Lehrstuhl für Rechnerarchitektur und Parallele Systeme

Grundlagenpraktikum: Rechnerarchitektur

Gruppe 266 – Abgabe zu Aufgabe A406
Sommersemester 2022

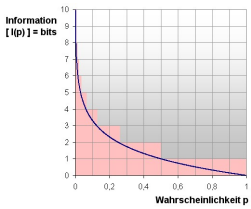
Jonathan Schilling Victor Lamberti Caridad Monder Rammouz

1 Einleitung

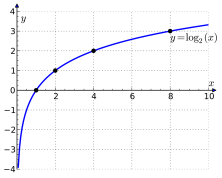
Der Begriff der Entropie wurde von Claude E. Shannon (1916-2001) im Jahr 1948 in „A Mathematical Theory of Communication“ eingeführt. Der Begriff prägt seitdem die moderne Informationstheorie, ein Feld, in welchem man sich allgemein mit dem Kodieren von Nachrichten aufgrund von statistischen Gegebenheiten beschäftigt.

1.0.1 Grundlagen

In der Informationsverarbeitung repräsentieren Signale, die von einer Quelle – dem Mund eines Sprechers, einer Nachrichtenübermittlung oder einer Tastatur ausgegeben werden, Symbole, die Information übertragen. Diese Information besteht in der Unsicherheit, Veränderungen eines Signals vorhersagen zu können. Der Informationsgehalt eines Zeichens x aus einer Ergebnismenge der möglichen Zeichen Ω, hängt von der Wahrscheinlichkeit $\rho(x) = Pr[X = x]$ ab, dass das informationstragende Signal zum Beobachtungszeitpunkt den zu diesem Zeichen zugeordneten Wert x annimmt. Je kleiner die Auftrittswahrscheinlichkeit $\rho(x) = Pr[X = x]$ eines Zeichens x ist, desto höher ist sein Informationsgehalt. Andersherum ist der Informationsgehalt eines Zeichens gering, wenn es oft vorkommt (1a).



(a) Informationsgehalt eines Zeichen mit Auftrittswahrscheinlichkeit $\rho(x) = Pr[X = x]$



(b) Der Logarithmus zur Basis 2

Da $Pr\colon \Omega \to [0, 1]$, und weil der Logarithmus zur Basis 2 im Intervall $[0, 1]$ negativ

und aufsteigend ist (1b), eignet sich die folgende Formel gut für die Definition des Informationsgehalts:

$$I(x) = \log_2 \frac{1}{Pr[X = x]} = -\log_2 Pr[X = x] = -\log_2(\rho(x))$$

In diesem Kontext ist die Entropie H ein Maß für den mittleren bzw. erwarteten Informationsgehalt jeder Nachricht bzw. jeder Zeichenkette, die von einer Quelle produziert wird. Formal bezeichnet man als Entropie den mittleren Informationsgehalt für eine diskrete Wahrscheinlichkeitsverteilung der Zufallsvariablen X:

$$H(X) = \sum_{x \in X} Pr[X = x] \cdot I(x) = \sum_{x \in X} -Pr[X = x] \cdot \log_2(Pr[X = x]) = \sum_{x \in X} -\rho(x) \cdot \log_2(\rho(x))$$

Die diskrete Wahrscheinlichkeitsverteilung der Zufallsvariablen X sei dabei die Abbildung $\rho\colon \Omega \to [0, 1]$, $\rho(x) = Pr[X = x]$ mit den folgenden von den Kolmogorov-Axiomen hergeleiteten Eigenschaften:

- Normiertheit: $\rho(\Omega) = 1$
- Die abzählbare Additivität: Für jede abzählbare Folge von paarweise disjunkten Mengen von Erreignissen $X_1, X_2, X_3, \ldots$ aus Ω gilt:

$$\rho(\bigcup_{X_i \in \Omega} X_i) = \sum_{X_i \in \Omega} \rho(X_i) = \sum_{X_i \in \Omega} \sum_{x_i \in X_i} \rho(x_i) = \sum_{X_i \in \Omega} \sum_{x_i \in X_i} Pr[X = x_i]$$

Man kann zeigen, dass für die Entropie die folgenden Grenzen gelten:

$$0 \leq H(X) \leq \log_2(|X|)$$

Wobei $|X|$ die Kardinalität (die Anzahl von den möglichen Zeichen) von X ist.
Die linke Ungleichung ist aufgrund der Eigenschaften von $\log_2$ im Intervall $[0, 1]$ trivial und $H(X) = 0$ tritt auf, wenn ein Zeichen a stets mit maximaler Wahrscheinlichkeit $\rho(a) = Pr[X = a] = 1$ auftritt (und die restlichen Zeichen nie auftreten), sodass die Entropie minimal ist, weil der Output der Quelle immer vorhersehbar ist:

$$H(X) = \sum_{x \in \Omega} -Pr[X = x] \cdot \log_2(Pr[X = x]) = -Pr[X = a] \cdot \log_2(Pr[X = a]) = -1 \cdot 0 = 0$$

Intuitiv ist die Entropie einer Quelle maximal, wenn die Auftrittwahrscheinlichkeit zwichen den Zeichen gleichverteilt ist, formal heißt dies:

$$\begin{aligned} \forall x \in \Omega : \rho(x) = Pr[X = x] = \frac{1}{|X|} \Leftrightarrow H(X) &= \sum_{x \in \Omega} -Pr[X = x] \cdot \log_2(Pr[X = x]) \\ &= |X| \cdot (-\frac{1}{|X|} \cdot \log_2(\frac{1}{|X|})) \\ &= \log_2(|X|) \end{aligned} \tag{1}$$

1.0.2 Bemerkenswerte Anwendungsfälle

Karl Küpfmüller bestimmte die Entropie der deutschen Sprache im Jahr 1954 auf 1,5 bit je Buchstabe – damit liegt die Redundanz der deutschen Sprache bei etwa 3,2 bit je Buchstabe. Anders ausgedrückt: etwa 68% der Zeichen eines deutschen Textes sind aus der Sicht des Informationsgehalts der Nachricht überflüssig. Bei der Wahl eines Passworts zum Beispiel kommt es insbesondere auf einen hohen Informationsgehalt an. Wird ein deutsches Wort als Passwort gewählt, liegt der Informationsgehalt nur bei ca. 1,5 bit je Buchstabe. Ein zehn Zeichen langes Passwort hat also einen Informationsgehalt von nur 15 bit. Ohne Sonderzeichen, Ziffern und eine zufällige Passwortwahl reduziert sich die Zahl möglicher Passworte damit auf nur etwa 33.000 – ein Kinderspiel für Passwortcracker.
Eine binäre Quelle zum Beispiel, welche auf nicht vorhersehbare Weise die Zeichen '0' oder '1' emittiert, hat die Entropie:

$$H(X) = \sum_{x \in \Omega} -Pr[X = x] \cdot \log_2(Pr[X = x]) = 2 \cdot (-\frac{1}{2} \cdot \log_2(\frac{1}{2})) = \log_2(2) = 1 \quad (2)$$

Allgemein erwartet man bei der Generierung einer höherer Anzahl an Zufallzahlen durch Pseudozufallszahlengeneratoren, wie rand aus der C-Standardbibliothek, eine (fast) maximale Entropie, da die Werte unvorsehbar sind. Dies ließ sich durch unsere Implementierung später bestätigen, als wir die Entropie von 10000 mit rand generierten Zahlen gemessen haben. Das Ergebnis ($\approx$ 13.0027 bits) war minimal kleiner als die maximale mögliche Entropie $\log_2(|X|) \approx 13.0773$.

2 Lösungsansatz

2.1 Logarithmus in der C-Standardbibliothek

Im Linux Programmer's Manual [1] wird beschrieben, dass die Basislogarithmusfunktion *(log())* den natürlichen Logarithmus einer Zahl x berechnet. Es gibt auch eine andere Funktion in dieser Bibliothek, die den Logarithmus zur Basis 2 berechnet. Auch die edge cases dieser Funktion werden angegeben.

Der Source-code dieser Funktion ist eher auf die Effizienz des Algorithmus als auf seine Lesbarkeit ausgelegt.

2.2 Erste Naive Implementierung

Zu Beginn der Arbeit haben wir uns einen sehr einfachen Algorithmus ausgedacht, um die Formel umzusetzen. Das größte Problem zu Beginn war die Implementierung einer Formel zur Berechnung des Logarithmus zur Basis 2. Als erstes haben wir uns einen Algorithmus ausgedacht, der einen Wert *pmin = 1* so lange mit 2 multipliziert, bis dieser Wert größer oder gleich der Zahl ist, für die wir den Logarithmus berechnen wollen (diese Multiplikation könnte auch durch Verschieben eines Bits nach links erfolgen). Mit dieser Formel kann der Logarithmus von Zahlen berechnet werden, deren Logarithmus zur Basis

2 eine ganze Zahl ergibt. Für alle anderen Zahlen funktioniert der Algorithmus jedoch nicht korrekt. Der Entropiefunktion werden jedoch nur Wahrscheinlichkeitswerte, also Werte zwischen 0 und 1 übergeben. Man benötigt eine präzise Implementierung für diesen Bereich.

2.3 Reihendarstellung der Logarithmusfunktion

Eine weitere Möglichkeit, den Logarithmus korrekt zu berechnen, ist die Reihendarstellung des natürlichen Logarithmus. Diese Methode funktioniert auch für Zahlen, deren Logarithmus keine ganze Zahl ist, und ist daher viel genauer als das naive Vorgehen. Um auf die mathematische Formel der Reihendarstellung zu kommen, muss die Taylorreihe berücksichtigt werden [2] , deren Formel wie folgt lautet:

$$P_{\mathrm{n}}(x) = P_{\mathrm{n}}(a) + \frac{P^{'}{}_{\mathrm{n}}(a)}{1!}(x-a) + \frac{P^{''}{}_{\mathrm{n}}(a)}{2!}(x-a)^2 + \ldots + \frac{P^{(\mathrm{n})}{}_{\mathrm{n}}(a)}{n!}(x-a)^{\mathrm{n}}$$

Mit dieser Formel kann die Funktion aus ihren Ableitungen ermittelt werden: Zunächst wird der Fall, in dem *f(x) = lnx* ist, verworfen, da der natürlichen Logarithmus für Zahlen gleich oder kleiner als 0 nicht definiert ist. Wir verwenden daher von Anfang an die Formel *f(x) = ln(x + 1)*. Berechnet man die ersten fünf Ableitungen dieser Funktion, so erhält man folgende Ergebnisse:

$$f^{'}(x) = \frac{1}{1+x}$$

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

$$f^{'''}(x) = \frac{2 \cdot 1}{(1+x)^3}$$

$$f^{''''}(x) = \frac{-3 \cdot 2 \cdot 1}{(1+x)^4}$$

$$f^{'''''}(x) = \frac{4 \cdot 3 \cdot 2 \cdot 1}{(1+x)^5}$$

Diese Ableitungen sind konstant und folgen einer Formel, die wie folgt lautet:

$$f^{\mathrm{n}}(x) = (-1)^{\mathrm{n}+1} \cdot \frac{(n-1)!}{(1+x)^{\mathrm{n}}}$$

Setzt man diese Ableitungen in die Taylor-Reihe ein mit *a=0*:

$$ln(x+1) = \frac{0!}{1!}x + \frac{-1!}{2!}x^2 + \frac{2!}{3!}x^3 + \frac{-3!}{4!}x^4 + \frac{4!}{5!}x^5 + \ldots$$

Substituiert man x mit *x-1*, erhält man die folgende Formel:

$$lnx = (x-1) - \frac{1}{2}(x-1)^2 + \frac{1}{3}(x-1)^3 - \frac{1}{4}(x-1)^4 \dots$$

Diese Formel funktioniert für Zahlen zwischen 0 und 2, da die Berechnung der Entropie Wahrscheinlichkeiten erfordert, d.h. Zahlen zwischen 0 und 1, ist die Reihendarstellung eine sehr gute Wahl für ihre Berechnung. Um den Logarithmus zur Basis 2 zu bestimmen, kann folgende Formel verwendet werden:

$$\frac{\ln x}{\ln 2} = \log_2 x$$

Die Reihe ist unendlich lang, und je länger sie ist, desto genauer wird der resultierende Logarithmus sein. Für diese Berechnungen ist es sinnvoll, einen Algorithmus zu entwickeln, der die Potenz einer Zahl errechnet. Zu Beginn haben wir diese Berechnung zu einer einfachen Berechnung mit einer Laufzeit von O(n) gemacht.

2.3.1 Optimierungen

Die Reihendarstellung, die für die korrekte Berechnung des Logarithmus mehrere Schleifen durchlaufen muss, braucht länger als das naive Vorgehen und die LUT, um das Ergebnis zu liefern. Um diese Zeit zu minimieren, kann die Funktion zur Berechnung des natürlichen Logarithmus verbessert werden, die der Formel der Reihendarstellung folgt. Um dies zu präzisieren, hat diese Funktion als Parameter n die Grenze, an der die Formel ankommt (die letzte Zahl, auf die x-1 erhöht wird). Anstatt die Berechnung für jede Schleife zu wiederholen, kann sie mit Hilfe von Intrinsics so optimiert werden, dass 4 Floats in einem xmm-Register behandelt werden.

Das Gleiche gilt für die Entropie. Wenn die Länge des Felds größer als 4 ist, kann die Summe der Wahrscheinlichkeiten (die am Ende gleich 1 sein muss) optimiert werden, ebenso wie die Multiplikation zwischen den Wahrscheinlichkeiten und dem Logarithmus zur Basis 2 davon.

2.4 Der Lookup Table Ansatz

In diesem Abschnitt versuchen wir, anstatt in linearer Laufzeit $O(n)$ (n sei die gewünschte Genauigkeit) wie bei der Reihendarstellung, $\log_2(x)$ in konstanter Laufzeit $O(1)$ und damit unabhängig von der gewünschten Genauigkeit zu berechnen. Weiterhin möchten wir den durch Absorption beim Inkrementieren in jeder Iteration der früheren Implementierung entstandenen Genauigkeitsverlust vermeiden.
Um diese Ziele zu erreichen, bietet der Lookup Table Ansatz die Möglichkeit, für häufig auftretende, identische Berechnungen ein Anteil der Komplexität im Speicher zu “lagern”, indem wir anhand einer endlichen Anzahl von vorberechneten Punkten (X, Y) eine Funktion f approximieren.
Ein Nachteil von diesem Ansatz ist aber der höhere Speicherverbrauch. Zusammen haben

wir uns für eine Tabellengröße von maximal 4 KiB entschieden, um besser von der räumlichen Cache-Lokalitätseigenschaften Gebrauch zu machen.
Die X-Werte werden Stützpunkte (Breakpoints) genannt. Wir approximieren den Wert der Funktion an einem Punkt x, indem wir zwischen zwei oder mehr Stützpunkten $x_1, x_2, x_3 \ldots$, die dem Punkt am nächsten liegen, interpolieren. Zusammenfassend besteht unsere Implementierung meistens aus zwei Tabellen mit je $2KiB$ ($2 * *9 = 512$ double precision floating point Werte): eine Pre-Lookup Table für die Breakpoints X, und die eigentliche Lookup Table für die dazugehörigen Funktionswerte Y.

2.4.1 Eine erste umsetzung: evenly spaced LUT

Eine erste simple Implementierung besteht aus einer LUT mit 512 gleichmäßig verteilten Funktionswerten, die durch die log2f Funktion aus math.h für $x \in [2^{-9}, 1-2^{-9}]$ berechnet wurden. Da die Stützpunkte einen gleichmäßigen Abstand von 2^{-9} haben, können wir die Pre-Lookup Tabelle löschen und für die Indexierung von der Lookup Tabelle die binäre Darstellung von x ausnutzen: $x = \sum_{i=1}^{9} 2^{-i}.(1+f), f \in [0,1[$ sei dabei die Abweichung vom ersten Breakpoint x_1. Wir multiplizieren zwar unser float x mit 2^{-9}, um genau wie bei einem left shift von 9 bits, die 9 erste Nachkommaziffern als Index zu verwenden. Damit bekommen wir unserem x nächstliegende Stützpunkte:

$$x_1 = 2^{-9} \cdot \lfloor 2^{-9} x \rfloor$$
$$x_2 = x_1 + 2^{-9}$$

Die dazugefhörigen Werte y_1 und y_2 werden weiterhin von der Lookup Tabelle gelesen:

$$y_1 = evenLUT[\lfloor 2^{-9} x \rfloor]$$
$$y_2 = evenLUT[\lfloor 2^{-9} x \rfloor + 1]$$

Für eine polynomielle Interpolation können wir somit weitere Punkte z.B. $(x3, y3)$ festlegen.

2.4.2 Zwischen LUT-Einträgen interpolieren

Weil der unmittelbare Nachbar in der LUT (nearest neightbour) ein zu ungenaues Ergebnis liefert, nutzen wir die von Isaac Newton begründete lineare Interpolation. Anhand von 2 durch LUT festgelegten Punkten $(x1, y1)$ und $(x2, x2)$ kann man den Wert von $y = log_2(x)$ mit der linear interpolierten Approximation $p(x)$ berechnen:

$$\frac{y - y_1}{x - x_1} \approx \frac{y_2 - y_1}{x_2 - x_1} \Leftrightarrow y \approx p(x) = y_1 + (x - x_1) \cdot \frac{(y_2 - y_1)}{x_2 - x_1}$$

Mit wenig zusätzlichem Rechenaufwand ist die polynomielle Interpolation mit einem weiteren Punkt $(x3, y3)$ anhand von der Lagrange Darstellung eine genauere Alternative:

$$p(x) = \sum_{i=0}^{n=3} y_i \cdot l_i(x) \text{ mit } l_i(x) = \frac{\prod_{k=0, k \neq i}^{n=3} (x - x_k)}{\prod_{k=0, k \neq i}^{n=3} (x_i - x_k)}$$

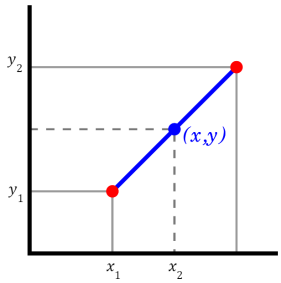


Abbildung 2: Veranschaulichung einer linearen Interpolante (blau) zwischen 2 Punkten.

Mit der Definition von den baryzentrischen Gewichten anstelle vom Nenner von lj:

$$w_i = \frac{1}{\prod_{k \neq i}(x_i - x_k)} \text{ mit } j = 0, ..., n$$

verringert man die Anzahl von Multiplikationen, mit der verbesserten Lagrange-Formel:

$$p(x) = l(x) \sum_{i=0}^{n=3} \frac{w_i}{x - x_i} y_i \text{ mit } l(x) = (x - x_0)(x - x_1)...(x - x_n)$$

Je mehr Stützstellen vorhanden sind, desto genauer ist die Interpolante $p(x)$ vom n-ten Grad, sogar mit geringen Rechenkosten.

2.4.3 Ein heuristischer Ansatz: unevenly spaced LUT

Durch den Vergleich mit den von log2f (aus math.h) gelieferten Werten haben wir erfahren, dass unsere Approximation stark davon abweicht, wenn die Abstände zwischen die Funktionswerten y_i von den Stützpunkten steigen. Das heißt, je höher die Ableitung $(log_2)'$ ist, desto ungenauer ist unsere Interpolante $p(x)$. Nun steigen bei log_2 die Abstände und die Ableitung für $x \to 0$ drastisch! Wir lösen dieses Problem, indem wir die Konzentration von den Stützpunkten mit $x \to 0$ steigern, in einer sogenannten unevenly spaced LUT (ungleichmäßig verteilte LUT). Wir stellen sicher, dass unsere ausgewählte Stützpunkte für eine maximale absolute Abweichung von $2^{-16} \approx 0.0000152587890625$ und einen maximalen Spreicherverbrauch von 2×4096 Bytes (eine Pre-Lookup Tabelle und eine Lookup Tabelle mit je 512 Single-Precision Floating Point Einträgen) sorgen, indem wir *Lookup Table Optimizer* von MATLAB für 1,5 Stunden laufen lassen. Dieser Algorithmus führt eine Brute-Force erschöpfende Suche, indem er sehr viele potenziellen Stützpunktverteilungen in $[2^{-16}, 1 - 2^{-16}]$ durchprobiert, bis eine gefunden ist, die die oben genannten Anforderungen erfüllt. *Lookup Table Optimizer* berührt daher auf schiere Rechenleistung und das Ausprobieren aller Möglichkeiten statt auf fortgeschrittene algorithmische Techniken.
Der folgende Graph repräsentiert die absolute Abweichung der linearen Interpolation, basierend auf die generierten Stützpunkten:

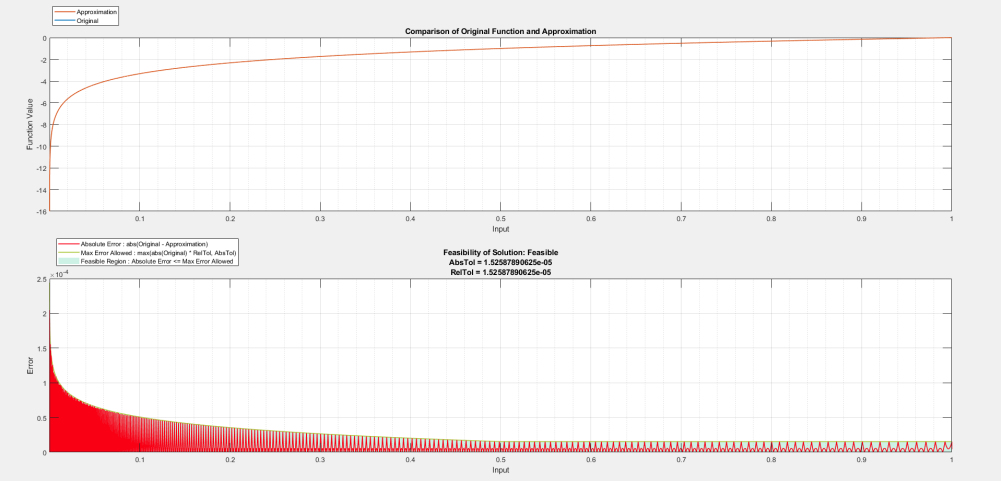


Abbildung 3: Von MATLAB berechnete absolute Abweichung (rot) zwischen Originalfunktion $log2$ und linearer Interpolation anhand von den generierten Stützpunkten.

Man merkt, dass die absolute Abweichung trotz höherer Konzentration von Stützpunkten für $x \to 0$ steigt, dadurch wurde die aber stark gedämpft!
Die ungleichmäßig verteilte Stützpunkte erfordern eine binäre Suche, um zu bestimmen, in welchem Intervall $[x1, x2]$ x liegt. Der Preis für einen signifikanten Genauigkeitssprung beträgt $log_2(512) = 9$ Iterationen innerhalb der binären Suche, mit je 1 Array-Zugriff auf die Pre-Lookup Tabelle.

2.4.4 Noch eine heuristische Anpassung

Für alle Implementierungen von log_2 addieren wir systematisch die Konstante c, die durchschnittliche relative Abweichung von math.h log2f über 1 Million Berechnungen für x in "]0,1]". Durch eine einzelne Addition lässt sich die Genauigkeit günstig verbessern. Dadurch lässt sich unsere Philosophie durchsehen: Wir bevorzugen das Genauigkeit/Performanz Verhalten vor mathematischem Rigorismus.

2.4.5 Weitere Herausforderungen

- Die Entropie ist eine Summe. Durch die vom Rounding Modus erzwungene Rundung kann das Ergebnis stark von der Realität abweichen, weil kleine Zahlen von größeren absorbiert werden. Dafür haben wir unsere Summanden mit dem Quicksort Algorithmus qsort von der C-Standardbibliothek sortiert (Komplexität $O(nlog(n))$, damit kleine Werte zuerst akkumuliert werden.

- Die uneven LUT Implementierung lässt sich sehr schlecht mit SSE4 optimieren. Zum Beispiel greift man durch die binäre Suche auf unterschiedlichen Stellen der Pre-Lookup Table zu. Das lässt sich nur in AVX parallelisieren. Darüber hinaus werden einzelne If-Statements durch mehrere Operationen auf Bitmasken ersetzt.

3 Performanzanalyse

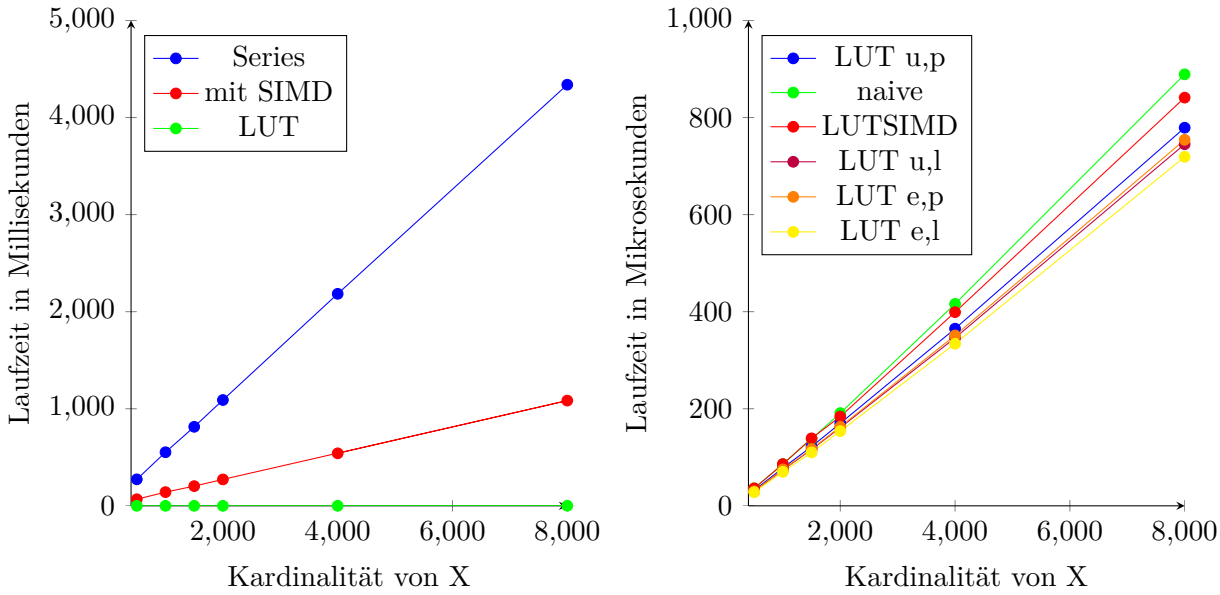


Abbildung 4: e - even, u - uneven, l - linear, p - polynomial

Gemessen wurde auf einem System mit einem Intel i5-8250U, 3.40GHz, 8 GB Arbeitsspeicher, Fedora Linux 36 (Workstation Edition) x86_64, Linux Kernel 5.18.13-200.fc36.x86_64. Kompiliert wurde mit (GCC) 12.1.1 20220507 (Red Hat 12.1.1-1) mit der Option -O3. Die Berechnungen wurden mit Eingabegrößen von 500 bis 8000 jeweils 5 mal durchgeführt, wobei die Berechnung intern ein bis 20000 mal wiederholt wurde. Der Durchschnitt jeder Messung wurde in die Diagramme eingetragen.

Für die einzelnen Versionen wurde die Berechnungsdauer in Abhängigkeit von der Eingabegröße gemessen. Es ist zu erkennen, dass bei jeder Implementierung die Laufzeit linear abhängig von der Eingabegröße ist.

Obwohl die Laufzeiten eine Obergrenze von O(n) haben, ist ein wesentlicher Unterschied zwischen den LUT-Versionen und den Series-Versionen zu erkennen. Je nach Eingabegröße und Version ist die LUT-Implementierung ca. 5000 bis 9000 mal schneller. Die Genauigkeit der Series-Implementierung ist von der Anzahl an Iterationen abhängig.

Unsere Implementierung arbeitet mit 1000 Iterationen. Jeder Summand benötigt also 1000 Schleifendurchläufe und die dazugehörigen Berechnungen. Die LUT-Implementierung benötigt hingegen nur 9 Iterationen für den Lookup und wenige (von der Optimierungstufe abhängige) weitere Operationen für einen Summanden.

Es ist deutlich zu erkennen, dass die Implementierung der Reihendarstellung im Gegensatz zur LUT Implementierung mithilfe von SIMD erfolgreich war. Die Laufzeit wurde um ungefähr 75% reduziert. Ein Grund warum die SIMD-Version der LUT Implementierung nicht schneller ist, sind Arrayzugriffe. Aus der LUT müssen Werte von

bestimmten, aber voneinander unabhängigen Positionen gelesen werden. Dies lässt sich nicht mit SSE parallelisieren.

Um den passenden Wert in der LUT zu finden wird ein binäres Suchverfahren genutzt. Damit dieses parallelisiert werden kann müssen if-Statements parallelisiert werden. Das geschieht mit Bitmasken. Um die Bitmasken korrekt zu setzen werden aber weitere Instruktionen benötigt. Dadurch wird kein Performanzvorteil erzielt.

4 Genauigkeit

Um die Genauigkeit zu testen haben wir die absoluten Abweichung unserer Implementierungen von der gcc Funktion "log2"gemessen. Es ist nicht notwendig die gesamte Entropiefunktion zu betrachten, weil die Berechnung des Logarithmus zur Basis 2 die einzige Stelle sein kann, an der Abweichungen auftreten können. Zum Vergleich haben wir eine Wahrscheinlichkeitsverteilung mit 100000 Werten erstellt, welche alle die gleiche Wahrscheinlichkeit (p = 0.00001) besitzen.

Zum Vergleich haben wir die Implementationen mit den Wahrscheinlichkeiten von 0.00001 bis 1 getestet, wobei die Schrittgröße 0.00001 betrug.

Die größte Abweichung wurde von der naiven Implementierung verursacht. Im Durchschnitt war das Ergebnis um ca. 2.41277 Einheiten falsch.Eine relativ große Abweichung war zu erwarten, da exakte Werte nur für Eingaben einer Zweierpotenz ausgegeben werden. Für die meisten Anwendungsfälle ist diese Genauigkeit jedoch nicht ausreichend Die restlichen Implementierungen sind wesentlich präziser. Die zweitschlechteste Genauigkeit hat die even-linear-LUT-Implementation. Im Durchschnitt lag die Abweichung bei 0.00720. Somit ist sie 335 mal genauer als die naive Implementierung. Die even-polynomial-Implementation hatte eine bessere Genauigkeit mit einer durchschnittlichen Abweichung von 0.00120. Beide Implementationen der Reihendarstellung hatten eine Abweichung von ca. 0.00062. Allgemein kann man sagen, dass die Ergebnisse der LUT-Implementierungen eine hohe Genauigkeit aufweisen. Beim beschriebenen Testfall gab es keine Abweichung der uneven-linear und der uneven-polynomial Implementationen. Bei exakteren Abweichungen lassen sich jedoch auch dort Fehler erkennen. Bei Verwendung der eben beschriebenen Wahrscheinlichkeitsverteilung mit 10000000 Werten (p=0.0000001) kann man eine Abweichung von 0.0000058 (uneven-linear) bzw. von 0.0000035 (uneven-polynomial) messen.

Aus den Ergebnissen kann man schlussfolgern, dass die Genauigkeit steigt, je größer die Werte der Wahrscheinlichkeitsverteilung werden. Im Umkehrschluss bedeutet das, dass die Genauigkeit sinkt, je kleiner die Werte werden. Die uneven-Implementationen sind so gestalten, dass sie das Problem verringen, da diese Lookuptabellen mehr Werte für kleinere Wahrscheinlichkeiten besitzen.

In der Reihendarstellung als auch in der LUT-Implementation ist es recht einfach die Genauigkeit zu erhöhen. Die Reihendarstellung benötigt mehr Iterationen, die LUT-Version mehr Einträge. Dadurch ergeben sich jedoch andere Probleme, wie eine erhöhte Laufzeit oder ein erhöhter Speicherverbrauch.

5 Zusammenfassung und Ausblick

Zusammenfassend kann man sagen, dass der Ansatz, eine Entropiefunktion mithilfe einer LUT zu implementieren, der bessere war. Die LUT-Implementierung waren schneller und genauer. Ihr einziger Nachteil im Vergleich zur Implementierung einer Reihendarstellung war der Speicherverbrauch. Da aber nur wenige Kilobyte verwendet wurden ist der Speicherverbrauch auf modernen Rechnern verschwindend gering. Außerdem ist es wesentlich leichter die LUT-Versionen durch größere Tabellen zu verbessern, als die Reihendarstellung weiter zu optimieren. Mithilfe von komplexeren Instruktionen und Intrinsics (welche sich außerhalb des Umfangs dieses Projekts befinden) sollten sich die Laufzeiten der LUT-Versionen weiter verringern lassen. Die Berechnung des mittleren Informationsgehaltes einer Quelle wird in vielen wichtigen Bereichen der Informatik angewendet. Deshalb ist es wichtig, die Entropie präzise und schnell berechnen zu können, zum Beispiel um die Stärke eines Hashingalgorithmus zu bestimmen. Falsche Entropieergebnisse können hier zu fatalen Fehlern führen.

Literatur

[1] Michael Kerrisk. Linux programmers's manual, 2021.

[2] Eric Weisstein. Taylor series, 1999.