LEHRSTUHL FÜR RECHNERARCHITEKTUR UND PARALLELE SYSTEME

Grundlagenpraktikum: Rechnerarchitektur

Gruppe 213 – Abgabe zu Aufgabe A326
Sommersemester 2023

Noah Schlenker Leon Baptist Kniffki Christian Krinitsin

1 Einleitung

Schon im 18. Jahrhundert v. Chr. beschäftigten sich die Babylonier mit der Quadratwurzel aus zwei. Mit $\frac{30547}{21600} \approx 1,\mathbf{41421}2962\ldots$ schafften sie eine Näherung, die nur ab der fünften Nachkommastelle vom eigentlichen Ergebnis abweichte [1]. Außerdem schafften sie es, mit dem Beweis der Irrationalität der Wurzel aus 2, den ersten Beweis dieser Art aufzustellen, der abseits von Intuition, eine Überlegung und genaue Beweisführung erforderte [2].

Der Nutzen der bekanntesten irrationalen Zahl ist heute im Alltag weitreichend. So ist beim internationalen Standard für Papierformate das Seitenverhältnis stets $\frac{1}{\sqrt{2}}$ [3]. Dies hat zur Folge, dass beim Halbieren "des Blattes entlang der längeren Seite wieder ein Blatt im DIN-A-Format [...] entsteht"[1].

In dieser Arbeit werden wir uns damit beschäftigen, wie man eine beliebige Anzahl an Nachkommastellen aus der Quadratwurzel aus zwei mit folgenden Verfahren berechnet:

$$\begin{pmatrix} 0 & 1 \\ 1 & 2 \end{pmatrix}^n = \begin{pmatrix} x_{n-1} & x_n \\ x_n & x_{n+1} \end{pmatrix} \Leftrightarrow \lim_{n\to\infty} 1 + \frac{x_n}{x_{n+1}} = \sqrt{2}\,.$$

Zu Beginn setzen wir uns dafür damit auseinander, wie man Zahlen mit beliebiger Größe speichern kann und arithmetische Operationen ausführen kann. Als nächstes wird beleuchtet, wie wir mithilfe von Matrizen und der schnellen Exponentiation an den Wert von $\sqrt{2}$ approximieren. Nun werden wir die Darstellung der Kommazahlen diskutieren und einen Divisionsalgorithmus vorstellen. Es wird die Genauigkeit der Endergebnisse erläutert und zum Schluss die Performanz des Programms mithilfe von Vergleichsimplementierungen bewertet.

2 Lösungsansatz

2.1 Big-Num

Der Anspruch der Arbeit liegt darin, $\sqrt{2}$ beliebig genau zu berechnen, herkömmliche Variablen mit festgelegter Größe können damit nicht verwendet werden. Hiermit wird die Datenstruktur *Big-Num* eingeführt. Sie ermöglicht die Darstellung von Zahlen mit beliebiger Größe und Genauigkeit, braucht in C aber eine eigene Implementierung, die im Folgenden erläutert wird.
Die Definition der Datenstruktur sieht wie folgt aus:

```
struct bignum {
    uint32_t *digits;
    size_t size;
    size_t fracSize;
};
```

Zahlen werden in einem Array von 32-Bit großen *digits* gespeichert. Die Reihenfolge der DoubleWords ist Little-Endian. *Size* gibt die Größe des Arrays an, *fracSize* bestimmt, wie viele Nachkommastellen in dieser Zahl vorhanden sind, dazu mehr in 2.5. Nun werden verschiedene Arithmetische Operationen ausgeführt.

Addition und Subtraktion

Bei der Addition und der Subtraktion werden die einzelnen Elemente aufeinander addiert bzw. subtrahiert. Entsteht ein Overflow, wird dieser auf die nächsten Blöcke übertragen.
Seien a und b zwei Big-Nums, unterteilt in ihre Blöcke von 0 bis m. Falls die digits eines Big-Nums kleiner sind als die des anderen, werden die entsprechenden Blöcke mit Nullen aufgefüllt. Für die Addition und Subtraktion gilt:

$$c = \sum_{i=0}^{m} \left(2^{32i}(a_i + b_i)\right) \quad \text{und} \tag{1}$$

$$c = \sum_{i=0}^{m} \left(2^{32i}(a_i - b_i)\right) \ . \tag{2}$$

Hiermit sind beide Operationen trivial mit einer Laufzeit von $\mathcal{O}(n)$ implementiert.

Multiplikation

Um zwei Zahlen miteinander zu multiplizieren läuft man bei der russischen Bauernmultiplikation des Multiplikators einmal den Multiplikanten ab, wenn man diesen auf das Zwischenergebnis addiert. Für zwei Zahlen der Längen $n, m \in \mathbb{N}$ hat die Bauernmultiplikation also Laufzeit von $\mathcal{O}(n * m)$ und damit im häufigen Fall von $n = m$ sogar $\mathcal{O}(n^2)$. Die Multiplikation spielt auf der untersten Ebene für die Matrixmultiplikation und somit auch für die Berechnung von $\sqrt{2}$ eine wichtige Rolle.
Mathematisch lässt sich dies mit den eingangs definierten Big-Nums a und b folgendermaßen darstellen:

$$c = \sum_{i=0}^{m} \left(2^{32i} b_i \sum_{j=0}^{m} 2^{32j} a_j \right) \ . \tag{3}$$

Im Folgenden wird ein Algorithmus zur schnelleren Berechnung des Produkts erläutert.

2.2 Karazuba-Multiplikation

Dank des Karazuba-Algorithmus kann die Laufzeit der Multiplikation auf $\mathcal{O}(n^{\log_2(3)}) = \mathcal{O}(n^{1.59})$[8] verringert werden. Dafür werden die zu multiplizierenden Zahlen in die Form $a_0+2^m a_1$ gebracht, wobei a_0 und a_1 maximal die Größe $\lceil \frac{n}{2} \rceil$ haben. Durch folgende Umformung kann ab mit nur noch drei $\lceil \frac{n}{2} \rceil$ großen Multiplikationen und sechs zu vernachlässigenden Additionen bzw. Subtraktionen und einigen shifts ermittelt werden:

$$\begin{aligned}
ab &= (a_0 + 2^m a_1)(b_0 + 2^m b_1) \\
&= a_0 b_0 + 2^m a_0 b_1 + 2^m a_1 b_0 + 2^m 2^m a_1 b_1 \\
&= a_0 b_0 + 2^m(\mathbf{a_0 b_1} + \mathbf{a_1 b_0}) + 2^{2m} a_1 b_1 \\
&= a_0 b_0 + 2^m(\mathbf{a_0 b_1} + \mathbf{a_1 b_0} + \mathbf{a_0 b_0} + \mathbf{a_1 b_1} - a_0 b_0 - a_1 b_1) + 2^{2m} a_1 b_1 \\
&= a_0 b_0 + 2^m((\mathbf{a_0} + \mathbf{a_1})(\mathbf{b_0} + \mathbf{b_1}) - a_0 b_0 - a_1 b_1) + 2^{2m} a_1 b_1 \,.
\end{aligned}$$

Kann eines der Zwischenprodukte immer noch nicht durch eingebaute CPU-Instruktionen berechnet werden, kann der Algorithmus rekursiv angewendet werden oder auch ab einer bestimmten Grenze auf die russische Bauernmultiplikation zurückgegriffen werden. Zur Veranschaulichung ein Beispiel mit $0x0001.0002 \cdot 0x0003.0004$ unter der Annahme, dass wir 16-bit Zahlen mit der CPU multiplizieren können:

$$\begin{aligned}
a &= (0002)_{16} + 2^4(0001)_{16}; \quad b = (0004)_{16} + 2^4(0003)_{16} \\
a_0 b_0 &= (0000.0008)_{16}; \quad a_1 b_1 = (0000.0003)_{16}; \quad (a_0 + a_1)(b_0 + b_1) = (0000.0015)_{16} \\
ab &= (0000.0003.0000.0008)_{16} + 2^4((0000.0015)_{16} - (0000.0008)_{16} - (0000.0003)_{16}) \\
&= (0000.0003.000a.0008)_{16} \,.
\end{aligned}$$

2.3 Matrixmultiplikation

Die Matrixmultiplikation kann wie nach Definition implementiert werden. Um zwei $\mathbb{N}^{2\times 2}$ Matrizen miteinander zu multiplizieren, werden acht Multiplikation und vier Additionen benötigt:

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{pmatrix} .$$

Da die Multiplikation in einer schlechteren Laufzeitklasse ist als die Addition, ist es erstrebenswert, die Anzahl der Multiplikationen zu minimieren.
Für allgemeine $\mathbb{N}^{2\times 2}$ Matrizen wären keine Optimierungen mehr möglich, allerdings befindet sich die Matrix

$$\begin{pmatrix} 0 & 1 \\ 1 & 2 \end{pmatrix} \quad \text{in der symmetrischen Form} \quad \begin{pmatrix} x_{n-1} & x_n \\ x_n & x_{n+1} \end{pmatrix} ,$$

die bei der Multiplikation mit sich selbst beibehalten wird. Diese Eigenschaft macht es nicht nur möglich, die Anzahl der gespeicherten Werte von vier auf drei zu reduzieren,

sondern auch die Anzahl der Multiplikationen auf vier zu halbieren:

$$\begin{pmatrix} x_{n-1} & x_n \\ x_n & x_{n+1} \end{pmatrix} \begin{pmatrix} y_{n-1} & y_n \\ y_n & y_{n+1} \end{pmatrix} = \begin{pmatrix} x_{n-1}y_{n-1} + x_n y_n & x_{n-1}y_n + x_n y_{n+1} \\ x_n y_{n-1} + x_{n+1} y_n & x_n y_n + x_{n+1} y_{n+1} \end{pmatrix}.$$

Multipliziert man Matrizen miteinander, die Potenzen der selben Basis sind, so gilt $x_{n-1}y_n + x_n y_{n+1} = x_n y_{n-1} + x_{n+1} y_n$. Durch das Einsparen der doppelten Berechnung von x_n und das Wiederverwenden von $x_n y_n$ sind nur noch fünf Multiplikationen nötig.

Außerdem kann in folgender Rechnung x_n auch als eine rekursiv definierte Folge mit $x_n = 2x_{n-1} + x_{n-2}$ und $x_0 = 0, x_1 = 1$ interpretiert werden:

$$\begin{pmatrix} 0 & 1 \\ 1 & 2 \end{pmatrix} \begin{pmatrix} x_{n-1} & x_n \\ x_n & x_{n+1} \end{pmatrix} = \begin{pmatrix} x_n & x_{n+1} \\ x_{n-1} + 2x_n = x_{n+1} & x_n + 2x_{n+1} = x_{n+2} \end{pmatrix}.$$

Bei der Multiplikation mit unserer Basismatrix müssen demnach lediglich x_{n-1} und x_n mit vier Multiplikationen berechnet werden. x_{n+1} erhält man durch einen shift und eine Addition.

2.4 Schnelle Exponentiation

Die schnelle Exponentiation nutzt Assoziativität und Potenzgesetze, um die Anzahl der Multiplikationen bei der Exponentiation von $\mathcal{O}(n)$ auf $\mathcal{O}(\log n)$ zu verringern. Naiv kann eine Potenz a^n mit $n \in \mathbb{N}$ nach der Schulmethode mit $\prod_1^n a$ berechnet werden. Dafür benötigt man allerdings $n-1$ Multiplikationen, was bei großen Werten für n zu einer langen Berechnung führt.

Um dieses Problem effizienter zu lösen, werfen wir einen Blick auf die Potenzgesetze für assoziative Operatoren. Denn sowohl eine Multiplikation, als auch eine Addition im Exponenten kann aufgeteilt werden mit

$$a^{n+m} = \overbrace{a \dots a}^{n+m} = \overbrace{(a \dots a)}^{n} \overbrace{(a \dots a)}^{m} = a^n a^m \quad \text{und} \tag{4}$$

$$a^{nm} = \overbrace{a \dots a}^{nm} = \underbrace{\overbrace{(a \dots a)}^{n} \dots \overbrace{(a \dots a)}^{n}}_{m} = \left(a^n\right)^m. \tag{5}$$

Wenn man also a^n und a^m effizienter als mit $n+m-2$ Multiplikationen berechnen kann, kann man auch a^{n+m} mit 4 effizient berechnen.

Bei der schnellen Exponentiation ermittelt man rechnerisch durch wiederholtes Quadrieren alle $a^{(2^k)}$ mit $2^k \leq n$. Denn nach 5 gilt:

$$\left(a^{(2^k)}\right)^2 = a^{(2 \cdot 2^k)} = a^{(2^{k+1})}.$$

Zur Berechnung von a^n mit $n = 2^k$ sind damit nur noch $k = \log_2 n$ Multiplikationen notwendig.

Um nun auch Potenzen mit $n \in \mathbb{N}$ berechnen zu können, nutzt man 4. Jede Zahl $n \in \mathbb{N}$ kann durch Addition von Zweierpotenzen dargestellt werden, siehe das Binärsystem.

Sei n in Binärdarstellung $2^0b_0 + 2^1b_1 + 2^2b_2 + \cdots + 2^nb_n$, so erhält man a^n laut 4 mit:

$$a^n = a^{2^0b_0+2^1b_1+2^2b_2+\cdots+2^nb_n} = a^{2^0b_0}a^{2^1b_1}a^{2^2b_2}\dots a^{2^nb_n}\,.$$

Da b_i nur die Werte 0 und 1 annehmen kann, ist es am Ende eine boolesche Entscheidung, ob der aktuelle Wert von $a^{(2^k)}$ auf das Zwischenergebnis aufmultipliziert wird.
Außerdem gilt:

$$a^n a^m = a^m a^n\,. \tag{6}$$

Auch wenn diese Gleichung auf den ersten Blick nach der Anwendung des Kommutativitgesetzes aussieht, gilt sie aufgrund der Assozitivität, da nur die Klammerung geändert wird:

$$\overbrace{(a\dots a)}^{n}\overbrace{(a\dots a)}^{m} = \overbrace{(a\dots a)}^{m}\overbrace{(a\dots a)}^{n}\,.$$

Demnach macht es keinen Unterschied, ob zuerst $a^{(2^k)}$ mit dem kleinsten oder dem größten k aufmultipliziert wird.

$(\mathbb{N}^{2\times2}, \cdot)$ ist eine Gruppe und damit assoziativ ist, deshalb kann die schnelle Exponentiation auch für das lösen von $a \in \mathbb{N}^{2\times2}$ genutzt werden. Zur Verdeutlichung berechnen wir das Beispiel a^3 für $a = \begin{pmatrix}0 & 1\\ 1 & 2\end{pmatrix}$. Die Binärdarstellung von 3 lautet $(11)_2$, wir multiplizieren also a^1 und a^2 aufeinander:

$$\begin{pmatrix}0 & 1\\ 1 & 2\end{pmatrix}\begin{pmatrix}0 & 1\\ 1 & 2\end{pmatrix}^2 = \begin{pmatrix}0 & 1\\ 1 & 2\end{pmatrix}\begin{pmatrix}1 & 2\\ 2 & 5\end{pmatrix} = \begin{pmatrix}2 & 5\\ 5 & 12\end{pmatrix}\,.$$

2.5 Darstellung von Kommazahlen und Division

Mithilfe der bereitgestellten Matrix werden nun die Werte ausgerechnet, die bei der Division an $\sqrt{2}$ konvergieren. Nun beschäftigen wir uns mit der Darstellung von Kommazahlen und einem Algorithmus zur Berechnung eines Quotienten in dieser Darstellung. Wir kennen zwei verschiedene Formen der Darstellung von Kommazahlen: Fließkommazahlen nach *IEEE-754* und Fixpunktzahlen. Der Vorteil von Fließkommazahlen ist ihr großer Wertebereich, der dafür mit schwankender Genauigkeit einherkommt, wie man in Abbildung 1 erkennen kann.

Fixpunktzahlen haben einen kleineren Wertebereich bei gleichem Speicherverbrauch, besitzen dafür eine schnellere und deutlich einfachere Arithmetik und haben eine gleichbleibende Genauigkeit im gesamten Wertebereich - zu sehen in Abbildung 2.

Ein weiterer Vorteil von Fixpunktzahlen liegt darin, dass man unter der Verwendung der in 2.1 vorgestellten Big-Nums unendlich viele Nachkommastellen darstellen kann. Aus den genannten Gründen lässt sich schließen, dass sich die Nutzung von Fixpunktzahlen besser eignet, um $\sqrt{2}$ mit beliebig vielen Nachkommastellen darzustellen.

Bei der Division wird ein naives Verfahren verwendet, das im Stile der schriftlichen Division das Ergebnis berechnet und das nur funktioniert, wenn Dividend $<$ Divisor gilt. Zu Beginn wird der Dividend einmal nach links geshiftet, danach werden in einer

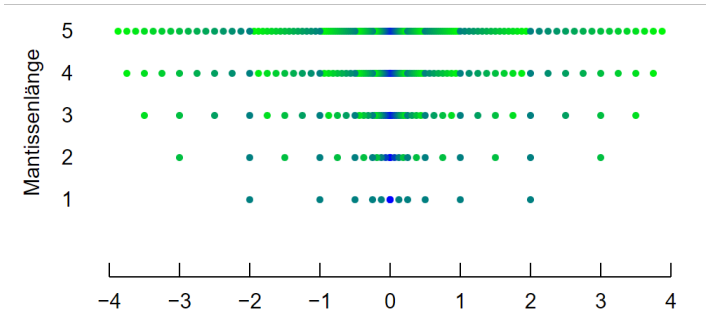


Abbildung 1: Exakt darstellbare Fließkommazahlen mit verschiedenen Mantissen (Entnommen aus [4])

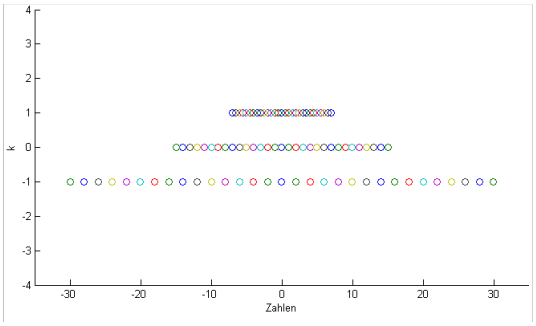


Abbildung 2: Exakt darstellbare Fixpunktzahlen mit k Nachkommastellen (Entnommen aus [5])

Schleife beide Zahlen verglichen. Die Anzahl der Wiederholungen der Schleife entspricht der Anzahl der benötigten Nachkommastellen. In der i-ten Ausführung der Schleife ergibt sich folgende Fallunterscheidung:

- Dividend $>$ Divisor: Setze das i-te Bit des Ergebnisses auf 1, subtrahiere den Divisor vom Dividenden, shifte den Dividenden einmal nach links
- Dividend $=$ Divisor: Setze das i-te Bit des Ergebnisses auf 1, beende den Algorithmus frühzeitig
- Dividend $<$ Divisor: Setze das i-te Bit des Ergebnisses auf 0, shifte den Dividenden einmal nach links

Als illustratives Beispiel berechnen wir $5_{10}/12_{10} = (101)_2/(1100)_2$ mit fünf Nachkommastellen. Abbildung 3 beschreibt das Vorgehen bei der Division. Das Ergebnis ist $(0,01101)_2 = (0,40625)_{10}$.

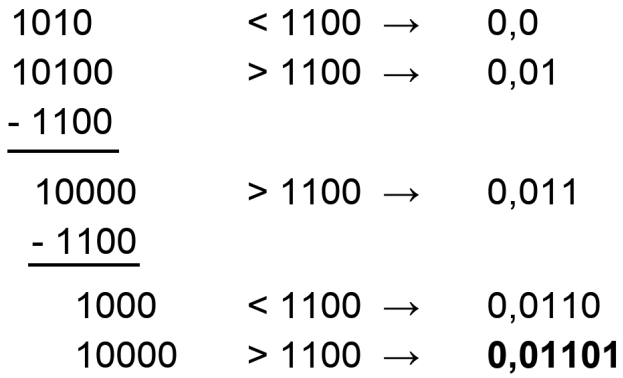


Abbildung 3: Die einzelnen Rechenschritte zur Berechnung der Division

Obwohl dieser Algorithmus zur Kategorie der langsamen Division gehört, bei dem in jedem Schleifendurchlauf nur jeweils eine weitere Nachkommastelle berechnet wird, wurde sich explizit dafür entschieden, diesen zu verwenden. Schnelle Divisionsalgorithmen wie die Newton-Raphson-Division können pro Schleifendurchlauf die Anzahl der Nachkommastellen zwar verdoppeln [9], haben aber aufgrund von wiederholten Multiplikationen und der benötigten initialen Approximation eine schlechtere Performanz, als der hier vorgestellte Algorithmus.

Wir sind in der Lage, Divisionen durchzuführen und effizient Werte auszurechnen, die das Ergebnis approximieren.

3 Genauigkeit

In dieser Sektion wird die Genauigkeit des Ansatzes und seiner Implementierung belegt und erläutert. Da das Belegen der Korrektheit des Gesamtansatzes über die Anforderungen dieses Projektes hinaus gehen, werden wir uns auf die Genauigkeit unserer Implementierung fokussieren. Die mathematische Korrektheit der Ansätze der Komponenten wurde bereits in Kapitel 2 belegt.

Wir haben eine Vielzahl an Methoden ergriffen, um die Korrektheit unserer Implementierung sicherzustellen. Einer dieser Aspekte sind Unit-Tests. Unser Code wurde in möglichst inkohärente Komponenten aufgeteilt, um diese einzeln auf ihre Korrektheit zu prüfen. Die jeweiligen Tests einer Komponente sind im `tests` Subordner zu finden.

Zusätzlich wurden automatisierte System-Tests ausgeführt, die alle Versionen und ihre Ausgaben bis auf 100.000 Dezimalnachkommastellen abgleicht (Implementierung/tests/lookup_comparison.py). Diese Vergleichs-Nachkommastellen wurden aus mehreren unabhängigen Quellen erhalten [6], [7]. Somit wird nicht nur die Korrektheit der einzelnen Komponenten festgestellt, sondern auch die des Gesamtsystems.

Wie bereits in 2.5 erwähnt, erlaubt die Struktur von `bignums` eine Darstellung von beliebig genauen Zahlen. Außerdem wurde gezeigt, dass bei der vorgestellten Divisionsmethode die Anzahl an errechneten Binärnachkommastellen beliebig bestimmbar ist. Da

in unserem Anwendungsfall die Multiplikation, Addition und Subtraktion von `bignum` nicht mit Fixkommazahlen rechnen müssen, sind diese per Definition (unter Annahme der Korrektheit) unendlich genau, also exakt. Da somit alle benötigten arithmetische Methoden für die Errechung von $\frac{x_n}{x_{n+1}}$ entweder exakt oder beliebig genau bestimmbar sind, ist dieser auf beliebig genaue Binärnachkommastellen bestimmbar.

Ein weiterer Aspekt der Genauigkeit ist die korrekte Übersetzung von Dezimalpräzision bzw. Hexadezimalpräzision in Binärpräzision, sowie das Bestimmen von beliebig genauen konvergenten Nachkommastellen. Es muss zwischen konvergenten und normalen Nachkommastellen unterschieden werden, da nicht alle errechneten Nachkommastellen gleich der Nachkommastellen von Wurzel zwei sind. Die Umwandlung von dezimaler Präzision n in binärer Präzision i lässt sich folgendermaßen formulieren:

$$\begin{aligned} 2^{-i} &\leq 10^{-n} \Leftrightarrow \\ -i \cdot \ln(2) &\leq -n \cdot \ln(10) \Leftrightarrow \\ i &\geq \left\lceil n \cdot \frac{\ln(10)}{\ln(2)} \right\rceil . \end{aligned}$$

Die Umwandlung von hexadezimaler Präzision n in binärer Präzision i lässt sich so formulieren:

$$\begin{aligned} 2^{-i} &\leq 16^{-n} \Leftrightarrow \\ -i \cdot \ln(2) &\leq -n \cdot \ln(16) \Leftrightarrow \\ i &\geq \left\lceil n \cdot \frac{\ln(16)}{\ln(2)} \right\rceil = n \cdot 4 . \end{aligned}$$

Diese Korrelationen wurden im Code umgesetzt (`decimal_place_converter.c`). Außerdem wurde durch Messungen die Anzahl an konvergenten Nachkommastellen in Abhängigkeit zum Matrixexponenten n ermittelt, die sich als logarithmisches Wachstum herausstellte. Dieser lässt sich gut durch lineares Wachstum annähern, da ein logarithmisches Wachstum sich zunehmend linear verhält. Somit lässt sich für eine gegebene Anzahl an konvergenten Binärnachkommastellen (Eingabe von `sqrt2`) ein approximatives Minimum des Matrixexponenten rückrechnen (siehe `sqrt2.c`). Die aktuelle Approximierung rechnet ca. 0.5% mehr konvergente Nachkommastellen aus als benötigt. Hierdurch ist unser Programm stets auf den geforderten Nachkommastellen präzise. Hier sind einige Beispiele, die dieses Verhalten demonstrieren:

```
$ ./main -d100
1.41421356237309504880168872420969807856967187537694807317667973799073247846210703885038753432764157 27
```

```
$ ./main -h10
0x1.6a09e667f3
```

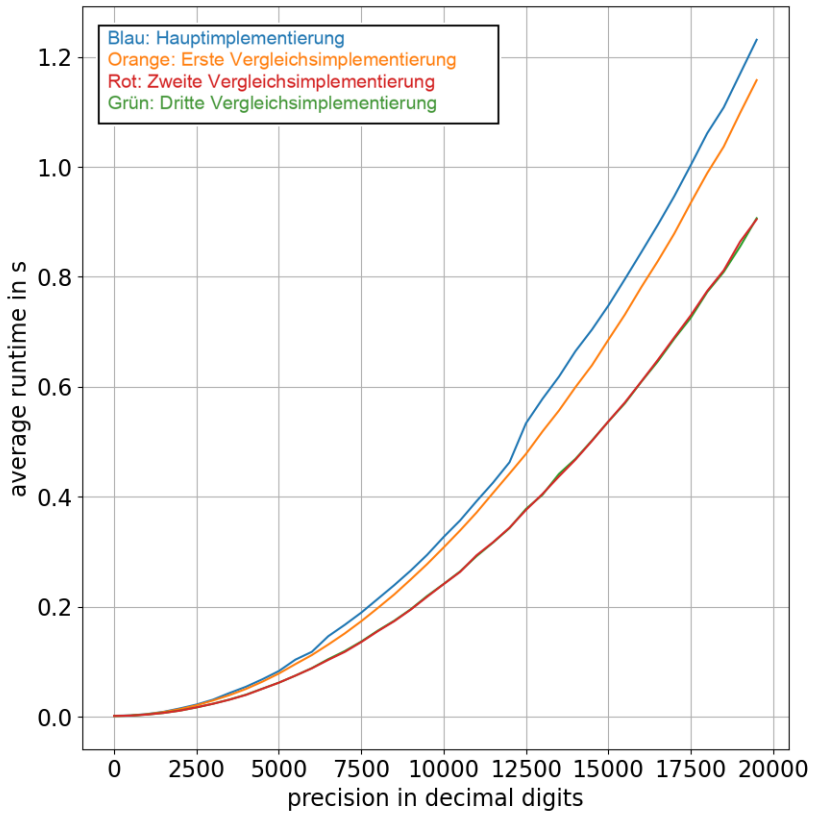


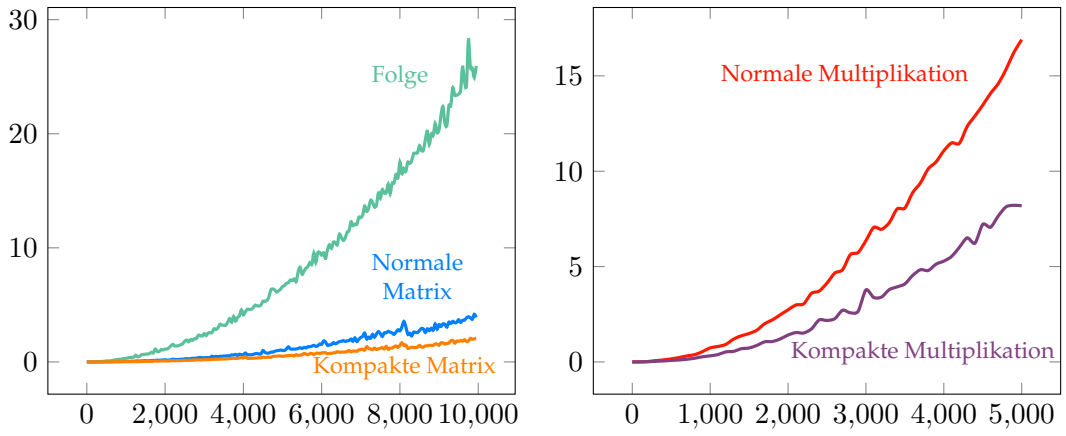
Abbildung 4: Performanz der einzelnen Implementierungen

4 Performanzanalyse

Im Folgenden werden drei Implementierungen ausgewählt und ihre Performanz verglichen, anschließend erklärt. Die Performanztests wurden auf einem System mit einem Intel i5-10210U Prozessor, 4.2GHz, 16 GB Arbeitsspeicher, Fedora 38 x86_64, Linux-Kernel 6.3.8 ausgeführt. Kompiliert wurde mit der Option -O1 mit GCC 13.1.1

Die Hauptimplementierung nutzt eine herkömmliche Matrix und alle naiven Implementierungen der Arithmetik von Big-Nums, die erste Vergleichsimplementierung nutzt kompakte Matrizen statt der herkömmlichen. Die zweite Vergleichsimplementierung nutzt neben kompakten Matrizen auch die Addition und Subtraktion in SIMD und die Karazuba-Multiplikation. Die Dritte gleicht der zweiten, nur mit dem Unterschied, dass statt der Karazuba-Multiplikation eine SIMD Implementierung der Multiplikation verwendet wird.

Die Berechnungen wurden mit Eingabegrößen von 1 bis 20000 dezimalen Nachkom-



(a) Matrixexponentiationen (b) Matrixmultiplikationen

Abbildung 5: Performanz einzelner Operationen

mastellen jeweils 10 mal durchgeführt und das arithmetische Mittel für jede Eingabegröße wurde in das Diagramm aus Abbildung 4 eingetragen.

An dem Diagramm ist zu erkennen, dass die Hauptimplementierung langsamer ist als die drei Vergleichsimplementierungen. Dies liegt zum Einen an der Tatsache, dass die kompakten Matrizen deutlich Rechenzeit einsparen, wie man im direkten Vergleich in Abbildung 5 erkennen kann. Zum Anderen liegt das daran, dass die arithmetischen Operationen in SIMD und die Karazuba-Multiplikation schneller sind als die naiven Implementierungen.

Aufgrund des nahezu identischen Graphenverlaufs der ersten und der zweiten Vergleichsimplementierung sieht man, dass die Karazuba-Multiplikation und die Multiplikation in SIMD ähnlich performant abschneiden. Dies steht entgegen unserer anfänglichen Annahme, dass die Karazuba-Multiplikation wegen ihrer Laufzeit von $\mathcal{O}(n^{1.59})$ schneller ist als die SIMD Multiplikation. Ein möglicher Grund dafür ist der Overhead, der durch die rekursiven Funktionsaufrufe entstehen kann.

Die Optimierungsstufe -O1 wurde bewusst gewählt. Mit der Option -O3 verlaufen alle Graphen identisch, alle Unterschiede werden also durch die Optimierungen des Compilers ausgeglichen.

5 Zusammenfassung und Ausblick

In dieser Arbeit beschäftigten wir uns mit der Berechnung der Quadratwurzel aus zwei. Es wurden verschiedene Algorithmen thematisiert, wie die Karazuba-Multiplikation oder Divisionsalgorithmus und ihr Einfluss auf die Laufzeit erläutert. Außerdem wurden SIMD Implementierungen berücksichtigt.

In einer weiteren Arbeit könnte man daran ansetzen und Big-Num Arithmetiken in AVX implementieren, um weiter die Performanz zu steigern. Des Weiteren könnte man

weitere Algorithmen auf die Laufzeit untersuchen und gegebenenfalls implementieren, so zum Beispiel die Goldschmidt-Division.

Das Ziel der Arbeit wurde erreicht. Nutzer können, abhängig von ihren Anforderungen oder auch Computerspezifikationen, die Wurzel aus zwei mit beliebigen Nachkommastellen berechnen.

Literatur

[1] `https://web.archive.org/web/20070509055700/http://www.do.nw.schule.de/mbr/2020/geschichte.htm`, Zugriff am 16.07.2023.

[2] `https://monde-diplomatique.de/artikel/!5918386`, Zugriff am 16.07.2023.

[3] `https://de.wikipedia.org/wiki/Papierformat`, Zugriff am 16.07.2023.

[4] `https://de.wikipedia.org/wiki/Datei:Exakt_darstellbare_Gleitkommazahlen.png`, Zugriff am 16.07.2023.

[5] `https://de.wikipedia.org/wiki/Datei:Fixpointnumbers.png`, Zugriff am 16.07.2023.

[6] `https://apod.nasa.gov/htmltest/gifcity/sqrt2.1mil`, Zugriff am 16.07.2023.

[7] `https://catonmat.net/tools/generate-sqrt2-digits`, Zugriff am 16.07.2023.

[8] Donald E. Knuth. In *The Art of Computer Programming*, page 295, Massachusetts, 1997. Addison-Wesley.

[9] Pawan Kumar Pandey, Dilip Singh, and Rajeevan Chandel. Fixed-point divider using newton raphson division algorithm. In Vijay Nath and J. K. Mandal, editors, *Proceeding of Fifth International Conference on Microelectronics, Computing and Communication Systems*, pages 225–234, Singapore, 2021. Springer Singapore.