LEHRSTUHL FÜR RECHNERARCHITEKTUR UND PARALLELE SYSTEME

Grundlagenpraktikum: Rechnerarchitektur

Gruppe 103 – Abgabe zu Aufgabe A300

Sommersemester 2023

Ngoc Kim Ngan Nguyen Yll Kryeziu Keihan Pakseresht

1 Einleitung

Der *Fast Inverse Square Root* Algorithmus, auch bekannt unter dem Namen *Quake* Algorithmus ist ein berühmter Algorithmus aus dem Bereich der Computergrafik. Erstmalige Berühmtheit erlangte er im Jahre 2005 als der Source-Code des 3D First-Person-Shooters *Quake III: Arena* veröffentlicht wurde. Der Algorithmus nähert sich auf Basis mathematischer Berechnungen mit einer hohen Genauigkeit an den Kehrwert der Wurzel einer 32-Bit Floating-Point Zahl an und macht sich dabei das IEEE 754 Format und die newtonsche Annäherung zunutze.

Die Motivation für diesen Algorithmus liegt hinter prozessorintensiven Echtzeitberechnungen für Ein- und Ausfallswinkel und Reflektionen im Zusammenhang mit Simulationen von Licht und Schatten im dreidimensionalen Raum. In diesem Zusammenhang war es oftmals nötig, Vektoren zu normalisieren beziehungsweise ihren zugehörigen Normalvektor $\hat{v}$ zu berechnen [1].

$$\hat{v} = \frac{\vec{v}}{\|\vec{v}\|}$$

Der Betrag des Vektors kann über den Satz des Pythagoras berechnet werden und entspricht der euklidschen Norm.

$$\|\vec{v}\| = \sqrt{(v_1)^2 + (v_2)^2 + (v_3)^2}$$

Zu Zeiten von *Quake III: Arena* war dieser Algorithmus um einiges schneller als die herkömmliche Wurzelfunktion und Division, deren Geschwindigkeit für die Anforderungen des Videospiels nicht ausreichte. Der geringe Präzisionsverlust war dabei vernachlässigbar.

In dieser Ausarbeitung wird der *Fast Inverse Square Root* Algorithmus hinsichtlich seiner Implementierung, mathematischen Grundlagen und Performanz analysiert und mit äquivalenten Alternativen verglichen. Im Zusammenhang wird ein Programm implementiert, dass den *Fast Inverse Square Root* Algorithmus, beziehungsweise eine SIMD-optimierte Version davon, auf aus dem Terminal oder aus einer Datei eingelesene Float- und Double-Zahlen anwendet und somit die Kehrwerte ihrer Wurzeln berechnet und ausgibt.

Da die Aufgabenstellung keine genauere Angaben darüber macht, ob und wie die Werte aus der Eingabedatei und dem Terminal kombiniert werden, wird im Rahmenprogramm lediglich eine der beiden Eingabemethoden erlaubt. Die zusätzliche Option *-t* startet die Ausführung der Tests und Benchmarks, die einige Minuten andauern und *.csv* Dateien mit Messwerten im Ordner *benchmark_outputs* generieren.

2 Lösungsansatz

2.1 Relevante Hintergründe

Das IEEE 754 Format und das Newtonverfahren sind wesentlich für die Funktionsweise des Algorithmus, weshalb die beiden Konzepte kurz erläutert werden, bevor diskutiert wird, wieso die Gleichung $y = \frac{1}{\sqrt{x}} \approx \text{MagicNumber} - (x >> 1)$ aus dem Algorithmus gelten kann.

IEEE 754 ist eine Standarddarstellung für Gleitkommazahlen. In diesem Format lassen sich Gleitkommazahlen durch drei Komponenten darstellen: Sign S, Exponent E und Mantisse M. Eine Gleitkommazahl wird als Kombination der Binärdarstellung der drei Komponenten dargestellt. Siehe Tabelle 1 für die Verteilung der Bits. Der Wert einer

	Sign	**Exponent**	**Mantisse**
Float (32 Bit single precision)	1 Bit	8 Bits	23 Bits
Double (64 Bit double precision)	1 Bit	11 Bits	52 Bits

Tabelle 1: Bitzahl der Komponenten je nach Datentyp.

im IEEE 754 Format gespeicherten Zahl ergibt sich aus Gleichung 1, indem man die Komponenten als numerische Werte betrachtet.

$$x = (-1)^S * \left(1 + \frac{M}{2^N}\right) * 2^{E-b} \tag{1}$$

Um den numerischen Wert der Mantisse zu erhalten werden die Mantissen Bits als reale Zahl betrachtet und durch 2^N dividiert, wobei N die Anzahl der Bits in der Mantisse ist. Da die Mantisse immer mit 1 beginnt, werden nur die Nachkommastellen in den Mantissenbits abzuspeichern, folglich hat die Mantisse einen Wertebereich von [0, 1). Der Exponent wird als vorzeichenlose Binärzahl gespeichert. Durch das Abziehen der sogenannten Bias b können negative Exponenten ohne das Zweierkomplement gespeichert werden. Die Bias beträgt für Floats 127 und für Doubles 1023 [2]. Mit Gleichung 2 erhält man die Integerdarstellung einer Zahl im IEEE 754 Format, wovon später Gebrauch gemacht wird.

$$x_{\text{bits}} = 2^N * \left(E + \frac{M}{2^N}\right) \tag{2}$$

Das Rahmenprogramm der Implementierung verhindert die Eingabe von negativen Zahlen und 0, somit ist das Sign Bit immer 0 und muss in weiteren Berechnungen nicht beachtet werden.

Das **Newton-Raphson-Verfahren** ist ein mathematischer Algorithmus zur numerischen Annäherung von Nullstellen einer Funktion. Es basiert auf der Verwendung von Tangenten an den Funktionsgraphen, um schrittweise bessere Schätzungen für die Nullstelle zu finden. Das Verfahren konvergiert in der Regel schnell gegen die genaue Lösung,

wenn eine gute Startschätzung vorhanden ist. Ausgehend von einer vorhergehenden Nullstelle x_n ergibt sich die nächste, präzisere Nullstelle aus

$$x_{n+1} = x_n * \frac{f(x_n)}{f'(x_n)} \tag{3}$$

wobei $f'(x_n)$ die Ableitung von $f(x_n)$ nach x_n ist. Diese Iteration wird so oft ausgeführt, bis die gewünschte Genauigkeit erreicht ist [3].

2.2 Funktionsweise des Algorithmus

Der originale Sourcecode des Algorithmus kann auf dem Github-Account von id-Software gefunden werden [4]. Zentral für den Algorithmus ist folgende Codezeile, wobei `i` die Integerdarstellung der Gleitkommazahl ist, von der der Kehrwert der Wurzel berechnet wird: `i = 0x5F3759DF - (i >> 1);`. Nach der Zuweisung ist `i` eine sehr guter Ausgangspunkt für anschließende Annäherung durch das Newtonverfahren. Im Folgenden wird der Grund dafür erläutert. Aus Gleichung 1 folgt durch Vereinfachung mit dem Logarithmus folgende Gleichung:

$$\begin{aligned} x &= \left(1 + \frac{M}{2^{23}}\right) * 2^{E-127} \\ \Rightarrow \log_2(x) &= \log_2\left(1 + \frac{M}{2^{23}}\right) + E - 127 \end{aligned} \tag{4}$$

Weiteres gilt für eine Zahl c in $[0, 1)$: $\log_2(1 + c) \approx c + \sigma$, wobei σ ein freier Parameter ist, der zum Optimieren der Annäherung verwendet wird [5]. Da die Mantisse nur in diesem Bereich liegt, gilt Gleichung 5. Hierbei ist die Integerdarstellung von x nützlich.

$$\begin{aligned} \log_2(x) &= E + \frac{M}{2^{23}} - 127 + \sigma \\ \Rightarrow \log_2(x) &= \frac{2^{23} * \left(E + \frac{M}{2^{23}}\right)}{2^{23}} - 127 + \sigma \\ \Rightarrow \log_2(x) &= \frac{x_{\text{bits}}}{2^{23}} - 127 + \sigma \end{aligned} \tag{5}$$

Gleichung 5 wird nun in die eigentliche Berechnung eingesetzt. Ebenfalls wird die Integerdarstellung von y benutzt.

$$\begin{aligned} y &= \frac{1}{\sqrt{x}} \\ \log_2(y) &= -\frac{1}{2} * \log_2(x) \\ \frac{y_{\text{bits}}}{2^{23}} - 127 + \sigma &= -\frac{1}{2} * \left(\frac{x_{\text{bits}}}{2^{23}} - 127 + \sigma\right) \\ y_{\text{bits}} &= \underbrace{\frac{3}{2} * 2^{23} * (127 - \sigma)}_{\text{MagicNumber}} - \underbrace{x_{\text{bits}} * \frac{1}{2}}_{\texttt{(i >> 1)}} \end{aligned} \tag{6}$$

Demnach korrespondiert $\frac{3}{2} * 2^{23} * (127 - \sigma)$ zur konstanten MagicNumber, die der Fehlerkorrektur durch das Shiften dient, und $x_{\text{bits}} * \frac{1}{2}$ zum Bitshift nach rechts, der einer Halbierung der Integerdarstellung von x entspricht. Die Berechnung einer geeigneten MagicNumber, sodass die hohe Genauigkeit des Algorithmus sichergestellt ist, erfolgt in Kapitel 2.3.

Nachdem ein angemessener Ausgangspunkt gefunden wurde, wird das Ergebnis nun mit der newton'schen Annäherung verbessert. Dies geschieht durch die Codezeile:

```
y = y * (threehalfs - (x2 * y * y));
```

Durch geeignete Umformungen wird der Zusammenhang zum Newton-Raphson-Verfahren ersichtlich. Es ist $y = \frac{1}{\sqrt{x}}$ zu bestimmen. Diese Aufgabe ist äquivalent dazu, die Nullstellen y der folgenden Funktion zu finden:

$$f(y) = \frac{1}{y^2} - x$$

Ist eine gute Startschätzung gegeben, erhält man die nächste, präzisere Nullstelle Einsetzen in die Formel des Newton-Raphson-Verfahrens:

$$y_{n+1} = y_n - \frac{f(y_n)}{f'(y_n)} = y_n - \frac{\frac{1}{y_n^2} - x}{\frac{-2}{y_n^3}} = y_n + \frac{y_n - x y_n^3}{2} = y_n * \left(\frac{3}{2} - \frac{x}{2} * y_n^2\right) \tag{7}$$

Mit den Zuweisungen `float threehalfs = 1.5f` und `float x2 = x * 0.5f` aus dem originalen Sourcecode ist die Funktion der Codezeile und schlussendlich des gesamten Algorithmus geklärt.

2.3 Die MagicNumber

In dieser Implementierung wird ein *Brute-Force*-Ansatz benutzt, um die MagicNumber zu bestimmen. Er basiert auf den Beobachtungen von Narayanareddy [6]. Es werden verschiedene Werte der MagicNumber für die Implementierung für Floats und Doubles getestet und herausgefunden, welche Zahl den geringsten maximalen relativen Fehler über alle Zahle gibt. Aus Kapitel 2.2 Gleichung 6 ist die Form der MagicNumber (ab jetzt mit R genannt) schon bekannt:

$$R = \frac{3}{2} * 2^{23} * (127 - \sigma) \tag{8}$$

wobei σ ein freier Parameter ist, der zum Optimieren der Annäherung $\log_2(1+c) = c+\sigma$ verwendet wird.

2.3.1 Eingrenzungen des Suchfelds

Da $\log_2(1 + c) \geq c$ für alle Werte von c in [0, 1) gilt und die Gleichheit nur an beiden Grenzen passiert, ist es leicht zu erkennen dass $\sigma > 0$ im Vergleich mit $\sigma \leq 0$ eine bessere Annäherung gibt.

Aus (8) folgt $R = \frac{3}{2} * 2^{23} * 127 - \frac{3}{2} * 2^{23} * \sigma < \frac{3}{2} * 2^{23} * 127 =$ 0x5F400000. Darüber hinaus ist σ ein sehr kleiner Wert, was bedeutet, dass das Suchfeld zunächst zwischen **0x5F300000** und **0x5F400000** eingegrenzt werden kann.

Anstatt alle möglichen Werte von x_{bits}, was der Integerdarstellungen aller Floats entspricht, zu prüfen, können die meisten Exponentenbits zunächst ignoriert werden, um die Anzahl der Testfälle zu reduzieren. Die Genauigkeit des Verfahrens bleibt größtenteils bestehen, da σ nur in der Näherungsformel der Mantisse $\log_2(1 + \frac{M}{2^{23}}) = \frac{M}{2^{23}} + \sigma$ vorkommt. Lediglich das niedrigstwertige Bit E_0 der Exponentenbits spielt eine große Rolle, weil es durch den Rechtsshift `i >> 1` in das Mantissenfeld des Ergebnisses fällt. Hierbei müssen also 2^{24} Tests für jede MagicNumber stattfinden.

Zuletzt können noch die Testwerte der Mantisse der MagicNumber R eingeschränkt werden. Anstatt alle Werte der MagicNumber einzeln durchzuprobieren, können die Testwerte Ziffer für Ziffer vom höchstwertigen Bit zum niedrigstwertigen Bit eingegrenzt werden. Dies geschieht durch eine Inkrementierung in Schritten von **0x10000**, dann **0x1000** und so weiter, bis alle Ziffern gefunden sind. Auf diese Weise werden statt eine Millionen nur etwa 160 Werte geprüft.

2.3.2 Ergebnis und Güte der bestimmten MagicNumber

Nach Implementierung eines C-Programms nach dem oben beschriebenen Verfahren ist $\boldsymbol{R =}$ **0x5F375A87** die gefundene MagicNumber für Floats, die über alle Floats einen maximalen relativen Fehler von 0.1751350679% hat. Das Verfahren zur Bestimmung der Genauigkeit wird in Kapitel 3 behandelt, aber es sei erwähnt, dass dies die Genaugigkeit nach einer Newtoniteraion ist. Die von uns berechnete MagicNumber ist sehr nah an Lomont's und Robertson's MagicNumber von **0x5F375A86**, die einen etwas geringeren maximalen relativen Fehler von 0.1751341630% liefert. Es ist ersichtlich, dass die Zahl etwas schlechter in Genauigkeit ist. Dies liegt vor allem daran, dass die Exponenten von x_{bits} im Laufe des Verfahrens ignoriert werden und somit eine Abweichung des maximalen relativen Fehlers von jedem Wert der MagicNumber entsteht.

2.3.3 Erweiterung auf Doubles mit 64 Bit

Analog wird hier ebenfalls ein *Brute-Force*-Ansatz mit Eingrenzungen des Suchbereichs benutzt. Allerdings muss es aus praktischen Gründen mehrere Einschränkungen der Testwerte geben.

Erste Startschätzung: Da $R =$ 0x5F375A87 einen geringen maximalen relativen Fehler liefert, kann festgestellt werden, dass σ, gegeben durch die Gleichung

$$\frac{3}{2} * 2^{23} * (127 - \sigma) = \text{0x5F375A87} \tag{9}$$

eine hinreichend gute Approximation für $\log_2(1 + c) = c + \sigma$ bezüglich der *Fast Inverse Square Root* Funktion anbietet. Aus (9) folgt der Wert von σ:

$$\sigma = 127 - \frac{R}{\frac{3}{2} * 2^{23}} \approx 0.0450332165 \tag{10}$$

Die Gleichung (8) zur Bestimmung der MagicNumber lässt sich auf Double Zahlen erweitern, wobei N die Anzahl der Bits der Mantisse und b die Bias seien:

$$R_d = \frac{3}{2} * 2^N * (b - \sigma) = \frac{3}{2} * 2^{52} * (1023 - \sigma)$$

Eine Substitution von σ durch Gleichung (10) ergibt folgendes Ergebnis:

$$\begin{aligned} R_d &= \frac{3}{2} * 2^{52} * \left(1023 - \left(127 - \frac{R}{\frac{3}{2} * 2^{23}}\right)\right) \\ &= (1023 - 127) * \frac{3}{2} * 2^{52} + R * 2^{52-23} = \text{0x5FE6EB50E0000000} \end{aligned} \tag{11}$$

Es ist zu beobachten, dass die unteren 29 Bits von R_d Nullen sind. Dies entspricht genau die Differenz zwischen der Anzahl der Mantissenbits von Floats und Doubles. Analog zu dem oben beschriebenen Verfahren können die Testwerte der MagicNumber Ziffer für Ziffer eingegrenzt werden. Es werden zunächst die Werte um $2^{32} =$ **0x100000000** kleiner oder größer von R_d getestet. Hierbei sind ungefähr 256 Werte von R_d zu testen.

Weitere Reduzierung der Anzahl der Testfälle: Eine Überprüfung aller möglichen Mantissen von Doubles ist zeitlich unmöglich. Stattdessen wird die Integerdarstellung nach jeder Überprüfung um $2^{28} = 268435456$ inkrementiert und es werden somit ungefähr $2^{53-28} = 2^{25}$ Doubles für jeden Wert von R_d überprüft.

Das implementierte Programm zur Bestimmung der MagicNumber nach dem beschriebenen Verfahren gibt die Zahl $\boldsymbol{R_d =}$ **0x5FE6EB50C7B33600** zurück. Nach dem in Kapitel 3 beschriebenen Verfahren zur Bestimmung der Genauigkeit ist diese Zahl mit dem maximalen relativen Fehler von 0.1751183669% scheinbar fast so gut wie die von Robertson berechnete MagicNumber mit dem Fehler von 0.1751183671%. Es ist dabei zu beachten, dass die Genauigkeit der hier bestimmten MagicNumber nicht exakt ist, da aus zeitlichen Gründen nicht alle möglichen Doubles überprüft werden können.

2.4 Implementierung und Optimierungen

Da das Ausnutzen des IEEE 754 Formats sowie das Newton-Raphson-Verfahren den Kern des Algorithmus bilden, basiert auch die Implementierung auf den beiden Konzepten. Im originalen Code wird durch einen Trick zwischen der Float- und Integerdarstellung gewechselt, da Bitshifts auf Integern funktionieren, bei Floats aber verboten ist. Dieses Konzept nennt man auch *type-punning*.

```
1 y = number;
2 i = *(long*) &y;
3 y = *(float*) &i;
```

Diese Operationen führen laut C ISO-Standard aber zu undefined behaviour und werden in der optimierten Implementierung durch ein union ersetzt, um *type-punning* trotzdem möglich zu machen [7]. Die Variablen oder Member von unions liegen im gleichen Speicherbereich. Auf diesen Speicherbereich kann man nun durch verschieden Typen zugreifen.

```
union {
    __m128 f;
    __m128i i;
} convSSE;
```

In der C-Programmierung werden die Datentypen `__m128` und `m128i` in der Regel verwendet, um 128-Bit-Vektoren in SIMD (Single Instruction, Multiple Data)-Operationen darzustellen. Der `__m128` Datentyp enthält vier 32 Bit Floats und der `__m128i` erlaubt die Anwendung von Integer-Operationen wie Shifts. Für die Implementierung des Algorithmus für Doubles wird auch der `__m128d` Datentyp benutzt, in den zwei 64 Bit Doubles passen.

SIMD-Operationen ermögliche eine effiziente parallel Verarbeitung, da nun mehrere Datenelemente, also vier 32 Bit Floats beziehungsweise vier 32 Bit Integer innerhalb des Vektors mit einer einzigen Instruktion gleichzeitig bearbeitet werden können. Da die Eingabewerte des Programms Arrays sind, eignen sich SIMD- beziehungsweise SSE-Operationen in Kombination mit *Loop unrolling*, es werden also, solange der Input noch groß genug ist, immer vier Floats beziehungsweise zwei Doubles geladen und gleichzeitig bearbeitet. Übrige Inputs werden skalar bearbeitet. Die wichtigen Bestandteile des Algorithmus, zuert die erste Schätzung des Ergebnisses durch Shifting und MagicNumber und eine Iteration der newton'schen Annäherung sehen in der SIMD-Implementierung für Floats folgendermaßen aus:

```
convSSE.i = _mm_sub_epi32(magicnumber, _mm_srli_epi32(convSSE.i, 1));
convSSE.f = _mm_mul_ps(convSSE.f, _mm_sub_ps(threehalfs,
            _mm_mul_ps(xhalf, _mm_mul_ps(convSSE.f, convSSE.f))));
```

Für die Implementierung der Berechnung mit Doubles werden äquivalente Befehle benutzt.

Lomont und Robertson haben in ihren Ausarbeitungen durch analytische Verfahren verbesserte Konstanten für die MagicNumber gefunden, die die Genauigkeit des Algorithmus verbessern [1]. In der Implementierung werden folglich die MagicNumbers **0x5F375A86** für Floats und **0x5FE6EB50C7B537A9** für Doubles übernommen.

2.5 Wurzelfunktion aus glibc

Im Folgenden geht es um die`__ieee754_sqrt (double x)` Funktion aus glibc Version 2.37, enthalten in `glibc/sysdeps/ieee754/dbl-64/e_sqrt.c`. Aufgrund fehlender Dokumentation können lediglich Vermutungen über die Funktionsweise der Funktion aufgestellt werden. In der Funktion gibt es vordefinierte Konstanten, die wahrscheinlich Teil einer Taylorreihe sind und optimiert wurden, um Fehler zu minimieren. Die erste Schätzung erfolgt nicht über eine MagicNumber, sondern über eine Look-Up-Tabelle und einer Taylorreihe. Anschließend wird die Schätzung vermutlich mit einer Newtoniteration verbessert. Abgesehen von der eigentlichen Berechnung behandelt die Funktion anschließend Rundung und Randfälle wie `NaN`, `+inf`, `-inf` und Eingaben ≤ 0.

3 Genauigkeit

Im Falle des *Fast Inverse Square Root* Algorithmus wird die Genauigkeit und nicht Korrektheit untersucht, da die Motivation hinter dem Algorithmus eine schnellere Berechnung ist, die eine nicht exakte Lösung in Kauf nimmt. Getestet wurde mit folgenden Komponenten: Intel i5-8257U 4x1.40GHz CPU, 8GB DDR3 2133MHz, MacOS 13.3.1 64Bit und Kernelversion Darwin 22.4.0. Kompiliert wurde mit GCC 11.3.0 mit der Option `-O2`.

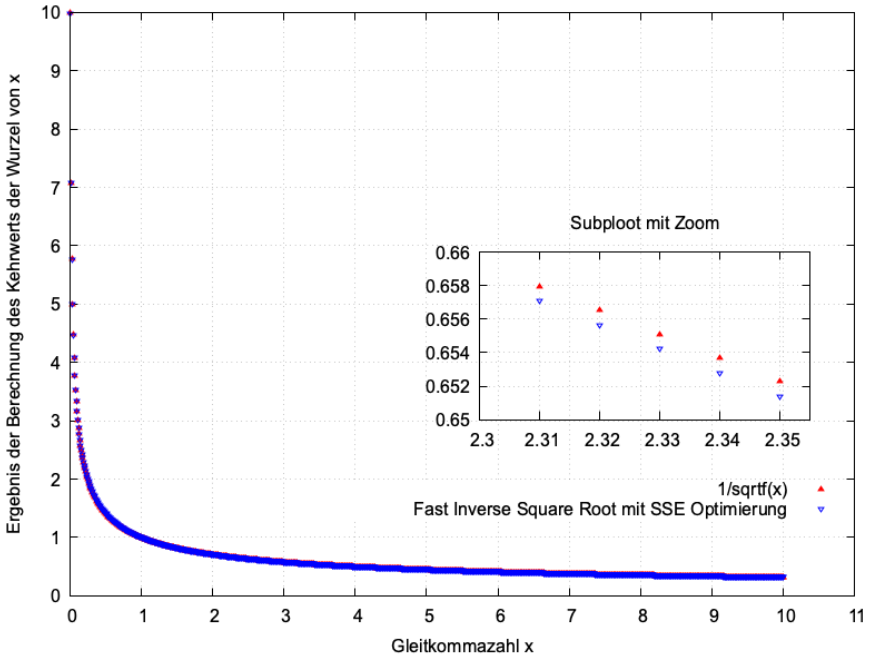

Abbildung 1: Vergleich von 1/sqrt(x) und Fast Inverse Square Root SIMD-optimiert

In Abbildung 1 ist zu erkennen, dass der Algorithmus sehr nah an die exakte Berechnung herankommt und die Kurven auf den ersten Blick übereinander liegen. Wie im Subplot zu sehen ist, gibt es trotzdem kleine Abweichungen. Die Funktion aus dem originalen Sourcecode mit der MagicNumber **0x5F3759DF** hat einen maximalen relativen Fehler von 0.17522874% [1]. Die Ermittlung der Genauigkeiten der optimierten MagicNumbers erfolg durch ein iteratives Verfahren. Der maximale Fehler ist unabhängig von der Größe der eingegeben Zahl, weshalb es keine bestimmten kritischen Punkte gibt, an denen der Fehler am größten ist. Demnach müssen alle möglichen Eingabewerte überprüft werden. Die berechneten Kehrwerte der Wurzeln werden verglichen mit dem exakten Ergebnis von `1/sqrtf(x)` beziehungsweise `1/sqrt(x)`.

Die Integerdarstellung aller positiven, normalisierten Floats reicht von **0x00800000** bis **0x7F800000**. Für die Überprüfung wird die Integerdarstellung bei jedem Durchlauf der Fehlerberechnung um 1 inkrementiert. Der Bereich der zu prüfenden Doubles reicht von **0x10000000000000** bis **0x7FF0000000000000**. Um alle Doubles zu überprüfen, müssten

also mehr als $9 * 10^{18}$ Doubles überprüft werden, was zeitlich nicht machbar ist. Stattdessen wird die Integerdarstellung nach jeder Überprüfung um 1073741824 (entspricht `1llu << 30`) inkrementiert und es werden somit ungefähr $8.6 * 10^9$ Doubles überprüft. Zusätzlich zur Genauigkeit des originalen skalar-implementierten Algorithmus werden auch die Genauigkeiten des SIMD-optimierten Algorithmus sowie des skalaren Algorithmus nach zwei Newtoniterationen geprüft. Wie bereits in Kapitel 2.3 erwähnt, werden in der Implementierung die optimierten MagicNumbers von Lomont beziehungsweise Robertson verwendet.

	Skalar	**Zweifache Newtoniteration**	**SIMD-Optimiert**
Float	0.1751341630	0.0004792558	0.1751387360
Double	0.1751183671	0.0004597281	0.1751183671

Tabelle 2: Maximale relative Fehler [%]

Auffallend ist, dass der maximale relative Fehler mit den optimierten MagicNumbers geringer ist als für die originale MagicNumber. Nach einer zusätzlichen Newtoniteration sinkt der Fehler drastisch. Der geringe Unterschied zwischen der skalaren Implementierung und der SIMD-Implementierung für Floats wird eventuell durch Präzisionsunterschiede der Instruktionen oder verschiedener Reihenfolge der Ausführung der Instruktionen hervorgerufen, da Floating-Point-Arithmetik nicht immer assoziativ ist. Der Unterschied ist jedoch unbedeutend und wird nicht weiter behandelt.

Insgesamt kann durch diese Untersuchung festgestellt werden, dass der Algorithmus und die verschiedenen Implementierungen davon geeignete Alternativen zur Wurzelfunktion von C darstellen und annähernde korrekte Ergebnisse für alle Floats und einen großen Bereich von Doubles liefern. Neben diesen Benchmarks werden im Programm mit der `-t` Option außerdem kleine Tests gestartet werden. In den Tests werden für beide Implementierungen für Float und Double mit willkürlich gewählten Zahlen und Edge-Cases Ergebnisse annähernd exakt berechnet, aber da der maximale relative Fehler dies bereits bestätigt, werden sie in dieser Ausarbeitung nicht gesondert angeführt.

4 Performanzanalyse

Im Folgenden werden Benchmarks der Geschwindigkeit von verschiedenen Implementierungen des *Fast Inverse Square Root* Algorithmus präsentiert. Genauer gesagt handelt es sich hierbei um Microbenchmarks, da die Geschwindigkeit von einzelnen Funktionen gemessen wird. Getestet wurde mit der gleichen Konfiguration wie in Kapitel 3.

Die gewählte Benchmarkmethode ist die Zeitmessung mithilfe der Funktion *clock_gettime* und der Uhr *CLOCK_MONOTONIC*. Für die Durchführung des Benchmarks werden die Berechnungen innerhalb einer Schleife öfter durchgeführt, damit die Zeit zwischen den Messungen lang genug ist. Die Messzeitpunkte sind direkt vor der Schleife und direkt nach der Schleife, somit werden fast keine für die Messung irrelevanten Instruktionen gemessen.

Die Korrektheit der Ergebnisse wurden durch die Überprüfung der Genauigkeit bereits sichergestellt. Weiters wurden für die Benchmarks der Intel-Turboboost und das CPU-Scaling deaktiviert, damit es zu keiner Verzerrung der Ergebnisse kommt, weil der CPU-Takt sich während des Benchmarks drastisch erhöht. Die Benchmarks wurden auf einem lokalen Rechner mit minimalen Hintergrundprozessen ausgeführt, damit die vollständigen Ressourcen zur Verfügung stehen.

Als Testwerte wurden 20 Arrays mit $1 * 10^6$ bis $10 * 10^6$ zufällig generierten Gleitkommazahlen genutzt. Die Berechnung der Kehrwerte der Wurzeln wurde für jedes Array mit der C-Funktion `1/sqrt()` und drei Varianten des Algorithmus, skalar, skalar mit 2-facher Newtoniteration und SIMD-optimiert, durchgeführt. Um mindestens eine Sekunde Abstand zwischen den Messungen sicherzustellen und Störungen zu dämpfen, wurde jede Berechnung einer Implementierung 200 mal durchgeführt. Anschließend wurde die mittlere Laufzeit jeder Implementierung gebildet. Die Ergebnisse sind in den folgenden Grafiken und Tabellen zu sehen.

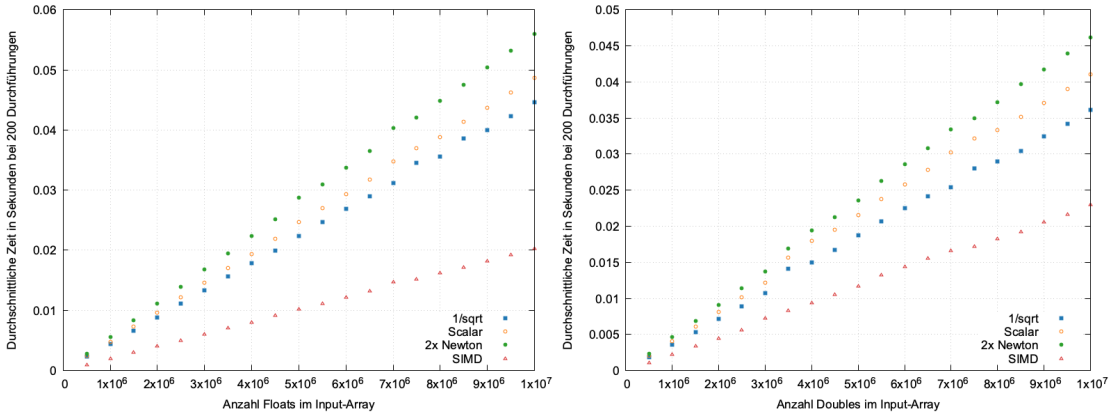

Abbildung 2: Laufzeit der Implementierungen für Float (l) und Double (r)

	Skalar	**Zweifache Newtoniteration**	**SIMD-Optimiert**
Float	0.9181	0.7980	2.2097
Double	0.8731	0.7835	1.5871

Tabelle 3: Durchschnittliches Laufzeitverhältnis von `1/sqrt()` zu Implementierungen

Die Zeitverläufe sind annähernd linear mit Ausnahme von kleinen Abweichungen, was darauf hindeutet, dass die Berechnungen nicht wegoptimiert wurden. Wie zu erwarten war, bringt die zweifache Newtoniteration eine etwas schlechtere Performanz mit sich. Die SIMD-Implentierungen für Floats und Doubles haben jeweils einem Speedup-Faktor von 2.406 und 1.818 zu ihren skalaren Gegenstücken. Auch, wenn für die SIMD-Implementierung für Floats eine höhere Beschleunigung erwartet wurde, bieten beide Faktoren eine bedeutungsvolle Steigerung der Recheneffizienz.

Verwunderlich ist jedoch, dass die Berechnung mit `1/sqrt()` schneller ist als mit dem

Fast Inverse Square Root Algorithmus. Wirft man auf den Blick des generierten Assembler Codes wird sichtbar, dass der GCC-Compiler `1/sqrt()` mit den SSE-Instruktionen `_mm_div_ss/ds und _mm_sqrt_ss/ds` ersetzt. Wird zusätzlich mit der Flag `-ffast-math` kompiliert, wird die Berechnung sogar durch `_mm_rsqrt_ss/sd` ersetzt. Diese Instruktionen für die Division und Wurzelberechnung wurden mit der Intel-SSE Extension veröffentlicht, basieren auf Hardware und liefern daher schnellere Berechnungszeiten, was den Geschwindigkeitsunterschied erklärt.

5 Zusammenfassung und Ausblick

Diese Ausarbeitung bietet einen Einblick in die Funktionsweise des *Fast Inverse Square Root* und präsentiert eine SIMD-optimierte Version Implementierung, auch erweitert für die Berechnung mit Doubles. Die wichtigsten Schritte und die mathematischen Grundlagen hinter dem Algorithmus und der MagicNumber wurden erläutert, wodurch die bemerkenswerte Kombination von Bithacking und numerischer Annäherung zur Geltung kommt. Durch einen *Brute-Force*-Ansatz wurde die MagicNumber für Floats $\boldsymbol{R} = \mathbf{0x5F375A87}$ und für Doubles $\boldsymbol{R_d} = \mathbf{0x5FE6EB50C7B33600}$ ermittelt, die auch eine hohe Genauigkeit liefern und nah an die bereits etablierten MagicNumbers von Lomont und Robertson kommen. Die Genauigkeit und Performanz des Algorithmus im Vergleich zu alternativen Berechnungen wurden durch wissenschaftliche Verfahren erfasst und die Ergebnisse wurden diskutiert. Dabei wurde festgestellt, dass heutige Prozessoren und Compiler dem Algorithmus durch Optimierung und Verwendung von SSE-Instruktionen wie `rsqrtss` oder sogar AVX-Instruktionen wie `vrsqrtss` überlegen sind, der Algorithmus aber für damalige Zeiten eine außerordentlich gute Annäherung mit einer bedeutend schnelleren Berechnungszeit liefern konnte. Weitere Optimierungsmöglichkeiten bestehen darin, die AVX-Extension zu nutzen und somit mehr als 4 Floats beziehungsweise 2 Doubles gleichzeitig zu verarbeiten und die resultierenden Konstanten der Newtoniteration und MagicNumber mit 3D-Optimierung zu verbessern, um noch genauere Ergebnisse zu erzielen [8].

Literatur

[1] Matthew Robertson. *A Brief History of InvSqrt*. The University of New Brunswick, 2012. https://mrober.io/papers/rsqrt.pdf, [Online; Stand 8. Juli 2023].

[2] Recognized as an American National Standard (ANSI). *IEEE Standard for Binary Floating-Point Arithmetic*, 1995. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=30711, [Online; Stand 8. Juli 2023].

[3] Werner Vogt. *Zur Numerik nichtlinearer Gleichungssysteme (Teil 1)*. Technische Universität Ilmenau, 2001. https://www.db-thueringen.de/servlets/MCRFileNodeServlet/dbt_derivate_00008992/IfM_Preprint_M_01_12.pdf#page=30&zoom=auto,-13,488, [Online; Stand 8. Juli 2023].

[4] Quake III Arena. id Software. *q_math.c*, 2012. https://www.github.com/id-Software/Quake-III-Arena/blob/master/code/game/q_math.c, [Online; Stand 8. Juli 2023].

[5] Charles McEniry. *The Mathematics Behind the Fast Inverse Square Root Function Code*. The University of New Brunswick, 2007. http://www.daxia.com/bibis/upload/406Fast_Inverse_Square_Root.pdf, [Online; Stand 8. Juli 2023].

[6] Preetham Narayanareddy. *How Fast Inverse Square Root actually works*, 2022. https://www.preethamrn.com/posts/fast-inverse-sqrt/, [Online; Stand 12. Juli 2023].

[7] *ISO/IEC9899:2017*, 2017. https://web.archive.org/web/20181230041359/http://www.open-std.org/jtc1/sc22/wg14/www/abq/c17_updated_proposed_fdis.pdf, Seite 59, Fußnote 97 [Online; Stand 8. Juli 2023].

[8] Jan Kadlec. *Improving the fast inverse square root*, 2010. http://rrrola.wz.cz/inv_sqrt.html, [Online; Stand 12. Juli 2023].