



ulm university universität
uulm

**Fakultät für
Mathematik und
Wirtschafts-
wissenschaften**

Institut für Numerische
Mathematik

Cache-optimierte QR-Zerlegung

Bachelorarbeit an der Universität Ulm

Vorgelegt von:

Florian Krötz
florian.kroetz@uni-ulm.de

Gutachter:

Dr. Michael Lehn
Dr. Andreas Borchert

Betreuer:

Dr. Michael Lehn

2018

Fassung 5. August 2018

© 2018 Florian Krötz

Satz: PDF- \LaTeX 2 _{ϵ}

Inhaltsverzeichnis

1	Einleitung	1
2	BLAS	3
2.1	Datenstruktur für Matrizen	3
2.2	Einige BLAS-Routinen	7
2.2.1	Matrix-Matrix Produkt (gemv)	7
2.2.2	Matrix-Vektor Produkt (gemv)	7
2.2.3	Rank1 update (ger)	8
2.2.4	Matrix-Matrix Produkt (trmm)	8
2.2.5	Matrix-Vektor Produkt (trmv)	8
3	QR-Zerlegung	10
3.1	Definition	10
3.1.1	Beispiel für eine Anwendung	10
3.2	Householder-Transformation	11
3.2.1	Householder Vector	13
3.2.2	Householder-Transformation anwenden	15
3.2.3	QR-Zerlegung mittels Housholder-Transformationen	15
3.3	Geblockte QR-Zerlegung	18
3.3.1	Berechnung der Matrix T	19
3.3.2	Anwenden von $I - VTV^T$	20
4	Implementierung und Benchmarks	24
4.1	Bibliothek	24
4.1.1	Algorithmus	26
4.2	Fehlerschätzer	26
4.3	Benchmarks	27
4.3.1	Aufwand	27
4.3.2	FLOPS	27
4.3.3	Test System	27

A Block Reflector	29
Literaturverzeichnis	33

1 Einleitung

Als QR-Zerlegung bezeichnet man die Zerlegung der Matrix A in eine orthogonale Matrix Q und eine obere Dreiecksmatrix R .

$$A = Q \cdot R$$

In dieser Arbeit wird die Berechnung der QR-Zerlegung mittels Householder-Transformation betrachtet.

Neben der Householder-Transformation kann die QR-Zerlegung auch mittels Givens-Rotationen oder mit dem Gram-Schmidtschen Orthogonalisierungsverfahrens berechnet werden.

Die QR-Zerlegung bietet folgende Möglichkeiten:

- Gleichungssysteme mit schlechter Kondition lassen sich mittels QR-Zerlegung stabiler als mit dem Gausschen Eliminationsverfahren (LR-Zerlegung) lösen.
- Lösen von linearen Ausgleichsproblemen durch die Methode der kleinsten Fehlerquadrate.
- Berechnung von Eigenwerten einer Matrix mittels QR-Verfahren. Im QR-Verfahren ist die QR-Zerlegung eine Kernoperation, da in jedem Iterationsschritt eine QR-Zerlegung berechnet wird.

Weil die QR-Zerlegung bei den aufgezählten Problemen häufig verwendet wird, empfiehlt es sich, die Berechnung der QR-Zerlegung zu optimieren.

Um die Prozessorkapazität optimal zu nutzen, ist es sinnvoll, die QR-Zerlegung mit einem cache-optimierten Algorithmus zu berechnen. Dadurch ist es möglich, die angegebene Maximalleistung des Prozessors zu erreichen.

Cache-Optimierung

Der Cache ist ein schneller Datenspeicher. Daten, die vom Prozessor verarbeitet werden sollen, müssen immer zuerst in den Cache geladen werden. Der Zugriff auf Daten, die im RAM liegen, dauert sehr viel länger als der Zugriff auf Daten, die im Cache liegen. Aus diesem Grund ist es sinnvoll, die Algorithmen so zu gestalten, dass eine optimale Übertragung erfolgt.

Ziele der Cache-Optimierung:

- Mehrfaches Laden von Daten soll vermeiden werden.
- Die Daten müssen in der Reihenfolge, in der sie verwendet werden, im RAM liegen.
- Sequentiell im RAM liegende Daten werden durch Prefetching effizient in den Cache geladen.

Prefetching ist die Eigenschaft des Prozessors, Zugriffsmuster auf den Speicher zu erkennen und vorherzusagen.

Intel MKL

Die Intel MKL (Math Kernel Library) [2] ist eine Bibliothek der Firma Intel, in welcher mathematische Funktionen enthalten sind. Die MKL implementiert diese Funktionen sehr effizient, damit der Prozessor die angegebene theoretische Maximalleistung erreichen kann.

In der MKL sind die beiden Programmbibliotheken BLAS (Basic Linear Algebra Subprograms) und LAPACK (Linear Algebra Package) implementiert. Diese Funktionen aus der Teilmenge der linearen Algebra sind für die Berechnung der QR-Zerlegung elementar erforderlich.

BLAS enthält grundlegende und LAPACK weiterentwickelte Funktionen der linearen Algebra.

2 BLAS

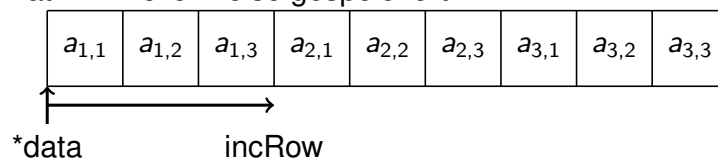
Die Abkürzung BLAS steht für Basic Linear Algebra Subprograms. BLAS-Bibliotheken enthalten elementare Operationen der linearen Algebra.

2.1 Datenstruktur für Matrizen

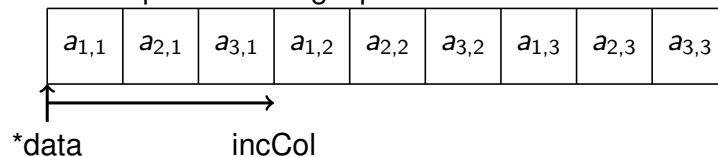
Vollbesetzte Matrizen werden bei BLAS entweder zeilen- oder spaltenweise abgespeichert. Das bedeutet, dass entweder die Zeilen- oder die Spalten der Matrix hintereinander im Speicher stehen.

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix} \in \mathbb{R}^{3 \times 3}$$

Matrix A Zeilenweise gespeichert



Matrix A Spaltenweise gespeichert



Eine Datenstruktur benötigt folgende Elemente:

- einen Zeiger auf eine Speicherfläche
- Information, ob die Matrix zeilen- oder spaltenweise gespeichert ist

- die Dimension der Matrix.

Eine derartige Datenstruktur könnte in C/C++ so aussehen.

```
1 struct Matrix {  
2     double * data;  
3     std::ptrdiff_t incRow, incCol;  
4     std::size_t numRows, numCols;  
5 }
```

Diese Datenstruktur ist für Matrixeinträge vom Type `double`. Die Variablen für die Speicherverwaltung (Zeile 3) sind vom Typ `ptrdiff_t` und die Variablen für die Dimensions Informationen sind vom Typ `size_t`. `ptrdiff_t` und `size_t` sind Datentypen für Ganzzahlen. Der Unterschied zwischen den Datentypen ist `size_t` kann nur Werte darstellen die ≥ 0 sind, `ptrdiff_t` dann auch negative Werte darstellen. Negative Werte von `incRow` und `incCol` könne nützlich sein, falls die Daten nicht auf Steigend sondern absteigend im Speicher liegen (siehe Beispiel) [10].

Für Intel-MKL-Routinen müssen die Matrizen zeilenweise gespeichert sein. Das bedeutet `incCol` ist gleich 1.

Beispiel

Angenommen, es wurde genügend Speicher für eine Matrix A belegt und die Werte 1 bis 25 liegen hintereinander im Speicher. Dann lässt sich die Matrix

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \\ 16 & 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 & 25 \end{pmatrix}$$

durch die folgende Datenstruktur repräsentieren.

```
1 struct Matrix A;  
2 A.data = /* pointer to data */;  
3 A.numRows = 5;  
4 A.numCols = 5;  
5 A.incRow = A.numCols;  
6 A.incCol = 1;
```


Die Matrix B soll den Matrixblock aus A darstellen, welcher in der zweiten Zeile und dritte Spalte beginnt. Zusätzlich soll der Block transponiert betrachtet werden. Die Matrix

$$B = \begin{pmatrix} 8 & 13 & 18 & 23 \\ 9 & 14 & 19 & 24 \\ 10 & 15 & 20 & 25 \end{pmatrix}$$

lässt sich mit folgender Datenstruktur repräsentieren.

```

1 struct Matrix B;
2 B.data = &A.data[ 1*A.incRow + 2*A.incCol ];
3 B.numRows = 3;
4 B.numCols = 4;
5 B.incRow = A.incCol;
6 B.incCol = A.incRow;
```

In Zeile 2 wird der Daten-Zeiger auf das Element der zweiten Zeile und dritten Spalte gesetzt. Um die Matrix zu transponieren, werden die Inkremente vertauscht. Dies geschieht in den Zeilen 5 und 6. In den Zeilen 3 und 4 werden die Matrix Dimensionen richtig gesetzt.

Die Matrix C soll ein Beispiel für negative Inkremente sein. Sie nutzt den selben Datenspeicher wie die Matrizen A und B , in dem die Werte 1 bis 25 hintereinander im Speicher liegen. Die Matrix

$$C = \begin{pmatrix} 25 & 20 & 15 & 10 \\ 24 & 19 & 14 & 9 \\ 23 & 18 & 13 & 8 \end{pmatrix}$$

lässt sich mit folgender Datenstruktur repräsentieren.

```

1 struct Matrix C;
2 C.data = &B.data[ 2*B.incRow + 3*B.incCol ];
3 C.numRows = B.numRows;
4 C.numCols = B.numCols;
5 C.incRow = -B.incRow;
6 C.incCol = -B.incCol;
```

Der data-Pointer wurde auf das letzte Element der Matrix B gesetzt. Die Inkremente werden negativ von der Matrix B übernommen.

Matrizen wie B und C , die den selben Datenspeicher wie A nutzen, werden als Matrixview bezeichnet. [10]

2.2 Einige BLAS-Routinen

Im Folgenden werden einige BLAS-Routinen beschrieben, die bei der QR-Zerlegung benutzt werden. BLAS-Routinen werden nach folgendem Schema benannt: Der erste Buchstabe im Namen gibt an, für welchen Datentyp die Funktion implementiert wurde. Der Rest beschreibt die Funktion.

Beispiel *dgemm*: Das *d* zeigt, dass die Funktion für Doubles gilt und *gemm* steht für *general matrix matrix*. Die Funktion berechnet also das Matrix-Matrix Produkt für Matrizen, deren Einträge Doubles sind.

2.2.1 Matrix-Matrix Produkt (gemm)

Die Funktion *gemm* berechnet das Matrix-Matrix Produkt. Der Funktion werden die Matrizen *A*, *B* und *C* und die Skalare α und β übergeben. Außerdem werden zwei Flags übergeben, ob die Matrizen *A* und *B* transponiert zu betrachten sind.

Die Funktion berechnet

$$C \leftarrow \beta C + \alpha AB \quad (2.1)$$

Falls $\beta = 0$, wird die Matrix *C* zuerst mit Nullen initialisiert. Falls *C* Einträge hat, die NaN (*Not a Number*) sind, werden diese mit Nullen überschrieben.

[10] Blas technical forum netlib

2.2.2 Matrix-Vektor Produkt (gemv)

Die Funktion *gemv* berechnet das Matrix-Vektor Produkt. Der Funktion werden die Matrix *A*, die Vektoren *x* und *y*, und die Skalare α und β übergeben. Außerdem wird ein Flag übergeben, das anzeigt, ob die Matrix *A* transponiert werden soll.

Die Funktion berechnet

$$y \leftarrow \beta y + \alpha Ax \quad (2.2)$$

Falls $\beta = 0$, wird der Vektor *y* zuerst mit Nullen initialisiert. Falls *y* Einträge hat, die NaN (*Not a Number*) sind, werden diese mit Nullen überschrieben.

2.2.3 Rank1 update (ger)

Die Funktion *ger* berechnet ein dyadisches Produkt aus den Vektoren x und y , skaliert die daraus resultierende Matrix mit α und addiert das Ergebnis auf A .

Der Funktion werden die Matrix A , die Vektoren x und y und das Skalar α übergeben.

Die Funktion berechnet

$$A \leftarrow A + \alpha xy^T \quad (2.3)$$

2.2.4 Matrix-Matrix Produkt (trmm)

Die Funktion *trmm* berechnet das Matrix-Matrix-Produkt einer Dreiecksmatrix mit einer voll besetzten Matrix. Der Funktion wird die Dreiecksmatrix A , die Matrix B und das Skalar α übergeben. Außerdem werden Flags mit übergeben, die anzeigen, ob A eine obere oder untere Dreiecksmatrix ist, ob A eine strikte oder unipotente Dreiecksmatrix ist, ob A von links oder rechts auf B multipliziert werden soll und ob A transponiert werden soll. Diese Eigenschaften werden unten in *op*(\cdot) zusammengefasst.

Die Funktion berechnet

$$B \leftarrow \alpha \cdot op(A) \cdot B \quad \text{oder} \quad B \leftarrow \alpha \cdot B \cdot op(A) \quad (2.4)$$

2.2.5 Matrix-Vektor Produkt (trmv)

Die Funktion *trmv* berechnet das Matrix-Vektor-Produkt für Dreiecksmatrizen. Die Funktion berechnet

$$x \leftarrow \alpha Ax \quad (2.5)$$

Dreiecksmatrizen, die von den Funktionen *trmm* und *trmv* verarbeitet werden, sind quadratische Matrizen, von denen nur der obere oder der untere Dreiecksteil betrachtet wird.

Das heißt, die Matrix liegt als quadratische Matrix im Speicher. Die Funktion beachtet nur die Einträge über oder unter der Diagonalen. Wird dem Algorithmus mit

einem Flag angezeigt, dass er die Matrix als eine unipotente Dreiecksmatrix betrachten soll, dann werden die Diagonaleinträge im Speicher nicht beachtet, sondern vom Algorithmus als 1 angenommen. [10]

3 QR-Zerlegung

3.1 Definition

Eine Matrix $A \in \mathbb{R}^{m \times n}$, $m \geq n$ besitzt eine eindeutige QR-Zerlegung

$$A = QR \quad (3.1)$$

mit einer orthogonalen Matrix $Q \in \mathbb{R}^{m \times m}$ und einer oberen Dreiecksmatrix $R \in \mathbb{R}^{m \times n}$ [6].

Eine QR-Zerlegung kann mit einer Householder-Transformation berechnet werden.

3.1.1 Beispiel für eine Anwendung

Lösung eines Minimierungsproblems

$$\min_{x \in \mathbb{R}^n} \|Ax - b\|_2^2 \quad (3.2)$$

mit Matrix $A \in \mathbb{R}^{m \times n}$ mit $\text{rang}(A) = n < m$, für die eine QR-Zerlegung existiert. R besitzt die Gestalt

$$R = \begin{pmatrix} * & * & * \\ & * & * \\ & & * \\ \hline & & 0 \end{pmatrix} = \begin{pmatrix} \hat{R} \\ 0 \end{pmatrix}$$

\hat{R} stellt eine obere Dreiecksmatrix dar. Damit kann man das Minimierungsproblem wie folgt mit $A = QR$ modifizieren

$$\min_{x \in \mathbb{R}^n} \|Ax - b\|_2^2 = \min_{x \in \mathbb{R}^n} \|Q^T(Ax - b)\|_2^2 = \min_{x \in \mathbb{R}^n} \|Rx - Q^T b\|_2^2 \quad (3.3)$$

Eine Lösung des Gleichungssystems

$$\hat{R}x = Q^T b \quad (3.4)$$

ist auch eine Lösung des Minimierungsproblem (3.2). Da R eine Dreiecksmatrix ist, lässt sich (3.4) leicht mit Rückwärtseinsetzen lösen.

3.2 Householder-Transformation

Eine Matrix $H \in \mathbb{R}^{n \times n}$

$$H = I - 2 \frac{vv^T}{v^T v} \quad (3.5)$$

wird als Householder-Transformation und der Vektor $v \in \mathbb{R}^n$ als Householder-Vektor bezeichnet. Eine Householder-Transformation $H = I - 2 \frac{vv^T}{v^T v}$ ist orthogonal und symmetrisch [6].

Die Householder-Transformation spiegelt den Vektor x auf die Achse x_1 . Dazu multipliziert man H von links auf x

$$Hx = \alpha e_1 \quad (3.6)$$

mit dem Skalar $\alpha \in \mathbb{R}$ und e_1 als ersten kanonischen Einheitsvektor. Der Householder-Vektor steht senkrecht auf der Ebene, an welcher x gespiegelt wird.

Die Abbildung 3.1 veranschaulicht die Spiegelung des Vektors x an der gestrichelt eingezeichneten Ebene auf die Achse x_1 .

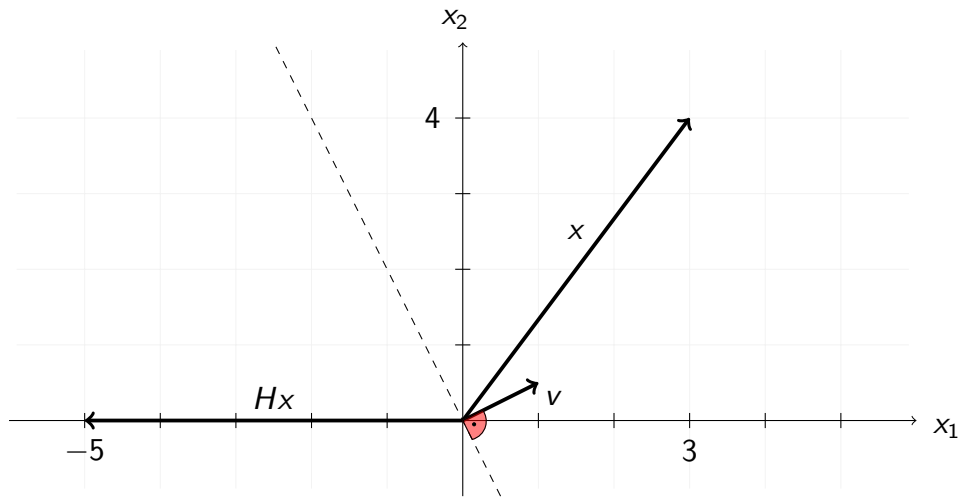


Abbildung 3.1: Beispiel Householder-Transformation mit $x = (3, 4)^T$

Beispiel

Rechnung zu Abbildung 3.1: Der Vektor $x = \begin{pmatrix} 3 \\ 4 \end{pmatrix}$ soll auf die x_1 Achse gespiegelt werden.

Zuerst wird der Householder-Vektor v und der dazugehörige Faktor τ wie in Abschnitt 3.2.1 berechnet.

$$\begin{aligned} \|x\|_2 &= \sqrt{3^2 + 4^2} = 5 \\ \alpha &= -1 \cdot \text{sign}(x_1) \cdot \|x\|_2 = -5 \\ \tau &= \frac{\alpha - x_1}{\alpha} = \frac{-5 - 3}{-5} = \frac{8}{5} \\ v &= \frac{x - \alpha e_1}{x_1 - \alpha} = \frac{\begin{pmatrix} 3 \\ 4 \end{pmatrix} - \begin{pmatrix} -5 \\ 0 \end{pmatrix}}{3 - (-5)} = \frac{1}{8} \begin{pmatrix} 8 \\ 4 \end{pmatrix} = \begin{pmatrix} 1 \\ 0,5 \end{pmatrix} \end{aligned}$$

Nun wird H auf x angewandt, wie in Kapitel 3.2.1 beschrieben.

$$\begin{aligned} Hx &= x - \tau v(v^T x) = \begin{pmatrix} 3 \\ 4 \end{pmatrix} - \frac{8}{5} \cdot \begin{pmatrix} 1 \\ 0,5 \end{pmatrix} \cdot \left(\begin{pmatrix} 1 \\ 0,5 \end{pmatrix}^T \cdot \begin{pmatrix} 3 \\ 4 \end{pmatrix} \right) \\ &= \begin{pmatrix} 3 \\ 4 \end{pmatrix} - \frac{8}{5} \cdot \begin{pmatrix} 1 \\ 0,5 \end{pmatrix} \cdot 5 = \begin{pmatrix} 3 \\ 4 \end{pmatrix} - \begin{pmatrix} 8 \\ 4 \end{pmatrix} \\ \iff Hx &= \begin{pmatrix} -5 \\ 0 \end{pmatrix} \end{aligned}$$

Der Vektor $x = \begin{pmatrix} 3 \\ 4 \end{pmatrix}$ wurde mit dem Householder-Vektor $v = \begin{pmatrix} 1 \\ 0,5 \end{pmatrix}$ auf die x_1 Achse gespiegelt.

3.2.1 Householder Vector

Damit (3.6) gilt, wird der Vektor berechnet, indem man (3.5) in (3.6) einsetzt

$$\begin{aligned} Hx &= x - 2 \frac{v v^T}{v^T v} x = x - 2 \underbrace{\frac{v^T x}{v^T v}}_{\lambda} v = x - \lambda v \stackrel{!}{=} \alpha e_1 \\ \implies v &\in \text{span}\{x - \alpha e_1\} \end{aligned}$$

Setzt man $v = t(x - \alpha e_1)$ in $Hx = \alpha e_1$ (3.6) ein, dann erhält man

$$\begin{aligned} Hx &= x - \frac{2}{v^T v} v(v^T x) = x - 2 \frac{v^T x}{v^T v} v \\ &= x - 2 \frac{t(x - \alpha e_1)^T x}{t(x - \alpha e_1)^T t(x - \alpha e_1)} t(x - \alpha e_1) = x - 2 \frac{(x - \alpha e_1)^T x}{(x - \alpha e_1)^T (x - \alpha e_1)} (x - \alpha e_1) \\ &= x - \frac{(x - \alpha e_1)^T x}{\|x - \alpha e_1\|_2^2} (x - \alpha e_1) \\ &= \underbrace{\left(1 - \frac{2(x - \alpha e_1)^T x}{\|x - \alpha e_1\|_2^2} \right)}_{\stackrel{!}{=} 0} x + \alpha e_1 \underbrace{\frac{2(x - \alpha e_1)^T x}{\|x - \alpha e_1\|_2^2}}_{\stackrel{!}{=} 1} \stackrel{!}{=} \alpha e_1 \end{aligned} \tag{3.7}$$

Damit das letzte = (3.7) gilt muss gelten.

$$\begin{aligned}
 1 &= \frac{2(x - \alpha e_1)^T x}{\|x - \alpha e_1\|^2} \\
 \Leftrightarrow (x - \alpha e_1)^T (x - \alpha e_1) &= 2x^T x - 2\alpha x_1 \\
 \Leftrightarrow x^T x - 2\alpha x_1 + \alpha^2 &= 2x^T x - 2\alpha x_1 \\
 \Leftrightarrow \alpha &= \pm \sqrt{x^T x}
 \end{aligned}$$

Das Vorzeichen von $\alpha = \pm \sqrt{x^T x}$ kann man frei wählen, um $v = x - \alpha e_1$ zu berechnen.

Wählt man das Vorzeichen positiv, kann Auslöschung auftreten, falls x annähernd ein positives Vielfaches von e_1 ist.

LAPACK [9] vermeidet die Auslöschung, indem das Vorzeichen entgegengesetzt gewählt wird. Das bedeutet, dass x immer auf die gegenüberliegende Seite gespiegelt wird.

Im Skript von Numerik 1 [6] wird das Vorzeichen immer positiv gewählt:

$\alpha = |\sqrt{x^T x}| = \|x\|_2$. Eine mögliche Auslöschung im Fall $x_1 > 0$ wird hier durch die folgende Umformung vermieden.

$$v_1 = x_1 - \|x\|_2 = \frac{x_1^2 - \|x\|_2^2}{x_1 + \|x\|_2} = \frac{-(x_2^2 + \dots + x_n^2)}{x_1 + \|x\|_2}$$

Um den Vektor v später auf der frei werdenden Diagonalen von A speichern zu können, wird er auf $v_1 = 1$ normiert. Dies geschieht mit

$$v = \frac{x - \alpha e_1}{x_1 - \alpha} \tag{3.8}$$

Mit der Normierung kann man den Faktor $\tau = \frac{2}{v^T v}$ berechnen. Setze dazu (3.8) in die Definition von τ ein.

$$\tau = \frac{2}{v^T v} = \frac{2(x_1 - \alpha)^2}{(x - \alpha e_1)^T (x - \alpha e_1)} = \frac{2(x_1 - \alpha)^2}{\underbrace{\|x\|_2^2}_{=\alpha^2} - 2\alpha x^T e_1 + \alpha^2} = \frac{2(x_1 - \alpha)^2}{2\alpha(\alpha - x_1)} = \frac{\alpha - x_1}{\alpha}$$

Mit dem Faktor $\tau = \frac{2}{v^T v}$ kann man die Householder-Transformation schreiben als

$$H = I - 2 \frac{v v^T}{v^T v} = I - \tau v v^T$$

Algorithmus 1 Housholder-Vector(LAPACK DLARFG)

Input: $x \in \mathbb{R}^n$

$$\alpha = -1 \cdot \text{sign}(x_1) \|x\|_2$$

$$\tau = \frac{\alpha - x_1}{\alpha}$$

$$v = \frac{x - \alpha e_1}{x_1 - \alpha}$$

Output: Householder-Vektor v , τ

3.2.2 Householder-Transformation anwenden

Ein aufwändiges Matrix-Matrix-Produkt kann bei der Anwendung einer Housholder-Transformation $H = I - \tau v v^T$ auf die Matrix A umgangen werden, indem man geschickt klammert.

$$HA = (I - \tau v v^T)A = A - \tau(v v^T)A = A - \tau v(v^T A)$$

Statt eines Matrix-Matrix-Produkts muss man nur ein Matrix-Vektor-Produkt und ein dyadisches Produkt berechnen.

3.2.3 QR-Zerlegung mittels Housholder-Transformationen

Um A in eine obere Dreiecksmatrix R zu transformieren, wird eine Folge von Householder-Transformationen auf A angewandt.

Zuerst wird aus der ersten Spalte der Matrix A ein Householder-Vektor berechnet, dann wird die Householder-Transformation auf die Matrix angewandt. Diese Housholder-Transformation erzeugt Nullen in der ersten Spalte unterhalb des ersten Eintrags. Damit eine obere Dreiecksmatrix entsteht, wird als nächstes die Matrix A ohne die erste Zeile und Spalte betrachtet. Aus der ersten Spalte der neu betrachteten Matrix wird wieder ein Householder-Vektor berechnet. Dieser Vektor erzeugt

eine Householder-Transformation \hat{H} . Um diese Transformation auf A anwenden zu können setzt man:

$$H_1 = (\hat{H}_1) \quad , \quad H_2 = \left(\begin{array}{c|c} I_1 & 0 \\ \hline 0 & \hat{H}_2 \end{array} \right) \quad , \quad H_i = \left(\begin{array}{c|c} I_{i-1} & 0 \\ \hline 0 & \hat{H}_i \end{array} \right)$$

I_{i-1} bezeichnet die $i - 1$ -dimensionale Einheitsmatrix, \hat{H}_i ist eine Householder-Transformation.

Diese Householder-Transformationen werden auf die Matrix angewandt. Führt man nach diesem Schema immer weiter fort, entsteht eine obere Dreiecksmatrix.

$$H_1 A = \begin{pmatrix} * & * & * & * \\ 0 & * & * & * \\ 0 & * & * & * \\ 0 & * & * & * \end{pmatrix} \quad , \quad H_2 H_1 A = \begin{pmatrix} * & * & * & * \\ 0 & * & * & * \\ 0 & 0 & * & * \\ 0 & 0 & * & * \end{pmatrix}$$

So erhält man die Faktorisierung

$$R = H_{n-1} H_{n-2} \cdot \dots \cdot H_1 A \Leftrightarrow A = (H_1 \cdot \dots \cdot H_{n-1}) R \Rightarrow Q = H_1 \cdot \dots \cdot H_{n-1}$$

Q ist das Produkt aller Householder-Transformationen. Diese Vorgehensweise führt zum Algorithmus 2.

Algorithmus 2 Ungeblockte Householder-Transformation.

Zur übersichtlicheren Beschreibung des Algorithmus werden die Bezeichnungen A_i und \hat{a}_i eingeführt. A_i zeigt auf einen Matrixblock der am i -ten Diagonalelement beginnt. \hat{a}_i zeigt auf die i -te Spalte unterhalb der Diagonalen. Matrizen sind 0-indiziert notiert.

```

1: Input:  $A \in \mathbb{R}^{m \times n}$ 
2: for  $i = 0, 1, 2, \dots, n-1$  do
3:    $(v_i, \tau_i) \leftarrow \text{householdervector}(\hat{a}_i)$ 
4:    $w \leftarrow v^T \dot{A}_i$  (dgemv)
5:    $A_i \leftarrow \tau * v * w + A_i$  (dger)
6:   if  $i < m$  then
7:      $\hat{a}_i \leftarrow v$ 
8:   end if
9: end for
10: Output:  $A$  QR zerlegt, Vektor  $\tau \in \mathbb{R}^n$ 
```

Der Algorithmus 2 überschreibt die Matrix A mit R . Aufgrund der Dreiecksstruktur

von R , können unter der Diagonale die Householder-Vektoren gespeichert werden. Die Householder-Vektoren haben die Form

$$v^{(j)} = (\underbrace{0, \dots, 0}_{j-1}, 1, v_{j+1}^{(j)}, \dots, v_m^{(j)})$$

Da die ersten $j - 1$ Einträge Null sind und der Vektor so normiert wurde, dass der j Eintrag gleich 1 ist, müssen die ersten j Einträge nicht gespeichert werden. Die Householder-Vektoren können somit unterhalb der Diagonalen gespeichert werden. Das geschieht im Algorithmus 2 in Zeile 7. Die Matrix A hat somit die Form

$$A = \begin{pmatrix} r_{1,1} & r_{1,2} & r_{1,3} \\ v_2^{(1)} & r_{2,2} & r_{2,3} \\ v_3^{(1)} & v_3^{(2)} & r_{3,3} \\ v_4^{(1)} & v_4^{(2)} & v_4^{(3)} \end{pmatrix}$$

Indem man Householder-Vektoren unterhalb der Diagonalen speichert, benötigt man keinen zusätzlichen Speicher für die Matrix Q .

Meistens ist es nicht notwendig, die Matrix Q explizit zu bestimmen, da man Q nur mit den Householder-Vektoren sehr effizient anwenden kann (siehe Kapitel 3.2.2).

3.3 Geblockte QR-Zerlegung

Ein geblockter Algorithmus ist sinnvoll, um bei großen Matrizen den Cache optimal zu nutzen.

Im Folgenden wird ein geblockter Algorithmus beschrieben wie er auch von LAPACK verwendet wird. Die entsprechende Funktion bei LAPACK heißt *DGEQRF* [8].

Die Idee beim geblockten Algorithmus ist, die Matrix in Blöcke aufzuteilen, die geblockte QR-Zerlegung für die Blöcke zu berechnen und die dabei verwendeten Householder-Transformationen auf den Rest der Matrix anzuwenden.

Betrachte dazu die Matrix $A \in \mathbb{R}^{m \times n}$ geblockt, mit einer geeigneten Blockgröße bs .

$$A = \left(\begin{array}{c|c} A_{0,0} & A_{0,bs} \\ \hline A_{bs,0} & A_{bs,bs} \end{array} \right) \quad (3.9)$$

Die Abbildung 3.2 zeigt schematisch die Partitionierung von A .

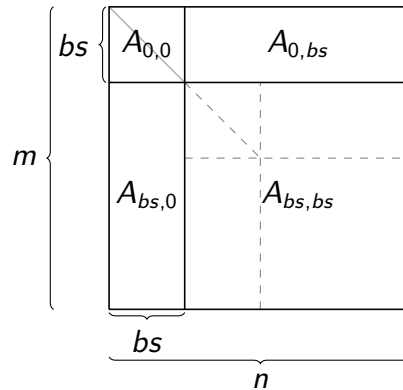


Abbildung 3.2: Aufteilung der Matrix A

Die Blockgröße bs wird so gewählt, dass die Geschwindigkeit der ungeblockten QR-Zerlegung für den Block $\begin{pmatrix} A_{0,0} \\ A_{bs,0} \end{pmatrix}$ optimal ist.

Für diesen Block wird die QR-Zerlegung mit dem ungeblockten Algorithmus (Algorithmus 2) berechnet.

$$\begin{pmatrix} A_{0,0} \\ A_{bs,0} \end{pmatrix} \leftarrow \begin{pmatrix} Q_{0,0} \backslash R_{0,0} \\ Q_{bs,0} \end{pmatrix} \quad (3.10)$$

Im Block $A_{0,0}$ steht auf und über der Diagonalen $R_{0,0}$. Unterhalb der Diagonalen und im Block $A_{bs,0}$ stehen die Householder-Vektoren.

Nun muss man die bei der ungeblockten QR-Zerlegung verwendeten Householder-Transformationen auf die restliche Matrix $\begin{pmatrix} A_{0,bs} \\ A_{bs,bs} \end{pmatrix}$ anwenden.

Das Produkt mehrerer Householder-Transformationen kann geschrieben werden als:

$$H_1 H_2 \cdots H_k = I - V T V^T \quad \text{mit} \quad H_i = I - \tau_i v_i v_i^T$$

[3]

Die Anwendung der Householder-Transformationen $I - V * T * V^T$ auf $\begin{pmatrix} A_{bs,bs} \\ A_{bs,bs} \end{pmatrix}$ erfolgt in zwei Schritten. Zuerst wird die Matrix T berechnet. Dann wird $I - V * T * V^T$ auf $\begin{pmatrix} A_{bs,bs} \\ A_{bs,bs} \end{pmatrix}$ angewandt.

$$\begin{pmatrix} A_{0,bs} \\ A_{bs,bs} \end{pmatrix} \leftarrow H^T \begin{pmatrix} A_{0,bs} \\ A_{bs,bs} \end{pmatrix} \quad (3.11)$$

Der Block $A_{bs,bs}$ wird erneut aufgeteilt. Das ist in Abbildung 3.2 gestrichelt dargestellt. Dies wird solange fortgesetzt, bis $A_{bs,bs}$ gleich der Blockgröße ist.

3.3.1 Berechnung der Matrix T

Die Matrix T wird in *LAPACK* von der Funktion *DLARFT* berechnet [7].

Sie bekommt eine Dreiecksmatrix $V \in \mathbb{R}^{m \times k}$, einen Vektor $\tau \in \mathbb{R}^k$ und eine Matrix $T \in \mathbb{R}^{k \times k}$ übergeben.

In der Dreiecksmatrix V stehen die Householder-Vektoren, im Vektor τ die zu den Householder-Vektoren gehörende τ_i .

Die Funktion berechnet eine obere Dreiecksmatrix T , so dass

$$H_1 H_2 \dots H_k = I - V T V^T \quad \text{mit} \quad H_i = I - \tau_i v_i v_i^T \quad (3.12)$$

Warum und wie das Verfahren funktioniert, wird in [3] beschrieben.

Algorithmus 3 Der Algorithmus berechnet die Matrix T so dass (3.12) gilt. Die untere Dreiecksmatrix V enthält die Householder-Vektoren. Der Vektor τ die dazugehörigen $\tau_i = \frac{2}{v_i^T v_i}$. Hinweise zur Notation: Kleine Buchstaben bezeichnen einzelne Matrixeinträge (Beispiel: $v_{i,j}$ ist der Eintrag der i-ten Zeile und j-ten Spalte der Matrix V). Die nach unten gestellten Indizes geben einen Block an, der betrachtet werden soll (Beispiel $V_{i:n,j:m}$ bezeichnet einen Block aus der Matrix V der von i-ten bis zur n-ten Zeile und von der j-ten bis zur m-ten Spalte geht).

```

1: Input  $V \in \mathbb{R}^{k \times n}, \tau \in \mathbb{R}^k, T \in \mathbb{R}^{k \times k}$ 
2: for  $i = 0, 1, 2, \dots, k$  do
3:   if  $\tau_i == 0$  then
4:      $T_{1:i,i} = 0$ 
5:   else
6:      $v_{ii} = v_{i,i}$ 
7:      $v_{i,i} = 1$ 
8:      $T_{0:i,i} = -\tau_i \cdot V_{i:n-i,0:i}^T \cdot V_{i:n-i,i}$  (dgmV)
9:      $v_{i,i} = v_{ii}$ 
10:     $T_{0:i,i} = T_{0:i,0:i} \cdot T_{0:i,i}$  (dtrmv)
11:     $t_{i,i} = \tau_i$ 
12:   end if
13: end for

```

Der Algorithmus 3 überschreibt die Matrix T folgendermaßen

$$T = \begin{pmatrix} \tau_1 & -\tau_1 \tau_2 (v_1^T v_2) & -\tau_1 \tau_2 \tau_3 (v_1^T v_2 v_2^T v_3) + \tau_1 \tau_3 (v_1^T v_3) \\ 0 & \tau_2 & -\tau_2 \tau_3 (v_2^T v_3) \\ 0 & 0 & \tau_3 \end{pmatrix}$$

Beispiel mit $k = 3$

3.3.2 Anwenden von $I - VTV^T$

Die Anwendung der Householder-Transformationen auf eine Matrix C wird in LAPACK von der Funktion *LARFB* implementiert.

Die Funktion bekommt eine untere Dreiecksmatrix $V \in \mathbb{R}^{m \times k}$, eine obere Dreiecksmatrix $T \in \mathbb{R}^{k \times k}$ und eine Matrix $C \in \mathbb{R}^{m \times n}$ übergeben.

In der Dreiecksmatrix V stehen die Householder-Vektoren und T ist die zuvor berechnete Matrix. Die Matrix C wird aktualisiert, indem die Matrix $I - VTV^T$ von rechts auf die Matrix C angewendet wird.

Ein weiterer Übergabeparameter gibt an, ob die Matrix $I - VTV^T$ transponiert werden soll. Die Funktion berechnet also

$$C \leftarrow HC = C - VTV^T C \quad \text{oder} \quad C \leftarrow H^T C = C - VT^T V^T C \quad (3.13)$$

Der Zweck der Funktion ist es, die Householder-Transformationen, die bei der Bereicherung der QR-Zerlegung für einen Block entstanden sind, auf die restliche Matrix anzuwenden. Die Abbildung 3.3 zeigt, wie die Matrix A für die Funktion partitioniert wird.

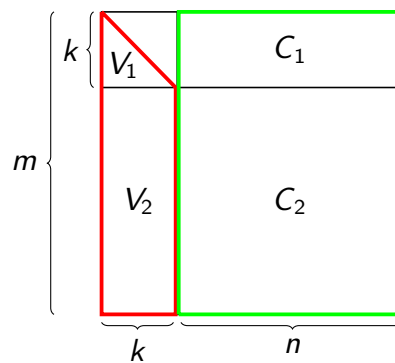


Abbildung 3.3: Partitionierung von A für larfb

Falls $m > k$, werden die Matrizen V und C aufgeteilt in

$$V = \begin{pmatrix} V_1 \\ V_2 \end{pmatrix} \quad \text{und} \quad C = \begin{pmatrix} C_1 \\ C_2 \end{pmatrix} \quad (3.14)$$

Dabei wird V genau so geteilt, dass $V_1 \in \mathbb{R}^{k \times k}$ der quadratische Dreiecksteil der Matrix ist und $V_2 \in \mathbb{R}^{(m-k) \times k}$ der Rest der Matrix. Die Matrix C wird in $C_1 \in \mathbb{R}^{k \times n}$ und $C_2 \in \mathbb{R}^{(m-k) \times n}$ aufgeteilt. Die Aufteilung ist so gewählt, dass das Matrix-Matrix-Produkt $V_1 \cdot C_1$ und $V_2 \cdot C_2$ möglich ist.

Diese Aufteilung ist notwendig, da die BLAS-Funktion `trmm` (*matrix-matrix product where one input matrix is triangular*) nur für quadratische Dreiecksmatrizen implementiert ist.

Im Fall $m = k$ ist die Aufteilung nicht notwendig, da V quadratisch ist.

Mit folgender Umformung kommt man vom (3.13) auf Algorithmus 4

$$\begin{aligned}
 C &\leftarrow C - VTV^T C \\
 C &\leftarrow C - (VTV^T C)^{TT} \\
 C &\leftarrow C - (C^T V T^T V^T)^T
 \end{aligned}$$

Die Matrizen C und V werden wie in (3.14) aufgeteilt.

$$\begin{aligned}
 \begin{pmatrix} C_1 \\ C_2 \end{pmatrix} &\leftarrow \begin{pmatrix} C_1 \\ C_2 \end{pmatrix} - \left(\begin{pmatrix} C_1 \\ C_2 \end{pmatrix}^T \cdot \begin{pmatrix} V_1 \\ V_2 \end{pmatrix} \cdot T \cdot \begin{pmatrix} V_1 \\ V_2 \end{pmatrix}^T \right)^T \\
 \begin{pmatrix} C_1 \\ C_2 \end{pmatrix} &\leftarrow \begin{pmatrix} C_1 \\ C_2 \end{pmatrix} - \left((C_1^T | C_2^T) \cdot \begin{pmatrix} V_1 \\ V_2 \end{pmatrix} \cdot T \cdot (V_1^T | V_2^T) \right)^T \\
 \begin{pmatrix} C_1 \\ C_2 \end{pmatrix} &\leftarrow \begin{pmatrix} C_1 \\ C_2 \end{pmatrix} - ((C_1^T \cdot V_1 + C_2^T \cdot V_2) \cdot T \cdot (V_1^T | V_2^T))^T
 \end{aligned}$$

$(C_1^T \cdot V_1 + C_2^T \cdot V_2)$ wird in Zeile 2-6 berechnet und das Ergebnis in W gespeichert. Die Matrix W ist eine Workspace-Matrix. In Zeile 7-11 wird $(C_1^T \cdot V_1 + C_2^T \cdot V_2) \cdot T$ berechnet, in dem T auf W multipliziert wird mit $W \leftarrow W \cdot T$.

$$\begin{aligned}
 \begin{pmatrix} C_1 \\ C_2 \end{pmatrix} &\leftarrow \begin{pmatrix} C_1 \\ C_2 \end{pmatrix} - (W \cdot (V_1^T | V_2^T))^T \\
 \begin{pmatrix} C_1 \\ C_2 \end{pmatrix} &\leftarrow \begin{pmatrix} C_1 \\ C_2 \end{pmatrix} - \begin{pmatrix} V_1 \\ V_2 \end{pmatrix} \cdot W^T \\
 \Rightarrow C_1 &\leftarrow C_1 - V_1 \cdot W^T \\
 C_2 &\leftarrow C_2 - V_2 \cdot W^T
 \end{aligned}$$

C_1 wird in Zeile 15 und 16 berechnet. C_2 wird in Zeile 13 berechnet.

Algorithmus 4 $I - VTV^T$ auf C anwenden.

Die Matrix W ist ein Workspace. Die Matrizen V und C werden geteilt in V_1, V_2 und C_1, C_2 wie oben beschrieben.

```
1: Input:  $V \in \mathbb{R}^{m \times k}, T \in \mathbb{R}^{k \times k}, C \in \mathbb{R}^{m \times n}$ 
2:  $W \leftarrow C_1^T$  (copy)
3:  $W \leftarrow W * V_1$  (trmm)
4: if  $m > k$  then
5:    $W \leftarrow W + C_2^T * V_2$  (gemm)
6: end if
7: if trans then
8:    $W \leftarrow W * T^T$  (trmm)
9: else
10:   $W \leftarrow W * T$  (trmm)
11: end if
12: if  $m > k$  then
13:   $C_2 \leftarrow C_2 - V_2 * W^T$  (gemm)
14: end if
15:  $W \leftarrow W * V_1^T$  (trmm)
16:  $C_1 \leftarrow C_1 - W^T$ 
```

4 Implementierung und Benchmarks

Es wurde der in Kapitel 3 beschriebene Algorithmus zur Berechnung der QR-Zerlegung mittels Householder-Transformation implementiert. Es wurde sich an der Implementierung von LAPACK orientiert.

Es wurde der ungeblockte und der geblockte Algorithmus implementiert. Beide Algorithmen wurden mit BLAS-Routinen der IntelMKL implementiert. Die Implementierung wurde dann mit den Algorithmen der IntelMKL verglichen.

Die Implementierung steht auf GitHub Verfügung [4].

Link: <https://github.com/Flousen/Bachelorarbeit>

4.1 Bibliothek

Die verwendete Bibliothek wurde in der Vorlesung High Performance Computing 1 entwickelt [5].

Die Bibliothek ist in C++ geschrieben. Es sind Klassen für Matrizen und Vektoren implementiert sowie einige BLAS-Routinen.

Die Matrix-Klassen erlauben den Zugriff auf Matrixblöcke. Der folgender Code soll beispielhaft den Zugriff auf Matrixblöcke veranschaulichen.

```
1 GenerelMatrix<T> A(5,5); // erzeugt Matrix-Objekt
2 init(A); // Matrix mit Zufallszahlen initialisieren
3 printf("A =\n"); print(A); // Matrix ausgeben
4 printf("A22 =\n"); print(A.block(2,2));
5 printf("A22T=\n"); print(A.block(2,2).view(Trans::view));
```

Dieser Code kann die folgende Ausgabe erzeugen.

A =

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

A22 =

13	14	15
18	19	20
23	24	25

A22T=

13	18	23
14	19	24
15	20	25

MKL Wrapper

```
14 double
15 dot(MKL_INT n, const double *x, MKL_INT incx ,
16     const double *y, MKL_INT incy)
17 {
18     return ddot(&n, x, &incx, y, &incy);
19 }
20
21 template <typename T, template<typename> class VectorX,
22           template<typename> class VectorY,
23           Require< Dense<VectorX<T>>,
24                   Dense<VectorY<T>> > = true>
25 T
26 dot(const VectorX<T> &x, const VectorY<T> &y)
27 {
28     return dot(x.length(), x.data(), x.inc(),
29               y.data(), y.inc());
30 }
```

4.1.1 Algorithmus

Beispiel: Ungeblockte QR-Implementierung vom Algorithmus 2

```

1 size_t mn = std::min(m,n);
2 DenseVector<T> work(mn);
3 for (std::size_t i = 0; i < m
4 n; ++i){
5     householderVector(A(i,i), A.col(i+1,i),tau(i));
6     if (i < n && tau(i) != 0) {
7         All = A(i,i);
8         A(i,i) = 1;
9         mv(1, A.block(i,i+1).view(hpc::matvec::Trans::view),
10            A.col(i,i), 0, work.block(i+1));
11         rank1(-tau(i), A.col(i,i), work.block(i+1),
12              A.block(i,i+1));
13         A(i,i) = All;
14     }
15 }
```

Die anderen Algorithmen wurden analog implementiert, siehe Anhang.

4.2 Fehlerschätzer

Um zu testen, ob die QR-Zerlegung korrekt ist, ist ein Fehlerschätzer notwendig. Es wurde der Fehlerschätzer von ATLAS [1] verwendet.

$$err = \frac{\|A - QR\|_i}{\|A\|_i \cdot \min(m, n) \cdot \varepsilon} \quad (4.1)$$

$\|\cdot\|_i$ ist eine passende Norm. Die Matrizen Q und R sind die QR-Zerlegung der Matrix $A \in \mathbb{R}^{m \times n}$. ε ist die kleinste darstellbare Zahl.

Die QR-Zerlegung ist gut genug, falls der Fehler kleiner 1 ist: $err < 1$.

Als Norm wurde die Zeilensummennorm $\|\cdot\|_\infty$ gewählt. Diese ist für eine Matrix $A \in \mathbb{R}^{m \times n}$ gegeben durch

$$\|A\|_\infty = \max_{i=1,\dots,m} \sum_{j=1}^n |a_{ij}|$$

Diese Norm wurde gewählt, da sie für zeilenweise gespeicherte Matrizen effizient berechnet werden kann.

4.3 Benchmarks

4.3.1 Aufwand

Der Aufwand zur Berechnung der QR-Zerlegung mittels Householder-Transformation einer Matrix $A \in \mathbb{R}^{m \times n}$, $m \geq n$ ist bei ATLAS [1] angegeben mit

$$\#QR = n \cdot \left(\frac{23}{6} + m + \frac{n}{2} + n \cdot \left(\frac{m-n}{3} \right) + n \cdot \left(\frac{5}{6} + \frac{n}{2} + m - \frac{n}{3} \right) \right) = \mathcal{O}(n^2 m)$$

$\#QR$ bezeichnet die Anzahl der Rechenoperationen, die zur Berechnung der QR-Zerlegung notwendig sind.

4.3.2 FLOPS

FLPOS (*floating point operations per second*) geben an, wie viele Fließkomma-Operationen pro Sekunde ausgeführt werden.

$$\text{FLPOS} = \frac{\#}{\Delta t}$$

Δt bezeichnet die Zeit in Sekunden und $\#$ die Anzahl der Rechenoperationen, die zur Berechnung benötigt werden.

4.3.3 Test System

Getestet wurde auf einem System mit einer Intel i5-3470 CPU mit 3.20GHz.

Die theoretische *peak performance* errechnet sich, auf dieser Architektur, aus der Taktrate mal die Registerbreite mal 2.

Die CPU des Testsystems hat eine Taktrate von 3.20GHz. Die AVX-Register sind 256-Bit groß. Darin haben 4 double Platz.

Taktrate · Registerbreite · 2 = 3,20 GHz · 4 · 2 = 25,6 GFLOPs

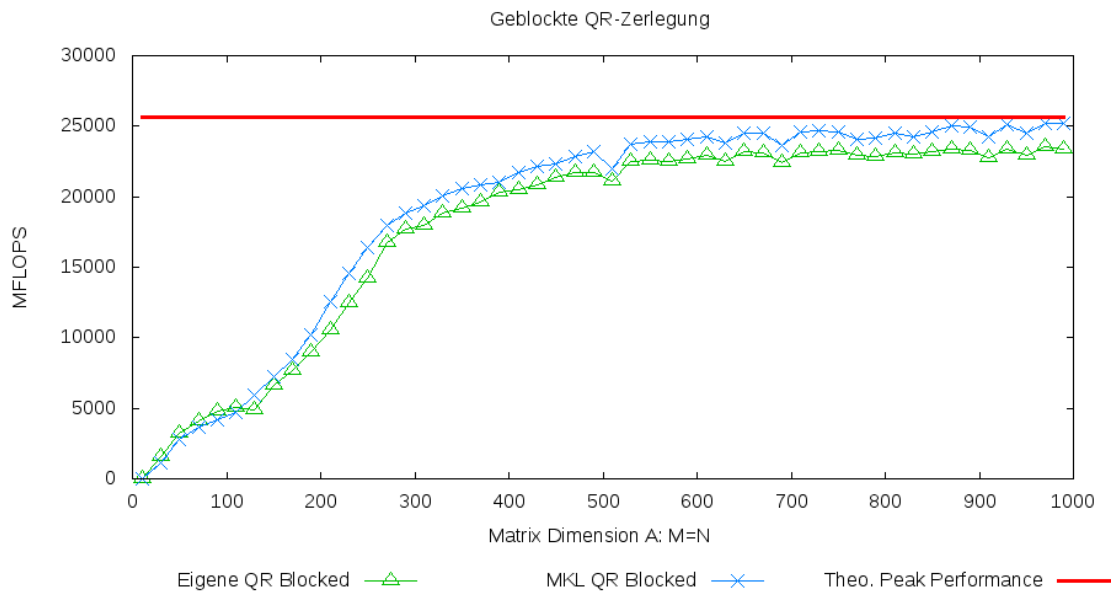


Abbildung 4.1: Benchmark geblockte QR-Zerlegung

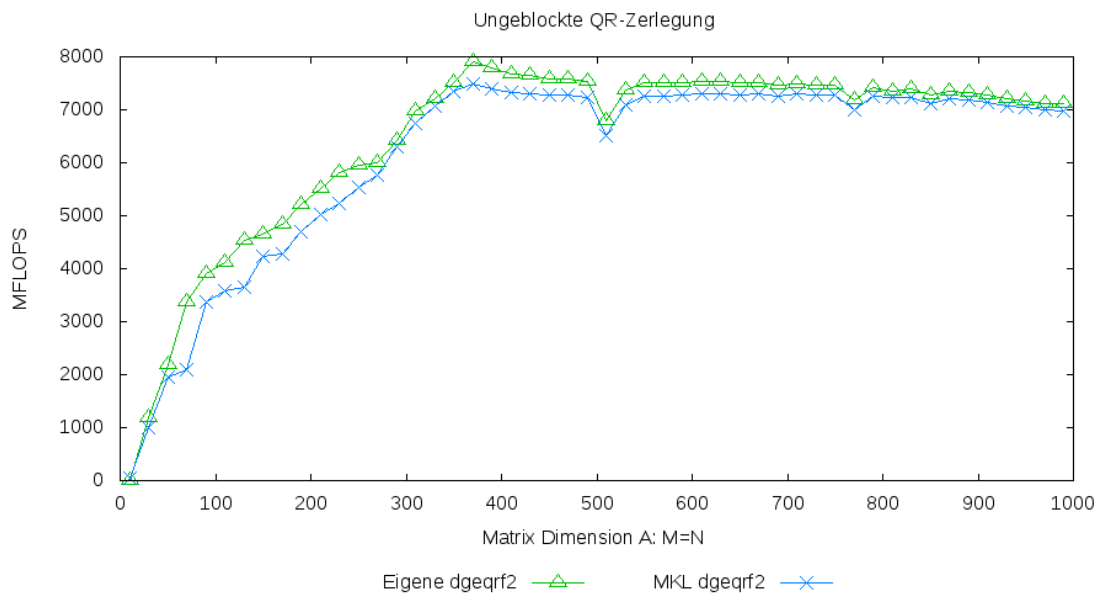


Abbildung 4.2: Benchmark ungeblockte QR-Zerlegung

A Block Reflector

Das Produkt aus Householder-Transformationen $H_1 \cdot \dots \cdot H_n$ lässt sich schreiben als

$$H_1 \cdot \dots \cdot H_n = I - VTV^T$$

mit einer unteren Dreiecksmatrix $V \in \mathbb{R}^{m \times n}$, die die Householder-Vektoren enthält und eine obere Dreiecksmatrix $T \in \mathbb{R}^{n \times n}$ [3]

Beweis:

$n=2$ vorwärts:

$$\begin{aligned} H_1 H_2 x &= (I - \tau_1 v_1 v_1^T)(I - \tau_2 v_2 v_2^T)x \\ &= (I - \tau_1 v_1 v_1^T - \tau_2 v_2 v_2^T + \tau_1 v_1 v_1^T \tau_2 v_2 v_2^T)x \\ &= x - \tau_1 v_1 v_1^T x - \tau_2 v_2 v_2^T x + \tau_1 \tau_2 v_1 (v_1^T v_2) v_2^T x \\ &= x - \tau_1 v_1 v_1^T x - \tau_2 v_2 v_2^T x + \tau_1 \tau_2 (v_1^T v_2) v_1 v_2^T x \end{aligned}$$

rückwärts:

$$\begin{aligned} H_{1,2} x &= (I - VTV^T)x = x - VTV^T x \\ &= x - (v_1, v_2) \begin{pmatrix} a & b \\ 0 & c \end{pmatrix} \begin{pmatrix} v_1^T \\ v_2^T \end{pmatrix} x \\ &= x - (v_1, v_2) \begin{pmatrix} a & b \\ 0 & c \end{pmatrix} \begin{pmatrix} v_1^T x \\ v_2^T x \end{pmatrix} \\ &= x - (v_1, v_2) \begin{pmatrix} av_1^T x + bv_2^T x \\ cv_2^T x \end{pmatrix} \\ &= x - v_1(av_1^T x + bv_2^T x) - v_2(cv_2^T x) \\ &= x - av_1 v_1^T x - bv_1 v_2^T x - cv_2 v_2^T x \end{aligned}$$

Koeffizientenvergleich

$$a = \tau_1$$

$$b = -\tau_1 \tau_2 (v_1^T v_2)$$

$$c = \tau_2$$

$$T = \begin{pmatrix} \tau_1 & -\tau_1 \tau_2 (v_1^T v_2) \\ 0 & \tau_2 \end{pmatrix}$$

n=3

vorwärts:

$$\begin{aligned}
 H_1 H_2 H_3 x &= (I - \tau_1 v_1 v_1^T)(I - \tau_2 v_2 v_2^T)(I - \tau_3 v_3 v_3^T)x \\
 &= (I - \tau_1 v_1 v_1^T - \tau_2 v_2 v_2^T + \tau_1 v_1 v_1^T \tau_2 v_2 v_2^T)(I - \tau_3 v_3 v_3^T)x \\
 &= (I - \tau_1 v_1 v_1^T - \tau_2 v_2 v_2^T - \tau_3 v_3 v_3^T \\
 &\quad + \tau_1 v_1 v_1^T \tau_2 v_2 v_2^T + \tau_1 v_1 v_1^T \tau_3 v_3 v_3^T + \tau_2 v_2 v_2^T \tau_3 v_3 v_3^T \\
 &\quad - \tau_1 v_1 v_1^T \tau_2 v_2 v_2^T \tau_3 v_3 v_3^T)x \\
 &= x - \tau_1 v_1 v_1^T x - \tau_2 v_2 v_2^T x - \tau_3 v_3 v_3^T x \\
 &\quad + \tau_1 \tau_2 (v_1^T v_2) v_1 v_2^T x + \tau_1 \tau_3 (v_1^T v_3) v_1 v_3^T x + \tau_2 \tau_3 (v_2^T v_3) v_2 v_3^T x \\
 &\quad - \tau_1 \tau_2 \tau_3 (v_1^T v_2 v_2^T v_3) v_1 v_3^T x
 \end{aligned}$$

rückwärts:

$$\begin{aligned}
 H_{1,2,3} x &= (I - V T V^T) x = x - V T V^T x \\
 &= x - (v_1, v_2, v_3) \begin{pmatrix} a & b & c \\ 0 & d & e \\ 0 & 0 & f \end{pmatrix} \begin{pmatrix} v_1^T \\ v_2^T \\ v_3^T \end{pmatrix} x \\
 &= x - (v_1, v_2, v_3) \begin{pmatrix} a & b & c \\ 0 & d & e \\ 0 & 0 & f \end{pmatrix} \begin{pmatrix} v_1^T x \\ v_2^T x \\ v_3^T x \end{pmatrix} \\
 &= x - (v_1, v_2, v_3) \begin{pmatrix} a v_1^T x + b v_2^T x + c v_3^T x \\ d v_2^T x + e v_3^T x \\ f v_3^T x \end{pmatrix} \\
 &= x - v_1 (a v_1^T x + b v_2^T x + c v_3^T x) \\
 &\quad - v_2 (d v_2^T x + e v_3^T x) \\
 &\quad - v_3 (f v_3^T x) \\
 &= x - a v_1 v_1^T x - b v_1 v_2^T x - c v_1 v_3^T x \\
 &\quad - d v_2 v_2^T x - e v_2 v_3^T x \\
 &\quad - f v_3 v_3^T x
 \end{aligned}$$

Koeffizienten Vergleich

$$a = \tau_1$$

$$b = -\tau_1 \tau_2 (v_1^T v_2)$$

$$c = -\tau_1 \tau_2 \tau_3 (v_1^T v_2 v_2^T v_3) + \tau_1 \tau_3 (v_1^T v_3)$$

$$d = \tau_2$$

$$e = -\tau_2 \tau_3 (v_2^T v_3)$$

$$f = \tau_3$$

$$T = \begin{pmatrix} a & b & c \\ 0 & d & e \\ 0 & 0 & f \end{pmatrix} = \begin{pmatrix} \tau_1 & -\tau_1 \tau_2 (v_1^T v_2) & -\tau_1 \tau_2 \tau_3 (v_1^T v_2 v_2^T v_3) + \tau_1 \tau_3 (v_1^T v_3) \\ 0 & \tau_2 & -\tau_2 \tau_3 (v_2^T v_3) \\ 0 & 0 & \tau_3 \end{pmatrix}$$

Mit Induktion kann man zeigen... siehe paper Im Paper wird gezeigt wie man das verallgemeinern kann. [3]

Literaturverzeichnis

- [1] *Automatically Tuned Linear Algebra Software (ATLAS)*. <http://math-atlas.sourceforge.net/>, . – [Online; zugegriffen 12-06-2018]
- [2] INTEL: *Intel MKL*. <https://software.intel.com/en-us/mkl>, . – [Online; zugegriffen 16-07-2018]
- [3] JOFFRAIN, Thierry ; LOW, Tze M. ; QUINTANA-ORTÍ, Enrique S. ; GEIJN, Robert van d. ; ZEE, Field G. V.: Accumulating Householder Transformations, Revisited. In: *ACM Trans. Math. Softw.* 32 (2006), Juni, Nr. 2, 169–179. <http://dx.doi.org/10.1145/1141885.1141886>. – DOI 10.1145/1141885.1141886. – ISSN 0098–3500
- [4] KRÖTZ, Florian: *Implementierung*. <https://github.com/Flousen/> Bachelorarbeit,
- [5] LEHN MICHAEL, Borchert A.: *Vorlesung High Performance Computing 1*. <http://www.mathematik.uni-ulm.de/numerik/hpc/ws17/>, 2017. – [Online; zugegriffen 31-05-2018]
- [6] STEFAN A. FUNKEN, Karsten U.: *Einführung in die Numerische Lineare Algebra*. Ulm, Germany, 2016
- [7] TENNESSEE, Univ. of California B. o. ; LTD., NAG: *DLARFT forms the triangular factor T of a real block reflector H of order n, which is defined as a product of k elementary reflectors*. <http://www.netlib.org/lapack/explore-3.1.1-html/dgeqrf.f.html#DGEQRF.1>, 2006. – [Online; zugegriffen 12-06-2018]
- [8] TENNESSEE, Univ. of California B. o. ; LTD., NAG: *LAPACK blocked QR*. <http://www.netlib.org/lapack/explore-3.1.1-html/dgeqrf.f.html>, 2006. – [Online; zugegriffen 31-01-2018]
- [9] TENNESSEE, Univ. of California B. o. ; LTD., NAG: *LAPACK unblocked QR*. <http://www.netlib.org/lapack/explore-3.1.1-html/dgeqr2.f.html>, 2006. – [Online; zugegriffen 31-01-2018]

- [10] TENNESSEE, University of: *BLAS Technical Forum*. <http://www.netlib.org/blas/blast-forum/>, 2001. – [Online; zugegriffen 03-06-2018]

Name: Florian Krötz

Matrikelnummer: 884948

Erklärung

Ich erkläre, dass ich die Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den

Florian Krötz