LEHRSTUHL FÜR RECHNERARCHITEKTUR UND PARALLELE SYSTEME

Praktikum Rechnerarchitektur

Gruppe 196 – Abgabe zu Aufgabe A328

Sommersemester 2020/21

Aleksandre Kandelaki Matthias Staritz Benjamin Liertz

1 Einleitung

Im Folgenden wird im Rahmen einer Projektarbeit im Fach Einführung in die Rechnerarchitektur an der TU München ein in der linearen Algebra häufig benutztes Verfahren genauer beschrieben, implementiert und enstprechend dokumentiert.

Das Verfahren LU-Zerlegung, auch LR-Zerlegung genannt, bietet eine Möglichkeit per Algorithmus lineare Gleichungssysteme zu lösen und Matrixinverse zu bestimmen. Dies findet z.B. Anwendung bei der Berechnung des Stromflusses in Schaltkreisen. Mit der LU-Zerlegung kann man, wenn man die Wiederstände einzelner Leitungen gegeben hat, den Stromfluss an jedem Wiederstand berechnen [5]. Dazu liefert die LU-Zerlegung für jedes eindeutig lösbare Gleichungssystem $A \cdot x = b$, also wenn A regulär ist, zwei Dreiecksmatrizen L und U und eine Pivot-Matrix P, wobei $P \cdot L \cdot U = A$ ergibt[4, 37]. Hierbei haben die Matrizen besondere Eigenschaften. L hat in allen Einträgen oberhalb der Diagonalen die Werte 0 (siehe Matrix 1). U hingegen hat in allen Einträgen unterhalb der Diagonalen die Werte 0 (siehe Matrix 2)[1, 5]. Die Matrix P ist eine Einheitsmatrix mit ggf. vertauschten Zeilen[2, 59].

$$Untere\ Dreiecksmatrix: \begin{bmatrix} l_{1,1} & 0 & 0 & \cdots & 0 \\ l_{2,1} & l_{2,2} & 0 & & \vdots \\ l_{3,1} & l_{3,2} & l_{3,3} & \ddots & \vdots \\ \vdots & \vdots & \vdots & \ddots & 0 \\ l_{n,1} & l_{n,2} & l_{n,3} & \cdots & l_{n,n} \end{bmatrix} \tag{1}$$

$$Obere\ Dreiecksmatrix: \begin{bmatrix} u_{1,1} & u_{1,2} & u_{1,3} & \cdots & u_{1,n} \\ 0 & u_{2,2} & u_{2,3} & \cdots & u_{2,n} \\ 0 & 0 & u_{3,3} & \cdots & u_{3,n} \\ \vdots & \vdots & & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & u_{n,n} \end{bmatrix} \tag{2}$$

Ein anschauliches Beispiel hierzu ist das lineare Gleichungssystem:

$$0x_1 + 3x_2 + 5x_3 + 7x_4 = 0 \tag{3}$$
$$2x_1 + 6x_2 + 10x_3 + 14x_4 = 0 \tag{4}$$
$$-4x_1 + 12x_2 + 15x_3 + -21x_4 = 0 \tag{5}$$
$$6x_1 + 9x_2 + -5x_3 + -7x_4 = 0 \tag{6}$$

Welches auch durch folgende Koeffizientenmatrix dargestellt werden kann:

$$A = \begin{bmatrix} 0 & 3 & 5 & 7 \\ 2 & 6 & 10 & 14 \\ -4 & 12 & 15 & -21 \\ 6 & 9 & -5 & -7 \end{bmatrix} \tag{7}$$

Nach Anwendung der LU-Zerlegung ergeben sich folgende Matrizen:

$$A = L \cdot U \cdot P$$

$$\begin{bmatrix} 0 & 3 & 5 & 7 \\ 2 & 6 & 10 & 14 \\ -4 & 12 & 15 & -21 \\ 6 & 9 & -5 & -7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -2 & 8 & 1 & 0 \\ 3 & -3 & 4 & 1 \end{bmatrix} \cdot \begin{bmatrix} 2 & 6 & 10 & 14 \\ 0 & 3 & 5 & 7 \\ 0 & 0 & -5 & -49 \\ 0 & 0 & 0 & 168 \end{bmatrix} \cdot \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{8}$$

2 Lösungsansatz

Zur Durchführung dieser Zerlegung haben wir uns entschieden ein Programm auf Basis des gaußschen Eliminationsverfahren zu enwickeln. Das gaußsche Eliminationsverfahren ändert die Einträge einer Matrix unter Verwendung elementarer Zeilenoperationen. Die verwendeten elementaren Zeilenoperationen sind das Tauschen zweier Zeilen (siehe Gleichung 9) und das Addieren von Vielfachen einer Zeile auf eine Andere (siehe Gleichung 10). Dabei wird das Gleichungssystem $A \cdot x = b$ verändert, die Lösung bleibt aber erhalten.

Mit diesem Verfahren wird die bereits genannte obere Dreiecksmatrix U generiert. So erhalten wir unsere U Matrix. Um nun auch die Matrizen L und P zu erhalten, müssen wir nur alle Schritte die zur Generierung der U Matrix beigetragen haben dokumentieren. Jedes mal, wenn wir eine Zeile auf eine Andere addieren, wird dies in der L Matrix festgehalten, indem dieselbe Operation mit invertiertem Vorzeichen auf dieser L Matrix durchgeführt wird. Die Zeilenvertauschung werden auf ähnliche Art in P festgehalten. Um nun sicher eine gültige L Matrix zu erhalten, welche oberhalb der Diagonalen nur die 0 als Einträge hat und somit die gesuchte untere Dreiecksmatrix bildet, folgt der

Algorithmus strikt der Vorgehensweise des Gauß Algorithmus.

Gehen wir davon aus, dass für jede der Matrizen L, U und P ein Speicherplatz vorhanden ist. Dann müssen zu Beginn die für den Algorithmus notwendigen Startwerte dort abgelegt werden. Für L und P sind das reguläre Einheitsmatrizen, während in U zu Beginn die Eingabematrix A gespeichert wird. Nun werden die Einträge unterhalb der Diagonalen in U spaltenweise von links nach rechts auf 0 gesetzt indem man immer ein genau passendes vielfaches einer oberen Zeile von allen darunterliegenden Zeilen abzieht. Die Wahl der Zeile hängt davon ab, welche Spalte gerade auf 0 gesetzt werden soll. Bei Spalte 1 ist es Zeile 1 bei Spalte 2 Zeile 2 und so weiter. Da man sich hierbei von oben nach unten bzw. von links nach rechts vorarbeitet, generiert man schrittweise die gewünschten L und U Matrizen. Nun kann es aber vorkommen, dass in der Zeile welche von den anderen Zeilen subtrahiert werden soll an dem Index der zu bearbeitenden Spalte eine 0 steht. Dies birgt das Problem, dass nun kein vielfaches dieser Zeile jemals die anderen Einträge der Spalte durch Subtraktion auf 0 bringen kann, da $0 \cdot x = 0$. Um dieses Problem zu vermeiden haben wir uns entschieden, unabhängig von der Eingabe, bevor wir mit der Nullung einer Zeile beginnen, immer die Zeile mit dem betragsmäßig größten Eintrag am entsprechenden Index nach oben zu tauschen (siehe Grafik). Dadurch wird garantiert, dass der Algorithmus bei einer regulären, quadratischen Matrix erfolgreich durchläuft. Im Abschnitt zur Genauigkeit wird noch genau darauf eingegeangen, warum wir immer die Zeile mit dem größten Eintrag nach oben tauschen und nicht nur irgendeine Zeile mit Eintrag ungleich 0. Zusätzlich sei gesagt, dass nicht jede Matrixzerlegung den Schritt der Zeilenvertauschung benötigt und, dass der Algorithmus auf diese Art oft unnötige Vertauschungen durchführt was Potential für Optimierungen bieten könnte[1, 10-13].

Hier ein Beispiel für Zeilenvertauschungen:

$$\begin{bmatrix} 0 & 3 & 5 & 7 \\ 2 & 6 & 10 & 14 \\ -4 & 12 & 15 & -21 \\ 6 & 9 & -3 & -9 \end{bmatrix} \begin{matrix} \leftarrow \\ \\ \\ \leftarrow \end{matrix} \rightarrow \begin{bmatrix} 6 & 9 & -3 & -9 \\ 2 & 6 & 10 & 14 \\ -4 & 12 & 15 & -21 \\ 0 & 3 & 5 & 7 \end{bmatrix} \tag{9}$$

Hier ein Beispiel für Zeilenaddition:

$$\begin{bmatrix} 6 & 9 & -3 & -9 \\ 2 & 6 & 10 & 14 \\ -4 & 12 & 15 & -21 \\ 0 & 3 & 5 & 7 \end{bmatrix} \begin{matrix} \\ \overset{\frac{2}{3}}{} \\ \\ \leftarrow + \end{matrix} \rightarrow \begin{bmatrix} 6 & 9 & -3 & -9 \\ 2 & 6 & 10 & 14 \\ 0 & 15 & 14 & -24 \\ 0 & 3 & 5 & 7 \end{bmatrix} \tag{10}$$

2.1 Stack vs. Heap

Bei der Wahl des Speicherortes muss noch eine weitere Gegebenheit beachtet werden und zwar haben wir hier die Wahl zwischen dem Heap und dem Stack. Aufgrund der besseren Zugriffszeiten würden wir den Stack als Speicherort präferieren, jedoch werden bei Linux-Betriebssystemen meistens nur ca. 8MB Speicherplatz für die Stackallokation zur Verfügung gestellt. Dies birgt das Problem, dass es bei großen Matrizen zu einem Segmentation Fault kommen kann, sofern der Stack als Speicher verwendet wird. Um trozdem die bessere Performance des Stacks nutzen zu können, haben wir eine Fallunterscheidung implementiert, welche je nach Größe der Matrix die Zerlegung auf dem Stack zerlegt oder den Speicherplatz auf dem Heap allozieren. Die Maximalgröße einer Eingabematrix, deren Zerlegung auf dem Stack durchgeführt werden kann, berechnen wir folgendermaßen: Die Größe der Eingabematrix A muss 4 mal alloziert werden, da temporär A, L, U und P abgespeichert werdem müssen. Der in der Methode übergebene Parameter n gibt die Anzahl Zeilen / Spalten der Matrix an, da unsere Matrizen immer quadratisch sein müssen. Dadurch ergibt sich eine Anzahl Einträge pro Matrix von $n \cdot n$. Jeder dieser Einträge ist in unserem Fall ein Floating Point Wert und somit 4 Bytes groß. Daraus ergibt sich die Formel 11, welche die benötigte Anzahl Bytes in Abhängigkeit unserer Eingabegröße n berechnet.

$$bytes = 4 \cdot 4 \cdot n \cdot n \tag{11}$$

Damit folgt, dass für eine Nutzung des Stacks $n \leq 707$ sein muss (12).

$$4 \cdot 4 \cdot n \cdot n \leq 8 \cdot 10^6 \tag{12}$$

Um etwas Spielraum zu haben haben wir uns entschieden schon ab einem $n \geq 700$ auf dem Heap zu allozieren.

2.2 Verschiedene Implementierungen

Wie eben schon erwähnt ist uns aufgefallen, dass die LU Zerlegung vieler Matrizen auch noch gelöst werden kann, wenn man die Pivotisierung weg lässt. In der Theorie könnte man dadurch einen Performancegewinn erzielen. Um dies zu überprüfen, haben wir eine Vergleichsimplementierung erstellt, welche Pivotisierungen nicht berücksichtigt. Diese kann dann zwar nicht mehr jede Matrix zerlegen, ist aber potentiell schneller. Ansonsten ist diese Implementierung analog zu obigen Ansatz.
Den Ansatz mit Pivotisierung haben wir auf 4 verschiedene Arten umgesetzt. Einmal als einfaches, lineares C Programm und einmal als C Programm unter Verwendung von Intrinsics. Ebenso haben wir zwei verschiedene Assembler Implementierungen geschaffen. Wieder eine lineare Implementierung und eine vektorisierte Version mit Hilfe von SIMD Operationen. Auf die speziefischen Unterschiede wird im Abschnitt 4 noch genauer eingegangen.

3 Genauigkeit

In diesen Abschnitt wird die Genauigkeit unserer Implementierung der LU-Zerlegung analysiert und anhand passender Beispiele demonstriert und getestet. Wir haben uns für die Genauigkeitsanalyse entschieden, da diese ein großes Problem bei der LU-Zerlegung darstellt, besonders wenn der Algorithmus ohne Pivotisierung arbeitet. In diesen Fall handelt es sich um einen instabilen Algorithmus und es kann zu komplett falschen Ergebnissen kommen. Das Problem der Genauigkeit ist auf die Kondition der Matrix A und auf die endliche Rechengenauigkeit von Computern zurückzuführen[3, 62-63].

Im folgenden arbeiten wir mit Floats. Floats sind Gleitkommazahlen mit einer Größe bzw. Genauigkeit von 32 Bit und können neben einem sehr großen Wertebereich auch sehr genau Zahlen darstellen. Dies wird durch eine flexible Position des Kommas erreicht. Dennoch ist die Genauigkeit der Gleitkommazahlen allein durch die Endlichkeit an Rechenleistung und Speicher begrenzt. Somit beträgt die Genauigkeit im Folgenden immer ungefähr 8 Stellen. Aber auch bei Zahlen wie 0,2 kommt es schon zu Rundungsfehlern, da diese in Gleitkommadarstellung im Binärsystem unendlich viele Stellen benötigen würde. Beim Darstellen von Zahlen als Gleitkommazahlen im Computer kommt es allgemein immer zu Störungen. So also auch in unserem Programm. z.B. beim Einlesen von A.
Neben den Darstellungsproblemen treten beim Rechnen mit Zahlen in Gleitkommadarstellung aber auch die Probleme der Absorption und Auslöschung auf, welche noch sehr viel größere Störungen bewirken können. Problematisch wird dies, weil in der LU-Zerlegung $\frac{n^2}{2} - \frac{n}{2}$ Additionen und genau so viele Multiplikationen auftreten[3, 51].

3.1 Kondition linearer Gleichungssysteme

Durch die Gleitkommadarstellung kommt es nun also bei der LU-Zerlegung immer und unvermeidlich zu einer Vielzahl an Rundungsfehlern. Diese führen zu Störungen im Gleichungssystem $Ax = b$. In wie fern diese Störungen der Eingabedateien auch die Lösung des Problems stören, nennt man Kondition[3, 46]. Die Frage ist nun ob das Gleichungssystem gut oder schlecht konditioniert ist?
Hierzu ein Beispiel:

$$\begin{bmatrix} 1 & 1 \\ 1 & 1-\epsilon \end{bmatrix} \cdot x = \begin{bmatrix} 4 \\ 4-\epsilon \end{bmatrix} = b \qquad x = \begin{bmatrix} 3 \\ 1 \end{bmatrix} \tag{13}$$

Störung der rechten Seite mit $0 < \epsilon \ll 1$:

$$\bar{b} = b + \begin{bmatrix} \epsilon \\ -\epsilon \end{bmatrix} = \begin{bmatrix} 4+\epsilon \\ 4-2\epsilon \end{bmatrix} \qquad \bar{x} = \begin{bmatrix} 1+\epsilon \\ 3 \end{bmatrix} \tag{14}$$

Hier lässt sich sehen, dass eine kleine Störung in b dazu führen kann, dass das Gleichungssystem eine stark abweichende Lösung hat. Dieses lineares Gleichungssystem ist also schlecht konditioniert. In der LU-Zerlegung stellt dies nun ein großes Problem dar, da hier durch die Rundungsfehler einige Störungen in Matrix A auftreten.

Um zu verstehen, wie und ob die LU-Zerlegung verbessert werden kann, muss man zuerst das Problem weiter analysieren und verstehen, wie die Lösung des linearen Gleichungssystem von Störungen der Eingabe abhängt. Um die Größe einer Störung und die Abhängigkeit der Lösung von der Störung messen zu können, braucht man ein Maß. Hierfür bietet sich das Konzept der Normen an. Es wird aber auch die Norm für Matrizen benötigt, diese nennt man die zugehörige Matrizen Norm, welche wie folgt definiert ist:
es sei $||.||$ eine beliebige Norm auf R^n dann ist

$$||A|| := sup_{x \neq 0} \frac{||Ax||}{||x||} \; \forall \; A \; \epsilon \; R^{n \times n} \tag{15}$$

wodurch dann $||Ax||$ wie folgt abgeschätzt werden kann

$$||Ax|| \leq ||A|| \cdot ||x|| \; \forall \; x \; \epsilon \; R^n \tag{16}$$

Mit dieser Definition einer Norm lässt sich nun die Kondition eines linearen Gleichungssystems genauer untersuchen[3, 47- 48]. Sei $Ax = b$ unser lineares Gleichungssystem, welches leicht in b gestört wird, sodass nun $A\bar{x} = \bar{b}$ gilt. Der relative Fehler zur Größe x entspricht also $\frac{||x-\bar{x}|}{||x||}$ und die relative Störung $\frac{||(b-\bar{b})||}{||b||}$. Dadurch ergibt sich nun

$$\frac{||x - \bar{x}||}{||x||} \leq \frac{||b||}{||x||} \cdot ||A^{-1}|| \cdot \frac{||b - \bar{b}||}{||b||} \leq ||A|| \cdot ||A^{-1}|| \cdot \frac{||(b - \bar{b})||}{||b||} \tag{17}$$

Hier definiert man nun $||A|| \cdot ||A^{-1}||$ als Kondition von A. Anders gesagt wird der relative Fehler durch die Kondition von A mal der relativen Störung von oben beschränkt. Daraus lässt sich schließen, dass wenn die relative Störung klein ist, der relative Fehler nicht auch zwingend klein ist, sondern nur dann, wenn die Kondition von A auch klein ist[3, 47]. Jetzt haben wir aber nur b gestört, wenn man nun auch die Matrix A stört, dann ist unser gestörtes System nun $\bar{A}\bar{x} = \bar{b}$. Für dieses gilt nun folgender Satz, wenn $ea \cdot cond(A) < 1$ und $\frac{||A-\bar{A}||}{||A||} \leq ea$ und $\frac{||b-\bar{b}||}{||b||} \leq eb$, wobei ea und eb hier den Störungen in A und b entsprechen, ist:

$$\frac{||x - \bar{x}||}{||x||} \leq \frac{cond(A)}{1 - ea \cdot cond(A)} \cdot (ea + eb) \tag{18}$$

Es ist nun also sicher zu sagen, dass bei einer kleinen relativen Störung des linearen Gleichungssystems, auch die Kondition von A klein sein muss, damit der relative Fehler klein bleibt, aber auch dass ein gewisser relativer Fehler unvermeidbar ist[3, 48-49]. Um dies zu veranschaulichen haben wir auch zwei Test-Matrizen erstellt. Eine hat eine sehr hohe Kondition und ist in der Datei big_cond.txt gespeichert. Führt man nun mit dem Befehl "./main –input=big_cond.txt -v 4" die LU-Zerlegung aus, und setzt dann die Matrix U in einen programmierbaren Taschenrechner ein, ergibt sich für $b = (1;\ 2;\ 3;\ 4)$ das Ergebnis $(2756,2307;\ 58671,9527;\ 275884,6061;\ 443340,9728;\ 227272,7273)$. Eigentlich sollte aber $x = (156,1544;\ 3459,1659;\ 16979,0842;\ 28390,1576;\ 15061,3944)$ heraus kommen. Hier ist gut zu sehen das die Rundungsfehler in der LU-Zerlegung, auch wenn sie relativ

klein sind, zu sehr großen Fehlern in x geführt haben, da die Kondition der Matrix A so groß ist. Nimmt man nun aber als Beispiel die Matrix aus der Datei low_cond.txt so erhält man als Lösung für $b = (1;\ 2;\ 3)(0,00012746;\ 0,00043313;\ 0,00008111)$, wobei die Lösung auch $(0,00018580;\ 0,00005654;\ 0,00002588)$ seien sollte. Hier weicht das Ergebnis nun nur sehr gering ab, obwohl Störungen in derselben Größenordnung aufgetreten sind, hier ist aber die Kondition der Matrix A klein gewesen.

3.2 Stabilität

Die Frage ist nun, ob die LU-Zerlegung stabil ist. Stabil ist ein Algorithmus, wenn er bei der Störung der Eingabedaten nur einen relativen Fehler macht, der in der selben Größenordnung liegt wie der unvermeidbare relative Fehler des Problems an sich. Allgemein heißt ein Algorithmus stabil im Sinne der Rückwärtsanalyse, falls $\hat{x}$ die exakte Lösung von $\bar{A}x = b$ ist mit

$$\frac{||A - \bar{A}||}{||A||} \leq C\ eps \qquad \frac{||b - \bar{b}||}{||b||} \leq C\ eps \tag{19}$$

mit einem nicht all zu großem C, wobei eps für die Maschinengenauigkeit steht. Die LU-Zerlegung ermittelt nun, selbst wenn die Eingabe ohne Rundungsfehler erfolgt, $\hat{L}$ und $\hat{U}$ mit $\hat{x}$ als Ergebnis, da Rundungsfehler bei dem Verfahren auftreten. $\hat{x}$ ist dann die korrekte Lösung des linearen Gleichungssystems $\bar{A}x = b$ mit Störung in $\bar{A}$ bei der folgende Ungleichung gilt:

$$|A - \bar{A}| \leq 2n\ eps\ |\bar{L}||\bar{R}| + O(eps^2) \tag{20}$$

Hierbei sind die Beträge und das $\leq$ komponentenweise zu verstehen. Nun wird ersichtlich, dass der Abstand zwischen A und $\bar{A}$ auch von $|\hat{L}| \cdot |\hat{U}|$ abhängt. Dies wird problematisch, da der relative Abstand von A zu $\bar{A}$ eigentlich durch $C \cdot eps$ abgeschätzt werden sollte. $|\hat{L}|$ und $|\hat{U}|$ lässt sich jedoch nicht sinnvoll beschränken, weshalb C hier auch sehr groß werden kann.Folglich ist die LU-Zerlegung ohne Spaltenpivotwahl instabil[6, 75-79]. Zu sehen ist dies auch am Beispiel mit einer Matrix A in der Datei stabil.txt die auch wieder eine kleine Kondition hat. Heißt also, wenn die LU-Zerlegung ohne Pivotisierung stabil wäre, müsste hier das errechnete x durch U auch sehr nahe an der korrekten Lösung liegen. Jedoch wenn man nun für $b = (1;\ 2;\ 3)$ diese Matrix mit der LU-Zerlegung ohne Pivotisierung (-v 4) durchführt, erhält man für $x = (100000,00027239;\ 0,00005655;\ -0,00000503)$ obwohl die korrekte Lösung $(0,00018303;\ 0,00005651;\ 0,00042160)$ ist.

Es lässt sich jedoch durch die Spaltenpivotisierung der Faktor $|\hat{L}|$ begrenzen. Dies liegt daran, dass nun für alle $i, j = 0, ..., n\ |I_{i,j}| \leq 1$ gilt. Bei $|\hat{U}|$ lässt sich weiterhin keine gute Abschätzung finden und so gilt nun folgende Ungleichung:

$$\frac{||A - \bar{A}||_\infty}{||A||_\infty} \leq 2n^3\ eps\ \frac{max_{i,j}|\hat{u}_{ij}|}{max_{i,j}|a_{ij}|} + O(eps^2) \tag{21}$$

Hier ist nun der Quotient das Problem bzw. entscheidet über die Stabilität der LU- Zerlegung. Für diesen kann jedoch nur: $\frac{max_{i,j}|\hat{u}_{ij}|}{max_{i,j}|a_{ij}|} \leq 2^{n-1}$ angenommen werden, folglich ist der Algorithmus auch durch die Spaltenpivotwahl über die Menge aller invertierbaren Matrizen nicht stabil. Da sich der Quotient in der Realität mit zufälligen Matrizen deutlich besser abschätzen lässt, gilt die LU-Zerlegung mit Pivotsuche als stabil. Beispielsweise wenn man nun obiges Beispiel stabil.txt mit der LU-Zerlegung mit Pivotisierung ausführt, erhält man für dasselbe b wie oben $x = (0,00007903;\ 0,00003762;\ 0,00127988)$ was sehr nahe an der korrekten Lösung liegt. Dennoch kann es wie erwähnt auch mit Pivotisierung bei ungünstigen Eingaben zu großen Fehlern kommen, die zum Beispiel mit anderen Algorithmen, wie der QR-Zerlegung, minimiert werden können aber auch mehr Rechenaufwand benötigen.

4 Performanzanalyse

Die Laufzeit berechnet sich über $\frac{1}{3}n^3 - \frac{1}{3}n$ und befindet sich somit in $O(n^3)$[4, 37-38]. In unserer Performanzanalyse vergleichen wir folgende Implementierungen:

- Lineare C Implementierung ohne Compileroptimierung und mit Pivotisierung vs. Lineare C implementierung ohne Compileroptimierung und ohne Pivotisierung.
- Lineare C Implementierung ohne Compileroptimierung vs. C Implementierung mit Vektorisierung durch Intrinsics.
- Lineare C Implementierung mit Compileroptimierung -O3 vs. Assembler Implementierung mit SIMD Vektorisierung.

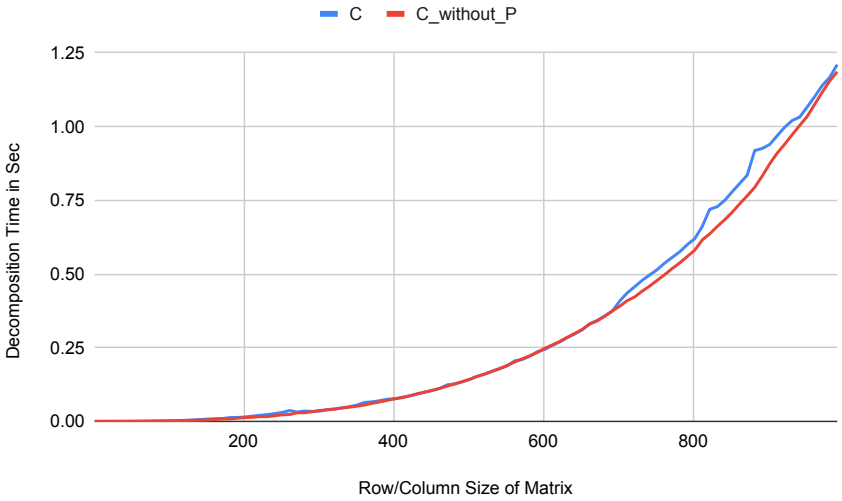
Alle Performance Tests wurden mit einem HP Spectre x360 Convertible mit einem i7-1165G7 @ 2.80GHz und dualchannel LPDDR4 4267MHz RAM durchgeführt.

Die Ergebnisse der Vergleiche werden in den folgenden Unterkapiteln analysiert.

4.1 Pivotisierung vs. keine Pivotisierung

Leider ist der Performanzgewinn wie man an Grafik 1 erkennen kann wenn höchstens marginal, obwohl bei den Testläufen aus der Grafik Worst-Case Eingaben verwendet wurden, bei denen $n - 1$ mal pivotisiert werden musste. Dies kann man vermutlich auf zwei grundlegende Effekte zurückführen. Erstens verändert das Weglassen der Pivotisierung nicht die asymptotische Laufzeit welche weiterhin bei $O(n^3)$ liegt. Der einzige Gewinn ligt also darin, dass pro Schleifendurchlauf etwas weniger Operationen ausgeführt werden müssen. Dies kann aber maximal einen linearen Speedup erbringen. Dies kann aber maximal zu einem linearen Speedup führen, welcher aber, wie an der Performance-Analyse erkennbar, nicht wirklich signifikant ist, da die Pivotisierung nur einen kleinen Prozentsatz der Laufzeit ausmacht.

Abbildung 1: Pivotisierend vs. Nicht-Pivotisierend

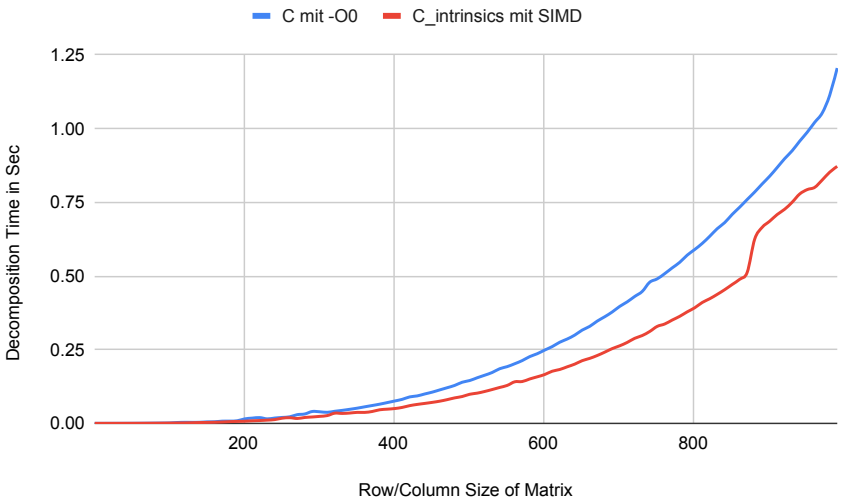


Unter Betrachtung dieser Erkenntnis zusätzlich zu den Ergebnissen der Genauigkeitsanalyse sind wir zu dem Schluss gekommen, dass eine Implementierung ohne Pivotisierung keinen Vorteil und signifikante Nachteile bringt und dieser Ansatz somit nicht weiter Verfolgt werden muss.

4.2 Linear vs. Parallel durch Intrinsics

Eine Performanzanalyse unserer linearen C Implementierung mit perf tool hat ergeben, dass ein Großteil der Rechenzeit in einer bestimmten Schleife verbracht wird. Außerdem zeigt perf tool uns, dass innerhalb der Schleife die meiste Zeit für **mov** Befehle aufgewandt wird, dicht gefolgt von arithmetischen Operationen. Diese Befehle sind in unserem Kontext Teil von Zeilenoperationen auf eine Matrix, was bedeutet, dass diese Operationen zu großen Teilen unabhänig voneinander sind. Also können wir diese Operationen relativ einfach, unter Zuhilfenahme von C Intrinsics, parallelisieren. Die Laufzeit der Implementierung, in der wir diese teuren Operationen vektorisiert haben, wird in Abbildung 2 mit der Laufzeit der unvektorisierten Implementierung verglichen. Durch den zusätzlichen Overhead der Vektorisieung sieht man bei kleineren Eingabegrößen keinen signifikanten Unterschied. Je größer aber die Eingabe wird, desto deutlicher sieht man an den sich von einander entfernenden Graphen, dass die Vektorisierung einen signifikanten Speedup generiert. Wir haben den Speedup punktuell bei einer Eingabegröße von $n = 1000$ berechnet, wo dieser bei ≈ 1.5 liegt.

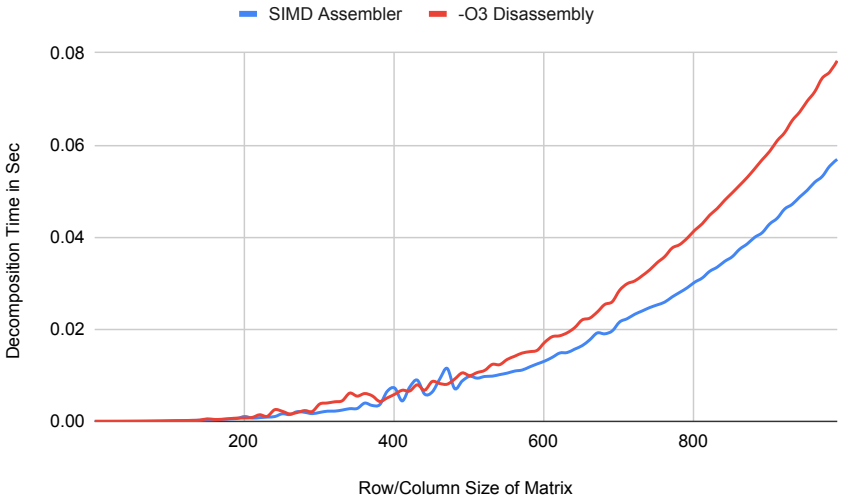
Abbildung 2: C-linear vs. C-vektorisiert



4.3 Compiler optimierter Code vs. ASM mit SIMD

Als letztes haben wir, um optimale Performanz zu erreichen eine reine ASM Implementierung entwickelt. In dieser Implementierung haben wir möglichst viel optimiert und vor allem die im vorherigen Kapitel als Laufzeit aufwendig identifizierten Stellen direkt mit xmm Instruktionen Vektorisiert. Zum Vergleich haben wir unsere einfache C Implementierung mit -O3 kompilliert um zu analysieren, ob unsere Implementierung besser, schlechter oder ähnlich gut wie der Compiler ist. Wie man an Abbildung 3 erkennen kann, bringt unsere händische Implementierung eine deutlich bessere Performance als die compileroptimierte Version. Dies ist ein für uns überraschendes Ergebnis, was uns aber zeigt, dass es manchmal tatsächlich noch sinnvoll sein kann, Assembler Code von Hand zu schreiben.

Abbildung 3: Compileroptimiert vs. ASM-vektorisiert



5 Zusammenfassung und Ausblick

In dieser Projektarbeit hatten wir die Aufgabe ein Programm zu entwickeln, dass eine reguläre Matrix A in 2 Dreiecksmatrizen L, U und eine Pivotmatrix P aufteilen kann. Wir entschieden uns dazu unser Programm auf Basis des Gaußalgorithmus zu implementieren, welcher eine Matrix mit Hilfe von elemantaren Zeilenoperationen wie Zeilenvertauschungen und Zeilenadditionen manipuliert. Einen alternativen Ansatz hätte der Dolittle Algorithmus geboten. Im Laufe dieser Ausarbeitung haben wir verschiedene Implementierungen der LU-Zerlegung erstellt und miteinander verglichen. In der Erwartung eines Performanzgewinns implementierten wir auch eine Version mit vereinfachtem Gaußalgorithmus, welcher keine Zeilen vertauscht. Es stellte sich aber heraus, dass dann der Algorithmus, aufgrund von floatingpoint arithmethischen Gegebenheiten, nicht mehr stabil ist und es so zu komplett falschen Ergebnissen kommen kann. Außerdem ist der Performanzgewinn dieser Implementierung nur marginal und somit hat eine nicht-pivotisierende Implementierung keinen Vorteil gegenüber einer pivotisierenden Implementierung. Performance technisch liegt der Algorithmus in $O(n^3)$. Wir haben die Interressante Erfahrung gemacht, dass unsere händisch vektorisierte ASM Implementierung am effizientesten lief, sogar besser als eine mit -O3 kompilierte C Version, was uns gezeigt hat, dass handgeschriebener ASM Code immer noch Anwendung finden kann.

Literatur

[1] Wolfgang Böhm. *Einführung in die Methoden der numerischen Mathematik*. Vieweg, 1977.

[2] Robert Plato. *Numerische Mathematik kompakt*. Vieweg, 2006.

[3] Thomas Richter. *Einführung in die Numerische Mathematik*. Springer, 2017.

[4] Hans Rudolf Schwarz. *Numerische Mathematik*. Vieweg, 2009.

[5] Prof. Dr. Barbara Wohlmuth. Lineare gleichungssysteme. `https://www-m2.ma.tum.de/foswiki/pub/M2/Allgemeines/EinfNumSoSe11TwoInOne/folien_lings.pdf`, Accessed: 08.07.2021.

[6] Walter Zulehner. *Numerische Mathematik*. Birkhäuser, 2008.