LEHRSTUHL FÜR RECHNERARCHITEKTUR UND PARALLELE SYSTEME

Praktikum Rechnerarchitektur

Gruppe 120 – Abgabe zu Aufgabe A301

Sommersemester 2020

Leonardo Maglanoc Julia Spindler Ronald Skorobogat

1 Einleitung

In diesem Projekt ist unsere Hauptaufgabe eine Verallgemeinerung des Geburtstagsparadox als Algorithmus möglichst effizient und performant zu implementieren. Das Geburtstagsparadox beschäftigt sich mit der Frage: wie viele Personen müssen sich in einem Raum befinden, damit mindestens zwei Leute mit einer Wahrscheinlichkeit von 50% denselben Geburtstag haben unter der Annahme der Gleichverteilung aller Geburtstage.

Intuitiv würde man eine sehr große Zahl für k annehmen, da die Wahrscheinlichkeit vom selben Geburtstag für zwei einzelne Personen $\frac{1}{365}$ beträgt. In Wirklichkeit werden nur 23 Leute benötigt, damit man eine 50%ige Wahrscheinlichkeit hat. Was viele nicht bedenken, ist dass man bei 23 Leuten eine hohe Anzahl von Zweier-Teilmengen hat, nämlich

$$\binom{23}{2} = \frac{23!}{2! \cdot 21!} = 253.$$

Allgemein kann man sagen, dass mit wachsender Zahl der Personen die Anzahl der möglichen Zweier-Mengen quadratisch ansteigt. Damit zwei Leute denselben Geburtstag haben, braucht man nur eine einzige Zweier-Teilmenge, die dies erfüllt und bei 23 Leuten ist das schon erreicht.

Wenn man allgemein die Wahrscheinlichkeit mit einer beliebigen Anzahl von Personen k, die dasselbe Element aus einer Grundmenge M, dessen Mächtigkeit n ist, berechnen will, hilft folgende Herleitung [3]:

$$P = 1 - \left(\underbrace{\frac{n}{n} \cdot \frac{n-1}{n} \cdot_{\dots} \cdot \frac{n-k+1}{n}}_{k\text{-mal}} \right) = 1 - \frac{n!}{n^k \cdot (n-k)!}.$$

Wenn man eine Mindestwahrscheinlichkeit von $\frac{1}{2}$ fordert und nach k auflöst, bekommt man folgende Formel. Diese gibt an, wie viele Leute sich in einem Raum befinden müssen, damit mit mindestens 50% Wahrscheinlichkeit gilt, dass zwei Personen in dem Raum das gleiche Element m aus einer beliebigen Menge M haben.

$$k \geq \frac{1 + \sqrt{1 + 8n \cdot \ln 2}}{2}$$

Die Projektaufgabe fordert die Implementierung obiger Formel mit n als Eingabeparameter. Dabei darf man aber nur die vier Grundrechenarten verwenden. Das heißt, man muss für den Wurzelausdruck ein Näherungsverfahren herleiten. Für die log-Funktion muss keine Approximation implementiert werden, da nur die Konstante $ln(2)$ verwendet wird, die im Code gespeichert werden kann. Zwei verschiedene Lösungsansätze des Wurzelausdrucks müssen umgesetzt werden: mit einer Reihendarstellung und mit einer Lookuptabelle. Wie man in Abb. 1 sieht, ist der Verlauf der Geburtstagsfunktion ähnlich zu einer skalierten Wurzelfunktion.

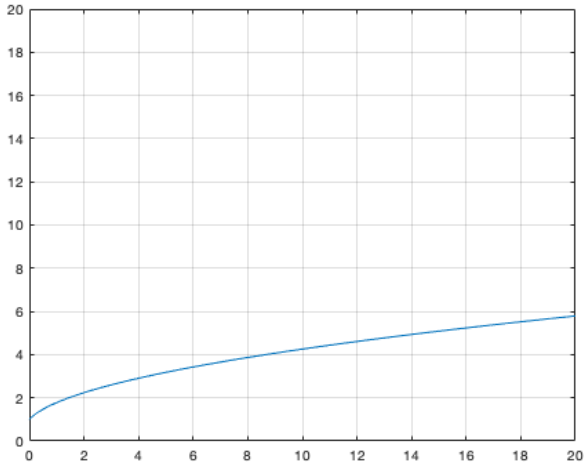

Abbildung 1: Verlauf der Geburtstagsfunktion

Hier sind einige Beispiele mit konkreten Anwendungsfällen zur Veranschaulichung:

Wenn M die Menge aller PINs mit 4 Ziffern ist, so gilt für k:

$$k \geq \frac{1 + \sqrt{1 + 8 \cdot 10^4 \cdot \ln 2}}{2} \approx 38$$

da es $n = 10^4 = 1000$ mögliche PINs gibt.

Wenn M die Menge aller MD5-Hashes ist, so gilt für k:

$$k \geq \frac{1 + \sqrt{1 + 8 \cdot 2^{128} \cdot \ln 2}}{2} \approx 2.18 \cdot 10^{19}$$

da es $n = 2^{128}$ mögliche MD5-Hashes gibt.

2 Lösungsansatz

2.1 Reihendarstellung

Eine mögliche Implementierung ist eine Taylorreihenentwicklung der Wurzelfunktion. Die Taylorreihe ist im Allgemeinen gegeben durch:

$$f(x) = \sum_{n=0}^{\infty} f^n(a) \frac{(x-a)^n}{n!}$$

Wobei a die Stelle ist an der entwickelt wird. Dabei muss man beachten, dass der Konvergenzbereich relativ gering ist und man zum Beispiel bei der Entwicklung an der Stelle $a = 1$ und $f(x) = \sqrt{x}$ effektiv nur Funktionswerte von $0 \leq x \leq 2$ berechnen kann. Um die Wurzel von größeren Zahlen zu berechnen ist es deswegen notwendig, sie zunächst in diesen Bereich zu bringen. Dafür muss man Vorberechnungen anstellen und sich für einen Bereich entscheiden in dem die Zahlen, von denen die Wurzel berechnet werden soll, liegen werden. In unserem Fall wird die Zahl unter der Wurzel in der Formel des Geburtstagsparadoxon nicht größer als $\approx 5.6 \cdot 2^{64}$ werden, was wir grob als 2^{66} abschätzen werden. Dieser Bereich muss nun in kleinere Intervalle aufgeteilt werden, dieser umfasst Zahlen von $2^n + 1$ bis 2^{n+2}, von 2^1 bis 2^{66}. Jedem dieser Intervalle wird ein Wert zugewiesen, eine Zweierpotenz, die sich aus der Wurzel des Wertes in der Mitte des Intervalls berechnet. Die Bestimmung des Intervalls kann dann relativ effizient geschehen, da für die Vergleiche nur Zweierpotenzen benötigt werden und diese mit Bitshifts vergrößert beziehungsweise verkleinert werden können. Außerdem wurde das Intervall mithilfe von binärer Suche implementiert, womit höchstens $log_2(64) = 6$ Vergleich benötigt werden.

Die Wurzel eines beliebigen Werts kann dann folgendermaßen ausgerechnet werden: Division des Wertes mit der passenden Zweierpotenz im Quadrat, Anwendung der Taylorreihe und anschließende Multiplikation mit der Zweierpotenz. Als Beispiel sei $x = 489$. Damit fällt x in das Intervall $]2^7, 2^9]$ und die zugehörige Zweierpotenz ist $\sqrt{2^8} = 2^4$. x' ist dann $x' = \frac{x}{2^{4^2}} \approx 1.91$ und liegt somit im richtigen Bereich für die Taylorreihe. Diese liefert uns das Ergebnis von $\sqrt{\frac{x}{2^{4^2}}} = \frac{\sqrt{x}}{2^4}$. Durch die Multiplikation von 2^4 erhalten wir somit das Ergebnis von $\sqrt{489}$. Wie man sieht, liegt bei diese Vorgehensweise der kleinste Wert für die Taylorreihe bei $\frac{2^{n-1}}{2^n} = 0.5$ und der größte bei $\frac{2^{n+1}}{2^n} = 2$. Da 2 schon am äußersten Rand des Konvergenzbereich liegt, falls man bei $a = 1$ entwickelt, haben wir uns entschlossen die Taylorreihe bei $a = 1.25$ zu entwickeln, da dies genauere Ergebnisse lieferte. Damit können Funktionswerte von $0.25 \leq x \leq 2.25$ berechnet werden. Außerdem haben wir uns dafür entschieden die Taylorreihe für neun Summenglieder zu berechnen, da mit SIMD eine gleichzeitige Berechnung von vier floats möglich ist. Damit erschien ein Vielfaches von vier sinnvoll, wobei das erste Summenglied keine Multiplikation benötigt. Der Funktionsterm für die Annäherung von $f(x) = \sqrt{x}$ im Intervall $[0.25, 2.25]$ lautet dann:

$$f(x) = 1.11803 + 0.447214 \cdot (x - 1.25) - 0.0894427 \cdot (x - 1.25)^2$$

$$+0.0357771 \cdot (x-1.25)^3 - 0.0178885 \cdot (x-1.25)^4 + 0.0100176 \cdot (x-1.25)^5$$
$$-0.00601055 \cdot (x-1.25)^6 + 0.0037786 \cdot (x-1.25)^7 - 0.002455 \cdot (x-1.25)^8$$

In Abb. 2 kann man sehen wie sich die Taylorreihe immer weiter an die Wurzelfunktion annähert, dies allerdings auf einen bestimmten Bereiche beschränkt ist. Hier liegt dieser zwischen 0.25 und 2.25.

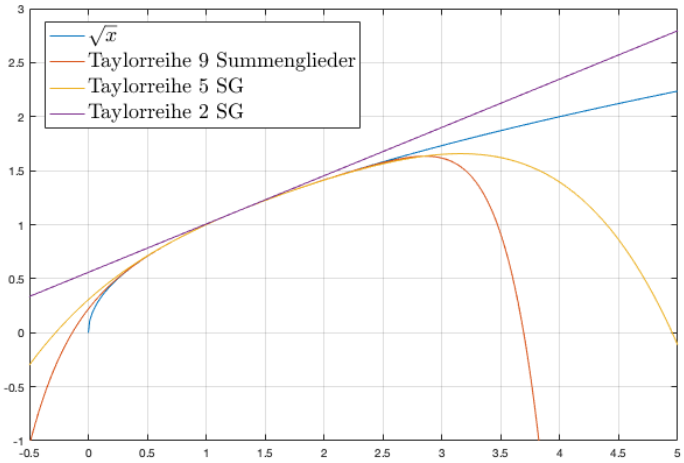

Abbildung 2: Taylorreihenentwicklung der Wurzelfunktion

2.2 Lookup-Tabelle

Eine reine Lookup-Tabelle wird nicht funktionieren, um die Wurzel numerisch zu berechnen. Die Wurzelfunktion ist stetig monoton steigend und hat keine periodischen Intervalle, aber man kann die Lookup-Tabelle in Kombination mit anderen Verfahren verwenden. Die Tabelle würde eine erste Annäherung angeben mit der dann weitergerechnet werden kann.

Insgesamt verwendet die Tabelle 1.284 KB Speicher. Die Methodensignatur wurde geändert, damit die Tabelle als Pointer an die Methode übergeben werden kann. Dadurch können die eigentlichen Werte der Tabelle in einer separaten C-Header Datei gespeichert werden. Sie verwendet den Float-Array-Datentyp und ist in zwei distinkte Hälften aufgeteilt. In der ersten Hälfte wurden alle Werte der gewurzelten Geburtstagsfunktion $\sqrt{1+8n \cdot ln(2)}$ von $n = 0$ bis einschließlich $n = 2^{16} - 1$ direkt berechnet, also sind dort 2^{16} Werte im Array. Der Eingabeparameter kann somit als Index für die Tabelle verwendet werden und in $\frac{1+x}{2}$ eingesetzt werden, wobei x der gespeicherte Wert aus der Tabelle ist. Dies ist eine sehr gute Optimierung, da der Prozessor bei Eingaben von 0 bis $2^{16} - 1$ effektiv nur sechs xmm-Befehle ausführen muss und somit in kurzer Zeit umgesetzt werden kann.

Wenn der Eingabeparameter größer als $2^{16} - 1$ ist, werden die Tabellenwerte als erste Approximierung von $\sqrt{1+8n \cdot ln(2)}$ verwendet. Diese zweite Hälfte der Tabelle

speichert auch 2^{16} Zahlen. Das Eingabeintervall $n \in \left[2^{16}, 2^{64} - 1\right]$ wird in 2^{16} Bereiche eingeteilt und für jeden Bereich wird ein Approximationswert in der Tabelle gespeichert. Da die Wurzelfunktion eine geringere Steigung bei größeren x-Werten hat, sollte man mehr Werte für kleinere x-Werte im Vergleich zu größeren x-Werten speichern. Deswegen wachsen die Abstände der Annäherungszahlen exponentiell, gegeben durch folgende Formel, wobei die kleinste Zahl $\sqrt{1 + 8 \cdot 2^{16} \cdot ln(2)} = \sqrt{363409,749}$ ist:

$$\sqrt{363408,749 + k^i}.$$

Die Konstante k muss so gewählt sein, sodass die Formel für $i = 2^{16} - 1$ ein wenig unter $\sqrt{5,6 \cdot 2^{64}}$ liegt. Durch Ausrechnen wurde $k = 1,00070330$ ermittelt. k^i gibt an, wie groß der Abstand zwischen der 0-ten und der i-ten Approximation ist. Wenn man den n-ten Approximationswert finden will, wertet man die Formel mit $i = n$ aus. Der letzte Eintrag der Tabelle wurde dann mit der höchstmöglichen Zahl, die die Wurzel annehmen kann, $\sqrt{1 + 8 \cdot ln(2) \cdot (2^{64} - 1)} \approx \sqrt{5,6 \cdot 2^{64}}$ gespeichert. Man muss beachten, dass die generierten Tabellenzahlen während der Berechnungen der Werte wegen dem Floating-Point Format an Genauigkeit verlieren.

Der Index wird mit folgender Vorschrift direkt berechnet, wobei $n \geq 2^{16}$ der Eingabeparamter und i der gesuchte Index ist:

$$m = n - 363408,749 = \text{Mantisse} \cdot 2^{\text{Exponent}-127}$$

$$m = k^i \implies i = \log_k(m) = \left\lfloor \frac{\text{Exponent} - 127}{\log_2(k)} + \log_k(\text{Mantisse}) \right\rfloor \approx \left\lfloor \frac{\text{Exponent} - 127}{\log_2(k)} \right\rfloor$$

Das Floating-Point-Format wird hier ausgenutzt, indem der Exponent per Bitshifts und Bitmasken extrahiert wird. Zudem wird $\log_2(k)$ als Konstante im Code gespeichert. Der zweite Summand wird weggelassen, da nur die vier Grundrechenarten verwendet werden dürfen. Wegen diesen Berechnungsungenauigkeiten kann nicht der passendste Index ausgerechnet werden, aber dies kann durch die schnelle Konvergenz der Heron-Iterationen ausgeglichen werden.

Nachdem der erste Approximationswert gefunden wurde, wird mit dem Heron-Verfahren die Zahl weiter verfeinert. Diese hat folgende Iterationsformel [1]:

$$x_{n+1} = \frac{1}{2} \cdot \left(x_n + \frac{q}{x_n} \right),$$

wobei q der Näherungswert der Tabelle und x die Werte der Iterationsschritte sind. Es hat sich ergeben, dass man mit drei Iterationsschritten eine gute Balance von Performanz und Genauigkeit hat.

Das Heron-Verfahren ist eine abgewandelte Form der sogenannten Newton-Formel, die durch folgende Vorschrift beschrieben wird:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

Mit dem Newton-Verfahren kann man ausgehend von einem Startwert x_0 iterativ die Nullstelle einer stetig differenzierbaren Funktion approximieren. Beim Heron-Verfahren verwendet man die Funktion $f(x) = x^2 + c$, wobei man sich der Nullstelle$\sqrt{c}$ annähert. Geometrisch kann man die Iterationen so verstehen, dass man eine Tangente am Graphen am Punkt $x = x_n$ der Funktion zeichnet und dann dessen Nullstelle als nächsten Approximationswert verwendet. Abb. 3 zeigt ein Iterationsbeispiel, wenn man $\sqrt{6}$ approximieren will mit dem ersten Annäherungswert $x_0 = 6$, wobei nach einem Schritt $x_1 = 3,5$ herauskommt. Dies kann man beliebig oft fortsetzen und die Approximationswerte nähern sich immer weiter der Nullstelle.

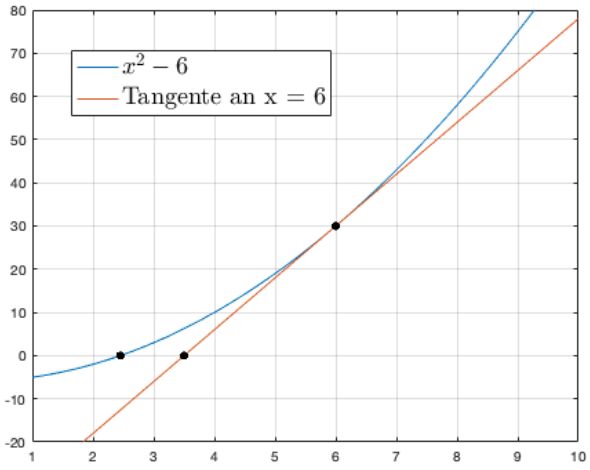

Abbildung 3: Annäherung durch Newtonverfahren

2.3 Funktionalität in der GLIBC

Die Funktionalität der Wurzelmethode in der Standard-C-Library (glibc) ist in der IEEE 754 spezifiziert. Die Wurzelfunktion verwendet die Standard Variante des Newton-Verfahrens mit der Funktion und ihrer Ableitung in Kombination einer Lookup-Tabelle. Die LUT speichert $2^9 = 512$ Approximation der Wurzel plus 512 weitere Annäherungen der Ableitung wegen der Newton-Formel. Zuerst wird per Bitshifts und Bitmasken am Double-Format der Index der ersten Approximation berechnet. Dann werden drei Newton-Iterationen verwendet und das Ergebnis der Iterationen ist die Ausgabe der glib sqrt-Methode. [4]

2.4 Rahmenprogramm

Die Funktionalität des Rahmenprogramms ist dem Nutzer die Möglichkeit zu geben, die Assemblerimplementierungen unseres Projekts auszuführen und zu testen.

2.4.1 Ausführungsweise

Das Rahmenprogramm kann mithilfe des Makefiles kompiliert werden. Im Terminal soll einfach „make“ im Verzeichnis „Implementierung“ eingegeben werden.
Beispiel: . . . /team120/Implementierung$ make

2.4.2 Spezifische Informationen zu Teilen des Rahmenprogramms

Eingabe von Nutzer (-s Option): Diese Option fragt die Größe der Grundmenge für das Ausrechnen der Geburtstagsparadoxformel ab.

Das Rahmenprogrammteil wird mit I/O-Operationen implementiert. Um möglichst alle Fehler wie Segmentation Faults, Endlosschleifen usw. zu vermeiden, werden keine unsicheren C-Funktionen wie scanf() verwendet. Um unerwartete Resultate, zum Beispiel wenn der Nutzer falsche Werte eingibt, zu vermeiden, werden sichere Funktionen wie fgets() verwendet.

Die Größe der Menge kann ausschließlich als eine ganzzahlige Dezimaldarstellung eingegeben werden. Das Programm akzeptiert nur Werte vom Typ unsigned long long. Alle anderen Eingabewerte, wie z.B. Zahlen in Hexadezimalschreibweise, Ox- -, oder Werte vom Typ char, double usw. werden als illegale Werte interpretiert.

Ein weiteres Beispiel: die Zahl 10010 wird in Dezimaldarstellung interpretiert und nicht als binäre Zahl.

Die Funktion gibt das Ergebnis anhand den beiden Assemblerimplementierungen der Wurzelfunktion und einer C-Referenzimplementierung aus.

Testfunktion (-t Option): Diese Funktion führt mehrere Tests der Assemblerimplementierungen aus. Das Resultat von beiden Implementierungen sowie ihre Abweichungen von einer C-Standardimplementierung und das Resultat von einer naiven Assemblerimplementierung werden ausgegeben.

Benchmarking (-b Option): Die Funktion zeigt die Laufzeit aller bisher erwähnten Assembler- und C-Implementierungen. Um eine möglichst genaue Laufzeitanalyse zu erreichen, soll die Laufzeit jeder Funktion mindestens 1 Sekunde lang dauern. Das kann mittels der Angabe von höheren Iterationen erreicht werden.

Hilfe (-h Option): Die letzte Funktion bietet Hilfe und allgemeine Informationen an.

Ausführungsbeispiel: . . . /team120/Implementierung$ ./main -b 100

3 Genauigkeit

Um die Genauigkeit der beiden Implementierungen zu testen, wurden einfach zufällige Zahlen jeweils in die Tabellenimplementierung, Reihenimplementierung und in eine C-Funktion gegeben, sowie in eine Assemblerfunktion, die die Assemblerinstruktion

$srtss$ benutzt, die die Wurzel eines floats berechnen kann [2]. Die Abweichung der Implementierungen wurden mit dem Ergebnis der C-Funktion verglichen, wobei alle Funktionen das Ergebnis der Geburtstagsfunktion berechnen, siehe Tabelle 1.

Eingabe	Abweichung Tabelle	Abweichung Reihe	Abweichung Asm
2	0.0000153%	-0.0003815%	0.0000153%
33	-0.0000153%	0.0009689%	-0.0000153%
73	0.0000153%	0.0000153%	0.0000153%
365	0%	0.0043488%	0%
737474	-0.0000153%	-0.0000153%	-0.0000153%
9898989	-0.0000153%	-0.0001450%	0%
9898242989	-0.0000153%	-0.0002060%	0%
532578665767	0%	-0.0024109%	0%
$2^{64}-1$	0%	0%	0%

Tabelle 1: Genaue Abweichungen

Beide Implementierungen liefern recht genaue Ergebnisse und haben eine Genauigkeit von $> 99.95\%$ im Vergleich zur C-Funktion. Bei Werten $\leq 2^{16}$ werden bei der Tabelle die genauen Werte für die Wurzel für die gegebene Eingabe abgespeichert. Abweichungen vom Ergebnis der C-Funktion könnten somit sogar daran liegen, dass die Tabellenimplementierung genauer ist, da sie im kleinen Bereich dieselben Werte wie die Assemblerreferenz hat. Bei der Reihenimplementierung fällt auf, dass die Größe der Abweichung schwankt. Dies wurde noch durch einen weiteren Test verdeutlicht, in dem Werte im Intervall $[2^{17}+1, 2^{19}-1]$ getestet wurden. Hierbei ist zu beachten, dass dabei nur das Ergebnis der Wurzelfunktion und nicht die der Geburtstagsfunktion verglichen wurde, siehe Abb. 4.

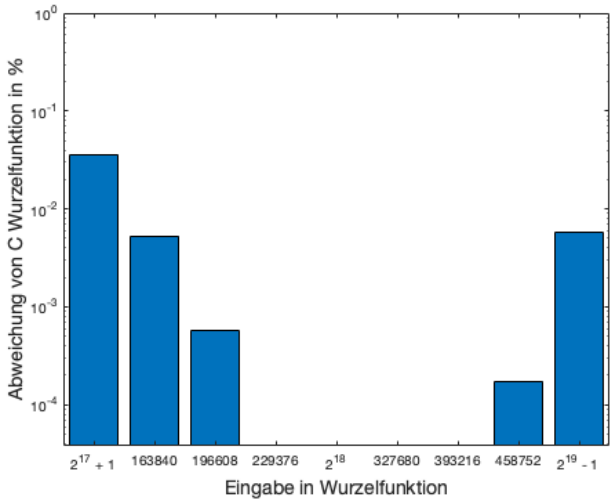

Abbildung 4: Genauigkeit der Reihenimplementierung

Das Intervall wurde so gewählt, da es genau die Werte umspannt, die die gleiche Zweierpotenz zugewiesen bekommen, um sie in den richtigen Bereich zu verkleinern. Man sieht, dass die Reihe an den Rändern weniger gute Ergebnisse liefert. Das liegt eben genau daran, dass die Taylorreihe das Ergebnis der Wurzelfunktion nur in einem bestimmten Bereich annähert und sich an den Ränder schon von der Funktion "wegbewegt". Bei sehr großen Zahlen merkt man bei beiden Implementierungen keinen Unterschied mehr zur C-Implementierung, was daran liegt, dass große Zahlen wegen dem Float-Format nicht sehr präzise dargestellt werden kann.

4 Performanzanalyse

Die Performanzanalyse wurde auf einem Rechner mit folgenden Spezifikationen ausgeführt: **CPU Intel Celeron T3500 @ 2.10GHz, Ubuntu 20.04, Linux-Kernel 5.4.0.** Um genauere Resultate zu bekommen, wurde das Programm mit GCC und Optimierungsstufe -O2 kompiliert. Um möglichst genaue Ergebnisse zu bekommen, wurde Benchmarking drei mal durchgeführt und jede Funktion mit $4 * 10^8$ Iterationen.

Es wurden Tests mit zwei verschiedenen Eingabeintervallen durchgeführt, weil die Tabellenmethode im Gegensatz zu den drei anderen Implementierungen je nach Eingabebereich eine andere Laufzeit hat. Die Reihen-, Assembler- und C-Implementierung ist unabhängig von der Eingabe ungefähr gleich schnell.

Der erste Laufzeitvergleich wurde in einem großen Bereich, $\left[2^{16}, 2^{64} - 1\right]$, ausgeführt. In größeren Wertebereichen kann man davon ausgehen, dass die Reihenimplementierung schneller als die Tabellenimplementierung ist. Die Laufzeit der Reihenimplementierung ist ungefähr 40% schneller im Vergleich zur Tabellenimplementierung. Das liegt daran, dass das Taylorpolynom mit SIMD parallelisiert berechnet werden kann, wohingegen bei den Iterationen des Heron-Verfahrens nicht parallel gerechnet werden kann. Trotzdem sind die Implementierungen in Assembler und C schneller, siehe Abb. 5.

Das Programm wurde auch mit Zahlen im kleineren Bereich, $\left[0, 2^{16} - 1\right]$, getestet. Wie man in Abb. 6 sieht, ist im kleineren Wertebereich die Tabellenimplementierung signifikant schneller als sowohl die Reihenimplementierung, als auch die Standardimplementierung in Assembler. Die Tabellenimplementierung ist mehr als 6 mal schneller als die Reihenimplementierung und 2 mal schneller als die Assemblerimplementierung. Die Ursache dafür liegt daran, dass der Wurzelausdruck direkt als Tabellenwert vorberechnet wurde und somit keine Heron-Iterationen durchgeführt werden müssen. Trotzdem ist die C-Implementierung zwei mal schneller als die Tabellenimplementierung. Die genauen Werte der Analyse sind in Tabelle 2 zu sehen.

Aus beiden Laufzeitvergleichen kann man schließen, dass die C-Implementierung im Allgemeinen am schnellsten ist. Das könnte daran liegen, dass die C-Standard-Library seit Jahrzehnten als Standard in der Industrie verwendet wird, und deswegen sehr gut optimiert ist. Die Tabellenimplementierung ist die zweitschnellste Implementierung im kleineren Bereich, aber ansonsten ist die Reihenimplementierung effizienter im größeren Bereich.

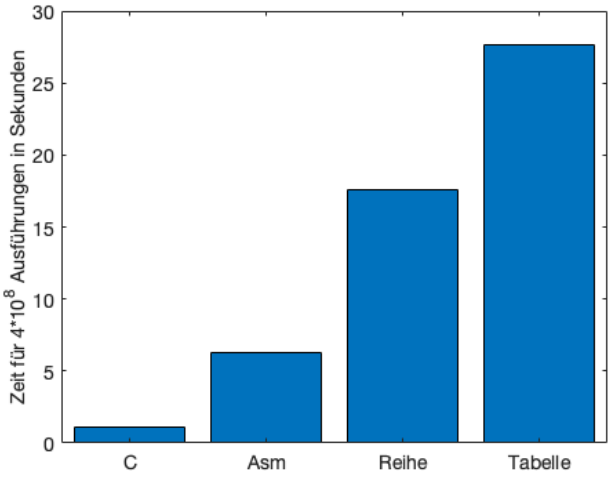

Abbildung 5: Performanzergebnisse im großen Wertebereich

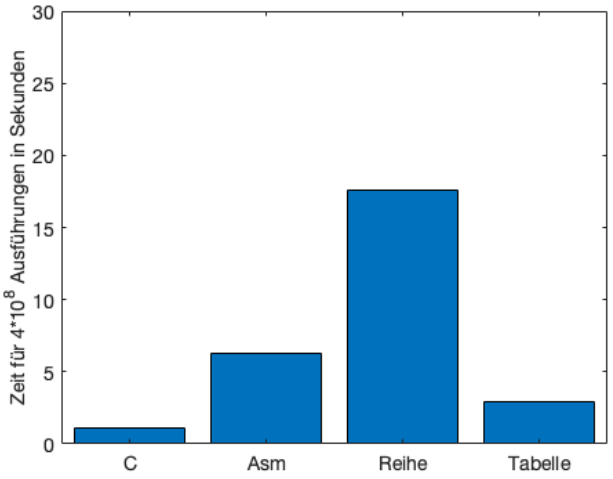

Abbildung 6: Performanzergebnisse im kleineren Wertebereich

Implementierung	großer Bereich	kleiner Bereich
C	$1.0720604446666s$	$1.247632289s$
Asm	$6.3228081073333s$	$6.31891209s$
Reihe	$17.561852893266s$	$17.63354452s$
Tabelle	$27.642242228s$	$2.884157807s$

Tabelle 2: Genaue Laufzeiten

5 Zusammenfassung und Ausblick

Man kann von diesen Ergebnissen entnehmen, dass Standardmathefunktionen wie z.B. die Wurzelfunktion so stark optimiert in sowohl Assembler als auch C implementiert sind, dass es kontraproduktiv ist, solche Funktionen erneut selbst zu erstellen.

Der Bottleneck in der Tabellenimplementierung liegt in den Heron-Iterationen. Eine Möglichkeit sie zu verbessern wäre die Berechnungen an sich effizienter zu schreiben, siehe GLIBC-Implementierung der Wurzelfunktion. Im Gegensatz zur Reihenimplementierung kann man hier nicht SIMD verwenden, da die iterativen Heron-Vorgänge nicht parallelisiert werden können. Eine weitere Möglichkeit der Verbesserung wäre die Anzahl der Heron-Iterationen in Abhängigkeit von der Genauigkeit zu bestimmen. Im Moment werden für Eingaben $\geq 2^{16}$ immer drei Iterationen durchgeführt, obwohl bei bestimmten Zahlen die Genauigkeit der Approximationen ohne Heron gut genug sein könnte.

Bei der Reihenimplemtierung ist das größte Problem die Ungenauigkeit am Rand des Konvergenzbereiches. Eine weitere Idee wäre es somit, die Intervalle noch kleiner zu fassen und somit das Ergebnis der notwendigen Division z.B irgendwo zwischen 0.7 bis 1.4 zu bringen. Somit lägen die Ränder dieses Intervalls weiter entfernt von den Ränder des Konvergenzbereichs und würden genauere Ergebnisse liefern. Das Problem wäre dabei allerdings natürlich eine größere Suche für den richtigen Bereich und die zugeordneten Zahlen für jeden Bereich wären nicht nur Zweierpotenzen sondern auch irrationale Zahlen. Beides würde wahrscheinlich zu einer schlechteren Performanz führen. Im Allgemeinen kann man sagen, dass eine Reihendarstellung eher ungeeignet ist, um die Wurzelfunktion numerisch zu berechnen aufgrund des kleinen Konvergenzbereiches.

Literatur

[1] Hans Bäckel. Das Verfahren von Heron. *Proseminar Analysis, Ruprecht-Karls-Universität Heidelberg*, Wintersemester 2008/9.

[2] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation, October 2019. `http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html`, visited 2020-07-18.

[3] John Haigh Mario Cortina Borja. The birthday problem. *significance*, september 2007.

[4] Oliver Krauss W.B. Langdon. Evolving sqrt into 1/x via software data maintenance. *Genetic and Evolutionary Computation Conference Companion*, July 8-12:3, 2020.