Fast Inverse Square Root

Ngoc Kim Ngan Nguyen, Yll Kryeziu und Keihan Pakseresht

Grundlagenpraktikum: Rechnerarchitektur

Gruppe 103 – Abgabe zu Aufgabe A300

Sommersemester 2023

Wozu Fast Inverse Square Root?

- Real-Time Rendering & Physikalische Simulation z.B.: **Quake III Arena**
- Unzählige Berechnungen von $\hat{v} = \frac{\vec{v}}{\|\vec{v}\|}$ und $\|\vec{v}\| = \sqrt{(v_1)^2 + (v_2)^2 + (v_3)^2}$

- **Problem:** Division und Wurzelfunktion waren **zu langsam**
- **Lösung:** **Fast Inverse Square Root Algorithmus**

Q_rsqrt

- Quake III Arena Sourcecode wird 2005 veröffentlicht:
- Gebrauch von IEEE 754 und Newton-Raphson-Methode

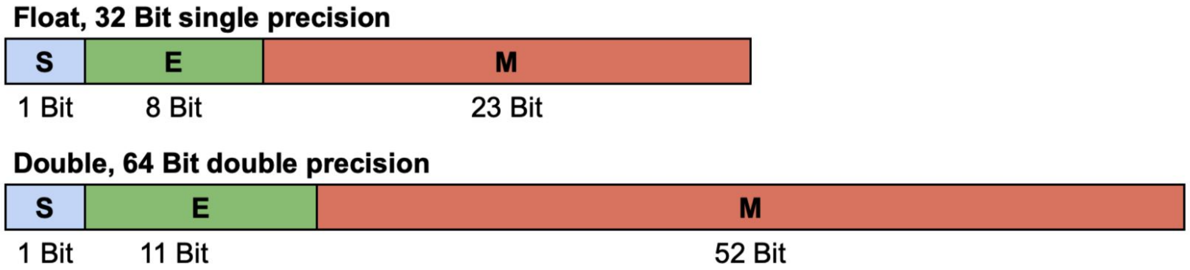
```c
float Q_rsqrt( float number )
{
    long i;
    float x2, y;
    const float threehalfs = 1.5F;

    x2 = number * 0.5F;
    y  = number;
    i  = * ( long * ) &y;                      // evil floating point bit level hacking
    i  = 0x5f3759df - ( i >> 1 );              // what the fuck?
    y  = * ( float * ) &i;
    y  = y * ( threehalfs - ( x2 * y * y ) );  // 1st iteration
//  y  = y * ( threehalfs - ( x2 * y * y ) );  // 2nd iteration, this can be removed

    return y;
}
```

IEEE 754 Standard

- Standardisierte Darstellung für Gleitkommazahlen: $-3*10^4$ statt -30 000
- Zusammensetzung aus **Vorzeichen, Exponent** und **Mantisse**
- Mantisse ist normalisiert, Bias für negative Exponenten



- Numerischer Wert: $$x = (-1)^S * \left(1 + \frac{M}{2^N}\right) * 2^{E-b}$$
- Integerwert: $$x_{\text{bits}} = 2^N * \left(E + \frac{M}{2^N}\right)$$

IEEE 754 Beispiel

Bits: 11000000110110011001100110011010

1 10000001 10110011001100110011010

S = $1_2 \rightarrow \mathbf{1}_{10}$ **E** = $10000001_2 \rightarrow \mathbf{129}_{10}$

M = $10110011001100110011010_2 \rightarrow \mathbf{5872026}_{10}$

$$x = (-1)^S * \left(1 + \frac{M}{2^N}\right) * 2^{E-b} \rightarrow (-1)^1 * \left(1 + \frac{5872026}{2^{23}}\right) * 2^{129-127} = -6.80000019\ldots$$

$$x_{\text{bits}} = 2^N * \left(E + \frac{M}{2^N}\right) \rightarrow 2^{23}\left(129 + \frac{5872026}{2^{23}}\right) = 1088002458$$

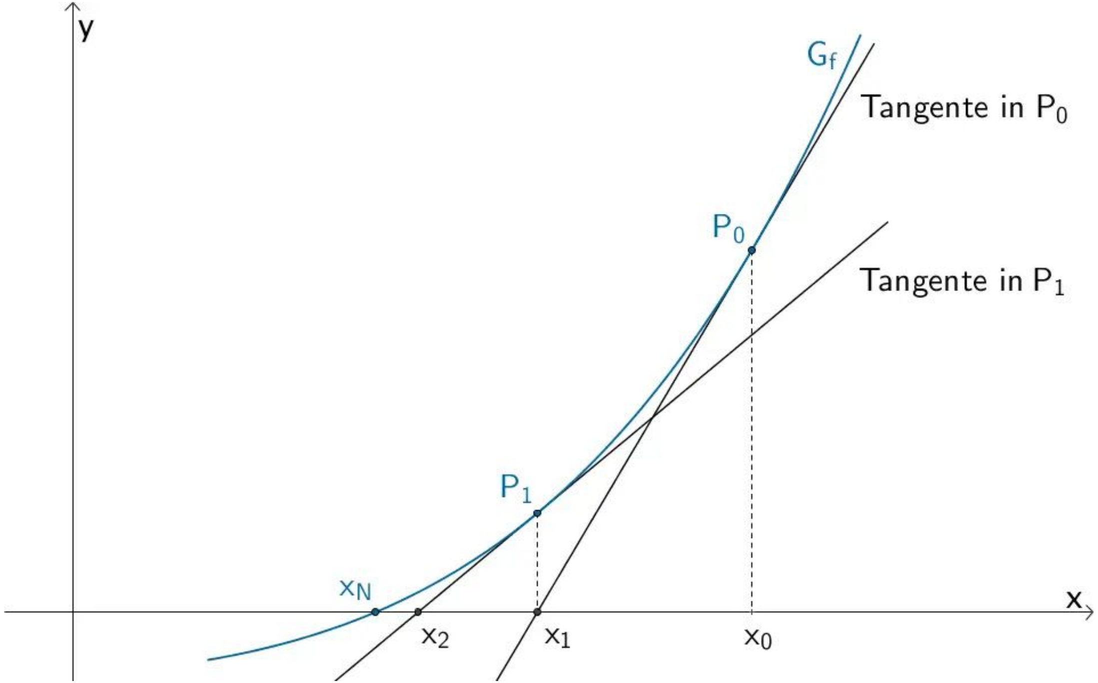
Newton-Raphson-Methode

- Iterative, numerische Methode zum Finden von **Nullstellen**

- 1. Ausgangswert als erste Schätzung wählen
- 2. Nähern an Nullstelle durch Formel
- 3. Wiederholen bis Ergebnis zufriedenstellend ist

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

Newton-Raphson-Methode



Funktionsweise von Q_rsqrt

```
i  = 0x5f3759df - ( i >> 1 );               // what the fuck?
```

i (10.125): 1 0 0 0 0 0 1 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

i >> 1: 0 1 0 0 0 0 0 1 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0x5F3759DF – (i >> 1): 1 0 1 1 1 1 1 0 0 1 1 0 1 1 1 0 1 0 1 1 0 0 1 1 1 0 1 1 1 1 1

– 0 1 0 0 0 0 0 1 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

----

0 1 1 1 1 1 0 1 0 1 0 0 1 1 0 0 1 0 1 1 0 0 1 1 1 0 1 1 1 1 1

$$\boxed{i \approx 0.3249 \approx \frac{1}{\sqrt{10.125}} = 0.31426968052}$$

Funktionsweise von Q_rsqrt

- Erkenntnis: i ist danach erste Schätzung für das Ergebnis, aber wie?
- Vereinfachen von x mit Logarithmus:

$$x = \left(1 + \frac{M}{2^{23}}\right) * 2^{E-127}$$

$$\Rightarrow \log_2(x) = \log_2\left(1 + \frac{M}{2^{23}}\right) + E - 127$$

Funktionsweise von Q_rsqrt

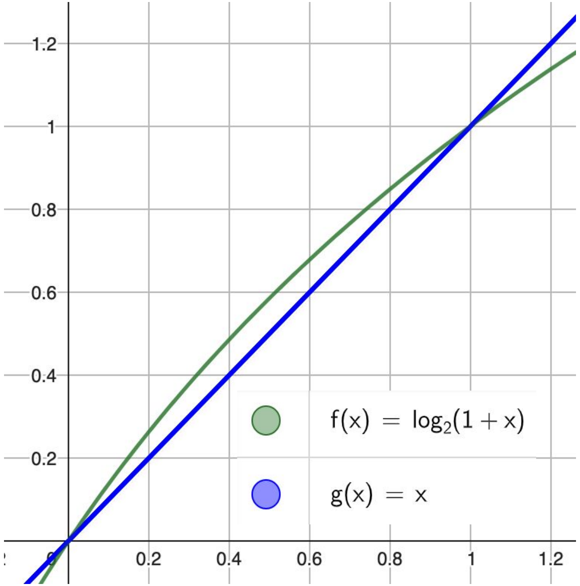
Für eine Zahl c in $[0,1)$ gilt $\log_2(1+c) \approx c + \sigma$

$$\log_2(x) = E + \frac{M}{2^{23}} - 127 + \sigma$$

$$\Rightarrow \log_2(x) = \frac{2^{23} * \left(E + \frac{M}{2^{23}}\right)}{2^{23}} - 127 + \sigma$$

$$\Rightarrow \log_2(x) = \frac{x_{\text{bits}}}{2^{23}} - 127 + \sigma$$

→ Zusammenhang zwischen
Logarithmus und Integerdarstellung



Funktionsweise von Q_rsqrt

→ Zusammenhang zwischen **Logarithmus und Integerdarstellung nutzen**

$$y = \frac{1}{\sqrt{x}}$$

$$\log_2(y) = -\frac{1}{2} * \log_2(x)$$

$$\frac{y_{\text{bits}}}{2^{23}} - 127 + \sigma = -\frac{1}{2} * \left(\frac{x_{\text{bits}}}{2^{23}} - 127 + \sigma\right)$$

$$y_{\text{bits}} = \underbrace{\frac{3}{2} * 2^{23} * (127 - \sigma)}_{\text{MagicNumber}} - \underbrace{x_{\text{bits}} * \frac{1}{2}}_{\texttt{(i >> 1)}}$$

Funktionsweise von Q_rsqrt

- Zusammenhang mit newton'scher Näherung

```
y  = y * ( threehalfs - ( x2 * y * y ) );   // 1st iteration
```

- Aus MagicNumber und Shift → erste Schätzung für **y**
- Zu **y** zugehöriges **x**:

$$y = \frac{1}{\sqrt{x}} \implies x = \frac{1}{y^2}$$

- Aufgabe: finde Nullstellen der Funktion

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

Funktionsweise von Q_rsqrt

- Einsetzen von **f(y) und f’(y)** in Formel von Newton-Raphson:

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

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

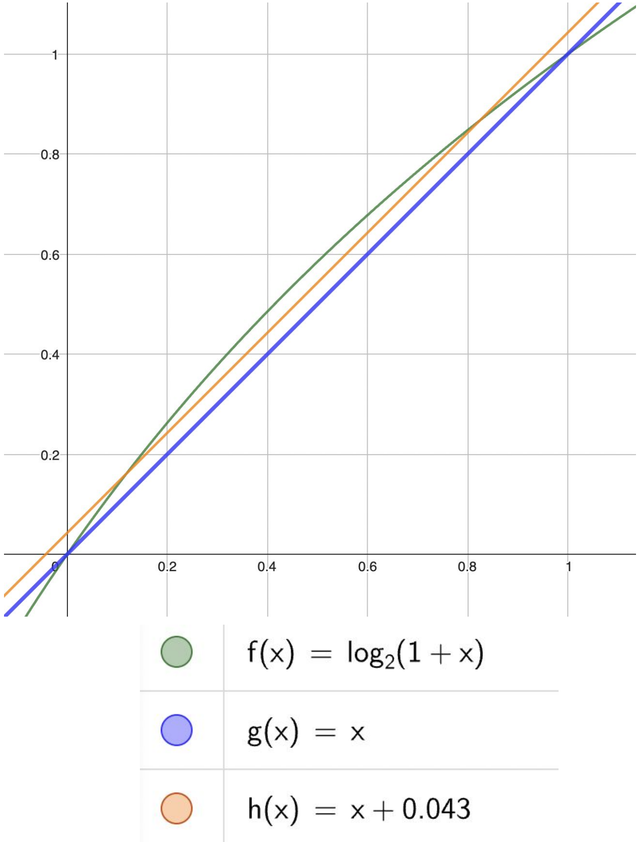
Die MagicNumber

1. Die MagicNumber für Floats

Ansatz: *Brute Force*

- Eingrenzung der Testwerte von *R*:
  zwischen 0x5F300000 und 0x5F400000

$$
\begin{aligned}
R &= \frac{3}{2} * 2^{23} * (127 - \sigma) \\
&= \frac{3}{2} * 2^{23} * 127 - \frac{3}{2} * 2^{23} * \sigma \\
&< \frac{3}{2} * 2^{23} * 127 = \text{0x5F400000}
\end{aligned}
$$



Die MagicNumber

1. **Die MagicNumber für Floats**

   Ansatz: *Brute Force*

- Eingrenzung der Testwerte von *R*: zwischen 0x5F300000 und 0x5F400000
- Teste von *R* in Schritten Ziffer für Ziffer: Inkrementierung von 0x10000, 0x1000,...
- Ignorierung der Exponenten der möglichen Floats

Ergebnis: $R = \text{0x5F375A87}$

Maximaler relativer Fehler: 0.1751350679%

Die MagicNumber

2. Die MagicNumber für Doubles

Ansatz: *Brute Force* mit weiteren Eingrenzungen

- Erste Startschätzung: $R = \frac{3}{2} * 2^{23} * (127 - \sigma)$

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

$$R_d = \frac{3}{2} * 2^{52} * \left(1023 - \left(127 - \frac{R}{\frac{3}{2} * 2^{23}}\right)\right)$$ = 0x5FE6EB50E0000000

→ Testen von Werten um 0x<u>1000</u>0000 kleiner oder größer von dieser Zahl und Inkrementierung von 0x<u>1000</u>0000 nach jeder Iteration

Die MagicNumber

2. Die MagicNumber für Doubles

Ansatz: *Brute Force* mit weiteren Eingrenzungen

- Erste Startschätzung: $R_d = \text{0x5FE6EB50E0000000}$
- Größere Inkrementierung: Inkrementierung der Integerdarstellung von Doubles nach jeder Überprüfung um 0x10000000 = 268435456

Ergebnis: $R_d = \text{0x5FE6EB50C7B33600}$

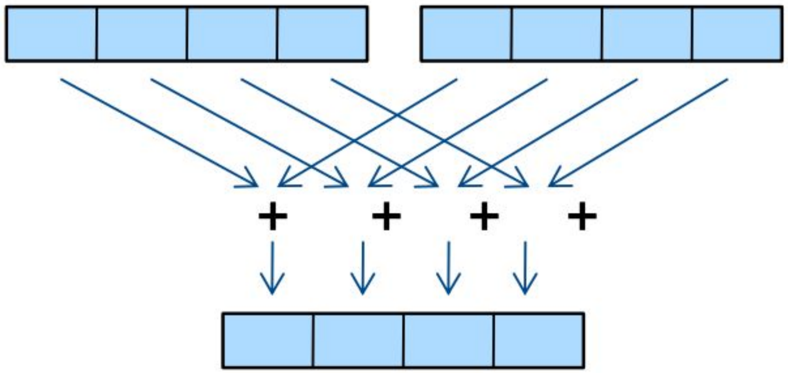
Maximaler relativer Fehler: 0.1751183669%

Die MagicNumber

Vergleich mit Robertsons MagicNumbers

Float	0x5F375A87	0.1751350679%
	0x5F375A86	0.1751341630%
Double	0x5FE6EB50C7B33600	0.1751183669%
	0x5FE6EB50C7B537A9	0.1751183671%

Implementierung und Optimierungen



- Inputs in Array → optimieren mit **SIMD**
- **Single Instruction, Multiple Data**
- z.B. Intel SSE, AVX…
- Parallelität ausnutzen → höhere Rechenleistung
- **Loop Unrolling** → die nächsten 4 Floats bzw. 2 Doubles gleichzeitig, Rest skalar abarbeiten
- 4 Floats bzw. 2 Doubles bearbeiten mit **SSE-Instruktionen**
- Algorithmus abarbeiten, optimierte MagicNumber von **Lomont/Robertson**

Implementierung und Optimierungen

- **union** für type-punning

```
y = number;
i   = * ( long * ) &y;
/*...*/
y   = * ( float * ) &i;
```

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

__m128

float 0	float 1	float 2	float 3
0...31	32...63	64...95	96...127

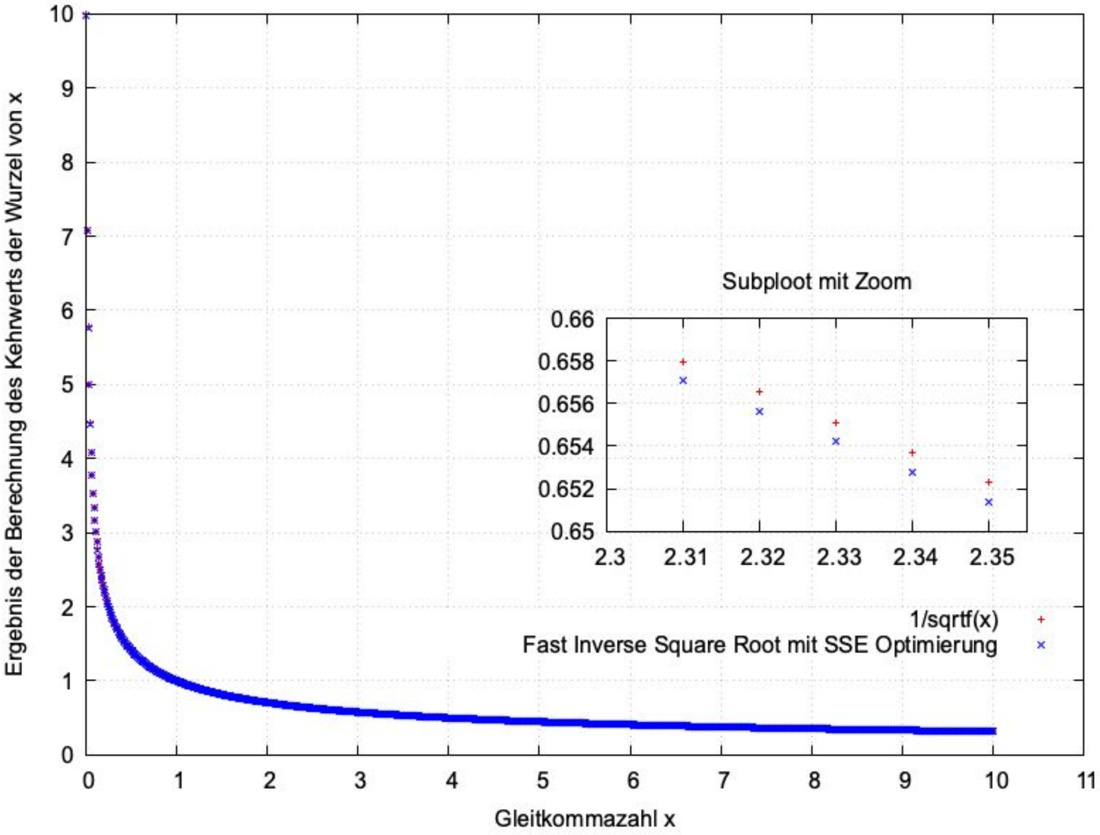
__m128d

double 0	double 1
0...63	64...127

Ganzer Code zu unübersichtlich, wichtige Instruktionen:

- Vektoren multiplizieren: **_mm_mul_ps**
- Vektoren subtrahieren: **_mm_srli_epi32**
- Vektoren shiften: **_mm_sub_ps**
- Äquivalente Instruktionen für Doubles

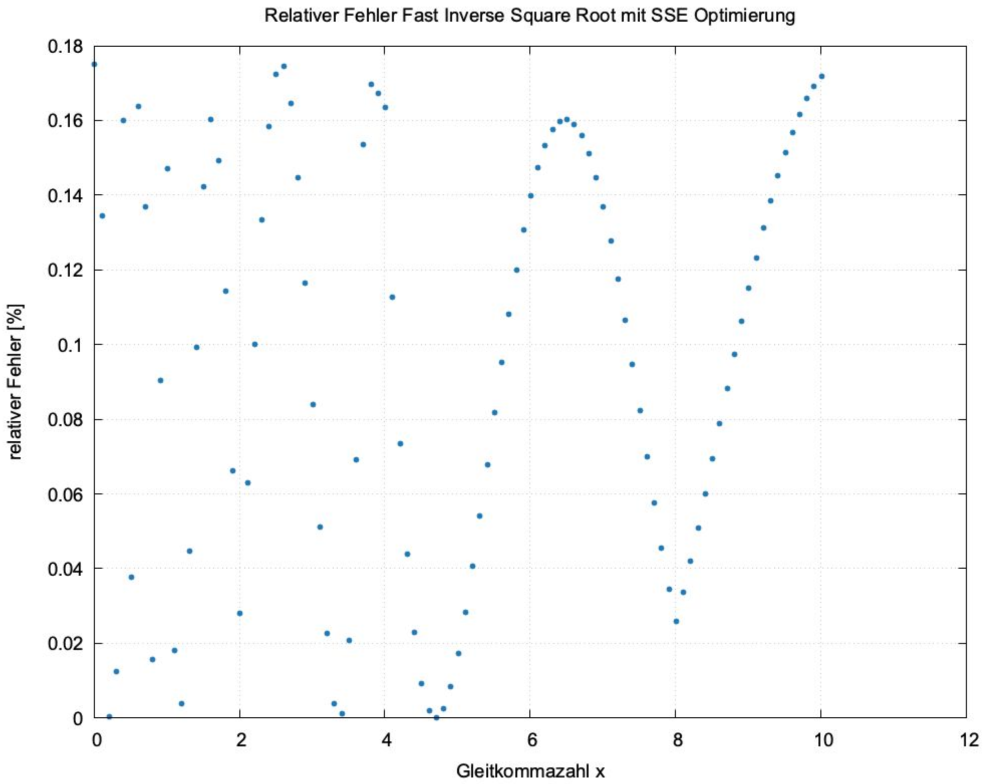
Bestimmung der Genauigkeit



→ sehr nah an exakter Berechnung

kompiliert mit -O2

Bestimmung der Genauigkeit



→ (fast) alle Inputs prüfen

Bestimmung der Genauigkeit

- Positive Floats von **0x00800000** bis **0x7f800000** mit Inkrement 1
- Positive Doubles von **0x10000000000000** bis **0x7ff0000000000000**
  - Inkrement 1 dauert zu lange, inkrementiere mit **1LLU<<30**
- MagicNumber **0x5F3759DF** hat maximalen rel. Fehler von: **0.17522874%**
- Fehler **SIMD-Implementierung** mit optimierten MagicNumbers:

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

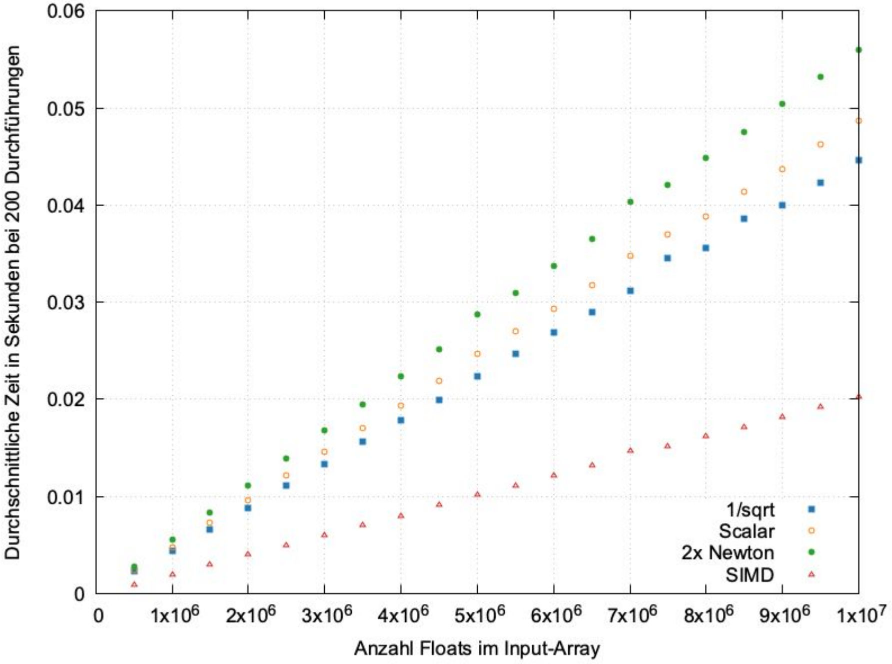
Maximale relative Fehler [%]

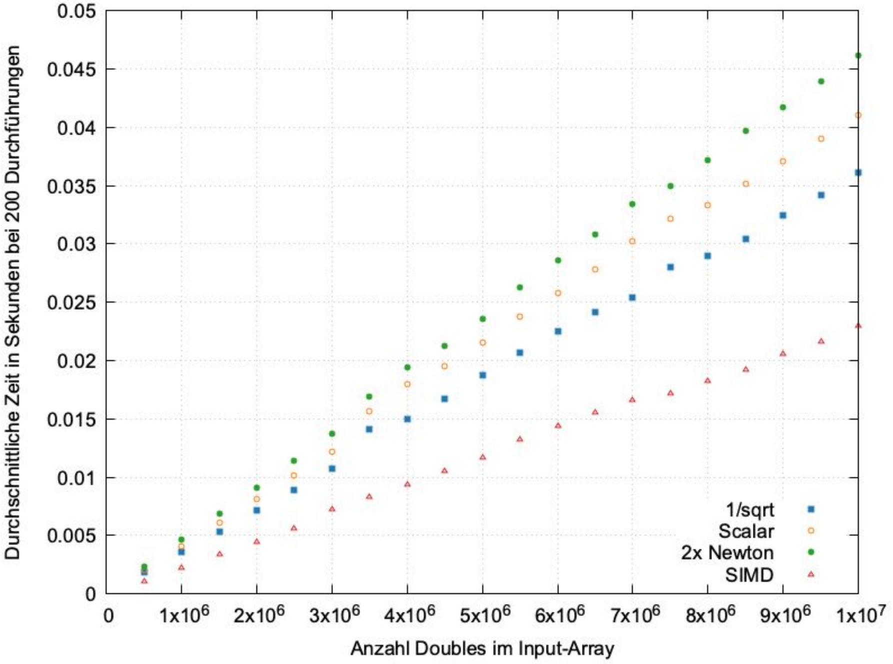
Performanzanalyse

- Zeitmessung mit **clock_gettime** und **CLOCK_MONOTONIC**
- Berechnung mehrmals durchführen und Mittelwert bilden
- Arrays verschiedener Größe mit zufälligen Werten bei jedem Durchgang
- Möglichst gleiche Bedingungen bei Durchführung sicherstellen
  - CPU-Scaling, Turboboost, geteilte Ressource vermeiden, Hintergrundprozess beenden

- Getestet: **Standard 1/sqrt, Skalar-Version,**
  2x Newton-Version und SIMD-Version mit Optimierung **-O2**
- **200** Durchführungen, **20** Arrays von Größe $\mathbf{10^6}$ bis $\mathbf{10^7}$

Performanzanalyse





Performanzanalyse

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

Durchschnittliches Laufzeitverhältnis von 1/sqrt() zu Implementierungen

- SIMD-Speedup zu skalarer Version:
  - Float: **2.406** und Double: **1.818**
- **Standard 1/sqrt ist schneller als skalarer Fast Inverse Square Root**
- Compiler optimiert **1/sqrt** zu Instruktionen **_mm_sqrt_ps** und **_mm_div_ss**
- mit **-ffast-math Flag** sogar Optimierung zu **_mm_rsqrt_ps**
- mit **AVX-Extension** auch **vrsqrtss**

Quellen

Q_rsqrt Source: Quake-III-Arena/code/game/q_math.c at master

IEEE 754 Beispiel: Fast Inverse Square Root - ppt download

SIMD: TUM ERA Vorlesung

Erklärung der MagicNumber Zeile: How Fast Inverse Square Root actually works - Preetham

Robertson FISQ Paper: A Brief History of InvSqrt