LEHRSTUHL FÜR RECHNERARCHITEKTUR UND PARALLELE SYSTEME

Praktikum Rechnerarchitektur

Gruppe 233 – Abgabe zu Aufgabe A406

Sommersemester 2021

Fikret Ardal Mert Corumlu Denis Paluca

1 Einleitung

Die Entropie (engl.: Shannon entropy, sym.: H) ist in der Informationstheorie ein Maß für den mittleren Informationsgehalt einer Nachricht. Das Informationsgehalt (auch Überraschungswert gennant) eines Ereignisses nimmt mit zunehmenden Wahrscheinlichkeit des Ereignisses ab. Je unwahrscheinlicher das Ereignis ist, desto überraschender ist es, falls es geschieht. Formel für die Entropie eines Zufallsvariable X:

$$\mathrm{H}(X) = -\sum_{x \in X} \mathrm{P}(X = x) \cdot \log_2(\mathrm{P}(X = x))$$

Mit der Entropiefunktion kann man sehen, wie gleichmäßig die Wahrscheinlichkeiten verteilt sind. Daher liegt die Entropie zwischen 0 und $\log_2(|X|)$. Es ist 0, wenn nur eine Ereignis möglich ist, und $\log_2(|X|)$, wenn es sich um eine Gleichverteilung handelt.

In der gegebenen Aufgabe geht es darum, die Entropie einer Zufallsvariable zu berechnen, gegeben sei die Wahrscheinlichkeitsverteilung der Zufallsvariable. Dafür muss man ein geeignetes Rahmenprogramm und die Entropiefunktion implementieren. Das Programm soll auf Genauigkeit und Performanz getestet werden.

Die Wahrscheinlichkeitsverteilung soll als Textdatei übergeben werden. Das Format ist wie folgt:

- Die Datei enthält eine Liste von Gleitkommazahlen
- Die Zahlen sind mittels Leerzeichen aufgeteilt
- Die Zahlen liegen zwischen 0 und 1 (inklusive)
- Die Summe der Zahlen ist gleich 1 (mit einer Fehlerspanne)

Die Datei wird von dem Rahmenprogramm, das in C programmiert ist, eingelesen. Die eingelesenen Zahlen werden in ein Array mit Fließkommazahlen gespeichert und als Pointer dem Assemblerprogramm übergeben. Das Assemblerprogramm berechnet die Entropie und danach gibt das Rahmenprogramm die Entropie aus. Die Performanz und die Genauigkeit der Implementierung werden mittels Vergleich zwischen einer Basisimplementierung und verschiedenen Implementierungen festgestellt.

In den folgenden Abschnitten wird der Lösungsansatz genauer erläutert. Insbesondere fokussieren wir auf die Logarithmusfunktion und auf die angewendeten Techniken zur

SIMD-Vektorisierung der Entropiefunktion. Danach besprechen wir die Genauigkeit und Performanz verschiedene Implementierungen und welche Art von Trade-offs sie eingehen. Wir schließen die Arbeit mit einer kurzen Zusammenfassung unserer Ergebnisse sowie mit einigen abschließenden Gedanken über mögliche zukünftige Entwicklungen.

2 Lösungsansatz

Bei der Berechnung der Entropie ist der zeitaufwendigste Teil im Code die Berechnung des Logarithmus. Zur Berechnung der Logarithmusfunktion gibt es drei Hauptmethoden:

- Approximation als Polynom
- Lookup Tabelle
- Approximation mit Hilfe von Lookup Tabelle (glibc Methode).

Da wir mehrere Logarithmusfunktionen ausprobieren möchten, haben wir die Signatur der Entropiefunktion entsprechend erweitert. Man soll jetzt auch den Pointer der Logarithmusfunktion angeben.

Die Entropiefunktion an sich ist auch nicht direkt implementiert, wie von der Gleichung spezifiziert, sondern es nutzt die Kahan-Summe-Algorithmus. [2] Die Kahan-Summe hilft die Auslöschung bei einer Reihe von Additionen zu vermeiden.

2.1 Approximation

Der erste Ansatz an dieses Problem war, eine Polynomdarstellung der Logarithmusfunktion zu finden, die nicht aufwendig zu berechnen ist und auch für kleine Werte von x schnell genug konvergiert.

Nach dem Testen einiger Möglichkeiten haben wir uns für zwei verschiedene Methoden zur Approximation der Logarithmusfunktion entschieden. Diese Methoden sollen leistungsfähig, präzise und vektorisierbar sein.

Da jede Gleitkommazahl die Form $x = 2^k \cdot z$ für ein $k \in \mathbb{Z}$ und $z \in \mathbb{Q}$ hat, können wir zuerst den Exponenten extrahieren und dann Logarithmus für z berechnen die zwischen 1 und 2 liegt.

$$\log_2 x = k + \frac{\ln z}{\ln 2}$$

Bei den folgenden Approximierungsmethoden haben wir immer berücksichtigt, wie anpassungsfähig diese Methoden an die SIMD-Version sind, was für den Performance-Gewinn eine große Rolle spielen wird.

2.1.1 ARTANH Approximation

Wir können $\ln(x)$ als eine konvergente Summe von Potenzreihen schreiben, indem wir die Taylorreihe von artanh verwenden. Die Partialsumme der Potenzreihe konvergiert für kleine Werte von x schneller als die normale Taylorreihe von $\ln(x)$.

$$\ln(x) = 2 \cdot \operatorname{artanh}\left(\frac{x-1}{x+1}\right) = 2\left(\sum_{k=0}^{\infty} \frac{1}{2k+1} \cdot \left(\frac{x-1}{x+1}\right)^{2k+1}\right) \tag{1}$$

$$\text{Sei } S_n = 2\left(\sum_{k=0}^{n} \frac{1}{2k+1} \cdot \left(\frac{x-1}{x+1}\right)^{2k+1}\right) \tag{2}$$

Obwohl die Artanh-Approximation eine Division erfordert, ist der absolute Unterschied zwischen Logarithmusfunktion und Artanh-Approximation mit S_2 fast 0, wenn die Zahl zwischen 1 und 2 liegt.

2.1.2 Remez Algorithmus

Der Remez-Algorithmus ist ein Minimax-Approximations-Algorithmus, das die maximale absolute Differenz zwischen dem Polynom und der gegebenen stetigen Funktion im Intervall $[a, b]$ minimiert [1].

Um Polynome zur Approximation von $\ln(x)$ im Intervall $[1, 2)$ zu erzeugen, haben wir ein Open-Source-Projekt namens „lolremez“ [4] verwendet. Wir haben 2 verschiedene Polynome mit dem Grad 2 und 4 implementiert.

$$\mathrm{P}_2(z) = -0.344845 \cdot z^2 + 2.024658 \cdot z - 1.674873 \tag{3}$$

$$\mathrm{P}_4(z) = -0.081616z^4 + 0.645142z^3 - 2.120675z^2 + 4.070091z - 2.512854 \tag{4}$$

Das Polynom mit Grad 2 liefert ungenauere Ergebnisse, gewinnt aber auch etwas an potenzieller Geschwindigkeit, da es 2 Multiplikationen weniger hat.

2.2 Lookup Tabelle

Ein anderer Ansatz besteht darin, eine sogenannte Lookup Tabelle zu benutzen. Anstatt dass wir der in Mantisse gespeicherte Wert zu $log_2(x)$ annähern, speichern wir vorher diese Werte in einer Tabelle und geben sie zurück gemäß der Mantisse. Für das Erhalten der Genauigkeit von 100% brauchen wir insgesamt 2^{23} Einträge in der Tabelle, welche aber nämlich

$$2^{23}.4\,\text{bytes} = 2^{15}\,\text{Kb} = 32\,\text{Mb}$$

Unter Berücksichtigung das Experiment von der Universität Berkeley [3] haben wir uns entschieden, nur erste 16 bits von der Mantisse für die Tabelle zu benutzen und die letzten 7 bits zu ignorieren.

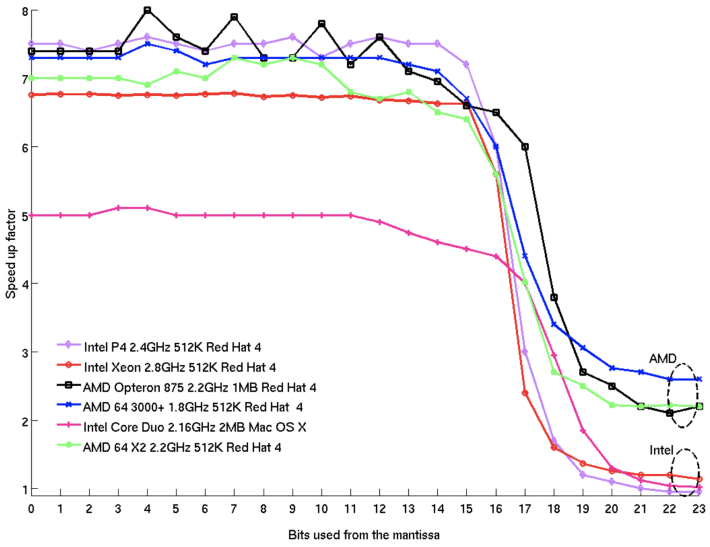

Abbildung 1: Performanz-Genauigkeit Trade-Off der logf mit Lookup Tabelle auf unterschiedliche Systeme

Weil wir die Tabelle somit verkleinert haben, können wir die Cache effizienter benutzen um Performanz zu erhöhen, jedoch verlieren wir die Genauigkeit im Durchschnitt um $6.55 \cdot 10^{-6}$. Das ist einer der Trade-Offs, die wir während der Entwicklung begegnet und sie werden in folgender Abschnitte eingegangen.

2.3 Glibc - log2f

Die glibc Implementierung von $log2f$ nutzt eine Kombination von Approximation und Lookup Tabelle. Erst ist x in Mantisse z und Exponent k geteilt.

$$log_2(x) = log_2(2^k \cdot z) = k + log_2(z)$$

Mittels z ist ein Index berechnet, mit dem man in der Lookup-Tabelle eine Variable c ($\approx z$) und $log_2(c)$ findet.

$$log_2(x) = k + log_2(z/c) + log_2(c)$$

Jetzt muss man nur $log_2(\frac{z}{c})$ berechnen, das wird durch die Taylor-Reihe approximiert. Das liefert genaue Ergebnisse, weil $\frac{z}{c}$ sehr nähe zu 1 ist, wo die Taylor-Reihe sehr schnell konvergiert.

2.4 SIMD

Für unsere Algorithmen sind Optimierungen durch SIMD deutlich möglich und sinnvoll. SIMD-Befehle nutzen XMM Registern, die 128 Bits haben. Unsere Algorithmen arbeiten mit Fließkommazahlen, die 32 Bits haben. Daher können wir mittels SIMD-Befehle 4 Einträge gleichzeitig bearbeiten. Dadurch kann man ein theoretischen maximalen Speedup von 4 erreichen.

Natürlich sind die Logarithmusfunktionen ebenfalls vektorisiert. Hier wird einer der größten Vorteile der Verwendung von Approximationen deutlich. Die Approximationsfunktionen verwenden nur eine Reihe von sehr einfachen Operationen, wie z.B. Addition, Multiplikation, Bitverschiebung und so weiter. Diese Operationen sind in SIMD mit wenig bis gar keinem Overhead replizierbar. Daher ist es möglich, sich dem maximalen Speedup anzunähern.

Im Gegensatz dazu ist die Vektorisierung von der Lookup-Version unsere Logarithmusfunktion deutlich ineffizienter. Den Index zu finden für jeden Eintrag, kann man noch gut vektorisieren. Mit 2 Befehle findet man das Index von alle 4 Einträge. Nachdem man die Indices hat, muss man jeden Eintrag der Lookup-Tabelle einzeln in das Register laden. Da diese Einträge in der Lookup-Tabelle höchstwahrscheinlich weit voneinander entfernt sind, fallen viele Cache-Misses an.

2.5 Umgang mit denormalen Zaheln

Um die oben beschriebenen Methoden an die denormalen Zahlen anzupassen ist eine spezielle Behandlung benötigt. Unsere Methoden funktionieren für die Zahlen $x = 2^k \cdot z$ wo $1 \leq z < 2$. Bei den denormalizierten Zahlen liegt z im Interval $[0, 1)$, wo unsere Approximationen sehr ungenau sind. Deswegen müssen die denormalen Zahlen erst normaliziert werden.

Dafür multiplizieren wir die Zahlen mit 2^{23}, deren Exponent gleich 0 ist. Unsere Zahl ist damit normaliziert, aber ist jetzt 2^{23} mal mehr als wir erwarten würden.

$$log_2(x \cdot 2^{23}) = log_2(x) - log_2(2^{23}) = log_2(y) - 23$$

Aus diesem Grund subtrahieren wir 23 von dem Exponent und führen wir die normalen Schritte für die Logarithmusfunktion.

3 Genauigkeit

Die Genauigkeit des Programms ist stark abhängig von der Genauigkeit unsere log2-Implementierung. Für die log2-Approximationen können wir für die Genauigkeit eine analytische Untersuchung durchführen. Wir sind auf die Differenz zwischen log2 und unseren Approximationen der log2 interessiert. Dann finden wir die Ableitung der Differenz, um das Maximum/Minimum im Bereich von 1 bis 2 herauszuziehen.

Log2 Approx DEG2 – $P_2(x)$ (3)

$$(\log_2(x) - \mathrm{P}_2(x))' = \frac{1}{x \cdot \ln(2)} + 0.68969 \cdot x - 2.024658$$

Nullstellen: $x_1 \approx 1.718089, x_2 \approx 1.217517$

Der größte absolute Fehler für diese Approximation ist ungefähr $4.941 \cdot 10^{-3}$.

Log2 Approx DEG4 – $P_4(x)$ (4)

$$(\log_2(x) - \mathrm{P}_4(x))' = \frac{1}{x \cdot \ln(2)} + 0.32646 \cdot x^3 - 1.93543 \cdot x^2 + 4.24135 \cdot x - 4.07009$$

Nullstellen: $x_1 \approx 1.08506, x_2 \approx 1.31912, x_3 \approx 1.62916, x_4 \approx 1.89513$

Der größte absolute Fehler für diese Approximation ist ungefähr. $8.752644 \cdot 10^{-5}$.

Log2 Approx ARTANH – $S_2(x)$ (2)

$$(\log_2(x) - \mathrm{S}_2(x) \cdot \frac{1}{ln(2)})' = \frac{x^6 - 6x^5 + 15x^4 - 20x^3 + 15x^2 - 6x + 1}{\ln(2)\, x\, (x+1)^6}$$

Nullstellen: $x = 1$

Auf die Stelle x gleich 1 ist die Differenz 0. Aber, der größte Wert im Bereich findet sich auf die Stelle x = 2, weil der Funktion steigend ist. Der größte absolute Fehler für diese Approximation ist ungefähr $2,063996 \cdot 10^{-4}$.

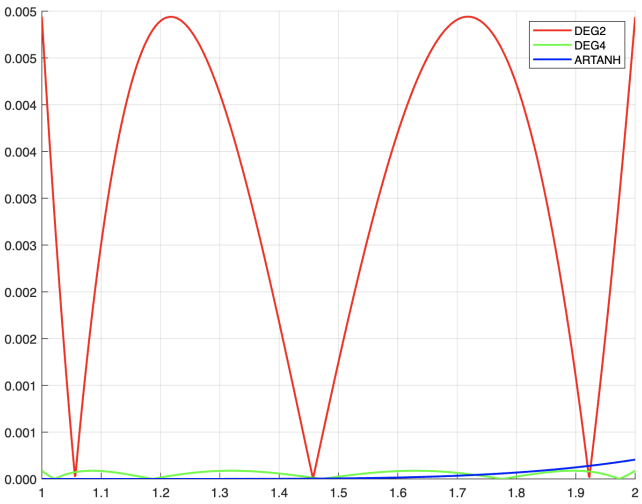

Abbildung 2: Absolute Fehler der Logarithmusfunktionen im $[1, 2)$

Wir erwarten, dass die DEG4-Approximation uns bessere Ergebnisse liefert, weil es den kleinsten Fehler hat. (Siehe Abbildung 3)

Wir haben nur den maximalen absoluten Fehler für die log2 im Bereich [1,2] berechnet. Am Ende wird der Fehler für die log2 Approximation noch kleiner, weil diese Zahl mit dem log2 vom Exponenten addiert wird. Der log2 des Exponenten ist im Intervall [-126,0] und der log2 der Mantisse ist im Intervall [0,1]. Daher sinkt der relative Fehler mit der Addition. Der Fehler für die Lookup Version der log2 mit der empfohlene 16-Bit-Mantisse ist $6.55 \cdot 10^{-6}$. [3]

Die Berechnung der Entropiefunktion enthält eine Multiplikation von der Logarithmusfunktion mit einem Wert x, der immer kleiner gleich 1 ist, welches die Abnahme des absoluten Fehlers verursacht.

Bei einer naiven Implementierung der Entropiefunktion gibt es auch einen Unterschied in Genauigkeit zwischen der Skalar- und SIMD-Version der Entropie. Wobei der SIMD Version genauer ist. Da der SIMD Version 4 Teil-summen berechnet, werden weniger Additionen von Auslöschung beeinflusst. Immer noch haben wir bemerkt, dass in beiden Fällen unsere Ergebnisse sehr stark von Auslöschung beeinflusst wurden.

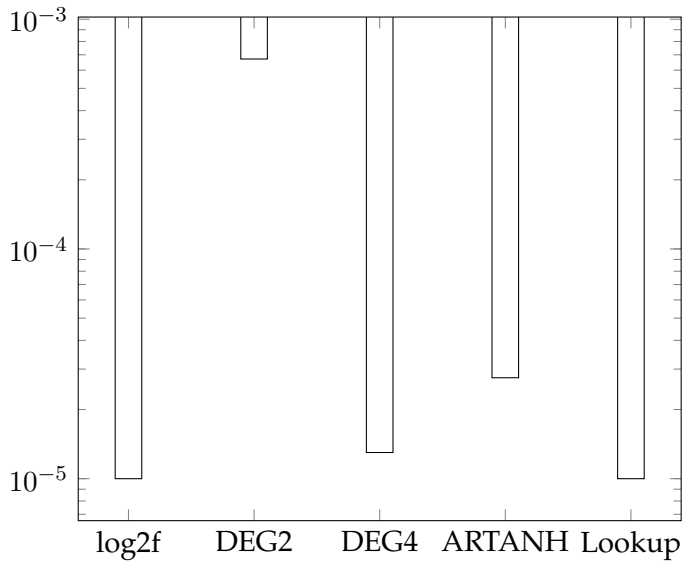

Abbildung 3: Entropiefehler Größenordnung (Durchschnitt bei zufälligen Daten)

Wie schon erwähnt in dem Ansatz, nutzen wir die Kahan-Summe [2], um Auslöschung zu vermeiden. Die Kahan-Summe speichert der unbehandelte Teil einer Addition in die Variable Kompensation und bei der nächste Addition wird die Kompensation aufaddiert. Die Kahan-Summe nimmt an, dass die Summe größer als der nächste zu addierende Zahl ist. Diese Annahme gilt bei uns auch. Wenn wir die Kahan-Summe verwenden sind die Ergebnisse bei SIMD und Skalar fast gleich.

4 Performanzanalyse

Die Performanz der Entropiefunktion hängt von vielen Faktoren ab. Wir wenden trotzdem einige Optimierungen an, die in der Theorie die Performanz verbessern sollen. In dieser Abschnitt geht es darum, wie unsere Optimierungen auf die Performanz auswirkt.

Die erste Optimierung liegt an SIMD Prinzip. Die Entropiefunktion lässt sich sehr einfach vektorisieren, außer der Verwendung von log2f-glibc und der Lookup-Tabelle.

Um unnötige Überprüfungen in unsere SIMD-Implmentation zu vermeiden, erweitern wir das Array der Fließkommazahlen, sodass die Länge durch 4 trennbar ist. Die zusätzliche Zahlen sind alle 0 und sie verändern nicht das Endergebnis. Außerdem allozieren wir das Array 16-Byte-Aligned, dann können wir Befehle, wie z.B. *movaps*, nutzen, die effizienter sind als ihre unaligned Gegenstücke.

Wir speichern die Konstanten für die Remez und Artanh Approximations im Speicher in der Reihenfolge der Verwendung. Das führt zu schnelleren Speciherzugriffen, weil dann die Konstanten in Cache geladen und von der Cache gelesen werden können.

In allen Implementationen behandeln wir die Spezialfälle nicht, nämlich NaN, $\pm$Inf, ± 0 und negative Zahlen nicht. Der Grund ist wir stellen uns in der Entropiefunktion sicher, dass in die Log-Funktion eingegebener Wert immer im Interval $[0, 1]$ liegt. Wenn der Wert 0 ist, spielt das Ergebnis von der Log-Funkiton keine Rolle denn es wird später mit 0 multipliziert. Damit sparen wir einige Branches in der Funktion, was gut auf die Performanz wirkt.

Unsere Logartihmusfunktionen sind stark für Entropieberechnung spezialisiert. Dazu lassen wir in der Assembly Implementation einige ABI Konventionen aus. Wir sichern Caller-Saved Registern in Entropiefunktion nicht denn wir wissen genau welche Registern in der Log-Funktion geändert wird. Dies erlaubt einen bemerkenswerten Gewinn an der Performanz.

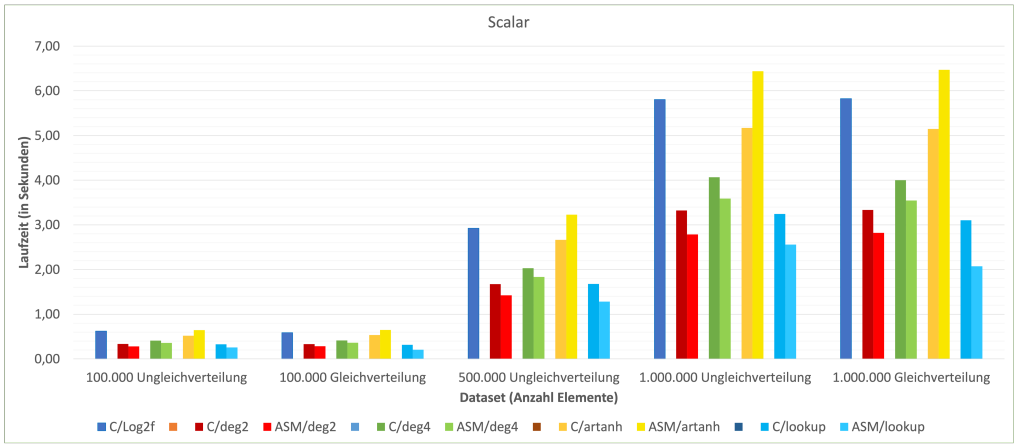

Abbildung 4: Performazgrafik zu skalarer Implementation mit 1000 Iterationen

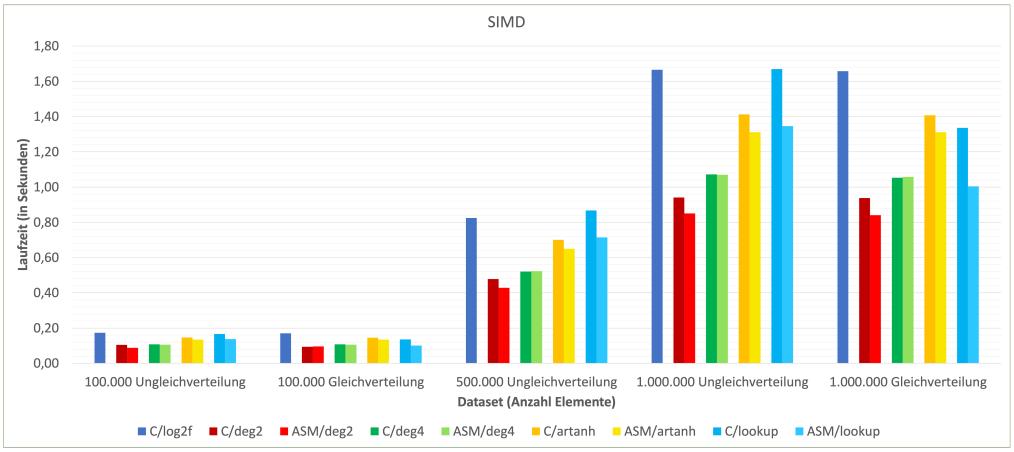

Abbildung 5: Performazgrafik zu SIMD Implementation mit 1000 Iterationen

Um Grafiken zu erstellen, haben wir das Programm mit GCC 11.2 und Flag $-O3$ kompiliert und die Zeitmessungen auf Arch Linux 3.2 mit Kernel Version 5.12.8 - Intel i7-10750H @ 2.6 GHz mit 1000 Iterationen durchgeführt.

In den Grafiken ist es zu sehen, dass unsere Logarithmusfunktionen leistungsfähiger als glibc Implementation. Darüber hinaus sind wir in der Lage, mittels Assembly unsere Funktionen noch besser als C implementieren.

Die Logarithmusfunktion mit Lookup Tabelle ist etwas schneller mit Gleichverteilungen gegen Ungleichverteilungen. Dies liegt daran, dass bei der Gleichverteilung immer der gleiche Wert von der Cache abgefragt wird.

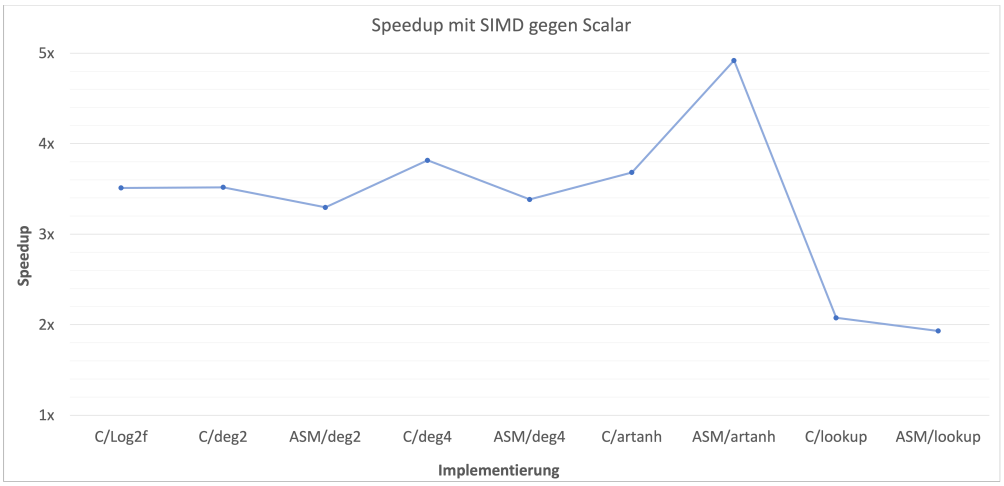

Abbildung 6: Speedup mit SIMD Implementation

Obwohl die Glibc-Implementierung der Logarithmusfunktion ebenfalls eine Lookup-

Tabelle verwendet, erreichen wir einen Speedup von 3.5. Der Grund dafür ist, dass, da die Lookup Tabelle eine Größe von 16 hat, fast alle Werte immer im Cache sind. Ein weiterer Grund ist, dass wir nicht auf Sonderfälle NaN, $\pm$Inf, ± 0 und negative Zahlen überprüfen, was uns einige Branches erspart.

Das durchschnittliche Speedup, das wir mittels unserer SIMD-Implementiereung außer Lookup-Tabelle ist 3.5. Zusätzlich ist die Speedup bei der Lookup Tabelle mittels SIMD gegen Skalar fast 2 Mal wegen der notwendigen Speicherzugriffe.

5 Zusammenfassung und Ausblick

Das Entropiefunktion wurde mit unterschiedlichen Algorithmen erfolgreich implementiert und mit einer Referenzimplementierung verglichen. Die Verwendung von Approximationen für die Logarithmusfunktion hat es uns leichter gemacht, vektorisierte Versionen unsere Algorithmen zu implementieren.

Der Einsatz von SSE-Registern und SIMD-Befehle hat für eine deutliche Performance-Steigerung gesorgt. Was wir nicht erwartet hatten, war die Verbesserung der Genauigkeit durch die Kahans-Summe. Wir haben auch festgestellt, dass die Leistung der Assembler-Implementierung in vielen fällen deutlich besser als der C-Implementierung war.

Zusammenfassend lässt sich sagen, dass wir ASM-SIMD-DEG4 Implementierung empfehlen, da es leistungsfähig und ausreichend genau ist. Nur in Fällen, wenn man noch höheren Genauigkeit braucht, empfehlen wir ASM-SIMD-Lookup Tabelle Implementierung.

Literatur

[1] E. Ya. Remez. Sur la détermination des polynômes d'approximation de degré donnée. *Comm. Soc. Math. Kharkov*, 1934.

[2] William Kahan. Further remarks on reducing truncation errors. *Communications of the ACM*, 8(40), 1965.

[3] O. Vinyals, G. Friedland, N. Mirghafori. Revisiting a basic function on current cpus: A fast logarithm implementation with adjustable accuracy. *Berkeley University*, June 2008. `http://www.icsi.berkeley.edu/pubs/techreports/TR-07-002.pdf`, visited 2021-06-16.

[4] @samhocevar. A remez algorithm implementation to approximate functions using polynomials. `https://github.com/samhocevar/lolremez`, visited 2021-06-16.