}_1\mathbf{M} = \begin{bmatrix} 1 & -\frac{2}{5} & -\frac{1}{5} & 0 \\ 0 & \frac{21}{5} & -\frac{2}{5} & 1 \\ -1 & 0 & 5 & 2 \\ 0 & 1 & 2 & 5 \end{bmatrix} \qquad \mathbf{E}_2 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad R_2 + 2R_1$$

$$\mathbf{E}_3\mathbf{E}_2\mathbf{E}_1\mathbf{M} = \begin{bmatrix} 1 & -\frac{2}{5} & -\frac{1}{5} & 0 \\ 0 & 1 & -\frac{2}{21} & \frac{5}{21} \\ -1 & 0 & 5 & 2 \\ 0 & 1 & 2 & 5 \end{bmatrix} \qquad \mathbf{E}_3 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{5}{21} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad R_2 \times \frac{5}{21}$$

$$\mathbf{E}_4 \cdots \mathbf{E}_1\mathbf{M} = \begin{bmatrix} 1 & -\frac{2}{5} & -\frac{1}{5} & 0 \\ 0 & 1 & -\frac{2}{21} & \frac{5}{21} \\ 0 & -\frac{2}{5} & \frac{24}{5} & 2 \\ 0 & 1 & 2 & 5 \end{bmatrix} \qquad \mathbf{E}_4 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad R_3 + R_1$$

$$\mathbf{E}_5 \cdots \mathbf{E}_1\mathbf{M} = \begin{bmatrix} 1 & -\frac{2}{5} & -\frac{1}{5} & 0 \\ 0 & 1 & -\frac{2}{21} & \frac{5}{21} \\ 0 & 0 & \frac{100}{21} & \frac{44}{21} \\ 0 & 1 & 2 & 5 \end{bmatrix} \qquad \mathbf{E}_5 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & \frac{2}{5} & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad R_3 + \frac{2}{5}R_2$$

$$\mathbf{E}_6 \cdots \mathbf{E}_1\mathbf{M} = \begin{bmatrix} 1 & -\frac{2}{5} & -\frac{1}{5} & 0 \\ 0 & 1 & 0 & \frac{5}{21} \\ 0 & 0 & 1 & \frac{11}{25} \\ 0 & 1 & 2 & 5 \end{bmatrix} \qquad \mathbf{E}_6 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{21}{100} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad R_3 \times \frac{21}{100}$$

$$\mathbf{E}_7 \cdots \mathbf{E}_1\mathbf{M} = \begin{bmatrix} 1 & -\frac{2}{5} & -\frac{1}{5} & 0 \\ 0 & 1 & -\frac{2}{21} & \frac{5}{21} \\ 0 & 0 & 1 & \frac{11}{25} \\ 0 & 0 & \frac{44}{21} & \frac{100}{21} \end{bmatrix} \qquad \mathbf{E}_7 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{bmatrix} \quad R_4 - R_2$$

$$\mathbf{E}_8 \cdots \mathbf{E}_1\mathbf{M} = \begin{bmatrix} 1 & -\frac{2}{5} & -\frac{1}{5} & 0 \\ 0 & 1 & -\frac{2}{21} & \frac{5}{21} \\ 0 & 0 & 1 & \frac{11}{25} \\ 0 & 0 & 0 & \frac{96}{25} \end{bmatrix} \qquad \mathbf{E}_8 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -\frac{44}{21} & 1 \end{bmatrix} \quad R_4 - \frac{44}{21}R_3$$

$$\mathbf{U} = \begin{bmatrix} 1 & -\frac{2}{5} & -\frac{1}{5} & 0 \\ 0 & 1 & -\frac{2}{21} & \frac{5}{21} \\ 0 & 0 & 1 & \frac{11}{25} \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \mathbf{E}_9 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & \frac{25}{96} \end{bmatrix} \quad R_4 \times \frac{25}{96}$$

Having found **U**, we can also find the lower triangular **L** by using (1.4), which yields

$$\mathbf{L} = \begin{bmatrix} 5 & 0 & 0 & 0 \\ -2 & \frac{21}{5} & 0 & 0 \\ -1 & -\frac{2}{5} & \frac{100}{21} & 0 \\ 0 & 1 & \frac{44}{21} & \frac{96}{25} \end{bmatrix}.$$

There exists a slightly improved algorithm for constructing the decomposition [12], and it can be summarized in the following steps

1. Reduce the matrix **A** to row echelon form **U** by using Gaussian elimination without row interchanges, while keeping track of of the row multipliers used to introduce leading ones and the multipliers used to introduce zeros below the leading ones.

2. For each element along the diagonal of **L**, place the reciprocal of the row multiplier that introduced the leading one in the corresponding element in **U**.

3. For each element below the diagonal of **L**, place the negative of the multiplier that was used to introduce the zero in the corresponding element in **U**.

By following this algorithm there is no longer any need to construct the elementary matrices, and thus from a computational standpoint, reducing both the time and storage requirement of the algorithm [12].

The LU decomposition we have defined up until this point is somewhat asymmetric since **U** is an upper triangular matrix with a unit diagonal, whereas **L** has a non-unit diagonal. This asymmetry in the decomposition is an issue if we want to exploit the symmetry of the original matrix to simplify the computation of the decomposition. If we want to obtain a **L** matrix with a unit diagonal, we have to factor out the diagonal of **L** into a new diagonal matrix **D** [12, 18, 24].

Definition 1.4: LDU decomposition

Let $\mathbf{A}$ be a $n \times n$ matrix which has a LU decomposition. Define the diagonal matrix $\mathbf{D} = \text{diag}(\mathbf{L})$ such that

$$\mathbf{D} = \begin{bmatrix} l_{11} & & & \\ & l_{22} & & \\ & & \ddots & \\ & & & l_{nn} \end{bmatrix}$$

and redefine the lower triangular $\mathbf{L}$ by dividing every column by its corresponding diagonal element in order to get a unit diagonal. The LDU decomposition can then be expressed as

$$\mathbf{A} = \mathbf{LDU}$$

What is more, if the matrix being decomposed is symmetric, the LDU decomposition can be further simplified by exploiting the symmetry.

Definition 1.5: LDL decomposition

Let $\mathbf{A}$ be a $n \times n$ symmetric matrix which has a LDU decomposition. Then it follows that $\mathbf{L} = \mathbf{U}^\mathsf{T}$, and hence the decomposition can be expressed as

$$\mathbf{A} = \mathbf{LDL}^\mathsf{T} \tag{1.5}$$

which is a unique decomposition [24].

Looking back at our earlier example where we found a concrete LU decomposition, we now want to find the corresponding LDU decomposition. We begin by finding the diagonal matrix,

$$\mathbf{D} = \begin{bmatrix} 5 & 0 & 0 & 0 \\ 0 & \frac{21}{5} & 0 & 0 \\ 0 & 0 & \frac{100}{21} & 0 \\ 0 & 0 & 0 & \frac{96}{25} \end{bmatrix}.$$

In order to calculate the redefined $\mathbf{L}$ we need to scale every column vector in the original matrix by its corresponding diagonal element. One way to express this is via the

Hadamard product [25] of the original matrix and a matrix whose row vectors equal the reciprocal of the diagonal of the original matrix. Thus, we have

$$\mathbf{L} = \begin{bmatrix} 5 & 0 & 0 & 0 \\ -2 & \frac{21}{5} & 0 & 0 \\ -1 & -\frac{2}{5} & \frac{100}{21} & 0 \\ 0 & 1 & \frac{44}{21} & \frac{96}{25} \end{bmatrix} \circ \begin{bmatrix} \frac{1}{5} & \frac{5}{21} & \frac{21}{100} & \frac{25}{96} \\ \frac{1}{5} & \frac{5}{21} & \frac{21}{100} & \frac{25}{96} \\ \frac{1}{5} & \frac{5}{21} & \frac{21}{100} & \frac{25}{96} \\ \frac{1}{5} & \frac{5}{21} & \frac{21}{100} & \frac{25}{96} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -\frac{2}{5} & 1 & 0 & 0 \\ -\frac{1}{5} & -\frac{2}{21} & 1 & 0 \\ 0 & \frac{5}{21} & \frac{11}{25} & 1 \end{bmatrix}$$

Additionally, since our example matrix $\mathbf{M}$ is symmetric, we know from (1.5) that it can be decomposed uniquely as

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -\frac{2}{5} & 1 & 0 & 0 \\ -\frac{1}{5} & -\frac{2}{21} & 1 & 0 \\ 0 & \frac{5}{21} & \frac{11}{25} & 1 \end{bmatrix} \begin{bmatrix} 5 & 0 & 0 & 0 \\ 0 & \frac{21}{5} & 0 & 0 \\ 0 & 0 & \frac{100}{21} & 0 \\ 0 & 0 & 0 & \frac{96}{25} \end{bmatrix} \begin{bmatrix} 1 & -\frac{2}{5} & -\frac{1}{5} & 0 \\ 0 & 1 & -\frac{2}{21} & \frac{5}{21} \\ 0 & 0 & 1 & \frac{11}{25} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.6}$$

Hence, if we know that we are working with a symmetric matrix, we do not need to calculate the upper triangular matrix $\mathbf{U}$ — we only need $\mathbf{L}$ and the diagonal elements of $\mathbf{D}$. Both of which can be computed directly without first finding the LU or LDU decomposition, yielding a reduction in both spatial and temporal complexity [26].

1.3. Cholesky decomposition

Before we define the Cholesky decomposition, we need to introduce another definition in order to guarantee that the Cholesky decomposition exists for that particular type of matrix [27].

Definition 1.6: Positive definite matrix

Let $\mathbf{A}$ be a $n \times n$ real, symmetric matrix. Then $\mathbf{A}$ is said to be *positive definite* if it satisfies the property

$$\boldsymbol{v}^{\mathsf{T}}\mathbf{A}\boldsymbol{v} > 0, \quad \forall \boldsymbol{v} \neq \mathbf{0} \in \mathbb{R}^n$$

However, if the quadratic form only satisfies

$$\boldsymbol{v}^{\mathsf{T}}\mathbf{A}\boldsymbol{v} \geq 0, \quad \forall \boldsymbol{v} \neq \mathbf{0} \in \mathbb{R}^n$$

then the matrix $\mathbf{A}$ is *positive semi-definite*.

Although we could stick to only this general definition of definiteness, it will be beneficial for us to consider a less general definition that does not require computing the quadratic form — it only requires checking diagonal elements of a LU decomposition, hence reducing computational complexity.

Definition 1.7

Let $\mathbf{A}$ be a $n \times n$ matrix that has a LU decomposition. Then $\mathbf{A}$ is positive definite if all diagonal elements in the decomposition are positive. However, if some of the diagonal elements are zero, then $\mathbf{A}$ is only positive semi-definite [18].

Using the property of positive definiteness as a requirement, we can finally define the aforementioned Cholesky decomposition.

Definition 1.8: Cholesky decomposition

Let $\mathbf{A}$ be a $n \times n$ symmetric, positive definite matrix. Then $\mathbf{A}$ has a unique decomposition given by

$$\mathbf{A} = \mathbf{R}^\mathsf{T}\mathbf{R} \tag{1.7}$$

where $\mathbf{R}$ is an upper triangular matrix with positive diagonal elements. $\mathbf{R}$ is sometimes referred to as the *Cholesky factor*.

Remark. If the matrix $\mathbf{A}$ is positive semi-definite, the Cholesky decomposition still exists, but is not unique.

There are several ways to compute the Cholesky decomposition; we will start by considering the approach based on the LDL decomposition (1.5). Since we know that $\mathbf{D}$ is a diagonal matrix with only positive elements, we can take the square root of the matrix and the result will remain real;

$$\mathbf{D} = \mathbf{D}^{1/2}\mathbf{D}^{1/2}$$

By substituting this into (1.5) we find that

$$\mathbf{A} = \mathbf{L}\mathbf{D}^{1/2}\mathbf{D}^{1/2}\mathbf{L}^\mathsf{T} = \mathbf{L}\mathbf{D}^{1/2}(\mathbf{L}\mathbf{D}^{1/2})^\mathsf{T},$$

and we see that this decomposition is equivalent to (1.7) by setting $\mathbf{R}^\mathsf{T} = \mathbf{L}\mathbf{D}^{1/2}$. It is worth noting that the LDL decomposition itself can be considered equivalent to the Cholesky decomposition, and may be preferred in some situations.

Continuing with our previous example matrix $\mathbf{M}$, we found that it has a LDL decomposition given by (1.6). This means that the Cholesky factor $\mathbf{LD}^{1/2}$ of $\mathbf{M}$ is

$$\mathbf{LD}^{1/2} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -\frac{2}{5} & 1 & 0 & 0 \\ -\frac{1}{5} & -\frac{2}{21} & 1 & 0 \\ 0 & \frac{5}{21} & \frac{11}{25} & 1 \end{bmatrix} \begin{bmatrix} \sqrt{5} & 0 & 0 & 0 \\ 0 & \sqrt{\frac{21}{5}} & 0 & 0 \\ 0 & 0 & \frac{10}{\sqrt{21}} & 0 \\ 0 & 0 & 0 & \frac{4\sqrt{6}}{5} \end{bmatrix}$$

$$= \begin{bmatrix} \sqrt{5} & 0 & 0 & 0 \\ -\frac{2}{\sqrt{5}} & \sqrt{\frac{21}{5}} & 0 & 0 \\ -\frac{1}{\sqrt{5}} & -\frac{2}{\sqrt{105}} & \frac{10}{\sqrt{21}} & 0 \\ 0 & \sqrt{\frac{5}{21}} & \frac{22}{5\sqrt{21}} & \frac{4\sqrt{6}}{5} \end{bmatrix}.$$

Since all diagonal elements of the decomposition are positive, we know that the matrix $\mathbf{M}$ is positive definite. Hence it follows that the Cholesky decomposition of the matrix $\mathbf{M}$ exists, and can be expressed as

$$\mathbf{M} = \begin{bmatrix} \sqrt{5} & 0 & 0 & 0 \\ -\frac{2}{\sqrt{5}} & \sqrt{\frac{21}{5}} & 0 & 0 \\ -\frac{1}{\sqrt{5}} & -\frac{2}{\sqrt{105}} & \frac{10}{\sqrt{21}} & 0 \\ 0 & \sqrt{\frac{5}{21}} & \frac{22}{5\sqrt{21}} & \frac{4\sqrt{6}}{5} \end{bmatrix} \begin{bmatrix} \sqrt{5} & -\frac{2}{\sqrt{5}} & -\frac{1}{\sqrt{5}} & 0 \\ 0 & \sqrt{\frac{21}{5}} & -\frac{2}{\sqrt{105}} & \sqrt{\frac{5}{21}} \\ 0 & 0 & \frac{10}{\sqrt{21}} & \frac{22}{5\sqrt{21}} \\ 0 & 0 & 0 & \frac{4\sqrt{6}}{5} \end{bmatrix}.$$

Instead of going via the LDL decomposition (or using it outright), it is possible to compute the Cholesky decomposition directly. There exists several different algorithms and nuanced variations to each, but we will limit our the one commonly referred to as the inner product form of the Cholesky decomposition [24, 27].

Algorithm 1.1 Cholesky decomposition (inner product form)

Require: $\mathbf{A}$ is a $n \times n$ symmetric, positive semi-definite matrix.
Ensure: $\mathbf{R}$ is a $n \times n$ upper triangular Cholesky factor of $\mathbf{A}$.

1: **for** $i = 1, \ldots, n$ **do**
2: $\quad r_{ii} \leftarrow \sqrt{a_{ii} - \sum_{k=1}^{i-1} a_{ki}^2}$
3: $\quad$ **for** $j = i+1, \ldots, n$ **do**
4: $\quad\quad r_{ij} \leftarrow r_{ii}^{-1}\left(a_{ij} - \sum_{k=1}^{i-1} r_{ki} r_{kj}\right)$
5: $\quad$ **end for**
6: **end for**

The routine in Algorithm 1.1 describes the inner product form, and yields the upper triangular $\mathbf{R}$ from (1.7). As we can see, the lower triangular part (excluding the diagonal) of $\mathbf{A}$ is never visited, thus saving some computational effort.

Applying Algorithm 1.1 in a step by step manner on the example matrix

$$\mathbf{M} = \begin{bmatrix} 5 & -2 & -1 & 0 \\ -2 & 5 & 0 & 1 \\ -1 & 0 & 5 & 2 \\ 0 & 1 & 2 & 5 \end{bmatrix}$$

gives the following matrix element computations

$$\begin{aligned}
r_{11} &= \sqrt{m_{11}} = \sqrt{5} \\
r_{12} &= r_{11}^{-1} m_{12} = -\frac{2}{\sqrt{5}} \\
r_{13} &= r_{11}^{-1} m_{13} = -\frac{1}{\sqrt{5}} \\
r_{14} &= r_{11}^{-1} m_{14} = 0
\end{aligned}$$

$$\begin{aligned}
r_{22} &= \sqrt{m_{22} - r_{12}^2} = \sqrt{5 - \tfrac{4}{5}} = \sqrt{\tfrac{21}{5}} \\
r_{23} &= r_{22}^{-1}(m_{23} - r_{12} r_{13}) = \sqrt{\tfrac{5}{21}}\left(0 - \left(-\tfrac{2}{\sqrt{5}}\right)\left(-\tfrac{1}{\sqrt{5}}\right)\right) = -\tfrac{2}{\sqrt{105}} \\
r_{24} &= r_{22}^{-1}(m_{24} - r_{12} r_{14}) = \sqrt{\tfrac{5}{21}}\left(1 - \left(-\tfrac{2}{\sqrt{5}}\right) \times 0\right) = \sqrt{\tfrac{5}{21}}
\end{aligned}$$

$$\begin{aligned}
r_{33} &= \sqrt{m_{33} - r_{13}^2 - r_{23}^2} = \sqrt{5 - \tfrac{1}{5} - \tfrac{4}{105}} = \tfrac{10}{\sqrt{21}} \\
r_{34} &= r_{33}^{-1}(m_{34} - r_{13} r_{14} - r_{23} r_{24}) = \tfrac{\sqrt{21}}{10}\left(2 - \left(-\tfrac{1}{\sqrt{5}}\right) \times 0 - \left(-\tfrac{2}{\sqrt{105}}\right)\left(\sqrt{\tfrac{5}{21}}\right)\right) = \tfrac{22}{5\sqrt{21}}
\end{aligned}$$

$$r_{44} = \sqrt{m_{44} - r_{14}^2 - r_{24}^2 - r_{34}^2} = \sqrt{5 - 0^2 - \tfrac{5}{21} - \tfrac{22}{525}} = \tfrac{4\sqrt{6}}{5}.$$

Comparing these elements with the $\mathbf{R}$ matrix computed in (1.6) reveals that the two methods give identical results, which is expected since we know from Definition 1.8 that the Cholesky decomposition of a symmetric, positive definite matrix is unique.

The efficiency of the Cholesky decomposition compared to the LU decomposition, when it comes to solving systems of equations, has been shown to be about twice as good [28]. Moreover, it is generally also more stable than the LU decomposition [29].

1.4. QR decomposition

The final decomposition we will investigate is the QR decomposition, and for us its importance is primarily rooted in its use in an eigenvalue algorithm known as the QR algorithm, which is covered in Chapter 2.

First of all, let us formally define the QR decomposition.

Definition 1.9: QR decomposition

Let $\mathbf{A}$ be a $n \times n$ non-singular matrix. Then the QR decomposition of $\mathbf{A}$ can be expressed uniquely as

$$\mathbf{A} = \mathbf{QR},$$

where $\mathbf{Q}$ is a $n \times n$ orthogonal matrix, and $\mathbf{R}$ is a $n \times n$ upper triangular matrix with positive diagonal elements.

Remark. If the matrix $\mathbf{A}$ is singular the decomposition can still exist, but its uniqueness is no longer guaranteed [24].

The decomposition is typically used, just like the LU decomposition, to solve linear systems of equations of the form $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$. However, the advantage of the QR decomposition lies in the knowledge that $\mathbf{Q}^\mathsf{T}\mathbf{Q} = \mathbf{I}$ and that $\mathbf{R}$ is upper triangular, which makes the decomposition more computationally efficient when used for e.g. solving systems of equations [18, 24].

Before we look at how to compute the QR decomposition, we want to briefly note a somewhat remarkable connection between the QR and Cholesky decompositions which will make it easier to study the convergence characteristics of the QR decomposition. Let us consider that we have a non-singular matrix $\mathbf{G}$ which has a QR decomposition $\mathbf{G} = \mathbf{QR}$. Hence we also know that matrix $\mathbf{G}^\mathsf{T}\mathbf{G}$ is positive definite, which in turn implies that it has a Cholesky decomposition. We can then show the connection between the QR and Cholesky decompositions via

$$\mathbf{G}^\mathsf{T}\mathbf{G} = (\mathbf{QR})^\mathsf{T}\mathbf{QR} = \mathbf{R}^\mathsf{T}\mathbf{Q}^\mathsf{T}\mathbf{QR} = \mathbf{R}^\mathsf{T}\mathbf{R}.$$

This means that $\mathbf{R}$ is the Cholesky factor of $\mathbf{G}^\mathsf{T}\mathbf{G}$, which is an attractive result since it is generally easier to study convergence characteristics of the Cholesky decomposition [2].

In order to compute the factors $\mathbf{Q}$ and $\mathbf{R}$ one typically uses either the *Gram-Schmidt process*, *Householder reflections*, or *Givens rotations* [18]. It is worth noting upfront that only the latter two are numerically stable when it comes to computing the QR

decomposition, whereas the classical Gram-Schmidt process is unstable in that the orthogonality is lost when the number of vectors to orthogonalize is sufficiently large [30, 31, 32]. We will not study the modified Gram-Schmidt process or other variants of the Gram-Schmidt process. Even though these do (to some extent) resolve the numerical instability and loss of orthogonality, they are also computationally expensive.

1.4.1 The Gram-Schmidt approach to QR decomposition

Regardless of the problems with the classical Gram-Schmidt process, let us briefly look at how it can be leveraged to compute a QR decomposition.

Definition 1.10: Gram-Schmidt process

Let $\mathbf{A}$ be a $n \times n$ non-singular matrix with column vectors $\mathbf{A} = [\boldsymbol{a}_1|\boldsymbol{a}_2|\cdots|\boldsymbol{a}_n]$. Then the sequence expressed as

$$\boldsymbol{u}_1 = \boldsymbol{a}_1, \quad \boldsymbol{u}_k = \boldsymbol{a}_k - \sum_{i=1}^{k-1} \boldsymbol{q}_i^\mathsf{T} \boldsymbol{a}_k, \quad k = 2, \ldots, n \tag{1.8}$$

constructs an orthogonal basis for the column space of $\mathbf{A}$. Finally, the Gram-Schmidt process is completed by normalizing the orthogonal basis vectors,

$$\boldsymbol{q}_k = \frac{\boldsymbol{u}_k}{\nu_k}, \quad \nu_k = \|\boldsymbol{u}_k\|, \quad k = 1, 2, \ldots, n, \tag{1.9}$$

which gives an orthonormal basis for the column space of $\mathbf{A}$.

From Definition 1.10 we have the relationships

$$\boldsymbol{a}_1 = \nu_1 \boldsymbol{q}_1, \quad \boldsymbol{a}_k = \nu_k \boldsymbol{q}_k + \sum_{i=1}^{k-1} \boldsymbol{q}_i^\mathsf{T} \boldsymbol{a}_k, \quad k = 2, \ldots, n, \tag{1.10}$$

which can also be expressed in matrix form [18] as

$$[\boldsymbol{a}_1|\boldsymbol{a}_2|\cdots|\boldsymbol{a}_n] = [\boldsymbol{q}_1|\boldsymbol{q}_2|\cdots|\boldsymbol{q}_n] \begin{bmatrix} \nu_1 & \boldsymbol{q}_1^\mathsf{T}\boldsymbol{a}_2 & \boldsymbol{q}_1^\mathsf{T}\boldsymbol{a}_3 & \cdots & \boldsymbol{q}_1^\mathsf{T}\boldsymbol{a}_n \\ 0 & \nu_2 & \boldsymbol{q}_2^\mathsf{T}\boldsymbol{a}_3 & \cdots & \boldsymbol{q}_2^\mathsf{T}\boldsymbol{a}_n \\ 0 & 0 & \nu_3 & \cdots & \boldsymbol{q}_3^\mathsf{T}\boldsymbol{a}_n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & \nu_n \end{bmatrix}. \tag{1.11}$$

Moreover, this result is nothing more than $\mathbf{A} = \mathbf{QR}$, where $\mathbf{Q}$ is an orthogonal matrix and $\mathbf{R}$ is an upper triangular matrix with positive diagonal elements. Hence we have shown that a QR decomposition can be computed using the Gram-Schmidt process.

To better understand the Gram-Schmidt orthogonalization process in the context of a QR decomposition, let us apply the process on our example matrix

$$\mathbf{M} = \begin{bmatrix} 5 & -2 & -1 & 0 \\ -2 & 5 & 0 & 1 \\ -1 & 0 & 5 & 2 \\ 0 & 1 & 2 & 5 \end{bmatrix} = [\boldsymbol{m}_1|\boldsymbol{m}_2|\boldsymbol{m}_3|\boldsymbol{m}_4].$$

Computing the orthogonal basis of $\mathbf{M}$ using (1.8) yields

$$\begin{aligned} \boldsymbol{u}_1 &= \begin{bmatrix} 5 & -2 & -1 & 0 \end{bmatrix}^\mathsf{T}, \\ \boldsymbol{u}_2 &= \tfrac{1}{3}\begin{bmatrix} 4 & 11 & -2 & 3 \end{bmatrix}^\mathsf{T}, \\ \boldsymbol{u}_3 &= \tfrac{1}{50}\begin{bmatrix} 11 & -1 & 57 & 27 \end{bmatrix}^\mathsf{T}, \\ \boldsymbol{u}_4 &= \tfrac{1}{4920}\begin{bmatrix} -5 & -7 & -11 & 25 \end{bmatrix}^\mathsf{T}, \end{aligned}$$

which we then normalize using (1.9) to produce an orthonormal basis;

$$\begin{aligned} \nu_1 &= \sqrt{30}, & \boldsymbol{q}_1 &= \sqrt{\tfrac{1}{30}}\begin{bmatrix} 5 & -2 & -1 & 0 \end{bmatrix}^\mathsf{T} \\ \nu_2 &= \sqrt{\tfrac{50}{3}}, & \boldsymbol{q}_2 &= \sqrt{\tfrac{1}{150}}\begin{bmatrix} 4 & 11 & -2 & 3 \end{bmatrix}^\mathsf{T} \\ \nu_3 &= \sqrt{\tfrac{656}{25}}, & \boldsymbol{q}_3 &= \sqrt{\tfrac{1}{4100}}\begin{bmatrix} 11 & -1 & 57 & 27 \end{bmatrix}^\mathsf{T} \\ \nu_4 &= \sqrt{\tfrac{2304}{205}}, & \boldsymbol{q}_4 &= \sqrt{\tfrac{1}{820}}\begin{bmatrix} -5 & -7 & -11 & 25 \end{bmatrix}^\mathsf{T}. \end{aligned}$$

Having computed the orthonormal basis, we apply (1.10) and (1.11) to find the QR decomposition of $\mathbf{M}$,

$$\mathbf{M} = \begin{bmatrix} \sqrt{\frac{25}{30}} & \sqrt{\frac{16}{150}} & \sqrt{\frac{121}{4100}} & -\sqrt{\frac{25}{820}} \\ -\sqrt{\frac{4}{30}} & \sqrt{\frac{121}{150}} & -\sqrt{\frac{1}{4100}} & -\sqrt{\frac{49}{820}} \\ -\sqrt{\frac{1}{30}} & -\sqrt{\frac{4}{150}} & \sqrt{\frac{3249}{4100}} & -\sqrt{\frac{121}{820}} \\ 0 & \sqrt{\frac{9}{150}} & \sqrt{\frac{729}{4100}} & \sqrt{\frac{625}{820}} \end{bmatrix} \begin{bmatrix} \sqrt{30} & -\sqrt{\frac{400}{30}} & -\sqrt{\frac{100}{30}} & -\sqrt{\frac{16}{30}} \\ 0 & \sqrt{\frac{50}{3}} & -\sqrt{\frac{64}{150}} & \sqrt{\frac{484}{150}} \\ 0 & 0 & \sqrt{\frac{656}{25}} & \sqrt{\frac{61504}{4100}} \\ 0 & 0 & 0 & \sqrt{\frac{2304}{205}} \end{bmatrix}.$$

When the decomposition is fully evaluated by multiplying the two factor matrices together, the result does indeed yield the original matrix. The interested reader can easily verify the result using applicable software.

1.4.2 QR decomposition using reflectors and rotations

In addition to using the Gram-Schmidt process to compute a QR decomposition, it is also possible to use other, equivalent techniques for orthogonalization. Two frequent techniques are the so-called *Householder reflections* and *Givens rotations*. Since they are more numerically stable compared to the classical Gram-Schmidt approach [33] they should generally always be preferred when computing a QR decomposition.

We will briefly introduce both approaches, but will not cover them in great detail. Therefore, for the interested reader, we recommend having a look at [33, 34, 35] for much greater detail on both algorithms.

Householder reflections

Similar to the Gram-Schmidt orthogonalization process, the construction of the Householder reflectors yields both a triangular matrix, and a set of orthonormal basis vectors. Furthermore, just as with Gram-Schmidt, these matrices constitute the QR decomposition of the matrix from which they are constructed. The difference between the Gram-Schmidt and the Householder processes are found in how we construct the factor matrices. The former, as presented in the previous section, is built on sequentially constructing orthogonal basis vectors, whereas the latter is built on the premise of constructing unitary matrices that transform the original matrix such that we ultimately end up with a triangular matrix. These matrices are what we call the *Householder reflectors*, and the idea is to construct a hyperplane that we can reflect across in order to introduce the desired zero components post-reflection.

Definition 1.11: Householder reflector

Let $\boldsymbol{v}$ be a nonzero vector of dimension n. Then the $n \times n$ matrix given by

$$\mathbf{H} = \mathbf{I} - \beta \boldsymbol{v}\boldsymbol{v}^{\mathsf{T}}, \quad \beta = \frac{2}{\boldsymbol{v}^{\mathsf{T}}\boldsymbol{v}} \tag{1.12}$$

is called a *Householder reflector*, and $\boldsymbol{v}$ is often referred to as a *Householder vector*.

From this definition is should be clear that we need to choose $\boldsymbol{v}$ such that it introduces the desired zero components.

Given a vector $\boldsymbol{x}$ that we want to reflect across the hyperplane defined by a given Householder reflector $\mathbf{H}$. Then we can zero out all components of $\boldsymbol{x}$ except the first one by defining the Householder vector as

$$\boldsymbol{v} = \boldsymbol{x} \pm \|\boldsymbol{x}\|\boldsymbol{e}_1, \tag{1.13}$$

where $\boldsymbol{e}_1$ is the standard Euclidean unit vector defined as

$$\boldsymbol{e}_1 = \begin{bmatrix} 1 & 0 & \cdots & 0 \end{bmatrix}^\mathsf{T}.$$

Combining this vector with (1.12) yields

$$\mathbf{H}\boldsymbol{x} = \left(\mathbf{I} - \beta\boldsymbol{v}\boldsymbol{v}^\mathsf{T}\right)\boldsymbol{x} = \mp\|\boldsymbol{x}\|\boldsymbol{e}_1. \tag{1.14}$$

after a bit of vector algebra. What this result tells us is essence that effect off applying the reflector to its associated vector is equivalent of storing the magnitude of the vector, or all the information if you will, in a single vector component. In other terms, no information is lost, it is simply relocated in a sense. Also note that the choice of adding or subtracting needs to be made with numerical stability in mind. Typically we would opt for subtraction as that will make the result a positive multiple of the original vector $\boldsymbol{x}$. Although, if $\boldsymbol{x}$ is close to $\boldsymbol{e}_1$, subtraction might lead to catastrophic cancellation. There also exist other approaches for constructing the Householder vector, and we refer the interested reader to Golub [27] which covers this in great detail.

Before we go further, let us consider a toy example for a vector given by

$$\boldsymbol{x} = \begin{bmatrix} 2 & \frac{2\sqrt{3}}{3} \end{bmatrix}^\mathsf{T}, \tag{1.15}$$

Using (1.13), one corresponding Householder vector can be expressed as

$$\boldsymbol{v} = \begin{bmatrix} 2 - \frac{4\sqrt{3}}{3} & \frac{2\sqrt{3}}{3} \end{bmatrix}^\mathsf{T},$$

since $\|\boldsymbol{x}\| = \frac{4\sqrt{3}}{3}$. Using this result we can then construct the reflector as

$$\mathbf{H} = \begin{bmatrix} \frac{\sqrt{3}}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{\sqrt{3}}{2} \end{bmatrix}.$$

It should already be trivial to see that multiplying this reflector with the vector $\boldsymbol{x}$ will yield the desired result, but for completeness,

$$\mathbf{H}\boldsymbol{x} = \begin{bmatrix} \boldsymbol{h}_1^\mathsf{T}\boldsymbol{x} & \boldsymbol{h}_2^\mathsf{T}\boldsymbol{x} \end{bmatrix}^\mathsf{T},$$

where

$$\boldsymbol{h}_1^\mathsf{T}\boldsymbol{x} = \frac{\sqrt{3}}{2} \times 2 + \frac{1}{2} \times \frac{2\sqrt{3}}{3} = \frac{4\sqrt{3}}{3},$$
$$\boldsymbol{h}_2^\mathsf{T}\boldsymbol{x} = \frac{1}{2} \times 2 - \frac{\sqrt{3}}{2} \times \frac{2\sqrt{3}}{3} = 0.$$

Hence we see that

$$\mathbf{H}\boldsymbol{x} = \begin{bmatrix} \frac{4\sqrt{3}}{3} & 0 \end{bmatrix}^\mathsf{T},$$

which also corresponds with (1.14) since we already know that $\|\boldsymbol{x}\| = \frac{4\sqrt{3}}{3}$.

Furthermore, Definition 1.11 can also be expressed in a more algorithmic, pseudocode manner;

Algorithm 1.2 Householder reflector

```
function HOUSEHOLDER(x)
    v ← x ± ‖x‖e_1          ▷ Sign chosen to ensure numerical stability.
    β ← 2/(v^T v)
    return (I − βvv^T)
end function
```

If we want an upper triangular matrix, we need to introduce zeros in the lower triangular minor of the matrix. This is exactly what we achieve by applying a sequence of Householder reflectors on the target matrix in a systematic, column by column fashion. In other words, we if we have an $n \times n$ matrix that we want to compute the QR decomposition of, we need to sequentially compute and apply n Householder reflectors. The result of applying all reflectors yield the upper triangular matrix

$$\mathbf{H}_n\mathbf{H}_{n-1}\cdots\mathbf{H}_1\mathbf{A} = \mathbf{R},$$

whereas the result of multiplying together all the reflectors gives us the orthogonal matrix

$$\mathbf{H}_1\cdots\mathbf{H}_n = \mathbf{Q}.$$

Hence we have shown how the QR decomposition can be computed via a series of Householder transformations. Note that we can also express this process in a more explicitly algorithmic fashion;

Algorithm 1.3 QR decomposition using Householder reflectors

Require: $\mathbf{A}$ is an $n \times n$ matrix.
Ensure: $\mathbf{Q}_n$ is an orthogonal matrix, and $\mathbf{R}$ is upper triangular such that $\mathbf{A} = \mathbf{Q}_n\mathbf{R}$.

1: $\mathbf{R} \leftarrow \mathbf{A}$
2: $\mathbf{Q}_0 \leftarrow \mathbf{I}$
3: **for** $j = 1, \ldots, n$ **do**
4: $\quad \mathbf{H}_j = \textsc{Householder}(\mathbf{R}(j{:}n, j))$
5: $\quad \mathbf{R}(j{:}n, j{:}n) \leftarrow \mathbf{H}_j\mathbf{R}(j{:}n, j{:}n)$
6: $\quad \mathbf{Q}_j \leftarrow \mathbf{Q}_{j-1}\mathbf{H}_j$
7: **end for**

Let us apply the Householder transformation on our usual example matrix,

$$\mathbf{M} = \begin{bmatrix} 5 & -2 & -1 & 0 \\ -2 & 5 & 0 & 1 \\ -1 & 0 & 5 & 2 \\ 0 & 1 & 2 & 5 \end{bmatrix}.$$

We begin by finding the first Householder reflection vector $\boldsymbol{v}_1$,

$$\boldsymbol{v}_1 = \boldsymbol{m}_1 - \|\boldsymbol{m}_1\|\boldsymbol{e}_1 = \begin{bmatrix} 5 - \sqrt{30} & -2 & -1 & 0 \end{bmatrix}^\mathsf{T}, \quad \beta_1 = \frac{1}{30}\Big(6 + \sqrt{30}\Big),$$

which we then use to construct the first reflector,

$$\mathbf{H}_1 = \mathbf{I} - \beta_1\boldsymbol{v}_1\boldsymbol{v}_1^\mathsf{T} = \begin{bmatrix} \sqrt{\frac{25}{30}} & -\sqrt{\frac{20}{150}} & -\sqrt{\frac{1}{30}} & 0 \\ -\sqrt{\frac{4}{30}} & \frac{1}{5} - \sqrt{\frac{80}{150}} & -\frac{2}{5} - \sqrt{\frac{4}{30}} & 0 \\ -\sqrt{\frac{1}{30}} & -\frac{2}{5} - \sqrt{\frac{20}{150}} & \frac{4}{5} - \sqrt{\frac{1}{30}} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Comparing this result with our previous QR decomposition of $\mathbf{M}$, we see that the first column is identical to the first column of the previously found $\mathbf{Q}$ matrix. Given the construction of the subsequent reflectors and the manner in which we will now form $\mathbf{Q}$, we know that the first column will remain unchanged by, thus we can see how this method will ultimately yield the same result as found via the Gram-Schmidt approach. In order to see whether we get the same upper triangular matrix $\mathbf{R}$, we must compute

the matrix product $\mathbf{H_1 M}$, which can be expressed as

$$\mathbf{R} = \mathbf{H_1 M} = \begin{bmatrix} \sqrt{30} & -\sqrt{\frac{400}{30}} & -\sqrt{\frac{100}{30}} & -\sqrt{\frac{16}{30}} \\ 0 & 1-\sqrt{\frac{256}{30}} & -2-\sqrt{\frac{64}{30}} & -\frac{3}{5}-\sqrt{\frac{16}{30}} \\ 0 & -2-\sqrt{\frac{64}{30}} & 4-\sqrt{\frac{16}{30}} & \frac{18}{15}-\sqrt{\frac{16}{30}} \\ 0 & 1 & 2 & 5 \end{bmatrix}.$$

As anticipated, the first column corresponds with the previous computation of the upper triangular matrix $\mathbf{R}$.

Computing the next Householder transformation is done on the 3×3 sub-matrix $\mathbf{R}(2{:}4, 2{:}4)$, which first gives us

$$\boldsymbol{v}_2 = \begin{bmatrix} 1-8\sqrt{\frac{2}{15}}-5\sqrt{\frac{2}{3}} & -2-\sqrt{\frac{64}{30}} & 1 \end{bmatrix}^{\mathsf{T}}, \quad \beta_2 = \frac{3}{50+16\sqrt{5}-5\sqrt{6}}.$$

Computing the reflector $\mathbf{H}_2$ can be done, but since the result contains somewhat convoluted coefficients expressed with nested radicals, we will not present the actual result here. However, applying the computed reflector on the sub-matrix $\mathbf{R}(2{:}4, 2{:}4)$ gives

$$\mathbf{R} = \mathbf{H_2 H_1 M} = \begin{bmatrix} \sqrt{30} & -\sqrt{\frac{400}{30}} & -\sqrt{\frac{100}{30}} & -\sqrt{\frac{16}{30}} \\ 0 & \sqrt{\frac{50}{3}} & -\sqrt{\frac{64}{150}} & -\sqrt{\frac{484}{150}} \\ 0 & 0 & \times & \times \\ 0 & 0 & \times & \times \end{bmatrix},$$

where we have omitted some coefficients for the sake of brevity due to the convoluted form of the exact values. Similarly, we find that

$$\mathbf{Q}_2 = \mathbf{H}_1 \mathbf{H}_2 = \begin{bmatrix} \sqrt{\frac{25}{30}} & \sqrt{\frac{16}{250}} & \times & \times \\ -\sqrt{\frac{4}{30}} & \sqrt{\frac{121}{150}} & \times & \times \\ -\sqrt{\frac{1}{30}} & -\sqrt{\frac{4}{150}} & \times & \times \\ 0 & \sqrt{\frac{9}{150}} & \times & \times \end{bmatrix}$$

From the first two steps of Householder approach to QR decomposition, we can readily see that repeating this process two more times will yield the same result for both matrices, $\mathbf{Q}$ and $\mathbf{R}$, as the ones we found when applying the Gram-Schmidt approach.

Givens rotations

Another alternative approach to orthogonalization is to use rotations instead of reflections. The basic premise is similar in some sense, but instead of constructing a hyperplane used to reflect across, we compute a rotation which achieves the same result. Consider a trivial 2×2 example using trigonometry. Any matrix of the form

$$\mathbf{Z} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}$$

can be said to be a rotation transformation, since multiplying a vector with this matrix is equivalent to rotating the vector by an angle θ. Equivalently we can consider any matrix of the form

$$\mathbf{Z} = \begin{bmatrix} \cos\theta & \sin\theta \\ \sin\theta & -\cos\theta \end{bmatrix}$$

to be a reflection transformation, since multiplying a vector with this matrix is equivalent to reflecting across the line formed by

$$\text{span}\left\{ \begin{bmatrix} \cos\theta/2 & \sin\theta/2 \end{bmatrix}^{\mathsf{T}} \right\}.$$

An intuitive way to think about this is that the result produced by rotating a vector 60° can also be achieved by reflecting the same vector across a 30° line [27].

Since we will not use, nor consider the Givens rotations further in this thesis, we simply refer the interested to e.g. Golub [27] for further details.

Chapter 2.

Eigenvalue algorithms

In this chapter we will look at a few iterative eigenvalue algorithms that yield approximate solutions. The first algorithm uses a direct approach, while the others are based on matrix decompositions. There exists many more approximation techniques than those we will cover — this is a large field of research, and instead of attempting to give a complete overview of the field, we will instead focus on a few key algorithms. For more details and further examples of eigenvalue algorithms, in addition to those covered here, see e.g. Trefethen [33] and Demmel [36].

2.1. Power method

The power method, which is also referred to as the *von Mises algorithm* or power iterations, was first introduced by Mises and Pollaczek-Geiringer in a seminal article in 1929 titled "Praktische Verfahren der Gleichungsauflösung." As we will see, the algorithm might not be as useful these days as when it was first introduced. Nonetheless, it is still used indirectly in many other algorithms, and it is in fact used in e.g. the Google PageRank algorithm [38].

Let us start off by defining the algorithm in a sequential notation;

Definition 2.1: Power method

Let $\mathbf{A}$ be a $n \times n$ non-singular matrix with eigenvalues

$$|\lambda_1| \geq |\lambda_2| \geq \cdots \geq |\lambda_n|.$$

Then given an arbitrarily chosen non-zero vector $\boldsymbol{r}_0$ the sequence expressed by

$$\boldsymbol{q}_k = \mathbf{A}\boldsymbol{r}_{k-1}, \quad \boldsymbol{r}_k = \frac{\boldsymbol{q}_k}{\|\boldsymbol{q}_k\|}, \quad \nu_k = \boldsymbol{r}_k^{\mathsf{T}}\mathbf{A}\boldsymbol{r}_k, \quad k = 1, 2, \ldots \tag{2.1}$$

converges to the dominant eigenpair $(\boldsymbol{v}_1, \lambda_1)$ of $\mathbf{A}$, where $\boldsymbol{v}_1 = \boldsymbol{r}_{k_{max}}$ and $\lambda_1 = \nu_{k_{max}}$.

As we can see, the algorithm is only capable of finding the dominant eigenpair, however most modern machine learning methods typically need all eigenvectors and eigenvectors (or at least k such) associated with a given matrix [39].

In order to better understand how and why the power method works, we need to consider the following definition.

Definition 2.2: Rayleigh quotient

Let $\mathbf{A}$ be a $n \times n$ matrix, and $\boldsymbol{v}$ be a real, non-zero vector of length n. Then the Rayleigh quotient is expressed by

$$r(v) = \frac{\boldsymbol{v}^{\mathsf{T}}\mathbf{A}\boldsymbol{v}}{\boldsymbol{v}^{\mathsf{T}}\boldsymbol{v}}. \tag{2.2}$$

Note that it can be shown that if $\boldsymbol{v}$ is an eigenvector of $\mathbf{A}$, then we get that

$$r(v) = \lambda,$$

where λ is an eigenvalue of $\mathbf{A}$. Furthermore, it can also be shown that the Rayleigh quotient yields a quadratically accurate estimate of an eigenvalue [33] — two very powerful results. Hence we get some further insight into how the power method works, since both (2.1) and (2.2) are equivalent because we are using normalized vectors.

It is possible to use the power method to find further eigenvalues and eigenvectors by using a trick often referred to as deflation. The concept is to reduce the effect of the dominant eigenpair, thus transforming the matrix in such a manner as to make the second-most dominant eigenpair into the dominant eigenpair [40, 41, 42]. However, this technique typically suffers problems with numerical stability partially introduced by

the deflation trick [36]. An alternative is to use the so-called simultaneous iterations algorithm, which is a direct extension of the power method [33].

Implementing Definition 2.1 as a sequential algorithm, can be done as follows.

Algorithm 2.1 Power method

Require: $\mathbf{A}$ is a $n \times n$ non-singular matrix, and $\boldsymbol{r}_0$ is a non-zero vector.
Ensure: $(\boldsymbol{v}, \lambda)$ is the dominant eigenpair of $\mathbf{A}$.

1: **for** $k = 1, 2, \ldots$ **do** ▷ Iterate until stopping criteria reached.
2: $\quad \boldsymbol{q}_k \leftarrow \mathbf{A}\boldsymbol{r}_{k-1}$
3: $\quad \boldsymbol{r}_k \leftarrow \boldsymbol{q}_k / \|\boldsymbol{q}_k\|$
4: $\quad \nu_k \leftarrow \boldsymbol{r}_k^{\mathsf{T}} \mathbf{A} \boldsymbol{r}_k$
5: **end for**
6: $(\boldsymbol{v}, \lambda) \leftarrow (\boldsymbol{r}_k, \nu_k)_{k=k_{max}}$

This is more or less a direct implementation of (2.1), but requires some choice of convergence criteria; one choice is to compare the relative convergence rate between two successive iteration steps and stopping if this rate is below some pre-defined threshold. Another choice could be to simply stop after reaching a desired number of iterations.

Let us look at a simple example of using the power method; given that we have our usual example matrix

$$\mathbf{M} = \begin{bmatrix} 5 & -2 & -1 & 0 \\ -2 & 5 & 0 & 1 \\ -1 & 0 & 5 & 2 \\ 0 & 1 & 2 & 5 \end{bmatrix},$$

and an initial, non-zero vector

$$\boldsymbol{r_0} = \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix}^{\mathsf{T}}.$$

Then applying the power method algorithm yields the following results at a few illustrative iteration steps, where we have rounded to 4 decimals for the sake of brevity;

$$\begin{aligned} \boldsymbol{r}_1 &= \begin{bmatrix} 0.9129 & -0.3651 & -0.1826 & 0.0000 \end{bmatrix}^{\mathsf{T}}, \quad \nu_1 = 6.67 \\ \boldsymbol{r}_2 &= \begin{bmatrix} 0.7972 & -0.5315 & -0.2657 & -0.1063 \end{bmatrix}^{\mathsf{T}}, \quad \nu_2 = 7.34 \\ \boldsymbol{r}_3 &= \begin{bmatrix} 0.7150 & -0.5863 & -0.3146 & -0.2145 \end{bmatrix}^{\mathsf{T}}, \quad \nu_3 = 7.65 \\ &\vdots \end{aligned}$$

$$
\begin{aligned}
\boldsymbol{r}_{10} &= \begin{bmatrix} 0.5278 & -0.5268 & -0.4716 & -0.4706 \end{bmatrix}^{\mathsf{T}}, \quad \nu_{10} = 7.99 \\
\boldsymbol{r}_{11} &= \begin{bmatrix} 0.5209 & -0.5204 & -0.4787 & -0.4782 \end{bmatrix}^{\mathsf{T}}, \quad \nu_{11} = 8.00 \\
\boldsymbol{r}_{12} &= \begin{bmatrix} 0.5157 & -0.5155 & -0.4840 & -0.4838 \end{bmatrix}^{\mathsf{T}}, \quad \nu_{12} = 8.00 \\
&\vdots \\
\boldsymbol{r}_{22} &= \begin{bmatrix} 0.5009 & -0.5009 & -0.4991 & -0.4991 \end{bmatrix}^{\mathsf{T}}, \quad \nu_{22} = 8.00 \\
\boldsymbol{r}_{23} &= \begin{bmatrix} 0.5007 & -0.5007 & -0.4993 & -0.4993 \end{bmatrix}^{\mathsf{T}}, \quad \nu_{23} = 8.00 \\
\boldsymbol{r}_{24} &= \begin{bmatrix} 0.5005 & -0.5005 & -0.4995 & -0.4995 \end{bmatrix}^{\mathsf{T}}, \quad \nu_{24} = 8.00 \\
&\vdots \\
\boldsymbol{r}_{32} &= \begin{bmatrix} 0.5001 & -0.5001 & -0.4999 & -0.4999 \end{bmatrix}^{\mathsf{T}}, \quad \nu_{32} = 8.00 \\
\boldsymbol{r}_{33} &= \begin{bmatrix} 0.5000 & -0.5000 & -0.5000 & -0.5000 \end{bmatrix}^{\mathsf{T}}, \quad \nu_{33} = 8.00 \\
\boldsymbol{r}_{34} &= \begin{bmatrix} 0.5000 & -0.5000 & -0.5000 & -0.5000 \end{bmatrix}^{\mathsf{T}}, \quad \nu_{34} = 8.00.
\end{aligned}
$$

From the results we see that the resulting vector changes rapidly in the beginning, but seems to have more or less converged around iteration 23. We can see the convergence trend already around the 12th iteration. Since we have no change in the vector components between iterations 33 and 34, we conclude that the algorithm has converged. In the case of the eigenvalue, it seems to converge after only 11 iterations. The performance of the algorithm with respect to convergence to the dominant eigenvalue and eigenvector is rather good. In fact, from Figure 2.1.1 we can see that the improvement in the eigenvector estimate after each iteration is exponential.

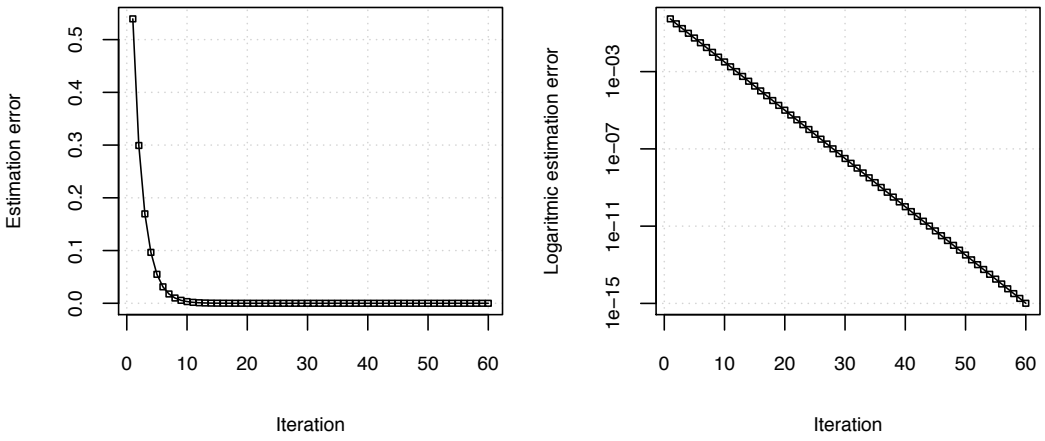


Figure 2.1.1. The figure depicts the eigenvector estimation error of using the power method in conjunction with the example matrix $\mathbf{M}$. Judging from the semi-logarithmic plot on the right, the improvement after each iteration is exponential.

Assuming that we have a $n \times n$ matrix with eigenvalues

$$|\lambda_1| \geq |\lambda_2| \geq \cdots \geq |\lambda_n|,$$

then the convergence characteristic of the power method is determined by

$$\left|\frac{\lambda_2}{\lambda_1}\right|. \tag{2.3}$$

If this ratio is close to 1, the power method algorithm will converge slowly. The smaller the ratio, the faster the algorithm will converge [26]. In other words, if the two most dominant eigenvalues are close in terms of magnitude, the power method will generally require many iterations before it converges. Looking at Figure 2.1.1, we see that given our example matrix $\mathbf{M}$, the algorithm has not converged even after 60 iterations when we use machine precision instead of rounding down to 4 decimals. In fact we found that we needed ≈ 128 iterations to reach convergence when using full machine precision.

2.2. QR algorithm

As we discussed in Section 2.1, the primary problem with the power method is that it in general only finds the dominant eigenpair, but we typically need to find the entire spectrum of a matrix. The QR algorithm [43, 44, 45] is one possible solution to the problem since it is capable of finding all eigenpairs at once. It does come with its own pitfalls however, as we shall see later.

Amongst the underlying building blocks of numerical eigenvalue approximation techniques such as the QR algorithm, we have so-called orthogonal similarity transformations [26, 46].

Definition 2.3: Orthogonal similarity transformation

Let $\mathbf{A}$ be a $n \times n$ matrix, and $\mathbf{V}$ be an orthogonal $n \times n$ matrix. Then a linear transformation T is said to be an orthogonal similarity transformation if

$$T(\mathbf{A}) = \mathbf{V}^\mathsf{T}\mathbf{A}\mathbf{V}.$$

From this definition it is possible to show that eigenvalues are preserved under this type of transformation [26], which implies that

$$\mathbf{A}\boldsymbol{v} = \lambda\boldsymbol{v} \Leftrightarrow \mathbf{V}^\mathsf{T}\mathbf{A}\mathbf{V}\boldsymbol{u} = \lambda\boldsymbol{u}, \qquad \boldsymbol{u} = \mathbf{V}^\mathsf{T}\boldsymbol{v}.$$

The QR algorithm is an iterative process based on the QR decomposition, and it exploits this property. Let us first define the QR algorithm formally.

Definition 2.4: QR algorithm

Let $\mathbf{A}$ be a $n \times n$ matrix which has a QR decomposition, and a spectral decomposition given by $\mathbf{A} = \mathbf{V}\mathbf{D}\mathbf{V}^{\mathsf{T}}$, where $\mathbf{V}$ is a matrix corresponding to the eigenvectors of $\mathbf{A}$, and $\mathbf{D}$ is a diagonal matrix with the eigenvalues of $\mathbf{A}$ placed along the diagonal. Then given that $\mathbf{A}_0 = \mathbf{A}$, the sequence expressed by

$$\mathbf{A}_{k-1} = \mathbf{Q}_k\mathbf{R}_k, \quad \mathbf{A}_k = \mathbf{R}_k\mathbf{Q}_k, \quad k = 1, 2, \ldots \tag{2.4}$$

has the following identities [2]

$$\begin{aligned} \mathbf{A}_\infty &= \mathbf{D} \\ \mathbf{Q}_\infty &= \mathbf{I} \\ \mathbf{Q}_0\mathbf{Q}_1 \cdots \mathbf{Q}_\infty &= \mathbf{V}. \end{aligned}$$

Since $\mathbf{Q}_k$ is orthogonal we know that $\mathbf{Q}_k^{\mathsf{T}}\mathbf{Q} = \mathbf{I}$, and thus we find that

$$\mathbf{R}_k = \mathbf{Q}_k^{\mathsf{T}}\mathbf{A}_{k-1}.$$

Hence we can partially reformulate the sequence in (2.4) as

$$\mathbf{A}_k = \mathbf{Q}_k^{\mathsf{T}}\mathbf{A}_{k-1}\mathbf{Q}_k, \tag{2.5}$$

which means we save some computational complexity since we do not have to calculate the full QR decomposition — we no longer require $\mathbf{R}$ to be computed. We also note that (2.5) is an orthogonal similarity transformation, so we know there is no loss or change of eigenvalues.

Algorithm 2.2 QR algorithm

Require: $\mathbf{A}_0$ is a $n \times n$ matrix with a QR decomposition $\mathbf{A}_0 = \mathbf{Q}_0\mathbf{R}_0$.
Ensure: $\operatorname{diag}(\mathbf{A}_{k_{max}})$ and $\mathbf{V}_{k_{max}}$ has converged to the eigenpairs of $\mathbf{A}_0$.

1: $\mathbf{V}_0 \leftarrow \mathbf{I}$ ▷ The $n \times n$ identity matrix.
2: **for** $k = 1, 2, \ldots$ **do** ▷ Iterate until stopping criteria reached.
3: $\mathbf{A}_{k-1} \rightarrow \mathbf{Q}_k\mathbf{R}_k$ ▷ Only need to compute $\mathbf{Q}_k$.
4: $\mathbf{A}_k \leftarrow \mathbf{Q}_k^{\mathsf{T}}\mathbf{A}_{k-1}\mathbf{Q}_k$
5: $\mathbf{V}_k \leftarrow \mathbf{V}_{k-1}\mathbf{Q}_k$
6: **end for**

Some intuition as to how the QR algorithm works can be gained by considering that the algorithm is a specialized implementation of the simultaneous iteration algorithm [33], which can be further understood as an extension to the power method [24].

Given its connection to the power method, the QR algorithm is to some extent bounded by the same convergence constraints as the power method. In particular it will generally converge slower when e.g. the dominant and second-most dominant eigenvalues are close in terms of magnitude. Additionally the QR algorithm may produce poor convergence rates for repeated eigenvalues. In Chapter 3 we will consider some techniques that can solve, or at the very least reduce, these issues by using e.g. permutation matrices.

Finally, implementing and executing Algorithm 2.2 on our example matrix

$$\mathbf{M} = \begin{bmatrix} 5 & -2 & -1 & 0 \\ -2 & 5 & 0 & 1 \\ -1 & 0 & 5 & 2 \\ 0 & 1 & 2 & 5 \end{bmatrix},$$

yielded the following estimated eigenvalues and eigenvectors, which have been truncated to 4 decimals for the benefit of the reader,

$$\mathbf{V} = \begin{bmatrix} 0.5000\ldots & -0.5000\ldots & -0.4999\ldots & -0.4999\ldots \\ 0.4999\ldots & 0.5000\ldots & 0.4999\ldots & -0.4999\ldots \\ -0.4999\ldots & -0.4999\ldots & 0.5000\ldots & -0.5000\ldots \\ 0.5000\ldots & -0.4999\ldots & 0.5000\ldots & 0.5000\ldots \end{bmatrix},$$

$$\mathbf{D} = \begin{bmatrix} 4.0000\ldots & -0.0000\ldots & -0.0000\ldots & -0.0000\ldots \\ -0.0000\ldots & 5.9999\ldots & 0.0000\ldots & 0.0000\ldots \\ -0.0000\ldots & 0.0000\ldots & 7.9999\ldots & -0.0000\ldots \\ 0.0000\ldots & -0.0000\ldots & -0.0000\ldots & 1.9999\ldots \end{bmatrix}.$$

From these results we can see that the algorithm has converged to approximate solutions of the true eigenvalues and eigenvectors of $\mathbf{M}$, up to some rounding error. In Figure 2.2.1, we can see the convergence performance of the algorithm measured by the error in the eigenvalue estimates at each iteration step. It seems that the convergence is mostly well-behaved with the exception of a sudden "jump" at the 5th iteration step. Studying the semi-logarithmic plot in the figure, it looks as if the algorithm has converged after approximately 48 iterations.

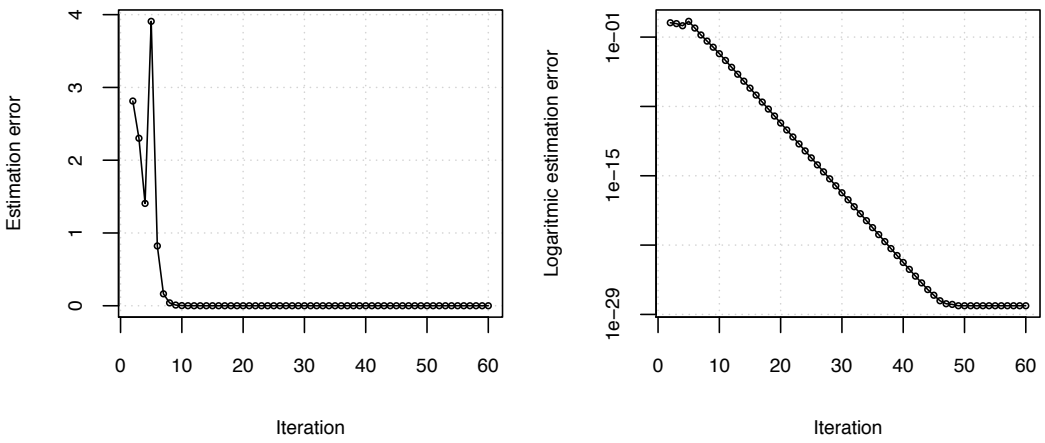


Figure 2.2.1. The two plots show the full eigenvalue estimation error of using the QR algorithm in conjunction with the example matrix $\mathbf{M}$. We see from the plot on the left that there is some instability that occurs around the 5th iteration. However, after that iteration the convergence behavior seems stable, and exponential. Convergence seems to be reached after about 50 iterations.

Chapter 3.

Optimization techniques

In this chapter we will consider a few optimization techniques that can improve both the numerical stability and convergence properties of eigenvalue algorithms, in particular the QR algorithm. Although before we look at how we can improve the algorithms, we need to briefly look at how we can determine the numerical stability of an eigenvalue algorithm.

3.1. Numerical stability

In order to know whether or not our algorithm will converge to a reasonable value, we generally need to assess the stability of the algorithm, and in our case the numerical stability in particular. The underlying primary causes of numerical stability are round-off and truncation errors, and both stem from the limits of the so-called machine precision inherent to how computers represent numbers [34]. While we can have essentially infinite precision when we work analytically, but when we perform computations on a computer, we are not so lucky. This is something we must always keep in mind, otherwise our algorithms might converge to the wrong result, or not converge at all; small round-off errors early in an iterative eigenvalue algorithm might propagate throughout the entire computation and produce wildly incorrect approximations of eigenvalues and eigenvectors. Let us look at some concrete ways to measure how well-behaved our computations are. Specifically, how small perturbations in vectors or matrices can in some scenarios be greatly magnified.

In order to determine how well-behaved a matrix-based computation is, it is common to consider the so-called condition number of the matrix. However, before we can define the condition number of a matrix, we must first define how to measure the magnitude of vectors, and a related concept for matrices. The magnitude of a vector is measured

by its p-norm, which can be expressed as

$$\|\boldsymbol{v}\|_p = \left(\sum_{i=1}^{n} |v_i|^p \right)^{1/p}.$$

For matrices we do not measure their "magnitude" as such, but rather the effect they have when operating on a vector. Therefore a matrix norm is often computed as an *operator norm* or *induced norm*. As such, the norm induced by the p-norm matrix is referred to as the matrix p-norm, and it can be formulated as

$$\|\mathbf{A}\|_p = \max_{\boldsymbol{v} \neq \mathbf{0}} \frac{\|\mathbf{A}\boldsymbol{v}\|_p}{\|\boldsymbol{v}\|_p}.$$

Intuitively we can think of the induced norm as a measure of how the matrix affects the magnitude of vector — e.g. does it increase or decrease?

Perhaps the two most important and commonly used induced matrix norms are the 1-norm and the infinity-norm,

$$\|\mathbf{A}\|_1 = \max_{1 \leq j \leq n} \sum_{i=1}^{n} |a_{ij}|, \tag{3.1}$$

$$\|\mathbf{A}\|_\infty = \max_{1 \leq i \leq n} \sum_{j=1}^{n} |a_{ij}|, \tag{3.2}$$

where $p = 1$ and $p = \infty$ respectively. From (3.1) we see that the 1-norm of a matrix is the largest column-wise sum of the absolute value of the entries. Whereas the infinity-norm, as expressed by (3.2), is the largest row-wise sum of absolute-valued entries.

Additionally we have the induced 2-norm, which is also known as the spectral norm of a matrix since it is defined as the square root of the largest eigenvalue of the matrix [47],

$$\|\mathbf{A}\|_2 = \sqrt{\lambda_{max}}.$$

This is an expensive norm to compute since it first requires computing the eigenvalues of the matrix, e.g. using the power method. It can be an important theoretical tool, but intractable in practical applications when working with large matrices.

As mentioned previously, when an algorithm is built around matrix and vector operations, it is common to consider the condition number of a matrix to get a sense of the numerical behavior of the algorithm.

Definition 3.1: Condition number

Let $\mathbf{A}$ be an $n \times n$ invertible matrix. Then the condition number of $\mathbf{A}$ is given by

$$\kappa_p(\mathbf{A}) = \left\|\mathbf{A}\right\|_p \left\|\mathbf{A}^{-1}\right\|_p,$$

where $\|\cdot\|_p$ can be any matrix p-norm. We say that a matrix is *ill-conditioned* if its condition number is much larger than 1. By "much larger", we typically mean more than an order of magnitude greater than 1.

The condition number of a matrix can be thought of as a measure of the maximum "magnification" resulting from the matrix operating on a vector. In other words, if the condition number is very large, even a very small perturbation in the vector will be magnified proportional to the condition number.

We commonly use the 1-norm or infinity-norm when we compute the condition number since both are tractable for reasonably large matrices, but note that they do require finding the inverse of the matrix. Therefore, if computing the inverse of the matrix is problematic, the condition number should rather be estimated. For details on how to estimate the condition number of a matrix, see e.g. Golub [27] and Sauer [34].

Even though it can be considered intractable for large matrices, computing the condition number can be done easily if we already know the eigenvalues of the matrix. In such scenarios, the condition number using the spectral norm can be computed trivially;

$$\kappa_2(\mathbf{A}) = \left|\frac{\lambda_{max}}{\lambda_{min}}\right|, \tag{3.3}$$

where λ_{min} and λ_{max} denote the smallest and largest eigenvalues of $\mathbf{A}$, respectively.

Considering (3.3) for a moment, let us get some intuition as to why a large difference between the smallest and largest eigenvalue — thus large condition number — is a problem. If we think in terms of the basis formed by the eigenvectors, the eigenvalue gives the magnitude (and direction) of its corresponding eigenvector. This means that if there is a large difference between the smallest and largest eigenvalues, there is an equivalent difference between the corresponding eigenvectors. Thus if this difference is large, the largest eigenvector will completely "dominate" the smallest eigenvector. In other words, there is a big disparity between the largest and smallest values in the matrix. An extreme example of such a scenario can be as simple as a matrix containing measurements of people such as height in meters, and weight in grams. For such a matrix there would be several orders of magnitude difference between the measurements of height compared to the weight. This ill-conditioned example could be improved by preconditioning, e.g. scaling or using PCA.

Once more, consider our usual example matrix,

$$\mathbf{M} = \begin{bmatrix} 5 & -2 & -1 & 0 \\ -2 & 5 & 0 & 1 \\ -1 & 0 & 5 & 2 \\ 0 & 1 & 2 & 5 \end{bmatrix},$$

which, as we know from Chapter 1, has the eigenvalues $\{2, 4, 6, 8\}$. In order to find the condition number of this matrix, we choose to use (3.3), which yields that

$$\kappa_2(\mathbf{M}) = 8/2 = 4.$$

Even though this condition number is four times greater than 1, it is still within the same order of magnitude, so the matrix $\mathbf{M}$ can be considered well-conditioned.

Next, let us look at another example matrix,

$$\mathbf{Z} = \begin{bmatrix} 1.0 & 0.0 \\ 1.000001 & 0.000001 \end{bmatrix}.$$

This matrix is what is sometimes referred to as almost-singular or numerically singular. Although it would be even more pronounced with more decimals, we can see how the two rows are almost linearly dependent. To see what effect this can have on an algorithm, let us compute the condition number of the matrix. We will need the inverse,

$$\mathbf{Z}^{-1} = \begin{bmatrix} 1.0 & 0.0 \\ -1.000001 \times 10^6 & 1.0 \times 10^6 \end{bmatrix}.$$

Computing the condition number is now simply a matter of finding the induced norms for the two matrices. Opting for the infinity-norm gives us

$$\|\mathbf{Z}\|_\infty = 1.000002, \quad \left\|\mathbf{Z}^{-1}\right\|_\infty = 2.000001 \times 10^6.$$

Therefore the condition number of $\mathbf{Z}$ is $\approx 2 \times 10^6$, which is a very large condition number. The condition number of a singular matrix is often defined to be ∞, so the closer we are to being numerically singular the larger the condition number will be — up to the maximum machine precision.

Another important problem, one that is of particular importance to us since most eigenvalue algorithms are based on matrix decompositions, is that of relatively small diagonal elements [27]. This can lead to an effect of that is sometimes referred to as swamping [24, 34]. Swamping can lead to huge numerical problems in e.g. Gaussian elimination. If the leading pivot is relative small, then when it is used to cancel rows, the other entries on those rows can grow or shrink vastly in several orders of magnitude.

To better understand this, let us consider a very simple example matrix, upon which we perform a LU decomposition.

$$\begin{bmatrix} 0.000001 & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1000000 & 1 \end{bmatrix} \begin{bmatrix} 0.000001 & 1 \\ 0 & -999999 \end{bmatrix}$$

Even for a small matrix like this, with a precision of only 6 decimals, we get some quite large values in the decomposition. We see how this type of problem can quickly lead to numerical instability with larger matrices due to the limits of the machine precision. Round-off and truncation errors are bound to happen, which means computational precision and subtle information in the data will be lost.

Now, what would happen if we simply permuted the two rows in the example above, before performing the LU decomposition?

$$\begin{bmatrix} 1 & 1 \\ 0.000001 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0.000001 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 0 & 0.999999 \end{bmatrix} \tag{3.4}$$

The resulting LU decomposition for the permuted matrix is much more reasonable in terms of numerical stability. Also note that since we have the explicit permutation matrix, we can still use this decomposition to e.g. solve systems of equations of the form $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$, or similar [34].

3.2. Permutation matrices

As alluded to in the previous section, the swamping problem in particular can usually be resolved by permuting the matrix in such a manner as to reduce the effect of small pivots.

Definition 3.2: Permutation matrix

Let $\mathbf{P}$ be a $n \times n$ identity matrix in which rows or columns have been permuted. Then the result of computing

$$\widetilde{\mathbf{A}} = \mathbf{PA}$$

is equivalent to performing the same row or column permutations on the matrix $\mathbf{A}$ directly. We refer to $\mathbf{P}$ as a *permutation matrix*.

Before we move on, let us construct a simple example to further demonstrate the concept of a permutation matrix. Starting with a 3×3 identity matrix, we permute the first and

second row which gives

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \mathbf{P}.$$

Given that we have an arbitrary 3×3 matrix defined as

$$\mathbf{A} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}.$$

Left-multiplying the permutation matrix with this matrix yields

$$\mathbf{PA} = \begin{bmatrix} d & e & f \\ a & b & c \\ g & h & i \end{bmatrix},$$

which is exactly the same as if we permuted the rows of $\mathbf{A}$ directly. Furthermore, if we right-multiply we get

$$\mathbf{AP} = \begin{bmatrix} b & a & c \\ e & d & f \\ h & g & i \end{bmatrix},$$

which is the column-permuted version of $\mathbf{A}$ where the first and second columns have been switched. Extending this one step further, the result of $\mathbf{APA}$ would yield a matrix where we first permute the first and second column, then the first and second row of the column-permuted matrix.

Note that in order to reduce the spatial complexity in the context of an actual machine implementation with large matrices, we do not have to store the entire permutation matrix. We only need to represent it as a vector which contains the index of the rows in the order corresponding to the permutation. In other words the permutation matrix above can be represented as

$$\boldsymbol{p} = \begin{bmatrix} 2 & 1 & 3 \end{bmatrix}^{\mathsf{T}}.$$

The use of a permutation matrix to improve the numerical stability of the LU decomposition by permuting rows, as we saw in (3.4), is usually referred to as LU decomposition with partial pivoting, or simply PLU decomposition. For further details, see e.g. Turing [22] and Sauer [34].

3.3. Symmetric QR algorithm with permutations

Expanding upon the idea of using permutation matrices to improve numerical stability, we want to consider how the use of permutation matrices can improve the convergence rate of the QR algorithm. Specifically, we want to look at the techniques described in Krishnamoorthy [2] (hereafter referred to as the *reference paper*).

As we know from Chapter 2, the QR algorithm can be thought of as an extension of the power method. Moreover, from (2.3), we know that the convergence performance of the power method is bounded by the ratio of the two most-dominant eigenvalues. This can extended for the QR algorithm such that the convergence can be seen as being bounded by the ratio of "neighboring" eigenvalues. Thus, the hope is that we can improve the convergence rate by optimally permuting the matrix decomposition during each iteration of the QR algorithm. The permutations can also lead to reduced round-off and truncation errors.

Two permutation schemes are proposed; *diagonal ordering* (DO), in which the matrix is permuted such that the diagonal elements are ordered in descending order based on absolute value, and *column ordering* (CO), which is similar to the diagonal ordering, but with the addition that the columns are permuted in descending order based on their norm. An important realization here is that the shape of the matrix is not preserved between successive iterations of the algorithm, so optimization techniques via construction of matrices with special forms will not work as expected [2].

As presented in the reference paper, the proposed symmetric QR algorithm with permutations requires that we compute the following additions at each k-th iteration,

1. Construct a permutation matrix $\mathbf{P}_k$ for the current eigenvalue estimate $\mathbf{A}_{k-1}$.
2. Perform the symmetric permutation of $\mathbf{A}_{k-1}$ before QR decomposition.
3. Permute $\mathbf{Q}_k$ prior to multiplication with current eigenvector estimate $\mathbf{V}_{k-1}$.

Augmenting the QR algorithm as expressed in Definition 2.4 with these additions means that the sequence in (2.4) becomes

$$\mathbf{P}_k\mathbf{A}_{k-1}\mathbf{P}_k = \mathbf{Q}_k\mathbf{R}_k, \quad \mathbf{A}_k = \big(\mathbf{P}_k^\mathsf{T}\mathbf{Q}_k\big)^\mathsf{T}\mathbf{A}_{k-1}\big(\mathbf{P}_k^\mathsf{T}\mathbf{Q}_k\big), \quad k = 1, 2, \ldots,$$

where we have incorporated the result from (2.5). The construction of a permutation matrix that will order a matrix according to the DO scheme, can be done as follows.

Algorithm 3.1 Permutation matrix for diagonal ordering

Require: $\mathbf{A}$ is a $n \times n$ positive-definite matrix.
Ensure: $\mathbf{P}$ is a $n \times n$ permutation matrix.

1: **function** DIORD($\mathbf{A}$)
2: $n = \text{NUMROWS}(\mathbf{A})$
3: $\boldsymbol{d} \leftarrow |\text{diag}(\mathbf{A})|$
4: $\boldsymbol{p} = \text{SORTINDICES}(\boldsymbol{d})$ ▷ The indices that would sort in descending order.
5: $\mathbf{P} = \text{PERMUTATIONMATRIX}(\boldsymbol{p})$ ▷ Creates a permutation matrix from a vector.
6: **return P**
7: **end function**

Constructing permutation matrices for the CO scheme can be achieved by invoking the DIORD function with $\mathbf{A}^2$ instead of $\mathbf{A}$ [2]. We note that computing this matrix power might be computationally expensive, so consideration needs to be given whether potential improvements in convergence is worth the additional expense. Furthermore, as mentioned in the reference paper, other permutation schemes are also bound to exist, but there will always be some level of trade-off between improved convergence and computational cost.

Putting all the pieces together, yields the augmented QR algorithm where the use of diagonal ordering permutations will hopefully yield improved convergence.

Algorithm 3.2 QR algorithm with permutations

Require: $\mathbf{A}_0$ is a $n \times n$ matrix with a QR decomposition $\mathbf{A}_0 = \mathbf{Q}_0\mathbf{R}_0$.
Ensure: $\text{diag}(\mathbf{A}_{k_{max}})$ and $\mathbf{V}_{k_{max}}$ has converged to the eigenpairs of $\mathbf{A}_0$.

1: $\mathbf{V}_0 \leftarrow \mathbf{I}$ ▷ The $n \times n$ identity matrix.
2: **for** $k = 1, 2, \ldots$ **do** ▷ Iterate until stopping criteria reached.
3: $\mathbf{P}_k = \text{DIORD}(\mathbf{A}_{k-1})$ ▷ Diagonal ordering (DO) scheme.
4: $\mathbf{P}_k\mathbf{A}_{k-1}\mathbf{P}_k^\mathsf{T} \rightarrow \mathbf{Q}_k\mathbf{R}_k$ ▷ Only need to compute $\mathbf{Q}_k$.
5: $\mathbf{U}_k \leftarrow \mathbf{P}_k^\mathsf{T}\mathbf{Q}_k$
6: $\mathbf{A}_k \leftarrow \mathbf{U}_k^\mathsf{T}\mathbf{A}_{k-1}\mathbf{U}_k$
7: $\mathbf{V}_k \leftarrow \mathbf{V}_{k-1}\mathbf{U}_k$
8: **end for**

As we can see from a high-level perspective, the algorithm does not seem to be very different from the original QR algorithm as expressed in Algorithm 2.2. However, there is quite a bit of additional computational complexity "hidden" within the algorithm. First and foremost, the construction of the permutation matrix in the call to DIORD is non-trivial and likely to be a computational hotspot. Moreover, the additional matrix

products will also be expensive to compute if the matrices are both dense and large. The added computational cost versus the improved convergence needs to be considered on an application-by-application basis.

In order to verify the improvements to the convergence, we mimic the simulation found in Krishnamoorthy [2], which means using a randomly generated set of 25000 symmetric, positive definite 4×4 matrices. For each matrix in the set, we run both the classical QR algorithm and the QR algorithm with permutations. Each algorithm runs for a total of 50 iterations on each matrix, and for each iteration we calculate the error in the current eigenvalue estimate. The eigenvalue error estimate is computed by

$$E_k^2 = \|\boldsymbol{\lambda}_k - \boldsymbol{\lambda}\|_2^2, \quad \boldsymbol{\lambda}_k = \text{sort}\left(\text{diag}\left(A_k\right)\right), \tag{3.5}$$

where $\boldsymbol{\lambda}_k$ are the current eigenvalue estimates, $\boldsymbol{\lambda}$ are the target eigenvalues for the matrix associated with the iteration. Both vectors are then sorted in ascending order.

In order to get a sense of the overall performance, the average error in the eigenvalue estimates is then computed. The error is computed on a per-algorithm basis across the results for all of the 25000 matrices, thus yielding a reasonable performance estimate.

The result of applying the permutations to the algorithm can be seen in Figure 3.3.1, and as we can see the results correspond well with the original results from Krishnamoorthy [2]. Both permutation schemes produce almost identical results with respect to convergence. This means that by using permutations to get some optimal reordering of the vectors during each iteration, we can improve the convergence rate of the QR algorithm by nearly a factor of two. This is a huge improvement. It essentially means we can run fewer iterations while still getting equivalent results when compared to the classical QR algorithm. However, the permutation variant of the algorithm does require additional computations. Hence, it is likely that even though we run with a lower number of iterations, the temporal improvements can be significantly diminished due to the increased computational complexity of each iteration. The factor with which the gain is diminished by the increased computational effort, depends in part on the cost of computing the permutation matrix, as mentioned in Chapter 3. We also note that CO permutation scheme will incur an additional cost compared to the DO scheme due to having to compute an additional matrix product.

In Chapter 7 we will attempt to overcome the cost of these additional computational complexities by implementing the algorithm on a GPU. Since many matrix and vector calculations are embarrassingly parallelizable, and given the appropriate conditions, can be extremely efficient to compute on a GPU [48], there is some hope that we can overcome the additional overhead. Although, we should keep in mind that the overall algorithm is sequential, which means that implementing the algorithm in an efficient manner on a GPU will not be trivial.

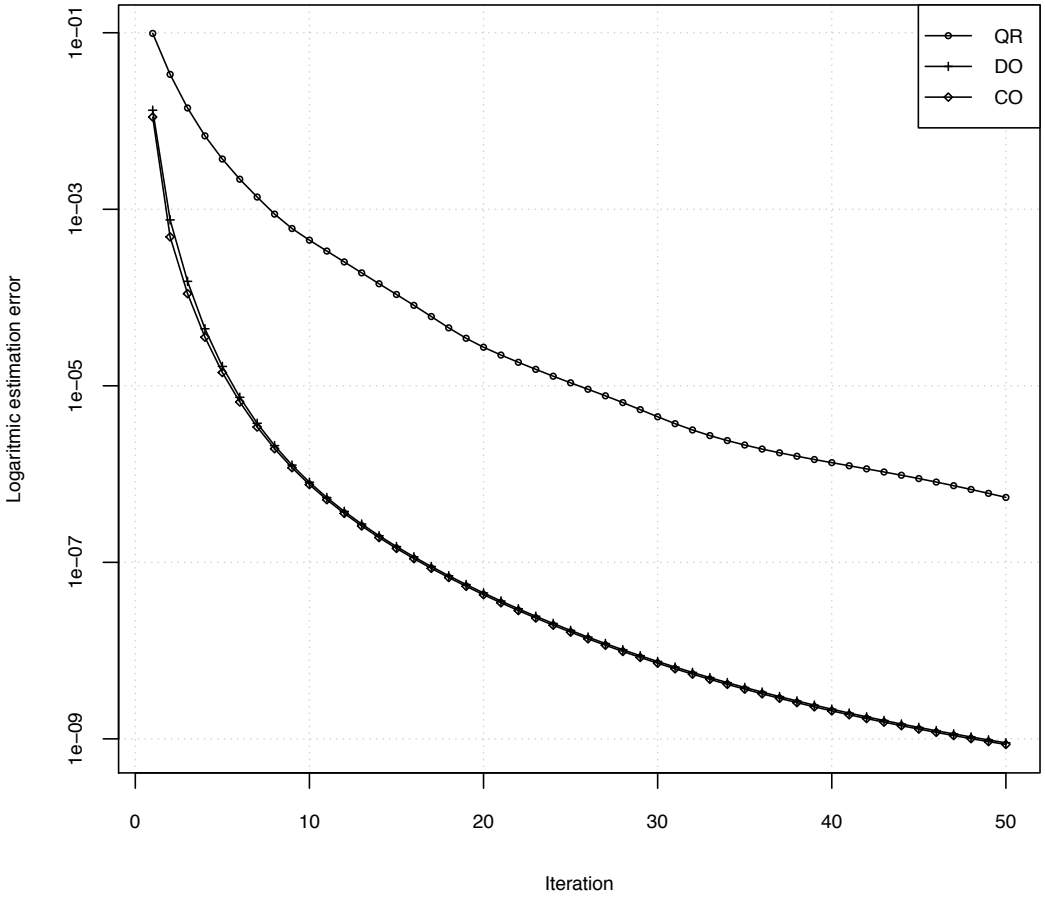


Figure 3.3.1. The plot shows the average of the squared estimation error defined by (3.5), of the "classical" QR algorithm compared to the symmetric QR algorithm with permutations. The latter algorithm shows an almost two-fold increase in convergence performance. Note that both permutation schemes, DO and CO, produce almost identical results.

Chapter 4.

Some examples of spectral methods

Preprocessing can often be a crucial step to successfully training a machine learning model, both in terms of reducing the amount of redundant or noisy data, but also to reduce the dimensionality of the input domain. Reducing the dimensionality of the training data can in many cases be directly correlated to reducing the number of parameters that need to be optimized in the model. Lowering the number of parameters will generally reduce the cost of the optimization problem. Redundancy reduction can also help in this regard, but can additionally help with overfitting, and play a role in ensuring that models are more generalized [49].

There exists a multitude of dimensionality reduction methods, but since this thesis is fundamentally based on the study of eigenvalue algorithms, we will limit ourselves to studying a limited set of (nonlinear) spectral methods.

4.1. Principal component analysis

One of the most commonly used linear dimensionality reduction techniques in data science is the so-called *principal component analysis* (PCA), which was introduced in Pearson [50]. Briefly, PCA can be summarized as first finding an orthogonal transformation based on the eigenvectors of the covariance structure of the input data that decorrelates the feature space by maximizing the variance. The transformation equals a change of basis that retains the dimensionality of the feature space. The dimensionality can be reduced by picking the largest k eigenvectors in descending order measured by the corresponding eigenvalues. Note that k can be chosen somewhat arbitrarily, but it is common practice to pick a value based on the total variance described by the chosen eigenvectors. This can be done in several ways, where one possibility is to use a scree plot of the eigenvalues in descending order, and then look for an "elbow joint". In Figure 4.1.1 we can see an example of such an "elbow joint" around the third eigenvector. In some situations this is a useful and intuitive visual approach for determing

where the relative magnitudes of the corresponding eigenvectors are rapidly decreasing, and thus determine a reasonable k. To understand why the length of an eigenvector in this context is important, we only need to realize that the eigenvalue describes not only the length of the corresponding eigenvector, but is also a measure of the variance for a unique linear combination of input features.

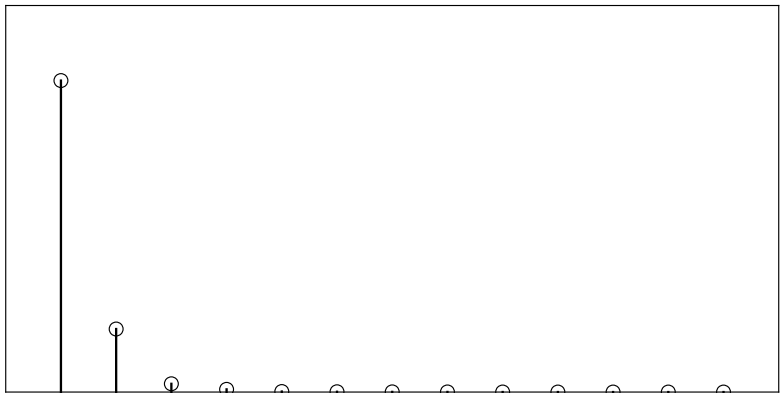


Figure 4.1.1. Elbow...

An alternative approach to the visual inspection of a scree plot is to simply pick the number of eigenvectors by calculating the accumulated contribution of each eigenvector to the total variance. Then we can decide to define k such that the resulting lower-dimensional representation accounts for e.g. at least 90% of the total variance; e.g.

$$\frac{\sum_{i=1}^{k} \lambda_i}{\sum_{i=1}^{N} \lambda_i} \geq 0.9,$$

where the eigenvalues λ_i are assumed sorted in descending order, and N denotes the total number of eigenvalues. If we think in terms of Bayesian statistics, we can view variance as a measure of information. Further, we can consider such a change of basis as preserving, or explaining, 90% of the information in original data even though the number of dimensions may have been greatly reduced. Note that for this to be reasonable we should either assume, or ensure, that the data has been normalized prior to finding eigenvalues and eigenvectors.

Once a new variance-maximizing basis has been found, mapping the original data to the basis can be achieved by multiplying the original data in matrix form with the transformation matrix which consists of the k chosen eigenvectors. Let us consider a simple example for some $m \times n$ matrix $\mathbf{X}$ which holds n observations with m features. The corresponding covariance matrix is denoted $\mathbf{K}$, is positive semi-definite, and therefore has an eigenvalue decomposition that can be expressed as

$$\mathbf{K} = \mathbf{V}\mathbf{D}\mathbf{V}^{\mathsf{T}}.$$

Here $\mathbf{V}$ is a $n \times n$ matrix that has the eigenvectors of $\mathbf{K}$ stored column-wise, and $\mathbf{D}$ is a diagonal matrix with the eigenvalues of $\mathbf{K}$ stored on the diagonal. If we assume that the eigenpairs have been sorted in descending order based on the eigenvalue, the a k-dimensional PCA transformation of $\mathbf{X}$ can be expressed as

$$\mathbf{X}_{\text{pca}} = \mathbf{X}^{\mathsf{T}}\mathbf{V}_k,$$

where $\mathbf{V}_k$ is a $m \times k$ matrix holding the k largest eigenvectors.

Figure 4.1.2 shows an example of a simple toy dataset consisting of two sets of observations, sampled from two distinct, bivariate Gaussian distributions. The two "blobs" are linearly separable, since it would be easy to draw a straight line that separates the two classes. Although, if we wanted to create a function that could discriminate between the two classes, we would need to use both features. Applying a PCA transformation of the dataset, yields the results seen in Figure 4.1.3, and Figure 4.1.4. From the latter we see that we could create a one-dimensional classifier, that would perfectly separate both classes from the two-dimensional input data.

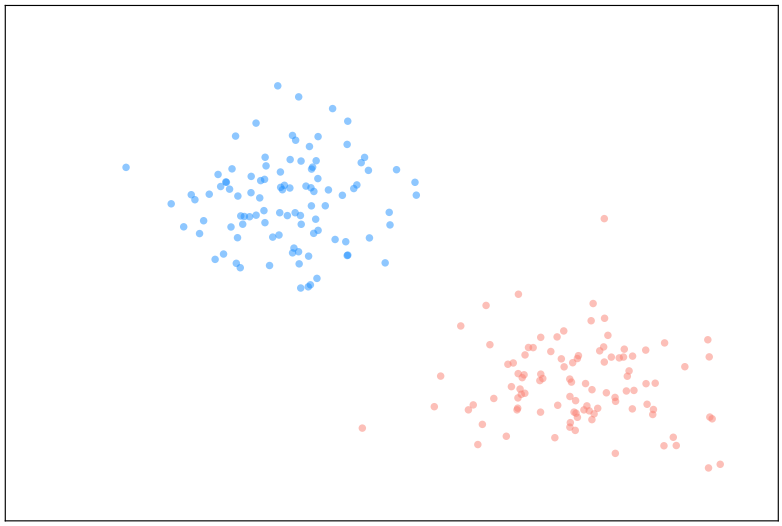


Figure 4.1.2. Synthetic dataset containing two sets of observations, sampled from two distinct, bivariate Gaussian distributions. Results produced using *scikit-learn* [51].

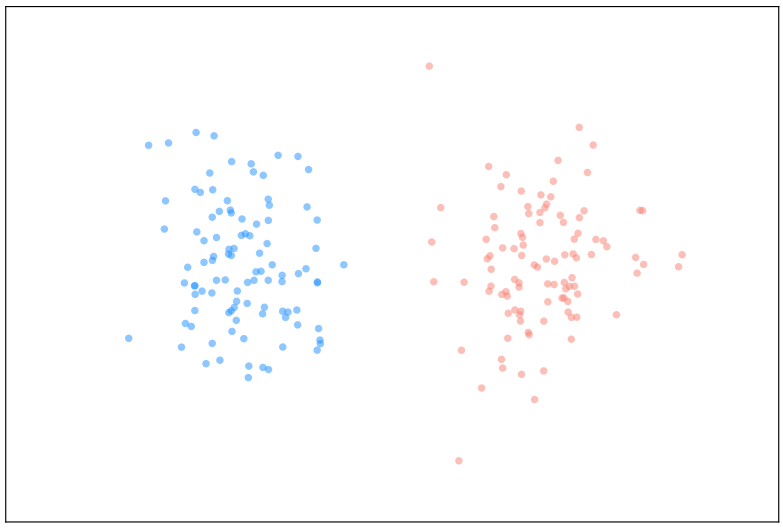


Figure 4.1.3. The result of projecting the Gaussian "blobs" from Figure 4.1.2 into the two-dimensional PCA space. Separability is maintained, and the rotation gives us a hint that the two classes might be separable in one dimension. Results produced using *scikit-learn* [51].

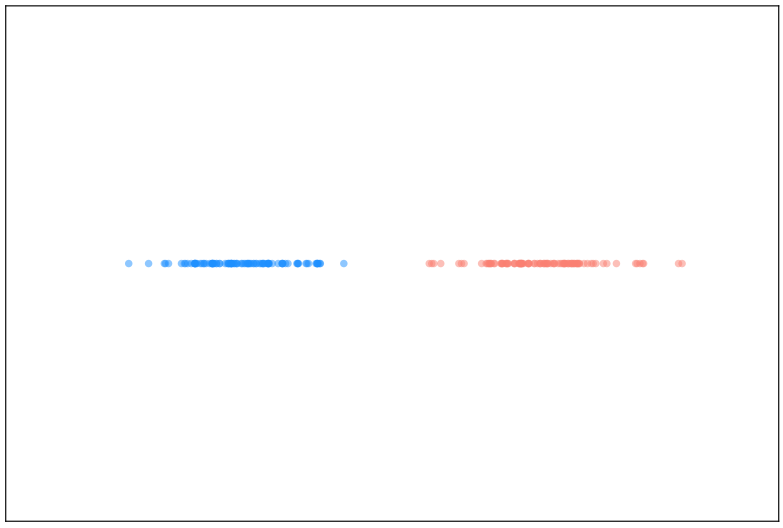


Figure 4.1.4. The two-dimensional, two-class problem after having been projecting onto the most dominant principal component. As we can see, we are now capable of separating the two classes of observations, using only one feature. Results produced using *scikit-learn* [51].

4.2. Kernel PCA

PCA is based on a linear transformation, which makes it a reasonable choice for data that is linearly separable, but what if we have a dataset such as the one we can see in Figure 4.2.1? It should be obvious that it is not separable using linear method such as PCA, and this can clearly be seen in both Figure 4.2.2, and Figure 4.2.3. Since PCA attempts to maximize the variance in the data, what we end up finding is a linear transformation that only slightly rotates the original dataset. In other words, if we hope to be able to separate the two classes, we will need a nonlinear method or transformation. One possible approach is to construct an (implicit) mapping from the input feature space into some higher-dimensional space where the data becomes linearly separable, such that we can leverage our usual linear methods — this is the premise behind kernel PCA.

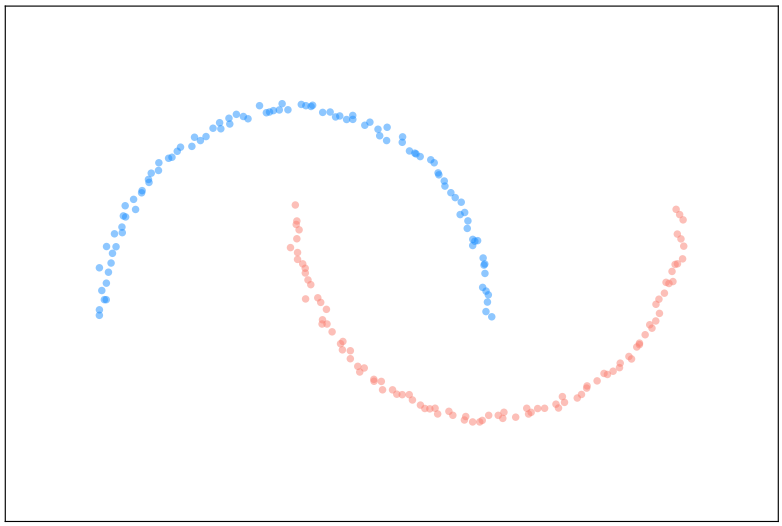


Figure 4.2.1. Randomly generated, two-class problem. The half moon structure makes the dataset non-separable using linear methods. Results produced using *scikit-learn* [51].

Let us assume for a moment that we have some observation $\boldsymbol{x}$ with n features, and some mapping function $\boldsymbol{\phi}(\cdot)$ such that

$$\boldsymbol{x} \in \mathbb{R}^n \mapsto \boldsymbol{\phi}(\boldsymbol{x}) \in H,$$

where H is a so-called Hilbert space. Hilbert spaces are complete vector spaces that possess the inner product [52], and can potentially be infinite-dimensional. Mercer's theorem tells us that if we ensure the mapping function preserves the inner products, then there exists an equivalent representation

$$\langle \boldsymbol{\phi}(\boldsymbol{x}) | \boldsymbol{\phi}(\boldsymbol{z}) \rangle = \kappa(\boldsymbol{x}, \boldsymbol{z}),$$

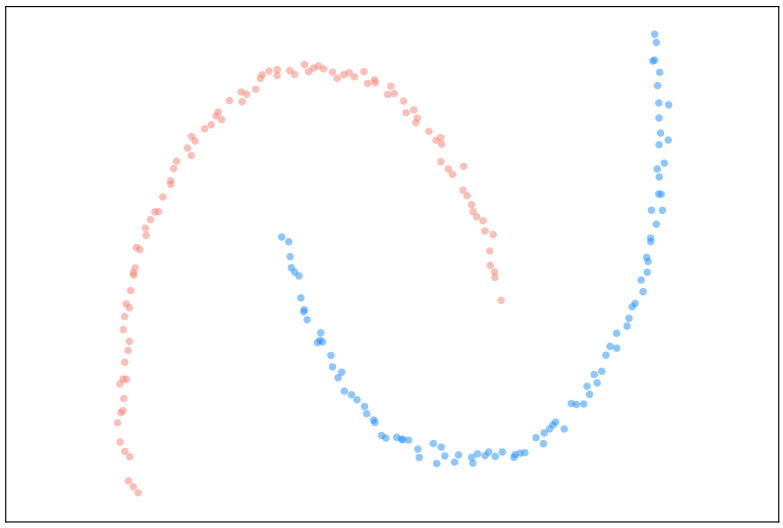


Figure 4.2.2. We can clearly see that the "half moon" dataset from Figure 4.2.1 is not linearly separable. The two-dimensional PCA transformation has only slightly rotated the data in some sense. Results produced using *scikit-learn* [51].

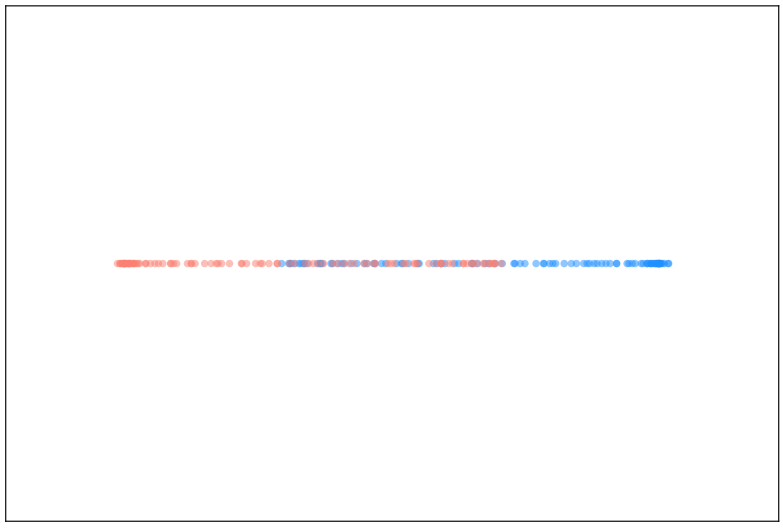


Figure 4.2.3. The one-dimensional linear PCA transformation of the "half moon" data. As we clearly see, there is no hope of being able to separate the two sets of observations. Results produced using *scikit-learn* [51].

where $\kappa(\cdot,\cdot)$ is a positive semi-definite kernel function [53], and H is a so-called *reproducing kernel Hilbert space* [54]. The kernel function must satisfy the following condition,

$$\iint_C \kappa(\boldsymbol{x},\boldsymbol{z})g(\boldsymbol{x})g(\boldsymbol{z})\,\mathrm{d}\boldsymbol{x}\,\mathrm{d}\boldsymbol{z} \geq 0,$$

where

$$\int_C g(\boldsymbol{x})^2\,\mathrm{d}\boldsymbol{x} < \infty_+ \mid \forall g(\boldsymbol{x}), \boldsymbol{x} \in C \subset \mathbb{R}^n.$$

The condition placed on the set of functions $g(\boldsymbol{x})$ essentially ensures the existence of the L^2 norm. Whereas the condition placed directly on the kernel gives us the positive semi-definite property of the kernel. We also have the so-called Moore-Aronszajn theorem, which tells us that any symmetric, positive definite kernel has a unique reproducing kernel Hilbert space [55]. The results granted by these two theorems are in some sense the foundation of our kernel-based methods.

What the observant reader might have noticed, is that we have no generalized toolkit at our disposal when it comes to defining these kernel functions. All we really know is that we must ensure the conditions from Mercer's theorem are met. In fact, finding such kernel functions is in itself an entire field of research, so we will not attempt to discuss this further. Instead we will simply present three commonly used kernel functions.

First we have the perhaps most frequently used kernel, the so-called radial basis function (RBF) kernel, which can also be referred to as a Gaussian kernel in some literature;

$$\kappa(\boldsymbol{x},\boldsymbol{z}) = \exp\bigl(-\gamma\|\boldsymbol{x}-\boldsymbol{z}\|_2^2\bigr), \quad \gamma > 0.$$

The RBF kernel is based on the squared Euclidean distance (the 2-norm) between the vectors, which is used as a measure of the (dis)similarity. Thus, we can see that if $\boldsymbol{x} = \boldsymbol{z}$ we get a coefficient equal to 1. The further apart the vectors are (i.e. less similar) the closer we get to 0, but the kernel does not converge to 0. It can be shown that the reproducing kernel Hilbert space of a RBF kernel is infinite-dimensional.

Next up, let us consider the polynomial kernel which can be defined as

$$\kappa(\boldsymbol{x},\boldsymbol{z}) = \bigl(b\boldsymbol{x}^\mathsf{T}\boldsymbol{z} + a\bigr)^q, \quad a \geq 0,\, b > 0,\, q > 0.$$

This is perhaps one of the simplest kernel functions, but it can as well shall see later work remarkably well. It is worth noting that if the vectors given to the kernel are orthogonal, the resulting scalar is simply a^q, and if $a = 0$ the kernel is sometimes referred to as homogeneous. Usage of polynomial kernels seem to be popular in the field of natural language processing [56, 57].

Last, but not least, we present the hyperbolic tangent kernel,

$$\kappa(\boldsymbol{x}, \boldsymbol{z}) = \tanh\big(\beta \boldsymbol{x}^\mathsf{T} \boldsymbol{z} + \alpha\big),$$

where α and β needs to be chosen such as to satisfy the conditions of a valid kernel function. This kernel belongs to a special class of kernels based on sigmoid functions, which are used in e.g. logistic regression methods.

Armed with a kernel framework, we can begin to consider the kernel PCA method. Let us first assume we have some dataset $\mathbf{X}$, and our goal is to perform an (implicit) mapping into a reproducing kernel Hilbert space such that our observations become linearly separable. Specifically,

$$\boldsymbol{x} \in \mathbf{X} \mapsto \boldsymbol{\phi}(\boldsymbol{x}) \in H.$$

PCA is based on an eigendecomposition of the covariance (or correlation) matrix, and the trick now is to perform this on the estimated covariance matrix in the implicitly mapped kernel space. To simplify matters, let us first assume the dataset $\mathbf{X}$ has zero mean, i.e. it is centered. This gives us that the sample covariance matrix can be expressed

$$\mathbf{Q} = \frac{1}{n} \sum_{i=1}^{n} \boldsymbol{\phi}(\boldsymbol{x}_i) \boldsymbol{\phi}(\boldsymbol{x}_i)^\mathsf{T}$$

As stated, our goal is to find the principal components of this matrix. We need to perform the eigendecomposition

$$\mathbf{Q}\boldsymbol{v} = \lambda \boldsymbol{v}.$$

From this, in combination with the definition of the sample covariance matrix, we get that

$$\lambda \boldsymbol{v} = \frac{1}{n} \sum_{i=1}^{n} \big(\boldsymbol{\phi}(\boldsymbol{x}_i)^\mathsf{T} \boldsymbol{v}\big) \boldsymbol{\phi}(\boldsymbol{x}_i).$$

Because $\mathbf{Q}$ is positive definite we can this rewrite as

$$\boldsymbol{v} = \sum_{i=1}^{n} \alpha_i \boldsymbol{\phi}(\boldsymbol{x}_i),$$

and it can be shown that the eigendecomposition above is equivalent performing an eigendecomposition of the so-called Gramian matrix of the kernel function [58]. The Gramian matrix can be defined as

$$\mathbf{K}(i, j) = \kappa(\boldsymbol{x}_i, \boldsymbol{x}_j),$$

where $\kappa(\cdot, \cdot)$ is the kernel function. As it turns out, the i-th eigenvector of $\mathbf{Q}$ corresponds to the i-th eigenvector of $\mathbf{K}$. This essentially boils down to that all we need to do to perform kernel PCA is to perform linear PCA on the Gramian matrix $\mathbf{K}$. For the full derivation of kernel PCA, including normalization considerations, and so forth, see e.g. Schölkopf, Smola, and Müller [59], and Theodoridis [39].

We can summarize kernel PCA as a sequence of steps;

1. Calculate the Gramian matrix $\mathbf{K}(i, j)$ for all observations, $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$.
2. (*Optional*) Ensure $\mathbf{K}$ has zero-mean by "centering" the matrix.
3. Eigendecompose $\mathbf{K}$ to find the k dominant eigenpairs.
4. Normalize the k dominant eigenvectors.
5. Project the data as represented by $\mathbf{K}$, onto the k normalized eigenvectors.

Returning to the toy dataset shown in Figure 4.2.1, applying kernel PCA with a RBF kernel ($\gamma = 20$) yields the results seen in Figure 4.2.4, and Figure 4.2.5. Comparing these to the earlier result from using traditional linear PCA, we can see the kernel-based method has successfully separated the two classes.

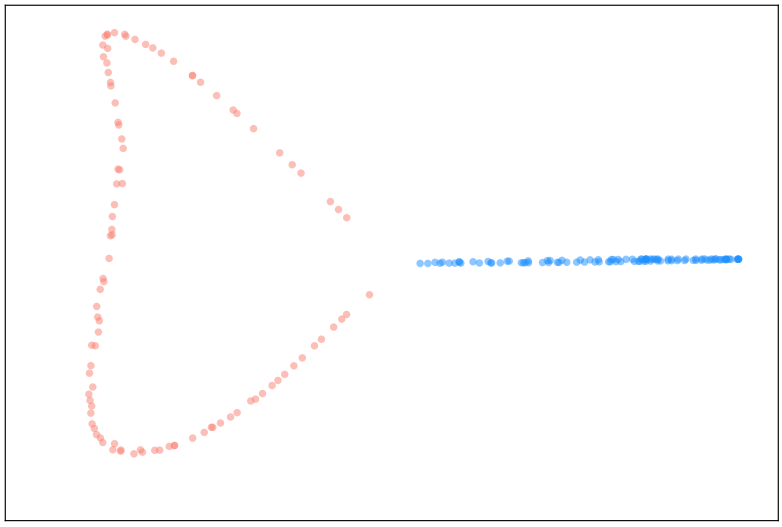


Figure 4.2.4. The two classes from the "half moon" dataset become linearly separable after being projected into the kernel PCA space. Results produced using *scikit-learn* [51].

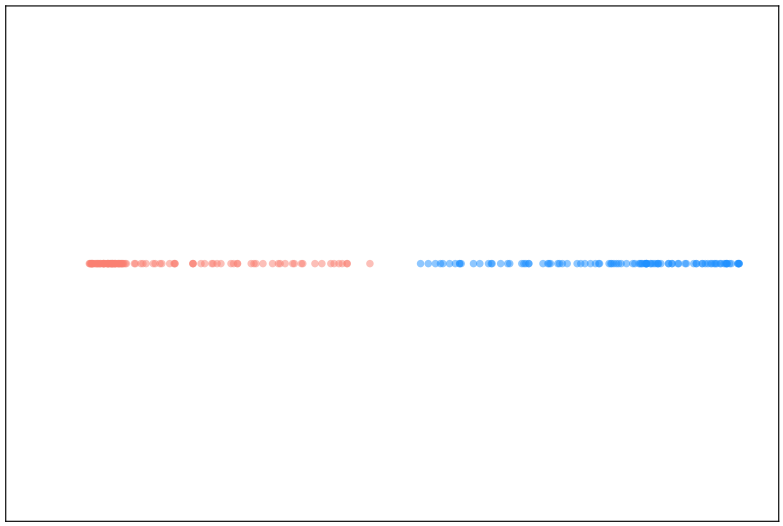


Figure 4.2.5. The two class "half moon" dataset becomes linearly separable using only a single feature after being projected into the kernel PCA space. Results produced using *scikit-learn* [51].

4.3. Kernel entropy component analysis

Kernel entropy component analysis is very similar to kernel PCA. As we know, PCA methods only take into account the variance of the data. The importance of an eigenvector in determined by variance maximization. While this in many cases is sufficient for performing classification, it might not always be sufficient for effective clustering techniques. KECA addresses this by taking into account the entropy of the input data, and uses entropy maximization as the scheme for picking eigenvectors. This approach makes KECA fundamentally different when we compare it to other spectral methods. KECA can reveal interesting structures in the data, which are not revelead by e.g. kernel PCA [60].

The KECA method is based on the Renyi quadratic entropy, which can be given as $H(p) = -\log V(p)$, where $V(p) = \int p^2(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}$, and $p(\boldsymbol{x})$ is the probability density function generating the input data [1]. Further, if the probability density function is estimated by a Parzen window density estimator, then it can be shown that an estimate for the Renyi entropy can be given by [61]

$$\widehat{V}(p) = \frac{1}{n^2}\mathbf{1}^\mathsf{T}\mathbf{K}\mathbf{1},$$

where $\mathbf{K}$ is the RBF kernel matrix. The kernel matrix has an eigendecomposition

$\mathbf{K} = \mathbf{V}\mathbf{D}\mathbf{V}^\mathsf{T}$, which means we can rewrite the entropy estimator as

$$\widehat{V}(p) = \frac{1}{n^2} \sum_{i=1}^{n} \left(\sqrt{\lambda_i} \boldsymbol{e}_i^\mathsf{T} \mathbf{1} \right)^2.$$

Importantly, it has been shown that the i-th term of this expression, corresponds to the i-th principal component in the kernel space [60]. Finally, the k-dimensional KECA transformation can be expressed by

$$\boldsymbol{\Phi}_k = \mathbf{D}_k^{\frac{1}{2}} \mathbf{V}_k^\mathsf{T}$$

For further details on KECA, including good examples illustrating the differences between kernel PCA and KECA, please see e.g. Jenssen and Storås [61].

Chapter 5.

General-purpose computing on graphics processing units

Typically when running any type of computation, we run on one or multiple CPUs, on one or multiple computers, and the computations might either be serial or parallel in nature. Historically CPUs have followed the oft-cited observation called *Moore's law*, which states that CPU chip complexity will double approximately every two years [62]. This complexity measure can also be thought of in terms of the CPU clock speed being doubled every two years. For several decades this predication came true, but in recent years this trend has stopped since the transistors have become so small, that by making them even smaller, they would become subject to various soft-errors. For instance, they would become subject to bit-flip errors caused by cosmic background radiation, due in part to a combination of reduced voltage and capacitance enforced by thermal limitations. The effect being that the charge carried by an alpha particle originating from cosmic radiation is enough to flip a bit at the transistor level in the CPU [63].

Since the CPU speed increase trend has flattened, CPU manufacturers have begun adding more CPU cores. Thus motivating programmers to parallelize algorithms to fully take advantage of the computing power available in multi-core CPUs. At the same time as this shift towards multi-core CPUs has taken place, we have also seen the advent of general-purpose computing on GPU (GPGPU). GPUs are highly specialized devices which these days typically have several thousand compute cores, and are designed for massively parallel tasks. Originally these devices were intended for rendering computer graphics, but have since become general-purpose computational devices. The usage of GPUs have been crucial to the success of some machine learning methods. One particular example of such a method, is the so-called convolutional neural network (CNN). This particular type of neural network lends itself very well to massively parallel computations since it is, as the name implies, based on convolutions. More importantly, the convolution operation can be performed as a series of matrix and vector operations [64, 65]. Furthermore, many linear algebra operations, such as the matrix-vector product, can be considered embarrassingly parallel [66].

5.1. NVIDIA CUDA

Currently there are a few different GPU hardware vendors and GPU platforms, including programming toolkits, etc. Although, the de facto standard in machine learning at the time of writing, is the CUDA platform from NVIDIA. It consists of both hardware and software, and have in recent years seen big improvements both in runtime performance and the overall development efficiency/effort. However, even though the development effort has been reduced in the last few years, it is still a highly complex task to write code that fully utilizes the compute resources offered by a modern GPU, and even more so when using multiple GPUs at once.

We remark that from this point onward, unless otherwise stated, the term GPU is implicitly referring to modern NVIDIA GPUs. Therefore, some of the terminology used, performance metrics, and so forth, might not apply to other GPU platforms.

5.1.1 GPU architecture

We conjecture that in order to fully utilize the entire compute potential afforded by a modern GPU, it is crucial to have at least a basic understanding of the underlying hardware architecture, and associated design choices. We cannot approach the GPU as if it is a CPU. As we will see, the two devices are very different at a fundamental level, and in order to use both types of devices as efficiently as possible, we need to be aware of these differences.

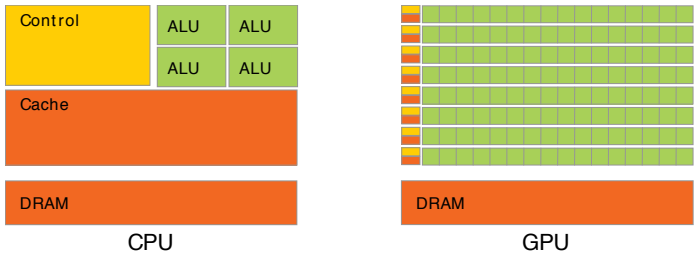


Figure 5.1.1. High-level hardware abstraction illustrating the differences in the design of a modern CPU chip compared to a modern GPU. From the CPU design we see that a big portion of the chip is dedicated to cache memory, as well as having a large control unit, and few advanced ALUs. The GPU design on the other hand has limited amounts of cache memory, and it is shared across many simple, cooperating ALUs. Each group of ALUs are governed by small control units. Reprint from *CUDA C Programming Guide* by NVIDIA [67].

A traditional CPU differs from a GPU in several ways, both conceptually and in how the actual hardware is designed. Figure 5.1.1 shows a simplified, high-level depiction of the chip differences. The CPU design has dedicated a rather big portion of the chip to cache,

as well as generally supporting much bigger backward and forward compatible scalar and vector instruction sets, novelties such as branch prediction, out of order execution, etc. All of this requires bigger and more complex control and logic units. On the other side, the GPU has limited amounts of cache memory, has little to no support for features such as branch prediction, out of order execution, etc. Additionally NVIDIA GPUs typically do not maintain any backward or forward compatibility in the instruction sets, and have a smaller instruction set in general by virtue of only having support for vector instructions. This yields much smaller and simpler control and logic units. The net result is that a GPU has more room on its chip dedicated to small, purely computationally-bound units. Modern workstation CPUs typically have between 6 and 10 compute cores, and have support for simultaneous multi-threading (SMT) which means that each core supports running 2 threads simultaneously. Hence, a high-end, workstation CPU can execute 20 threads at the same time. State of the art server-oriented CPUs currently support up to 44 hardware threads [68]. The equivalent exists on the GPU side of the spectrum, but at a much larger scale with the current state of the art GPU supporting up to 60 cores, each core supporting up to 64 simultaneous threads [69]. These cores are part of what is referred to as a streaming multiprocessor (SM), which is essentially responsibly for executing GPU threads. A GPU with 60 cores can in theory execute 3840 threads simultaneously (although due to certain design limitations the actual number is 3584). This is nearly two orders of magnitude greater than the comparable CPU, which has 44 threads. Although it is important to keep in mind that a single GPU core is much slower when compared to a single CPU core.

The differences between CPU and GPU chip designs stem primarily from the fact that the CPU is designed to finish computing each of its tasks as fast as possible, and since reading from system DRAM is orders of magnitude slower than reading from on-chip cache memory [70], the CPU therefore has multiple layers of cache to help reduce the latency incurred by reading from system DRAM. Additionally modern CPUs have functionality such as prefetching which in some situations can translate to a CPU effectively having “infinite” cache memory [70, 71]. GPUs have largely been optimized for maximum throughput — each computation does not have finish executing as fast as possible. Instead the GPU is designed to run as many computations in parallel as possible. In order to achieve this design goal while using an equivalent chip size, the amount of cache memory is sacrificed at the expense of fitting more compute units on the chip [72, 73].

The chip design differences lead to a vastly improved computational throughput in the case of the GPU. In Figure 5.1.2 we can see the how the theoretical number of floating point operations per second has evolved for both CPU and GPU devices in recent years. Currently a state of the art GPU has about 10 times more theoretical compute power when compared to a state of the art CPU. We see a similar story when it comes to the theoretical memory bandwidth, in Figure 5.1.3.

Keep in mind that these performance metrics are the theoretical, optimal performance

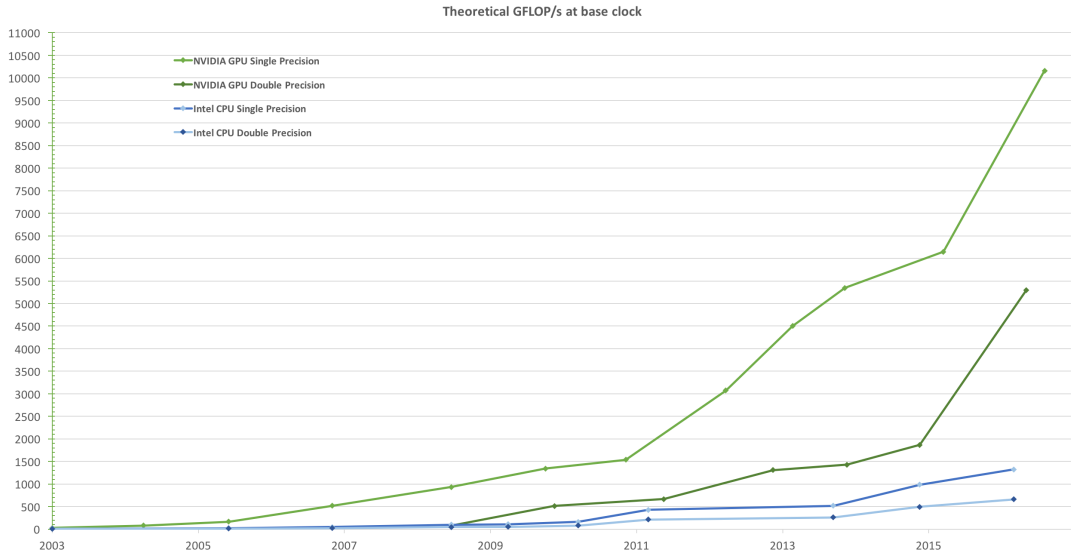


Figure 5.1.2. The theoretical maximum number of single and double precision floating point operations per second for comparable CPU and GPU chips, running at base clock speeds, since 2003. As we can see, initially they were operating within the same order of magnitude, but in recent years GPUs have gained a huge increase in performance. Today the fastest GPU has approximately 10 times more computational throughput versus a comparable CPU. Reprint from *CUDA C Programming Guide* by NVIDIA [67].

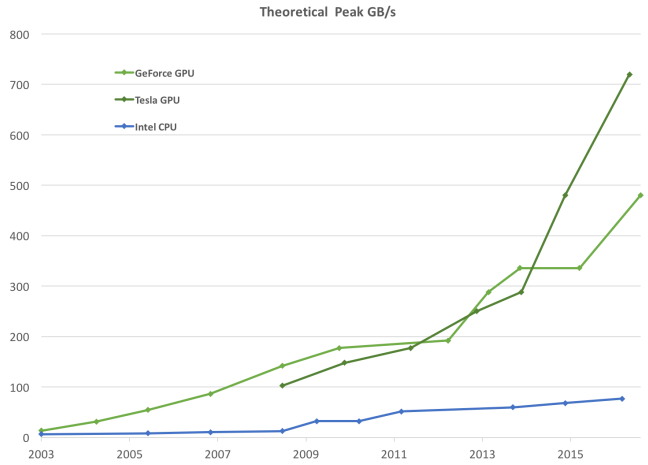


Figure 5.1.3. Theoretical peak processing power in terms of data processing bandwidth measured in bytes, comparing consumer GPU, workstation GPU, and comparable CPU chips, since 2003. Both categories of GPU have since their respective introductions, consistently had more bandwidth than CPUs. It is worth noting that for a while the consumer GPUs held the lead, with workstation GPUs only taking the lead in the last few years. This can be attributed primarily to the reduced bandwidth of the error-correcting code (ECC) memory used in the workstation GPUs. However, in the last few years this overhead has been resolved. Reprint from *CUDA C Programming Guide* by NVIDIA [67].

characteristics for both types of compute devices. Implementing an algorithm on a GPU will not trivially lead to a ten-fold speedup when compared to a CPU implementation — GPU parallelization is not a panacea. Instead it is a highly specialized tool that can work remarkably well when applied correctly to appropriate problems.

As we know from earlier, reading from DRAM, the so called device memory, is relatively slow. It can typically take more than 100 clock cycles on a CPU to read from DRAM, and this is why a CPU has a lot of cache memory. Regardless, we have to use DRAM to work with data, and in the realm of CPU-bound computations, having access to large amounts of device memory is common. The amount of memory can usually be expanded easily, and affordably. CPU-bound DRAM can be said to be a commodity resource. For GPUs however, the story is different — memory on commercially available GPUs can not be expanded. Thus we are limited to the amount of DRAM shipped on the device. The current state of the art GPU only comes with 16 GB of DRAM [69], and this means memory is a relatively scarce resource. One key take-away here is noting that a parallel algorithm executing on a CPU, with a companion dataset that fits entirely in CPU-bound memory, might not easily be ported to a GPU since the dataset might not fit in memory. If we are lucky, the algorithm can be batch processed. Otherwise we might have to rewrite parts of the algorithm, or perhaps approach the problem in an entirely different manner.

It is important to mention at this point that a GPU is dependent upon a CPU for getting access to data, receiving instructions to execute GPU programs, as well as making its computation results available to non-GPU-bound programs. All of this communication happens via the PCI Express (PCIe) bus. In other words, any communication between CPU and GPU is limited by the transfer speed/bandwidth of the PCIe bus which connects them. From the current hardware specification the maximum theoretical transfer speed of the PCIe bus is approximately 31.5 GB/s [74], but we conjecture that this is unrealistic. The key points from this insight is to realize that

a) Some level of synchronization between CPU and GPU is always required.
b) The CPU and GPU can operate in parallel with respect to each other, albeit limited by the previous point.
c) Since communication is limited by the speed of the PCIe bus, we need to ensure the GPU is given as much work as possible to fully saturate it with work, in order to reduce the amount of synchronization and data transfer between the GPU and CPU as much as possible.

Thus, if we want to completely utilize all the compute performance afforded by a GPU, we must keep these points in mind when developing/implementing algorithms for GPU acceleration.

Copying data from the CPU memory to GPU memory is handled by units called copy engines, which are located on the GPU. Modern GPUs have up to two copy engines, and

they work independently of the SMs. This implies that both copying data and performing computations can be performed in tandem. A GPU with two copy engines is capable of fully saturating the PCIe bus, using one engine per direction of transfer [75].

5.1.2 Compute infrastructure

GPU kernels and the thread hierarchy

The basic building block of GPU programs is referred to as a *kernel*. Note that this type of kernel is not related to the kernel functions introduced in Chapter 4. From this point on, the kernel term will refer to a GPU kernel unless otherwise stated. GPU kernels are more akin to kernel matrices in the context of digital image processing etc. There, kernels are commonly defined in relation to filtering based on on the convolution operation. Executing a GPU kernel can be thought of as convolving a kernel matrix which represents a filter, across a matrix which represents an image.

Let us briefly look at an example where we compute a grayscale version of a RGB color image. Assume that the three color channels of the original $m \times n$ image are given as three vectors. Storing the matrices as vectors is a trick to unroll the convolution operation into a single loop. The filter can then be implemented as follows

```
void convert_to_grayscale(float* gray, float* r, float* g, float* b)
{
  for (auto i = 0; i < m * n; ++i) {
    gray[i] = 0.2126 * r[i] + 0.7152 * g[i] + 0.0722 * b[i];
  }
}
```

The logic here is trivial; loop over all m rows and n columns of the image. Since the matrices have been unwrapped, we instead loop over all components of the vectors. Finally, the grayscale image is computed as a weighted sum of all three color components. The equivalent GPU kernel turns out to be quite similar;

```
__global__
void convert_to_grayscale(float* gray, float* r, float* g, float* b)
{
  auto i = threadIdx.x + n * blockIdx.x;
  gray[i] = 0.2126 * r[i] + 0.7152 * g[i] + 0.0722 * b[i];
}
```

As we can see, the loop is missing, and that is because GPU kernels are executed differently compared to regular functions. Highly simplified, when executing a kernel we have to specify how many times we want the GPU to invoke the kernel. In our simple

example, we want the kernel to be executed $m \times n$ times. Then, instead of having a loop that produces the vector index, we calculate it based on the index of the current execution. Think of this in terms of the GPU performing the iteration on our behalf.

Executing a GPU kernel cannot be considered trivial in the same sense as invoking a conventional CPU function, which usually only needs to be given references to (or copies of) the required data. When invoking a kernel we also need to declare how we want the thread execution hierarchy to be organized.

The smallest unit of work is a thread, which is organized into something called a *warp*. Warps consists of 32 threads [67]. It is important to note that warps are implemented as hardware threads on the GPU, and thus the smallest unit of execution is always 32 threads.

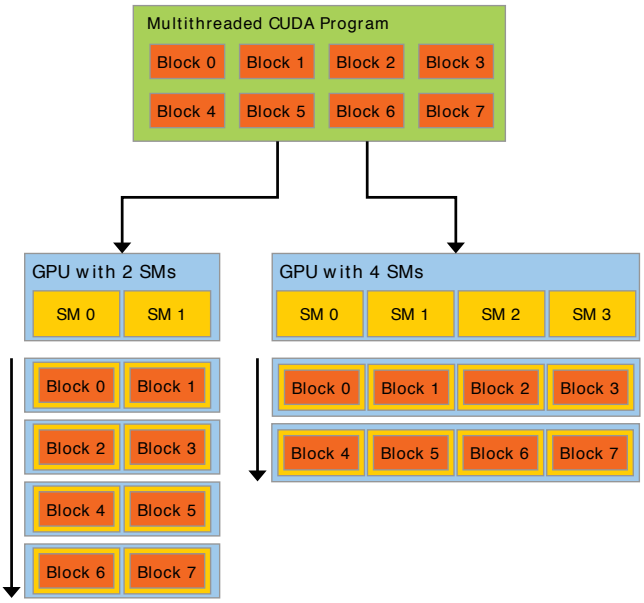


Figure 5.1.4. Example outlining how a kernel split in 8 thread blocks is scheduled for execution on a GPU with two SMs versus a GPU with four SMs. Note that the actual order of execution is only meant to be illustrative, and will likely differ in an actual simulation. Reprint from *CUDA C Programming Guide* by NVIDIA [67].

Warps are organized into something called blocks, which are structured as either one-, two-, or three-dimensional groups of threads that consists of at most 32 warps. The implication of this is that the "volume" of a thread block can not exceed a total of 1024 threads. Therefore we can for example define a block as a vector with a length of 1024, or as a matrix with dimensions such as 32×32, or 64×16. Thinking in terms of multi-dimensional matrices, or tensors, we can for instance define a block as $16 \times 16 \times 2$ threads. Usually we opt for a block structure that maps directly to the organization of the underlying data, and we also stress that we do not have to define thread blocks

such that they reach the maximum of 1024 threads. The important part is to choose an organization that makes sense for an algorithm, while attempting to maximize the occupancy on the GPU. The term occupancy can be in simple terms be explained as the ratio of active SMs in comparison to the total number of SMs on the GPU. In essence, it measures how much of the GPU is utilized. For details, please see Wilt [75]. We will consider the consequences of not optimizing the thread block dimensions at a later stage, but the curious reader can skip ahead to Figure 5.2.1 on page 76 for a sneak-peak.

Just as threads/warps are organized into blocks, blocks are organized as grids. Following the same concept as blocks, grids are organized into either one, two, or three dimensions. The maximum number of blocks per grid is 2^{20} [67]. In Figure 5.1.4 we can see a two-dimensional example of the block and grid hierarchy, and also how blocks are scheduled for execution across multiple SMs. The number of blocks executing simultaneously is affected by the number of SMs available on the GPU. Additionally, a block is not scheduled for execution until the entire block can fit on the device. That means that once a block is executing, we can rest assured that all threads that comprise the block will be executing together, sharing memory. This is very important since this means we then get access to intra-block synchronization which, as we shall see later, is necessary for certain algorithms to operate efficiently since they can cooperate and share the workload to some extent. At this point is makes sense to mention that there is no out of the box support for inter-block synchronization. This means that while we can easily synchronize work within a thread block, there is no equivalent way of synchronizing work between multiple blocks.

Keeping in mind this thread hierarchy we have outlined, it should not be too difficult to understand why having knowledge of these details can be of crucial importance when implementing a low-level GPU program. If we do not configure an appropriate hierarchy of threads and blocks, we run the risk of large portions of the compute potential of the GPU going to waste. Whereas an optimal thread configuration can help ensure the GPU is wholly saturated with work.

Memory hierarchy

Memory on the GPU can also be divided into a hierarchy, and we can see a simple depiction of this in Figure 5.1.5. The smallest unit of work, a GPU thread, has access to a finite amount of private, thread-local memory. Since thread-local memory is stored in device memory, the amount of available local memory is primarily bounded by the capabilities of the device. Since the local memory is persisted in the device memory, it obviously suffers from the same latency and bandwidth issues [67]. This is to some extent overcome by the fact that local memory access is coalesced, such that adjacent threads access adjacent, coalesced memory which can be cached, thus reducing latency considerably. The term coalesced memory in this context refers to how CUDA devices

coalesces global memory access, meaning it merges loads and stores of memory, based on how threads are grouped into warps. This is done to reduce device memory bandwidth and latency. Since memory coalescence is a difficult topic, and the conditions under which it occurs on CUDA devices is even more difficult, we refer the interested reader to Wilt [75], and NVIDIA [67].

Thread blocks have access to shared memory which is shared among all threads in the block. The amount of shared memory available to a SM depends both on the hardware on the device, as well as runtime configuration available via the CUDA API. The relevant API functionality makes it possible to adjust the ratio of memory used for L1 cache memory, and shared memory. Typical default values for shared memory is 48 KB or 64 KB, and L1 cache memory usually defaults to 16 KB or 24 KB [67].

Once we get to the grid level of hierarchy, the only memory accessible is so called global memory. This memory behaves slightly differently performance-wise depending upon whether it is determined by the kernel compiler, or explicitly marked, as read-only. If the memory is considered to be read-only, it may end up in a read-only cache which will greatly improve access times. All global memory access is cached in L2 cache memory [67]. Thus, we should always strive for access memory in a cache-optimal manner to reduce the cost of accessing device memory. We will however not cover in any detail how this can be achieved, since that topic is outside the scope of this thesis, and is highly dependent upon actual algorithms being implemented.

There are other types of memory available, e.g. texture and surface memory, as well as varied applications for the different types of memory. Additionally, modern GPUs provide functionality which makes it possible to access CPU-bound memory in various ways. Covering these concepts is beyond the scope of this thesis; for the further details on these topics, we refer the interested reader to Wilt [75], and NVIDIA [67].

CUDA compute capability

As we have alluded to earlier, NVIDIA GPUs are classified by something called compute capability [67]. This term can briefly be summarized as describing the interplay between the hardware and software. The actual chip design of specific NVIDIA GPU models adhere to the specifications for a specific compute capability specification. These hardware-level specifications naturally affect our implementations due to the tight bond between hardware and software. These capabilities are to some extent abstracted by the CUDA application programming interface (API), but we also need to take them into consideration when writing kernels. The reason being that we have to take the compute capability into consideration when compiling kernels. The net result of this is then deciding what compute capabilities we need, and perhaps which would be nice to have. Some capabilities might not be necessary for implementing a certain algorithm,

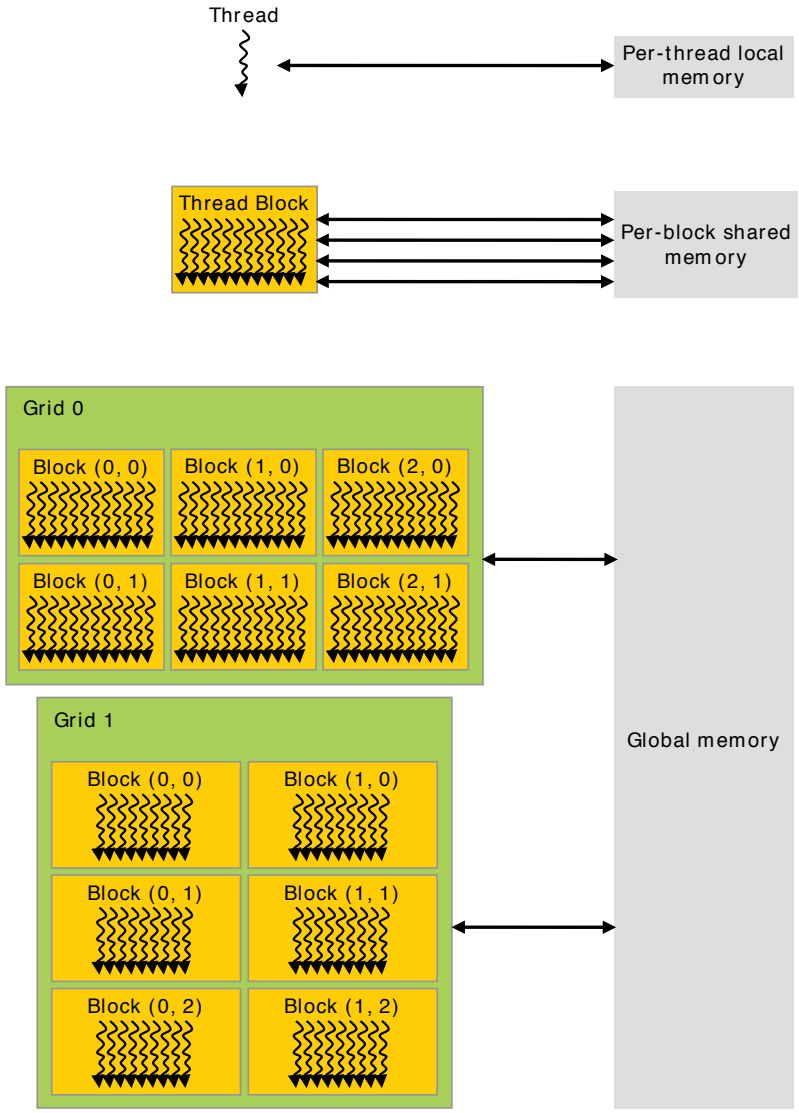


Figure 5.1.5. Somewhat simplified depiction of the CUDA memory hierarchy. Reprint from *CUDA C Programming Guide* by NVIDIA [67].

but if used can lead to improved computational performance. The compute capability we target naturally affects the models of GPUs that are capable of running the compiled kernel, since the capability is built into the GPU hardware. As we shall see later, this multi-targeting is not too difficult to implement, but we note that it can be difficult to test and performance benchmark since using this approach typically translates to having to test the implementation across the supported GPU models. Therefore we recommend targeting a single compute capability version, specifically the highest capability afforded by the GPUs on which we intend to implement our algorithms. This will lead to a less complicated implementation effort, and may result in better performance since newer compute specifications can lead to better global memory access, etc.

Compiling GPU programs

Even though we do not always have to write and compile GPU kernel code ourselves in order to leverage GPUs, it is important to consider how it is typically done. In particular what tools do we have at our disposal, and in simple terms what happens when we compile a GPU kernel.

The CUDA development toolkit naturally comes bundled with all the necessary tools in order to compile kernel code. The kernel code itself is written in a specialized C++ grammar, and the majority of the CUDA API is written in C. Files containing CUDA GPU kernels are usually affixed with the .cu extension. If a code file contains both CPU and GPU code, referred to as host and device code respectively, we typically refer to the combined form as mixed-mode code. The C++ features supported in kernel code consists of a very minimal subset of the full feature set available in modern C++ code [67]. And since the underlying API is written in pure C, we can not use features of modern C++ which prevents e.g. memory leaks, such as compiler-guaranteed resource allocation and cleanup, and so forth [76, 77, 78].

During compilation the code is split into host and device code. Host code can be written as pure C++, while the device code is governed by the aforementioned limitations. The original code is referred to as mixed-mode code. After being preprocessed the host code is sent to a common compiler such as GCC or Clang, depending upon configuration and operating system. The device code which consists primarily of kernel code, is sent through a specialized CUDA compiler. Traditionally this CUDA compiler has been the nvcc compiler which is provided by NVIDIA as part of the CUDA development kit. However, the open source Clang/LLVM compiler has recently received support for compiling CUDA code. The primary advantage of using this compiler is that is means being able to use the same compiler for the entire code base. Furthermore, it seems one of the goals of this effort is to provide nearly full support for modern C++. Note that this it is still bounded by the limitations enforced by the GPU architecture as mentioned earlier. In other words, certain features of C++ simply cannot be used on a

GPU, regardless of compiler infrastructure. For further details, we refer the interested reader to Wu et al. [79] which is a precursor to Clang/LLVM getting CUDA support.

We will not go into further details on how either of the mentioned compilers operate, but both end up generating an intermediate representation referred to as PTX (parallel thread execution) assembly. This PTX code is compiled into PTX binary that is either shipped as-is along with the rest of the program, or sent through another tool which called *ptxas*. This tool generates executable GPU microcode instructions for a specific GPU, based on the PTX [80]. The former approach is sometimes referred to as *online compilation* since the result is just-in-time compilation at the time of GPU program execution. Logically, the latter approach is referred to as *offline compilation* [75]. We mentioned in a previous section that GPUs have foregone forward and backward compatibility in their instruction set. Simplistically, it is because of this level of indirection afforded by the PTX that this design choice is feasible. All that is necessary to ensure a kernel runs on a future GPU is for the NVIDIA to ship an updated ptxas tool as part of the GPU driver that is capable of translating an existing PTX binary into executable microcode for the new GPU. An example of a typical compilation pipeline of a mixed-mode file can be seen in Figure 5.1.6.

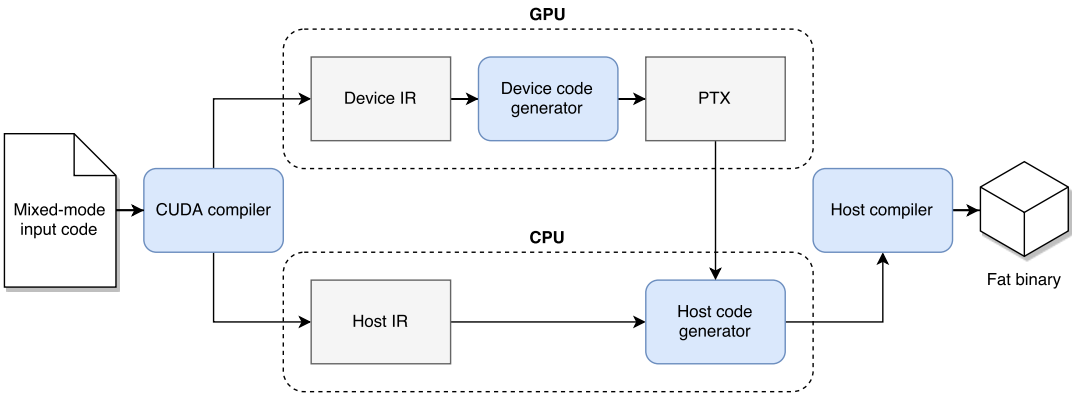


Figure 5.1.6. Simplified overview of the compilation pipeline employed by CUDA compilers when compiling GPU programs. The mixed-mode input is separated into device (GPU) and host (CPU) code, which is then usually processed by separate workflows, before being compiled into a single "fat binary" which can be executed by a CPU just like any other program.

Before we move on, be aware that there exists other alternative approaches to writing low-level GPU code, such as OpenCL [81], and HCC [82, 83]. Although the latter seems to work more or less as a code generator, sometimes referred to as a transpiler. Briefly put, it generates GPU code that is then subsequently compiled by an applicable compiler. Such a layer of indirection can lead to sub-optimal code compared to a tuned, hand-written GPU kernel. This is an inevitable effect of having to target multiple GPU architectures which differ at both the hardware and software level.

5.2. Computations on GPU

Performing computations on a GPU can be achieved in numerous ways, from writing custom kernels to using higher-order abstractions in languages such as Python. There also exists countless libraries that are designed to simplify the task of interacting with the GPU. Since these are typically well documented, and because we are primarily interested in custom CUDA kernels, we will not discuss any GPU libraries in this thesis. The interested reader is encouraged to have a look at the *Thrust* library (which is bundled with the CUDA toolkit), and PyCUDA. As part of our experiments later in this thesis, we will be exploiting the MAGMA library, in addition to a custom C++ header-only library[1].

Typical steps involved in executing a GPU-accelerated program can be summarized as follows;

1. Initialize device, and allocate device memory.
2. Copy memory from host to device.
3. Launch kernel(s).
4. Copy memory from device to host.
5. Release resources.

Note that most of these steps are asynchronous, which means that when we invoke some CUDA API function, execution will typically happen *out-of-process*, and control is immediately returned to the caller. Additionally, almost all of these API calls can result in errors — e.g. memory allocation might fail. Combining this with the asynchrony of interacting with the GPU, requires special API calls to check the error state of the GPU.

Let us now look at a simple GPU program that uses standard CUDA API functions, and follows the steps outlined above (sans the error checking). We remark that we will not attempt to give detailed explanations of every aspect. The intent is to show what is involved in writing custom GPU programs. Imagine we have a CUDA kernel that simply sets every element in a given vector to some given constant value;

```
__global__ void init_const(float* some_vector, float some_constant)
{
  const auto idx = threadIdx.x + blockDim.x * blockIdx.x;
  some_vector[idx] += some_constant;
}
```

[1] *Cudalicious* is a C++ header-only library we developed during the course of implementing the various experiments in this thesis. The library is open source, and can be found at `https://github.com/thomasjo/cudalicious`.

Looking at the kernel, we see that it needs be given two things, the vector we want to initialize, and a scalar that we want to initialize the vector with. This means that we need to allocate device memory which is sufficiently large to hold our vector of floats, and can be done as follows,

```
float* dev_vector;
cudaMalloc(&dev_vector, sizeof(float) * 32);
```

The code allocates enough memory on the device to hold a vector of length 32. Note that when we pass simple types, such as the single constant value we want to hand to the kernel, memory is automatically allocated on our behalf. This code handles step 1 and 2. Launching the kernel can be done as follows

```
constexpr auto pi = 3.14159f;
init_const<<<1, 32>>>(dev_vector, pi);
```

Without going into details, the "chevron" syntax specifies the grid and block hierarchy we have discussed earlier. In this case we want to launch a single block of 32 threads, one thread per vector component. Please note that kernel launches are asynchronous, so unless a blocking call is made after launching the kernel, the CPU part of the program will finish executing before the GPU has finished processing. Notably, copying memory is normally a blocking call. In our example, copying the vector from device memory to host memory can be achieved by

```
float result[32];
cudaMemcpy(&result, dev_vector, sizeof(float) * 32, cudaMemcpyDeviceToHost);
```

Finally, we need to clean up device resources, which in our case can be done as

```
cudaFree(dev_vector);
```

If we were to execute this code, we would end up with a 32-dimensional vector with every component initialized with our specified value.

The example given is trivial, and unlikely to be very useful, but illustrates in some sense what is typically involved in writing custom CUDA programs. To get a sense of what might be involved when writing a high-performance CUDA kernel, we refer to Listing 5.2.1. The code presented there performs, in a very efficient manner, what is referred to as an *add reduce* operation. The end result is a summation of all the components in the vector, yielding a single scalar. Additionally, working with GPUs might also involve concepts such as unified address space, streams, multi-GPU setups, and so forth.

Listing 5.2.1. Example showcasing a very efficient CUDA kernel implementation of *add reduce*.

```
constexpr auto STRIDE = 4u;

template<typename T>
__global__ void add_reduce(T* output, const T* input, const size_t num)
{
  auto sum = T{0};
  auto start_idx = (threadIdx.x + blockIdx.x * blockDim.x) * STRIDE;

  #pragma unroll
  for (auto idx = start_idx; idx < start_idx + STRIDE && idx < num; ++idx) {
    sum += __ldg(input + idx);
  }

  #pragma unroll
  for (auto delta = warpSize / 2; delta >= 1; delta /= 2) {
    sum += __shfl_down(sum, delta);
  }

  __shared__ T shared_sum;
  if (threadIdx.x == 0) {
    shared_sum = T{0};
  }

  __syncthreads();
  if (threadIdx.x % warpSize == 0) {
    atomicAdd(&shared_sum, sum);
  }

  __syncthreads();
  if (threadIdx.x == 0) {
    atomicAdd(output, shared_sum);
  }
}
```

5.2.1 Kernel execution parameters

The parameters used when launching a GPU kernel often play a critical role in ensuring optimal temporal performance of GPU computations, since they both affect and are affected by notions such as occupancy, memory coalescence, etc. To maximize the occupancy of a kernel we need to ensure we keep as many threads as possible busy all the time. This is affected by the number of SMs on the GPU, since that will limit the number of concurrent blocks that can be executed. We need to ensure that each block will saturate the GPU with work.

Consider for a moment a kernel that efficiently computes matrix-matrix products as seen in Listing 5.2.2. The result of launching the kernel with different block and grid dimensions on matrices of varying sizes can be seen in Figure 5.2.1. As we can see, there is a strong correlation between the number of threads per block and the runtime characteristics of the kernel. This effect is very much affected by the number of SMs on the GPU. The GPU used for the benchmark only had 2 SMs. The implication of a block size of 16 threads on such a GPU means running two blocks, each consisting of a single warp. Depending upon our kernel implementation this might also mean we only utilize half of each warp. That will end up having a detrimental effect on the performance since a large portion of the GPU will not be utilized.

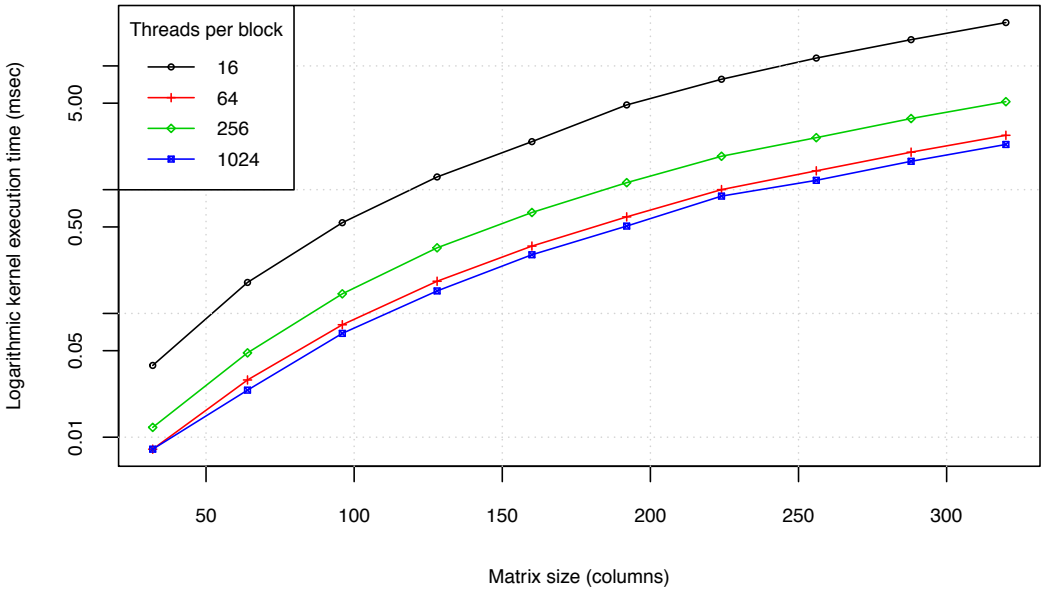


Figure 5.2.1. The plotted lines depict the kernel execution time for matrices of varying sizes using different block sizes. The correlation is linear, as expected.

Listing 5.2.2. Kernel that computes matrix-matrix product using blocked sub-matrix multiplication.

```
template<int BLOCK_SIZE>
__global__ void matrix_mul(float *c, float *a, float *b, int dim_a, int dim_b)
{
  // Block indices.
  const auto bx = blockIdx.x;
  const auto by = blockIdx.y;

  // Thread indices.
  const auto tx = threadIdx.x;
  const auto ty = threadIdx.y;

  // Indices and strides of sub-matrices processed by thread block.
  const auto a_begin = dim_a * BLOCK_SIZE * by;
  const auto a_end = a_begin + dim_a - 1;
  const auto a_step = BLOCK_SIZE;
  const auto b_begin = BLOCK_SIZE * bx;
  const auto b_step = BLOCK_SIZE * dim_b;

  auto c_sub = 0.f;  // Stores computed sub-matrix coeffcient

  // Loop over all the sub-matrices required to compute blocked sub-matrix.
  for (auto a_idx = a_begin, b_idx = b_begin; a_idx <= a_end;
       a_idx += a_step, b_idx += b_step) {
    // Shared memory arrays used to store the sub-matrices of a and b.
    __shared__ float a_sub[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float b_sub[BLOCK_SIZE][BLOCK_SIZE];

    // Load the sub-matrices from device memory to shared memory;
    // each thread loads one element of each matrix.
    a_sub[ty][tx] = a[a_idx + dim_a * ty + tx];
    b_sub[ty][tx] = b[b_idx + dim_b * ty + tx];

    __syncthreads();  // Ensure the matrices are loaded

    // Multiply the two matrices together;
    // each thread computes one element of the block sub-matrix.
    #pragma unroll
    for (auto k = 0; k < BLOCK_SIZE; ++k) {
      c_sub += a_sub[ty][k] * b_sub[k][tx];
    }

    __syncthreads();  // Ensure all threads have finished computing sub-matrix
  }

  // Write the block sub-matrix to device memory; each thread writes one element.
  const auto c_idx = dim_b * BLOCK_SIZE * by + BLOCK_SIZE * bx;
  c[c_idx + dim_b * ty + tx] = c_sub;
}
```

Part III.

Method & analysis

Chapter 6.

Spectral methods on GPU

In this chapter we will perform some experiments in order to study and validate our proposal with regards to the benefits of performing computations on GPU, in particular spectral methods. After some preliminary benchmark experiments with eigensolvers on both CPU and GPU, we will experiment with a KECA implementation on GPU. The implementation will be leveraging the MAGMA library, in addition to a CUDA kernel we have designed that computes the RBF kernel matrix $\mathbf{K}$ needed as part of the KECA algorithm.

Experiment infrastructure

Our final experiment results were produced by running experiments on a computer designed for experimenting with deep learning [84, 85], methods and was equipped with a NVIDIA GeForce GTX TITAN X GPU, and an Intel Xeon E5-2630 CPU. This GPU model has 12 GB of memory, and a total of 3072 CUDA cores. The CPU has 8 cores, where each core supports 2 threads, which yields a total of 16 simultaneous threads. This setup is referred to as the *primary setup*. The majority of our initials experiments were done on a MacBook Pro (mid-2012) with an Intel i7 CPU, and an NVIDIA GeForce GT 650M GPU with 1 GB of memory. We will refer to this setup as the *secondary setup*. While the CPU in this setup only has 8 threads spread across 4 cores, the most noteworthy difference between the two setups is the fact that the GPU only has 384 CUDA cores. Moreover, the GPUs are from different architecture generations, and therefore have different CUDA compute capabilities. At this point we want to stress that not having early access to more modern GPU hardware ultimately led to multiple issues which had a significant impact on the experimental aspect of the thesis. Therefore, heed our warning; ensure access to modern GPU infrastructure when performing GPU-related experiments.

When it comes to implementation details of the experiments, all experiments were

implemented using modern C++ and compiled with both the Clang/LLVM compiler provided via Xcode 8.0 on macOS, and the GCC 4.8 compiler on Ubuntu 14.04 LTS. Both CUDA toolkit 7.5 and 8.0 were tested, but neither yielded a significant performance difference in any of the experiments. Since CUDA hardware currently yields better performance when using 32-bit floating point operations, all experiments were performed on 32-bit data.

6.1. Experiments with eigendecomposition on GPU

In order to gauge the potential performance impact that using GPU computation can have on spectral methods, we begun our journey by experimenting solely with eigenvalue solvers on GPU versus CPU. The goal, to uncover both benefits and pitfalls of using GPU-based solvers. For this purpose, we opted for using the MAGMA library, which is pitched as a hybrid implementation that leverages both multi-CPU and multi-GPU infrastructure. The library is designed to be similar enough to LAPACK [86] to make it as easy as possible to transition to MAGMA. Since the objective was to show the benefit of using GPUs, we decided to compare the GPU accelerated MAGMA library against an equivalent CPU-based library. We chose the Eigen library, which is a C++ header-only linear algebra library capable of leveraging BLAS and LAPACK routines. The use of BLAS/LAPACK should in theory make it capable of computing eigenvalue decompositions in a highly optimized manner.

6.1.1 Comparison of CPU and GPU performance

To compare the performance of the CPU and GPU eigenvalue solvers, we measured the execution time of the methods using randomly generated, $n \times n$ positive semi-definite matrices. Using a randomized scheme should help even out minor numerical differences across the different implementations, as well as differences in the CPU and GPU hardware. The size of the matrices started at 32×32, and then increasing up to 2048×2048 in increments of 32 along each dimensions. Additionally, to counter semi-random effects that might reduce CPU or GPU performance on the machine during execution, we simulated with random 10 matrices per increment. In other words, we generated 10 different $n \times n$ matrices. By doing this we can find the average computation time of each implementation, per matrix increment. Since CUDA hardware currently performs best with 32-bit data, all randomly generated data was 32-bit. Furthermore, to ensure each of the implementations being tested produced correct computations (in an approximate, numerical sense), we compared the 10 most dominant eigenvalues produced in some of our experiment iterations, against the equivalent eigenvalues produced by a stable and proven, proprietary software solution. The results of that validation

process are not reproduced in this thesis, since the intent was only to validate the numerical stability of the library implementations, and verify the correctness of our overall approach.

For the the purely CPU-bound implementation built on the Eigen library, we used their eigenvalue solver made for symmetric (self-adjoint) matrices. According to their documentation, this is the fastest and most stable of their solvers. During our experiments we found it to be numerically stable, and efficient. On the GPU side, we used two slightly different implementations. The first uses a hybrid CPU/GPU implementation provided by MAGMA, which uses CPU computations for matrices smaller than some defined data dimensionality threshold. When the size of the problem/matrix exceeds the threshold, computations are transitioned to GPU. The second MAGMA implementation does not have this heuristic, and instead runs mostly on the GPU. We use the term mostly because the MAGMA library uses a hybrid CPU/GPU approach. Roughly speaking, this means they aim to fully leverage the benefits of both compute architectures. Note that both of the MAGMA implementations use the same underlying divide and conquer algorithm. The only major difference is seemingly whether the matrix size heuristic is employed. Using the non-heuristic version requires manually managing device memory, whereas the other handles this on our behalf.

During the implementation of our experiments, we initially tried using the QR iteration based methods provided by MAGMA, but we encountered several problems. Some of the problems are believed to be attributed to numerical instability. Additionally the MAGMA implementation requires several tuned parameters, and it is possible the problems might stem from an non-optimal parameter. The fact these parameters have to be given is a drawback of using the MAGMA implementation. Regardless, we ended up not running our experiments with a QR-based scheme, and instead opted for the divide-and-conquer approach. We also experienced severe problems while attempting to run the experiments on the secondary setup, but we are not sure what this can be attributed to. Whether problems were caused by an implementation detail in MAGMA, or possibly a problem at the hardware level. These exact problems luckily did not manifest themselves on the primary setup.

Results of the CPU versus GPU eigensolver experiment can be seen in Figure 6.1.1. First of all we observe that we have not reached the limit of the GPU during the simulation. This observation is based on the trend in the GPU graphs, which only increase by a small factor for each iteration. Even though the size of the matrices being decomposed increases as a factor of n^2, the execution time of the GPU implementations is not increasing quadratically. For the CPU implementation however, we can see that it is affected to a much greater extent by the increase in matrix size. The execution time for the CPU-based test seems almost quadratic. For small matrices, around 128×128, we can see the effect of the heuristic employed by MAGMA — from the clear jump in execution time we can clearly see where the implementation switches from a CPU to GPU computation scheme. Also note that it seems the results of the non-heuristic implementation is slightly

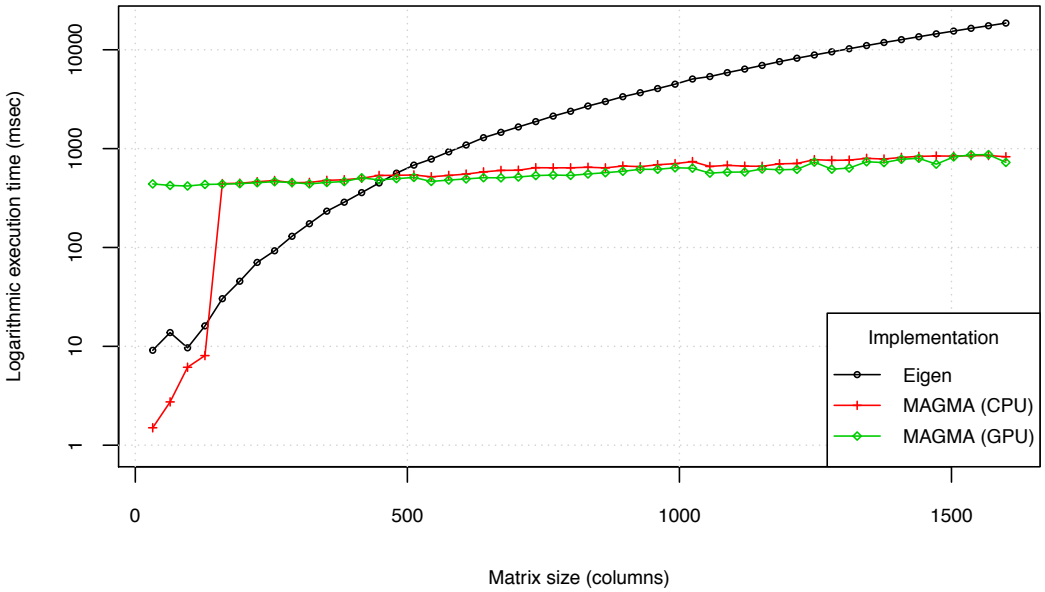


Figure 6.1.1. Comparison of the execution time of an entirely CPU-bound eigenvalue decomposition algorithm implemented in the Eigen library, compared to two GPU-accelerated algorithms found in the MAGMA library. The MAGMA implementations differ in that the one labeled "CPU" employs a heuristic to determine whether to run the computation on CPU or GPU.

better. We hypothesize that this is attributed to some overhead in the heuristic.

The second observation that can be made is that based on where the plotted lines intercept, we are able to determine approximately how large a matrix needs to be in order to get a computational benefit of using a GPU. Given our the hardware used in our primary setup, it looks like the limit is around 500×500. Comparing this result with the heuristic employed by the MAGMA implementation, we propose that the heuristic can be improved to account for modern GPUs, based on the results we have produced.

From the plotted results it can also be observed that what we are primarily measuring during the GPU executions is the inherent overhead of copying data to and from the GPU, and launching GPU kernels, etc. Likely there is some level of CPU-GPU synchronization taking place, which will also have its own associated overhead. Testing these hypotheses would require further experiments, and most likely a study of the actual implementation of the MAGMA library. Therefore, no such attempts were made, and we simply leave it as an interesting observation.

6.1.2 Testing performance bounds of the GPU implementation

Next, we ran an experiment to uncover some practical dimensionality bounds of matrices when only using the MAGMA library. The experiment was executed on the primary setup, and was essentially a repeat of the earlier experiment, sans the CPU-based implementation. We increased the previously set upper-bound on the matrix dimensions up to 32768×32768, in increments of 256. In terms of memory usage, the random matrices were approximately 4 GB in size at the upper limit. This was believed to be large enough to test the implementation both in terms of memory requirement and computational effort. We remark that since the execution time after some size increments can be measured in minutes, we did not have enough time to repeat the computations for each size increment multiple times. Hence, we are unable calculate mean execution times, as in the previous experiment.

In Figure 6.1.2 we can see the results of the simulation. Even though the plotted

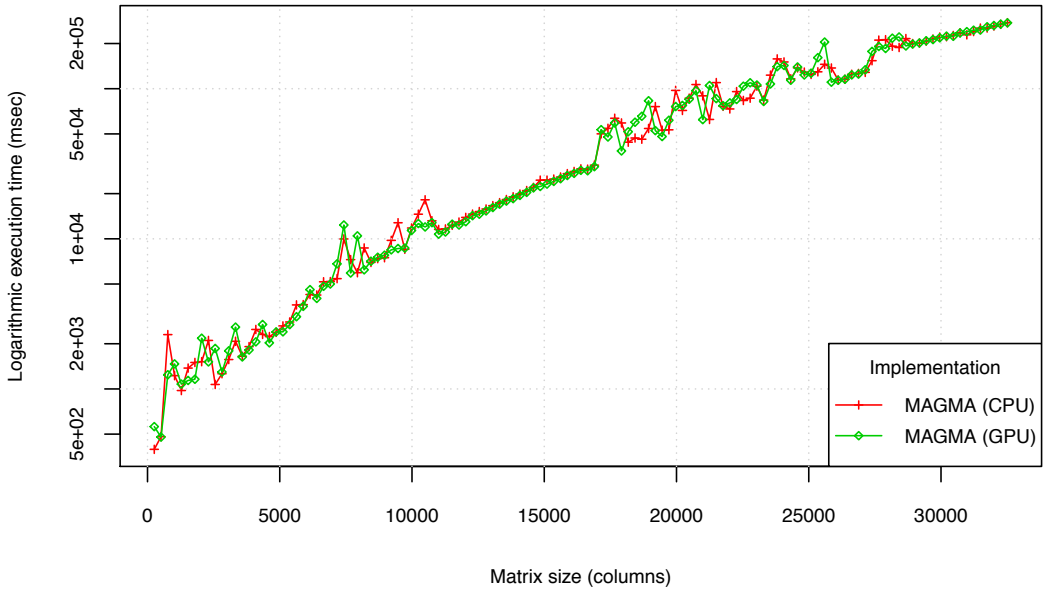


Figure 6.1.2. The plotted curves show the temporal performance of the two MAGMA library functions used in our experiments to perform eigenvalue decomposition. Both seem to have more or less identical performance, and scale roughly equivalent the Eigen implementation seen in Figure 6.1.1. The noise in the sample is thought the be caused by effects such as thermal throttling, or similar.

results are somewhat noisy, we can still see a clear trend. Once again, we see that the algorithms scale by some factor which seems to be approximately n^2. This result is expected since running computations on a GPU does not alter the complexity of the underlying algorithm, it will usually only change some constant factor. It is also worth

noting that with matrices of this scale, the temporal performance does not seem affected by whether we use the heuristic MAGMA implementation or not. In both cases the results are more or less overlapping. The noise in the data is thought to be caused by e.g. thermal throttling, which is managed at the hardware level, and out of our control. Repeating the simulation several times, and averaging, would almost surely yield less noisy data, as in our earlier experiment.

Before we move on, we would also briefly like to remark that during the final execution, using a 32768×32768 matrix, we experienced a runtime issue caused by illegal memory access. Our hypothesis is that the MAGMA algorithms we used need about three times more memory than the input matrix itself. Therefore, when running the execution with a matrix approximately 4 GB in size, the amount of memory needed by the algorithm exceeded what was available on the GPU, which was approximately 12 GB. This is something that must be taken into account when using the MAGMA implementation.

6.2. Implementing KECA on GPU

To illustrate the benefit of using GPU-accelerated eigenvalue solvers, we chose to implement the KECA algorithm in combination with a RBF kernel. Initially we implemented the algorithm using purely CPU-bound computations, using C++. This effort was necessary to understand what is involved in actually computing the kernel matrix, sorting and picking eigenvectors based on entropy, and so forth. Our initial implementation was based on the original implementation[1]. For our experiments we used the *Frey Face* dataset, since it is one of the datasets used in Jenssen [1], and it is sufficiently large to use for GPU benchmarks.

As part of our initial CPU-based implementation effort, we realized that the RBF kernel matrix computation would be a reasonable candidate for GPU-acceleration. We stress that our implementation is somewhat naive in the sense that it is not properly coalesced, and the manner in which we compute the pairwise distance measures can almost surely be implemented in a more optimal manner. Nonetheless, due to the massively parallel nature of GPU computations, our implementation is much faster than the CPU implementation. The comparison benchmark between both implementations can be seen in Figure 6.2.1. Both algorithms scale equivalently, but the GPU implementation is much faster because it is massively parallel. We hypothesize that the speedup will be even greater with a more efficient memory access pattern in the GPU kernel.

The eigendecomposition in our KECA implementation is performed by the previously discussed Eigen and MAGMA implementations. Therefore we will not benchmark them

[1]Implementation of KECA found at `http://ansatte.uit.no/robert.jenssen/software.html`.

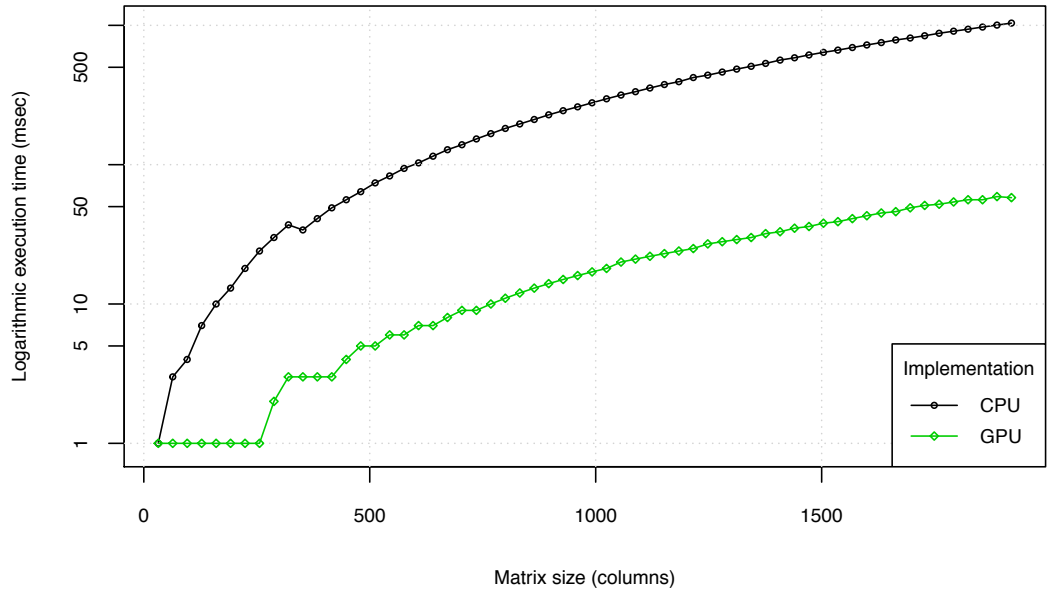


Figure 6.2.1. Results show the execution time of computing a RBF kernel matrix for a varying number of observations (columns).

again here, but rather we want to show the overall benefit of performing both the computation of the kernel and the eigenvalue decomposition on GPU. Note that there are still parts of the KECA implementation that are executing solely on a CPU, such as the entropy-based eigenvector selection criterion, and the final subspace projection. The comparison of running our KECA implementation entirely on CPU in comparison with mostly on GPU, can be seen in Figure 6.2.2. The speedup is significant — for the full-size dataset the difference is two orders of magnitude.

Finally, to verify the correctness of our implementations, we ran the original, plus both of our implementations on the full dataset, and plotted the result of projecting down to three dimensions. All three results can be seen in Figure 6.2.3. From the results we can see that all three results are equivalent. The slight difference in our CPU implementation is merely an effect of the direction of the computed eigenvectors.

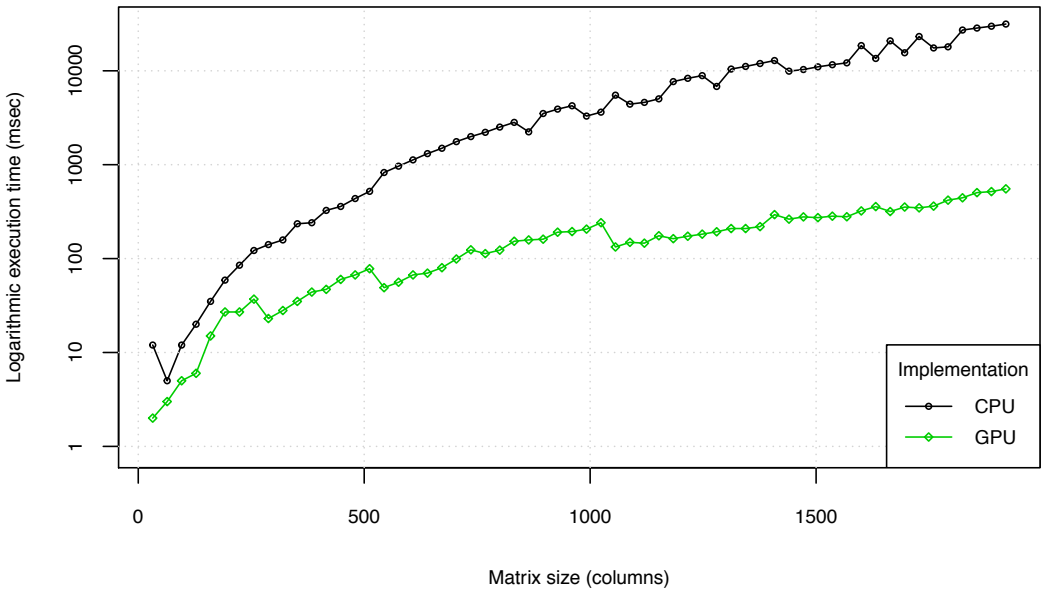


Figure 6.2.2. Benchmark results comparing running a full KECA implementation on CPU versus an implementation that mostly runs on a GPU. The speedup is approximately two orders of magnitude.

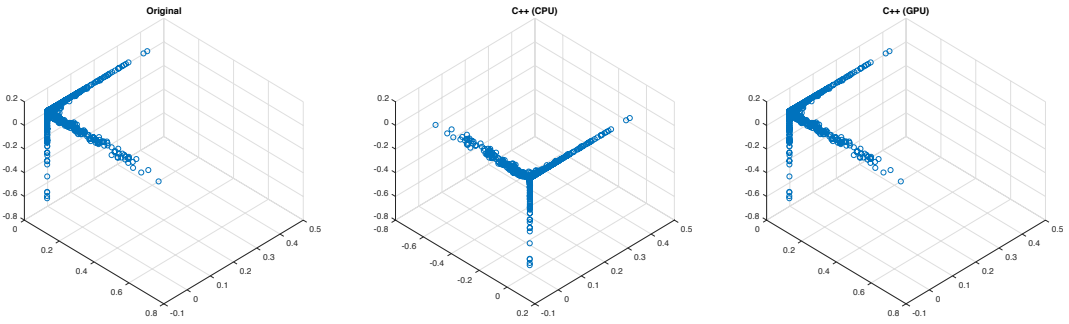


Figure 6.2.3. The three plots show the result of using KECA to project the *Frey Face* dataset down to three dimensions, where the result on the left was made by the original MATLAB implementation, and the other two using our C++ implementations.

Chapter 7.

QR algorithm on GPU

We have stated that it might be necessary to implement our own QR algorithm on GPU in order to efficiently implement the proposed permutations that improve the convergence properties of the algorithm. Therefore, in this chapter we will look at some preliminary experiments we performed in order to uncover what will be involved in such an effort, as well as discover problems. Our aim was to facilitate future efforts at implementing a GPU-based symmetric QR algorithm with permutations.

7.1. Preliminary investigation

As we saw in Chapter 6, using the QR algorithm provided by the MAGMA library has several drawbacks, and therefore we decided to investigate implementing our own QR algorithm. Instead of having to also implement the QR decomposition ourselves, we ended up using the QR decomposition implementations provided in the *cuSOLVER* library that is bundled with the CUDA toolkit. The implementation we used is based on a Householder reflection scheme that is based on the equivalent LAPACK implementation. It has one major drawback; it does not explicitly form the orthogonal matrix $\mathbf{Q}$, which is needed as part of each iteration in the QR algorithm. Instead what it yields, is a set of Householder vectors. As we know from Section 1.4.2, we can form Householder reflectors from the Householder vectors, and from the reflectors we can form the matrix $\mathbf{Q}$. The main problem is that due to computational complexity, we can not explicitly form the Householder reflectors and multiply them together to form $\mathbf{Q}$. Briefly summarized, for an $n \times n$ matrix, the Householder QR decomposition yields n Householder vectors. Therefore we would need to perform n outer products simply to form all the Householder reflectors. Furthermore, we would then need to multiply all n Householder reflectors together. The temporal and spatial complexity of this is non-trivial. Because of this, we instead propose that $\mathbf{Q}$ can be formed in an "implicit" manner; for each coefficient of the matrix $\mathbf{Q}$, perform all the computations necessary to form that single cofficient, as if it had been given as a result of performing all of the explicit computations.

7.2. The proposed GPU implementation

In order to to make our implementation easier to reason about during development, in particular investigating errors, we decided to limit the implementation to running entirely on a single thread block. Doing this allowed us to efficiently synchronize threads to make it easier to debug, etc. Therefore, the max dimensions of matrices is currently 32×32. Initial implementations were done on even smaller matrices, and scaled up to the maximum size towards the end of the experiment.

The preliminary implementation for forming the $\mathbf{Q}$ matrix on a GPU using an "implicit" reflector computation scheme is proposed,

```
__global__
void construct_q_matrix(float* q, float* source, float* tau, int n)
{
  auto row = blockDim.x * blockIdx.x + threadIdx.x;
  auto col = blockDim.y * blockIdx.y + threadIdx.y;
  auto idx = col * n + row;
  if (row >= n || col >= n) return;

  for (auto k = 0; k < n; ++k) {
    auto inner_product = 0.0f;

    for (auto i = 0; i < n; ++i) {
      auto row_coeff = q[i * n + row];
      auto col_coeff = get_reflector_coefficient(source, tau, n, k, i, col);

      inner_product += row_coeff * col_coeff;
    }

    __syncthreads();
    q[idx] = inner_product;
  }
}
```

For every k-th Householder vector given in `source`, an inner product between a row vector of $\mathbf{Q}$, and a column vector of $\mathbf{H}_k$ is computed. Which row and column is determined based on the coefficient being processed by the kernel thread. After all threads in the block have finished step k, each thread updates its coefficient in $\mathbf{Q}$. This necessity to synchronize threads is caused by the dependency in how all the reflectors are multiplied together. We remark that this will have to be handled in the future when implementing support for larger matrices. Additionally, we hypothesize the memory access scheme is not coalesced.

The necessary device-only function that computes the reflector coefficient for a given row and column position, and given reflector is proposed as follows,

```
__device__
float get_reflector_coefficient(
    float* source, float* tau, int n, int reflector_index, int row, int col)
{
  if (row < reflector_index || col < reflector_index) {
    return (row == col) ? 1.0f : 0.0f;
  }

  auto pre_coeff = (row == col) ? 1.0f : 0.0f;
  auto row_coeff = (row == reflector_index)
                       ? 1.0f : source[reflector_index * n + row];
  auto col_coeff = (col == reflector_index)
                       ? 1.0f : source[reflector_index * n + col];

  return pre_coeff - tau[reflector_index] * row_coeff * col_coeff;
}
```

For the complete, working implementation of the proposed, preliminary GPU-accelerated QR algorithm, please see Appendix B. We stress that while the implementation might look trivial in some sense, there was significant effort involved in "unrolling" the computations that construct $\mathbf{Q}$, and translating them into simple inner products.

When running the proposed implementation through the NVIDIA Visual Profiler, we discovered that approximately 72% of the computation effort is spent finding the QR decomposition, which is computed by the cuSOLVER library. About 15% is spent executing our $\mathbf{Q}$ matrix kernel, and the remaining time is spent on computing the matrix-matrix products. These numbers are promising, and at least give us an indication that our proposed kernel is working reasonably in terms of efficiency. The performance analysis also indicated that the occupancy is good. Indications are that the block size, register usage, and shared memory usage of the kernel will allow it to fully utilize all warps on the device. Some of the results produced by the analysis tool can be seen in Figure 7.2.1, Figure 7.2.3, and Figure 7.2.2.

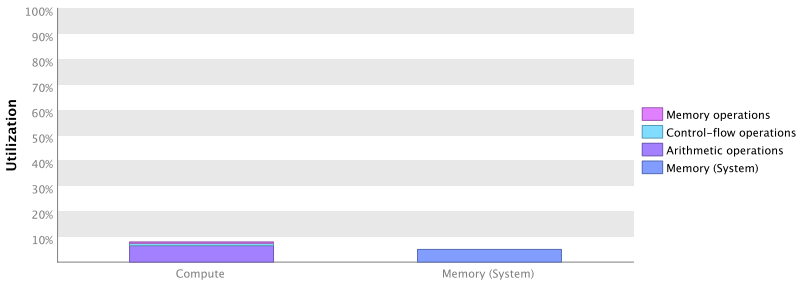


Figure 7.2.1. Utilization result for our $\mathbf{Q}$ matrix kernel. Generated by NVIDIA Visual Profiler.

The results as seen in the aforementioned figures are not particularily conclusive, due in part to the small block size and low memory usage, but they are nonetheless promising.

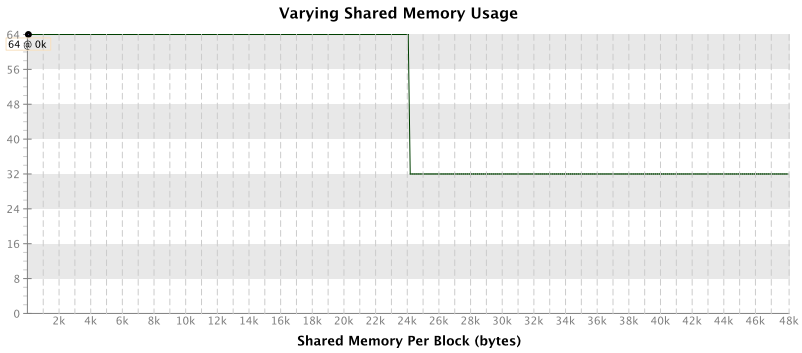


Figure 7.2.2. Shared memory usage for our **Q** matrix kernel. Generated by NVIDIA Visual Profiler.

To get more conclusive results, we would likely have to increase the block size, but that would mean going outside the intended scope of our experiments. Therefore we leave further investigation as a future effort.

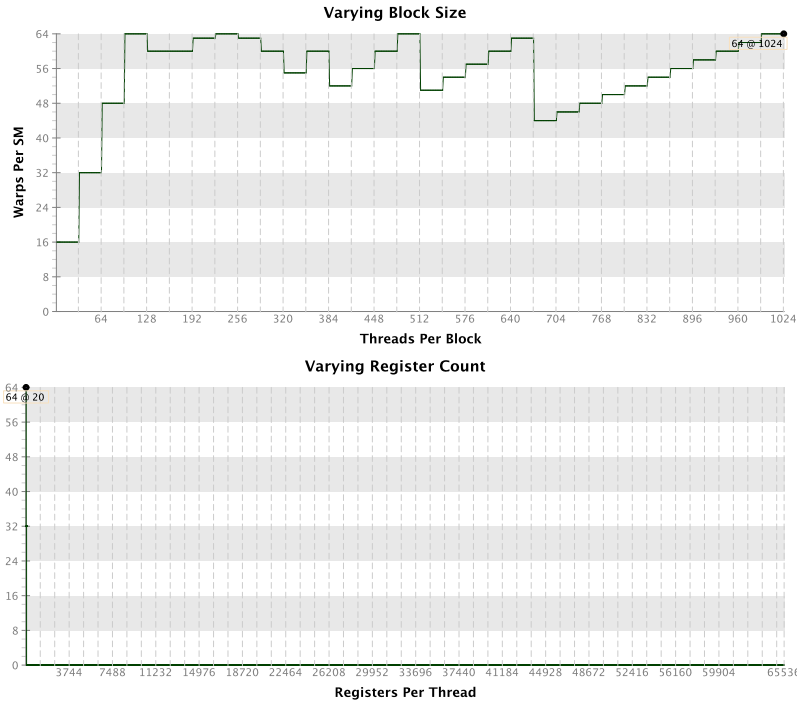


Figure 7.2.3. Occupancy result for our **Q** matrix kernel. Generated by NVIDIA Visual Profiler.

Part IV.

Discussion

Chapter 8.

Concluding remarks

In this thesis we have investigated both the benefits and pitfalls of performing eigendecomposition on GPU. Our investigation was performed as a series of small experiments designed to study important features of GPU-based eigensolvers. This primarily involved leveraging the MAGMA library, and specifically their divide-and-conquer implementation for computing an eigendecomposition. Our plan was originally to use the QR algorithm implemented in MAGMA for the eigendecomposition, but we found that it had problems with numerical stability, and therefor had to rely on the divide-and-conquer implementation instead. In order to show the proposed benefits of GPU computation, we compared against a CPU-bound eigensolver provided by the Eigen library. Our experiments showed that using a GPU-based eigensolver is two orders of magnitude faster for matrices larger than approximately 500×500, when running on an NVIDIA TITAN X GPU, and a Intel Xeon E5-2630 CPU. For small matrices, we have shown that using CPU computation is faster. We also performed an experiment that uncovered one of the limitations of the MAGMA implementation. What we found is that the employed implementation requires approximately three times the amount of memory needed to simply store the matrix, to perform its computations. This means that on a GPU with 12 GB of memory, the matrix cannot exceed 4 GB in size. Once the initial experiments were done, we then combined our GPU-based eigensolver approach with our own C++ implementation of KECA. As part of this effort we also proposed a GPU implementation to compute the RBF kernel, which is at the core of a machine learning algorithm called KECA. The computation of our proposed GPU kernel was shown to be about two orders of magnitude faster than the equivalent CPU version, since the estimation of the kernel function is the leading operation. The overall, proposed KECA implementation using both GPU-accelerated RBF kernel matrix computation, and GPU-accelerated eigensolver, was also shown to be two orders of magnitude faster than the equivalent CPU implementation.

Our second objective in this thesis was to perform a preliminary study into what is involved in implementing the symmetric QR algorithm with permutations on GPU. The main advantage of this QR algorithm is that the permutations improve the convergence

rate of the eigenvalue estimation by a factor of two. During our initial study we discovered a problem caused by the fact that the QR decomposition routine we used from the cuSOLVER library, does not explicitly form the orthogonal matrix $\mathbf{Q}$ that we require in the QR algorithm. Therefore we performed an experiment in which we implemented a GPU kernel that forms the required matrix from the Householder vectors given by the QR decomposition implementation. Instead of forming the matrix by constructing the Householder reflectors for the given vectors, then multiplying the constructed matrices to form the $\mathbf{Q}$ matrix, we instead proposed an "implicit" approach. Our proposed algorithm is based on constructing each coefficient in the final matrix by the series of computations that would yield that coefficient if we performed the explicit formation. We then combined this with an overall QR algorithm GPU implementation, and we showed that our proposed implementation works under the limitations for which it was designed. Our proposed implementation is intended to fascilitate a future implementation of the symmetric QR algorithm with permutations. Given our first-hand experience with numerical instability in the QR algorithm implemented in MAGMA, we remark that attempting to implement the permutation approach on GPU seems valuable, since it addresses numerical instability.

8.1. Future work

There are several additional paths that can be explored in future work. For instance, when it comes to implementing a permutation scheme, we have alluded to the fact it might be more efficient to perform a manner of implicit permutation. By this we mean that to perform permutation of one matrix, before multiplying said matrix with some other matrix, it is not strictly necessary to compute the permutation explicitly. It should instead be possible to implement a specialized matrix-matrix multiplication operation instead. Such that when multiplying two matrices together, we can "translate" row or column indices according to a permutation. If done efficiently, this type of implicit permutation should be faster than an explicit permutation. There is one caveat however; we typically employ block matrix multiplication when multiplying together large matrices. Combining block matrix multiplication with implicit permutation on GPU, while ensuring e.g. coalesced memory access, will undoubtedly be very challenging. However, given the benefits of improved convergence rates, this is worth exploring.

In the field of manifold learning, several algorithms are based on the computation of eigensystems. We briefly considered on such algorithm that is based on the computation of the gradient of the Hessian matrix for a small neighborhood around a point in the dataset, and this is done for all datapoints. Calculating the gradient in this case involves computing an eigendecomposition of the Hessian matrix. Importantly, these calculations are independent between all datapoints. Based on this, we performed a few experiments during the thesis, to attempt to accelerate the algorithm with GPU-based

eigendecomposition. However, the algorithm computes all the eigendecompositions inside of an iterative Runge-Kutta scheme. Therefore our experiments were unsuccesful, but parallel eigendecompositions of many small matrices is still a very interesting idea.

Our proposed GPU kernel implementation that computes the $\mathbf{Q}$ matrix uses thread synchronization, which only works intra-block. In order to scale up the kernel to handle matrices larger than maximum block dimensions, we will need to investigate some manner of inter-block synchronization scheme. We are aware of one proposed implementation, presented in Xiao and Feng [87], that would be a prime candidate for further investigation and experimentation.

Part V.

Appendix

Appendix A.

KECA on GPU

What follows are a few, noteworthy parts of the KECA GPU implementation.

A.1. RBF kernel matrix computation on GPU

```
__global__
void compute_kernel_matrix(float* kernel_matrix, const float* input, const float alpha, const
 ↪  size_t m, const size_t n)
{
  const auto i = threadIdx.y + blockDim.y * blockIdx.y;  // Maps to rows ~ "down"
  const auto j = threadIdx.x + blockDim.x * blockIdx.x;  // Maps to cols ~ "right"
  if (i >= n || j >= n) { return; }

  // We only want to perform work if we're in the upper diagonal of the kernel matrix...
  // Exit early if we're in the lower diagonal part.
  if (i > j) { return; }

  // Calculate index of within the kernel matrix for the i-th row, and j-th column.
  const auto idx = i + n * j;

  // If we're on the diagonal, simply set kernel coefficient to 1, and exit early.
  if (i == j) {
    kernel_matrix[idx] = 1.f;
    return;
  }

  // Calculate the pairwise Euclidean distance between vectors "i" and "j".
  auto sum = 0.f;
  for (auto k = 0; k < m; ++k) {
    const auto idx_i = k + m * i;  // i-th observation
    const auto idx_j = k + m * j;  // j-th observation
    const auto temp = input[idx_i] - input[idx_j];
    sum += temp * temp;
  }
```

```
  // RBF kernel coefficient based on: K = exp{-alpha * X.^2}
  // Note that we've omitted the idempotent operation of squaring a square root.
  const auto coeff = std::exp(-alpha * sum);

  // Set kernel coefficient in both the upper and lower diagonal.
  kernel_matrix[idx] = coeff;        // Upper diagonal
  kernel_matrix[j + n * i] = coeff;  // Lower diagonal
}
```

A.2. MAGMA-based eigensolver

```
auto solve_eigensystem(const std::vector<float>& matrix, const size_t n)
{
  // Initialize MAGMA.
  magma_init();
  magma_int_t info = 0;
  magma_device_t device;
  magma_getdevice(&device);
  magma_queue_t queue;
  magma_queue_create(device, &queue);

  // Allocate device memory.
  float* dev_matrix;
  magma_smalloc(&dev_matrix, n * n);

  // Copy from CPU to GPU...
  magma_ssetmatrix(n, n, matrix.data(), n, dev_matrix, n, queue);

  std:vector<float> eigvals(n);
  int num_eigvals;

  float* work_matrix;
  magma_smalloc_cpu(&work_matrix, n * n);

  // Figure out what the optimal workspace size is.
  float lwork;
  int liwork;
  magma_ssyevdx_gpu(
    MagmaVec,             // jobz [in]
    MagmaRangeAll,        // range [in]
    MagmaLower,           // uplo [in]
    n,                    // n [in]
    dev_matrix,           // dA [in,out]
    n,                    // ldda [in]
    0,                    // vl [in]
    0,                    // vu [in]
    0,                    // il [in]
    0,                    // iu [in]
```

```
  &num_eigvals,         // m [out]
  eigvals.data(),       // w [out]
  work_matrix,          // wA [in]
  n,                    // ldwa [in]
  &lwork,               // work [out]
  -1,                   // lwork [in]
  &liwork,              // iwork [in]
  -1,                   // liwork [out]
  &info                 // info [out]
);

// Initialize workspace with the optimal size.
float* work;
magma_smalloc_cpu(&work, std::ceil(lwork));

int* iwork;
magma_imalloc_cpu(&iwork, liwork);

// Execute the eigendecomposition.
magma_ssyevdx_gpu(
  MagmaVec,             // jobz [in]
  MagmaRangeAll,        // range [in]
  MagmaLower,           // uplo [in]
  n,                    // n [in]
  dev_matrix,           // dA [in,out]
  n,                    // ldda [in]
  0,                    // vl [in]
  0,                    // vu [in]
  0,                    // il [in]
  0,                    // iu [in]
  &num_eigvals,         // m [out]
  eigvals.data(),       // w [out]
  work_matrix,          // wA [in]
  n,                    // ldwa [in]
  work,                 // work [out]
  lwork,                // lwork [in]
  iwork,                // iwork [in]
  liwork,               // liwork [out]
  &info                 // info [out]
);

if (info > 0) {
  std::cerr « magma_strerror(info) « " (" « info « ")" « "\n";
}

std::vector<float> eigvecs(n * n);
magma_sgetmatrix(n, n, dev_matrix, n, eigvecs.data(), n, queue);

// Release host memory.
magma_free_cpu(work);
magma_free_cpu(iwork);
magma_free_cpu(work_matrix);
```

```
  // Release device memory.
  magma_free(dev_matrix);

  magma_queue_destroy(queue);
  magma_finalize();

  return std::make_tuple(eigvecs, eigvals);
}
```

Appendix B.

QR algorithm on GPU

Complete, experimental implementation of QR algorithm on GPU. Requires a C++ header-only library codenamed "Cudalicious", which can be found at `https://github.com/thomasjo/cudalicious`.

```
#include <cassert>
#include <cstdlib>
#include <iomanip>
#include <iostream>
#include <vector>

#include <cudalicious/blas.hpp>
#include <cudalicious/core.hpp>
#include <cudalicious/solver.hpp>

constexpr auto MAX_ITER = 50;

struct Eigensystem {
  const std::vector<float> eigenvectors;
  const std::vector<float> eigenvalues;

  Eigensystem(const std::vector<float>& eigenvectors, const std::vector<float>& eigenvalues)
    : eigenvectors(eigenvectors)
    , eigenvalues(eigenvalues)
  {}
};

template<typename T>
void print_matrix(const std::vector<T>& matrix, const int n)
{
  for (auto i = 0; i < n; ++i) {
    for (auto j = 0; j < n; ++j) {
      const auto idx = j * n + i;
      std::cout << std::setprecision(7) << std::setw(12) << matrix[idx];
    }
    std::cout << "\n";
  }
```

```
}

__device__
float get_reflector_coefficient(const float* source, const float* tau, const int n, const int
↪  reflector_index, const int row, const int col)
{
  if (row < reflector_index || col < reflector_index) return (row == col) ? 1.f : 0.f;

  const auto row_coeff = (row == reflector_index) ? 1.f : source[reflector_index * n + row];
  const auto col_coeff = (col == reflector_index) ? 1.f : source[reflector_index * n + col];

  return ((row == col) ? 1.f : 0.f) - tau[reflector_index] * row_coeff * col_coeff;
}

__global__
void construct_q_matrix(float* q, const float* source, const float* tau, const size_t n)
{
  // Remember that `x` gives the row index, and `y` gives the column index.
  const auto row = blockDim.x * blockIdx.x + threadIdx.x;
  const auto col = blockDim.y * blockIdx.y + threadIdx.y;
  const auto idx = col * n + row;

  if (row >= n || col >= n) return;

  for (auto k = 0U; k < n; ++k) {
    auto inner_product = 0.f;

    for (auto i = 0U; i < n; ++i) {
      const auto row_coefficient = q[i * n + row];
      const auto col_coefficient = get_reflector_coefficient(source, tau, n, k, i, col);

      inner_product += row_coefficient * col_coefficient;
    }

    __syncthreads();
    q[idx] = inner_product;
  }
}

Eigensystem compute_eigensystem(const std::vector<float>& matrix, const size_t n)
{
  assert(matrix.size() == n * n);  // Ensure matrix is square

  if (n > 32U) {
    std::cerr « "Maximum supported matrix size is 32x32!\n";
    std::exit(1);
  }

  const dim3 threads_per_block(32, 32);
  const dim3 block_size(
    std::ceil(n / threads_per_block.x) + (((n % threads_per_block.x) == 0 ? 0 : 1)),
    std::ceil(n / threads_per_block.y) + (((n % threads_per_block.y) == 0 ? 0 : 1))
```

```
);

// Create cuSOLVER and cuBLAS handles.
auto solver_handle = cuda::solver::initialize();
auto blas_handle = cuda::blas::initialize();

// Allocate device memory.
auto dev_matrix = cuda::copy_to_device(matrix);
auto dev_qr = cuda::allocate<float>(matrix.size());
cuda::copy_on_device(dev_qr, dev_matrix, matrix.size());

// Determine workspace size.
auto workspace_size = cuda::solver::geqrf_buffer_size(solver_handle, n, n, dev_matrix, n);

auto dev_tau = cuda::allocate<float>(n);
auto dev_workspace = cuda::allocate<float>(workspace_size);
auto dev_info = cuda::allocate<int>(1);

std::vector<float> identity(n * n, 0);
for (auto i = 0U; i < n; ++i) {
  identity[i * n + i] = 1.f;
}

auto dev_q = cuda::copy_to_device(identity);
auto dev_eigvecs = cuda::copy_on_device(dev_q, matrix.size());

for (auto iter = 0; iter < MAX_ITER; ++iter) {
  // Compute QR factorization.
  cuda::solver::geqrf(solver_handle, n, n, dev_qr, n, dev_tau, dev_workspace,
  ↪  workspace_size, dev_info);

  cuda::copy_to_device(dev_q, identity.data(), identity.size());
  construct_q_matrix«<block_size, threads_per_block»>(dev_q, dev_qr, dev_tau, n);
  cuda::device_sync();

  constexpr auto alpha = 1.f;
  constexpr auto beta = 0.f;

  // Compute A_k = Q_k^T * A_(k-1) * Q_k -> A_k converges to eigenvalues of A_0.
  cuda::blas::gemm(blas_handle, n, n, n, alpha, dev_q, n, dev_matrix, n, beta, dev_qr, n,
  ↪  true);
  cuda::blas::gemm(blas_handle, n, n, n, alpha, dev_qr, n, dev_q, n, beta, dev_matrix, n);

  // Compute L_k = Q_k * Q_(k-1)..Q_0 -> L_k converges to eigenvectors of A_0.
  cuda::blas::gemm(blas_handle, n, n, n, alpha, dev_eigvecs, n, dev_q, n, beta, dev_qr, n);

  cuda::copy_on_device(dev_eigvecs, dev_qr, matrix.size());
  cuda::copy_on_device(dev_qr, dev_matrix, matrix.size());
}

std::vector<float> eigvecs(n * n);
cuda::copy_to_host(eigvecs, dev_eigvecs);
```

```
  std::vector<float> eigvals(n * n);
  cuda::copy_to_host(eigvals, dev_matrix);

  cuda::free(dev_eigvecs);
  cuda::free(dev_q);
  cuda::free(dev_info);
  cuda::free(dev_workspace);
  cuda::free(dev_tau);
  cuda::free(dev_qr);
  cuda::free(dev_matrix);

  cuda::blas::release(blas_handle);
  cuda::solver::release(solver_handle);

  auto trace = [](const std::vector<float>& matrix, const size_t n) {
    std::vector<float> tr;
    for (auto i = 0U; i < n; ++i) { tr.emplace_back(matrix[i * n + i]); }
    return tr;
  };

  return Eigensystem(eigvecs, trace(eigvals, n));
}

int main()
{
  const auto n = 4;
  const std::vector<float> matrix {
     5, -2, -1, 0,
    -2,  5,  0, 1,
    -1,  0,  5, 2,
     0,  1,  2, 5,
  };

  std::cout « "\nInput matrix:\n";
  print_matrix(matrix, n);

  const auto eigensystem = compute_eigensystem(matrix, n);

  std::cout « "\nComputed eigenvectors:\n";
  print_matrix(eigensystem.eigenvectors, n);
  std::cout « "\nComputed eigenvalues:\n";
  for (auto v : eigensystem.eigenvalues) std::cout « v « "\n";
}
```

Bibliography

[1] Robert Jenssen. "Kernel entropy component analysis." In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 32.5 (2010), pp. 847–860.

[2] Aravindh Krishnamoorthy. "Symmetric QR Algorithm with Permutations." In: *arXiv preprint arXiv:1402.5086* (2014).

[3] Stanimire Tomov, Jack Dongarra, and Marc Baboulin. "Towards dense linear algebra for hybrid GPU accelerated manycore systems." In: *Parallel Computing* 36.5 (2010), pp. 232–240.

[4] Stanimire Tomov et al. "Dense linear algebra solvers for multicore with GPU accelerators." In: *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*. IEEE. 2010, pp. 1–8.

[5] Jack Dongarra et al. "Accelerating numerical dense linear algebra calculations with GPUs." In: *Numerical Computations with GPUs*. Springer, 2014, pp. 3–28.

[6] Rajib Nath, Stanimire Tomov, and Jack Dongarra. "An improved MAGMA GEMM for Fermi graphics processing units." In: *International Journal of High Performance Computing Applications* 24.4 (2010), pp. 511–515.

[7] Rajib Nath, Stanimire Tomov, and Jack Dongarra. "Accelerating GPU kernels for dense linear algebra." In: *International Conference on High Performance Computing for Computational Science*. Springer. 2010, pp. 83–92.

[8] Azzam Haidar et al. "A framework for batched and gpu-resident factorization algorithms applied to block householder transformations." In: *International Conference on High Performance Computing*. Springer. 2015, pp. 31–47.

[9] Ahmad Abdelfattah et al. "High-Performance Tensor Contractions for GPUs." In: *Procedia Computer Science* 80 (2016), pp. 108–118.

[10] Yinan Li, Jack Dongarra, and Stanimire Tomov. "A note on auto-tuning GEMM for GPUs." In: *International Conference on Computational Science*. Springer. 2009, pp. 884–892.

[11] Jakub Kurzak, Stanimire Tomov, and Jack Dongarra. "Autotuning GEMM kernels for the Fermi GPU." In: *IEEE Transactions on Parallel and Distributed Systems* 23.11 (2012), pp. 2045–2057.

[12] Howard Anton and Chris Rorres. *Elementary Linear Algebra : with Supplemental Applications*. Wiley, 2011. ISBN: 978-0-470-56157-7.

[13] Leonhard Euler. "Du mouvement d'un corps solide quelconque lorsqu'il tourne autour d'un axe mobile." In: *Histoire de l'Académie royale des sciences et des belles lettres de Berlin. [Formerly Miscellanea Berolinensia]. Avec les mémoires.* Akademie der Wissenschaften der DDR., 1767, pp. 176–227. URL: https://books.google.no/books?id=XLQEAAAAQAAJ.

[14] David Hilbert. "Grundzüge einer allgeminen Theorie der linaren Integralrechnungen. (Erste Mitteilung)." In: (1904). URL: http://www.digizeitschriften.de/dms/img/?PID=GDZPPN002499967.

[15] Svante Wold, Kim Esbensen, and Paul Geladi. "Principal component analysis." In: *Chemometrics and intelligent laboratory systems* 2.1-3 (1987), pp. 37–52.

[16] Jianbo Shi and Jitendra Malik. "Normalized cuts and image segmentation." In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 22.8 (2000), pp. 888–905.

[17] Amy N Langville and Carl D Meyer. "A survey of eigenvector methods for web information retrieval." In: *SIAM review* 47.1 (2005), pp. 135–161.

[18] C. D. Meyer. *Matrix analysis and applied linear algebra*. Society for Industrial and Applied Mathematics, 2000. ISBN: 978-0-898714-54-8.

[19] Paolo Ruffini. *Teoria generale delle equazioni: in cui si dimostra impossibile la soluzione algebraica delle equazioni generali di grad superiore al quarto*. Vol. 1. Nella stamperia di S. Tommaso d'Aquino, 1799.

[20] Niels Henrik Abel. *Mémoire sur les équations algébrique: où on démontre l'impossiblité de la résolution de l'equation générale du cinquième dégré*. Librarian, Faculty of Science, University of Oslo, 1824.

[21] Niels Henrik Abel. "Démonstration de limpossibilité de la résolution algébrique des équations générales qui passent le quatrieme degré." In: *Journal für die reine und angewandte Mathematik* 1 (1826), pp. 65–96.

[22] Alan M Turing. "Rounding-off errors in matrix processes." In: *The Quarterly Journal of Mechanics and Applied Mathematics* 1.1 (1948), pp. 287–308.

[23] R. M. Corless and D. J. Jeffrey. "The Turing Factorization of a Rectangular Matrix." In: *SIGSAM Bull.* 31.3 (Sept. 1997), pp. 20–30. ISSN: 0163-5824. DOI: 10.1145/271130.271135. URL: http://doi.acm.org/10.1145/271130.271135.

[24] David S. Watkins. *Fundamentals of matrix computations*. Wiley-Interscience, 2002. ISBN: 0-471-21394-2.

[25] Denis Serre. *Matrices theory and applications*. Springer, 2002. ISBN: 0-387-95460-0.

[26] Lars Eldén. *Matrix methods in data mining and pattern recognition*. Society for Industrial and Applied Mathematics, 2007. ISBN: 978-0-898716-26-9.

[27] Gene Golub. *Matrix computations*. Johns Hopkins University Press, 1996. ISBN: 0-8018-5414-8.

[28] William Press. *Numerical recipes in C : the art of scientific computing*. Cambridge University Press, 1992. ISBN: 0-521-43108-5.

[29] Nicholas J. Higham. "Cholesky factorization." In: *Wiley Interdisciplinary Reviews: Computational Statistics* 1.2 (Sept. 2009), pp. 251–254. ISSN: 1939-0068. DOI: 10.1002/wics.18. URL: http://dx.doi.org/10.1002/wics.18.

[30] James W Daniel et al. "Reorthogonalization and stable algorithms for updating the Gram-Schmidt QR factorization." In: *Mathematics of Computation* 30.136 (1976), pp. 772–795.

[31] Åke Björck. "Numerics of Gram-Schmidt orthogonalization." In: *Linear Algebra and Its Applications* 197 (1994), pp. 297–316.

[32] Luc Giraud, Julien Langou, and Miroslav Rozloznik. "The loss of orthogonality in the Gram-Schmidt orthogonalization process." In: *Computers & Mathematics with Applications* 50.7 (2005), pp. 1069–1075.

[33] Lloyd Trefethen. *Numerical linear algebra*. Society for Industrial and Applied Mathematics, 1997. ISBN: 0-89871-361-7.

[34] Tim Sauer. *Numerical analysis*. Boston: Pearson, 2012. ISBN: 978-0321783677.

[35] Leslie Hogben. *Handbook of linear algebra*. Chapman & Hall/CRC, 2007. ISBN: 1-58488-510-6.

[36] James Demmel. *Applied numerical linear algebra*. Society for Industrial and Applied Mathematics, 1997. ISBN: 0-89871-389-7.

[37] RV Mises and Hilda Pollaczek-Geiringer. "Praktische Verfahren der Gleichungsauflösung." In: *ZAMM-Journal of Applied Mathematics and Mechanics/Zeitschrift für Angewandte Mathematik und Mechanik* 9.2 (1929), pp. 152–164.

[38] Lawrence Page et al. *The PageRank Citation Ranking: Bringing Order to the Web*. Technical Report 1999-66. Stanford InfoLab, Nov. 1999. URL: http://ilpubs.stanford.edu:8090/422/.

[39] Sergios Theodoridis. *Pattern Recognition*. Academic Press, 2009. ISBN: 978-1-59749-272-0.

[40] Thomas E Booth. "Power iteration method for the several largest eigenvalues and eigenfunctions." In: *Nuclear science and engineering* 154.1 (2006), pp. 48–62.

[41] Toshihiro Yamamoto. "Convergence of the second eigenfunction in Monte Carlo power iteration." In: *Annals of Nuclear Energy* 36.1 (2009), pp. 7–14.

[42] Filippo Maria Bianchi et al. "An agent-based algorithm exploiting multiple local dissimilarities for clusters mining and knowledge discovery." In: *Soft Computing* (2015), pp. 1–23.

[43] John GF Francis. "The QR transformation a unitary analogue to the LR transformation (part 1)." In: *The Computer Journal* 4.3 (1961), pp. 265–271.

[44] John GF Francis. "The QR transformation (part 2)." In: *The Computer Journal* 4.4 (1962), pp. 332–345.

[45] Vera N Kublanovskaya. "On some algorithms for the solution of the complete eigenvalue problem." In: *USSR Computational Mathematics and Mathematical Physics* 1.3 (1962), pp. 637–657.

[46] S. K. Godunov. *Modern aspects of linear algebra*. American Mathematical Society, 1998. ISBN: 0-8218-0888-5.

[47] Granville Sewell. *Computational methods of linear algebra*. Wiley-Interscience, 2005. ISBN: 0-471-73579-5.

[48] Junjie Li, Sanjay Ranka, and Sartaj Sahni. "GPU matrix multiplication." In: *Multicore Computing: Algorithms, Architectures, and Applications* 345 (2013).

[49] Michael Cogswell et al. "Reducing Overfitting in Deep Networks by Decorrelating Representations." In: *arXiv preprint arXiv:1511.06068* (2015).

[50] Karl Pearson. "On lines and planes of closest fit to systems of points in space." In: *Philosophical Magazine Series 6* 2.11 (1901), pp. 559–572. DOI: 10.1080/14786440109462720.

[51] Fabian Pedregosa et al. "Scikit-learn: Machine learning in Python." In: *Journal of Machine Learning Research* 12.Oct (2011), pp. 2825–2830.

[52] Naum Ilich Akhiezer and Izrail Markovich Glazman. *Theory of linear operators in Hilbert space*. Courier Corporation, 2013.

[53] James Mercer. "Functions of positive and negative type, and their connection with the theory of integral equations." In: *Philosophical transactions of the royal society of London. Series A, containing papers of a mathematical or physical character* 209 (1909), pp. 415–446.

[54] Kenji Fukumizu, Francis R Bach, and Michael I Jordan. "Dimensionality reduction for supervised learning with reproducing kernel Hilbert spaces." In: *Journal of Machine Learning Research* 5.Jan (2004), pp. 73–99.

[55] Nachman Aronszajn. "Theory of reproducing kernels." In: *Transactions of the American mathematical society* 68.3 (1950), pp. 337–404.

[56] Yoav Goldberg and Michael Elhadad. "splitSVM: fast, space-efficient, non-heuristic, polynomial kernel computation for NLP applications." In: *Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics on Human Language Technologies: Short Papers*. Association for Computational Linguistics. 2008, pp. 237–240.

[57] Yin-Wen Chang et al. "Training and testing low-degree polynomial data mappings via linear SVM." In: *Journal of Machine Learning Research* 11.Apr (2010), pp. 1471–1490.

[58] Gert RG Lanckriet et al. "Learning the kernel matrix with semidefinite programming." In: *Journal of Machine learning research* 5.Jan (2004), pp. 27–72.

[59] Bernhard Schölkopf, Alexander Smola, and Klaus-Robert Müller. "Kernel principal component analysis." In: *International Conference on Artificial Neural Networks*. Springer. 1997, pp. 583–588.

[60] Luis Gómez-Chova, Robert Jenssen, and Gustavo Camps-Valls. "Kernel entropy component analysis for remote sensing image clustering." In: *IEEE Geoscience and Remote Sensing Letters* 9.2 (2012), pp. 312–316.

[61] Robert Jenssen and Ola Storås. "Kernel Entropy Component Analysis Pre-images for Pattern Denoising." In: *Scandinavian Conference on Image Analysis*. Springer Berlin Heidelberg. 2009, pp. 626–635.

[62] Gordon E Moore et al. "Progress in digital integrated electronics." In: 1975.

[63] Shekhar Borkar. "Design challenges of technology scaling." In: *IEEE micro* 19.4 (1999), pp. 23–29.

[64] Sharan Chetlur et al. "cuDNN: Efficient primitives for deep learning." In: *arXiv preprint arXiv:1410.0759* (2014).

[65] Stefan Hadjis et al. "Caffe con Troll: Shallow Ideas to Speed Up Deep Learning." In: *Proceedings of the Fourth Workshop on Data analytics in the Cloud*. ACM. 2015, p. 2.

[66] Barry Wilkinson and Michael Allen. *Parallel programming*. Vol. 999. Prentice hall New Jersey, 1999.

[67] NVIDIA Corporation. *CUDA C Programming Guide*. 2016. URL: https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf (visited on 11/11/2016).

[68] Intel Corporation. *Intel Xeon processor E5-2699 v4*. Datasheet. 2016. URL: http://ark.intel.com/products/91317/Intel-Xeon-Processor-E5-2699-v4-55M-Cache-2_20-GHz (visited on 11/14/2016).

[69] NVIDIA Corporation. *NVIDIA Tesla P100. The most advanced datacenter accelerator ever built*. Whitepaper. 2016. URL: https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf (visited on 11/14/2016).

[70] Alan Jay Smith. "Cache memories." In: *ACM Computing Surveys (CSUR)* 14.3 (1982), pp. 473–530.

[71] Saied Zangenehpour. *Method of varying the amount of data prefetched to a cache memory in dependence on the history of data requests*. US Patent 5,146,578. Sept. 1992.

[72] Erik Lindholm et al. "NVIDIA Tesla: A unified graphics and computing architecture." In: *IEEE micro* 28.2 (2008), pp. 39–55.

[73] Craig M Wittenbrink, Emmett Kilgariff, and Arjun Prabhu. "Fermi GF100 GPU architecture." In: *IEEE Micro* 31.2 (2011), pp. 50–59.

[74] Intel Corporation. *PHY interface for the PCI Express, SATA, and USB 3.1 architectures*. Datasheet. 2016. URL: http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/phy-interface-pci-express-sata-usb30-architectures-3.1.pdf (visited on 11/16/2016).

[75] Nicholas Wilt. *The CUDA Handbook. A comprehensive guide to GPU programming*. Addison-Wesley, 2013. ISBN: 978-0-321-80946-9.

[76] Herb Sutter and Andrei Alexandrescu. *C++ coding standards: 101 rules, guidelines, and best practices*. Pearson Education, 2004.

[77] Scott Meyers. *Effective C++: 55 specific ways to improve your programs and designs*. Pearson Education, 2005.

[78] Bjarne Stroustrup. "Foundations of C++." In: *European Symposium on Programming*. Springer. 2012, pp. 1–25.

[79] Jingyue Wu et al. "gpucc: an open-source GPGPU compiler." In: *Proceedings of the 2016 International Symposium on Code Generation and Optimization*. ACM. 2016, pp. 105–116.

[80] Henry Wong et al. "Demystifying GPU microarchitecture through microbenchmarking." In: *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*. IEEE. 2010, pp. 235–246.

[81] John E Stone, David Gohara, and Guochun Shi. "OpenCL: A parallel programming standard for heterogeneous computing systems." In: *Computing in science & engineering* 12.1-3 (2010), pp. 66–73.

[82] Ben Sander et al. *HCC: A C++ Compiler For Heterogeneous Computing*. Tech. rep. Open Standards, 2015.

[83] Advanced Micro Devices, Inc. *AMD Launches 'Boltzmann Initiative' to Dramatically Reduce Barriers to GPU Computing on AMD FirePro Graphics*. URL: http://www.amd.com/en-us/press-releases/Pages/boltzmann-initiative-2015nov16.aspx (visited on 11/30/2016).

[84] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning." In: *Nature* 521.7553 (2015), pp. 436–444.

[85] Jürgen Schmidhuber. "Deep learning in neural networks: An overview." In: *Neural Networks* 61 (2015), pp. 85–117.

[86] Edward Anderson et al. *LAPACK Users' guide*. Vol. 9. Siam, 1999.

[87] Shucai Xiao and Wu-chun Feng. "Inter-block GPU communication via fast barrier synchronization." In: *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE. 2010, pp. 1–12.