Rechnerarithmetik

Vorlesung im Sommersemester 2008

Eberhard Zehendner

FSU Jena

Thema: Logarithmische Zahlensysteme

Logarithmische Zahlensysteme: Einsatzbereich

Logarithmische Zahlensysteme (logarithmic number systems LNS) dienen der Vereinfachung von

- Multiplikation (wird zu gewöhnlicher Addition)
- Division (wird zu gewöhnlicher Subtraktion)
- Potenzierung (wird zu gewöhnlicher Multiplikation)
- Radizieren (wird zu gewöhnlicher Division)

Dagegen sind Addition und Subtraktion in logarithmischer Darstellung komplizierter als in gewöhnlicher Darstellung.

Logarithmische Zahlensysteme sind insbesondere gut geeignet für Signalverarbeitung mit geringen Genauigkeitsanforderungen und beabsichtigtem reduziertem Energieaufwand

Typische Verteilungen von Filter-Koeffizienten passen empirisch zu logarithmischer Darstellung

Laufendes Projekt: European Logarithmic Microprocessor (ESPRIT-Projekt, Beginn 1999)

European Logarithmic Microprocessor

- Coleman, J.N.; Softley, C.I.; Kadlec, J.; Matousek, R.; Licko, M.; Pohl, Z.; Hermanek, A.:
  The European Logarithmic Microprocessor - a QR RLS application.
  Conference Record of the Thirty-Fifth Asilomar Conference on Signals, Systems and Computers, Volume 1, 2001 pp. 155–159.

  Summary: In contrast to all other microprocessors, which use floating-point for their real arithmetic, the European Logarithmic Microprocessor is the world's first device to use the logarithmic number system for this purpose. Simulation work has already suggested that this can deliver approximately twofold improvements in speed and accuracy. This paper describes the ELM device, and illustrates its operation using an example from a class of RLS algorithms.
- Coleman, J.N.; Softley, C.I.; Kadlec, J.; Matousek, R.; Tichy, M.; Pohl, Z.; Hermanek, A.; Benschop, N.F.:
  The European Logarithmic Microprocesor.
  IEEE Transactions on Computers, April 2008 (Vol. 57, No. 4) pp. 532–546.

  Abstract: In 2000 we described a proposal for a logarithmic arithmetic unit, which we suggested would offer a faster, more accurate alternative to floating-point procedures. Would it in fact do so, and could it feasibly be integrated into a microprocessor so that the intended benefits might be realised? Herein we describe the European Logarithmic Microprocessor, a device designed around that unit, and compare its performance with that of a commercial superscalar pipelined floating-point processor. We conclude that the experiment has been successful; that for 32-bit work logarithmic arithmetic may now be the technique of choice.

Logarithmische Zahlensysteme: Konversion

Die Konversion zwischen logarithmischen Zahlensystemen und Standarddarstellungen erfordert die Berechnung von Logarithmen und Antilogarithmen.

Wegen der dabei auftretenden Approximationsfehler sind die Operanden bzw. Ergebnisse in logarithmischer Darstellung ungenau.

Typische Anwendungsfelder (z. B. digitale Filter) erfordern wenige Konversionen, aber viele Multiplikationen oder Divisionen.

Von der Darstellung alleine her gesehen, sind logarithmische Zahlensysteme sogar etwas genauer als Gleitkommasysteme annähernd gleichen Zahlenbereichs.

Vorzeichen-Logarithmus-Darstellung (SL)

Die *Vorzeichen-Logarithmus-Darstellung* (sign logarithm SL) einer zu codierenden reellen Zahl X besteht aus dem Vorzeichenbit S_X und dem Logarithmus L_X des Betrags von X,
$X = (-1)^{S_X} \times R^{L_X}$.

Falls $|X| < 1$ zugelassen ist, muss L_X auch negative Werte annehmen können.

Da $\log 0$ nicht definiert ist, muss außerdem die Null eine spezielle Darstellung besitzen.

Meist wird $R = 2$ gewählt und L_X in einer binären Festkommadarstellung angegeben:

$$S_X\, L_X = S_X\, x_{k-1}\, x_{k-2} \ldots x_1\, x_0 . x_{-1}\, x_{-2} \ldots x_{-n}$$

Der Nachkommateil besitzt hier n Bits, der ganzzahlige Anteil k Bits (einschließlich eines eventuellen Vorzeichenbits des Logarithmus).

Beispiel: SL mit $k = 4$, $n = 3$

Mit L_X in 2-Komplement-Darstellung ohne Bias gilt z. B.

$00111.111 = +2^{(8-\frac{1}{8})}$	$\approx +234.753_{10}$	(größte positive Zahl)
$00001.010 = +2^{(1+\frac{1}{4})}$	$\approx +2.37841_{10}$	
$01110.100 = +2^{-(1+\frac{1}{2})}$	$\approx +0.35355_{10}$	
$01000.000 = +2^{-8}$	$\approx +0.003906_{10}$	(kleinste positive Zahl)
spezielle Darstellung		(Null)
$11000.000 = -2^{-8}$	$\approx -0.003906_{10}$	(betragskleinste negative Zahl)
$11110.100 = -2^{-(1+\frac{1}{2})}$	$\approx -0.35355_{10}$	
$10001.010 = -2^{(1+\frac{1}{4})}$	$\approx -2.37841_{10}$	
$10111.111 = -2^{(8-\frac{1}{8})}$	$\approx -234.753_{10}$	(betragsgrößte negative Zahl)

Vereinfachung arithmetischer Operationen

Operationen ohne Rundungsfehler:

Multiplikation	$(S_X, L_X) \times (S_Y, L_Y) = (S_X \oplus S_Y, L_X + L_Y)$	(Festkomma-Addition)
Division	$(S_X, L_X)/(S_Y, L_Y) = (S_X \oplus S_Y, L_X - L_Y)$	(Festkomma-Subtraktion)
Kehrwert	$1/(S_X, L_X) = (S_X, -L_X)$	(Komplement)
Quadrat	$(S_X, L_X)^2 = (0, 2 \times L_X)$	(Linksverschiebung)
ganze Potenz	$(S_X, L_X)^Z = (S_X \wedge (Z \bmod 2), Z \times L_X)$	(Festkomma-Multiplikation)

Operationen, bei denen ein Rundungsfehler auftreten kann:

Wurzel	$\sqrt{(0, L_X)} = (0, L_X/2)$	(Rechtsverschiebung)
Potenz	$(0, L_X)^Y = (0, Y \times L_X)$	(Festkomma-Multiplikation)

Überlauf und Unterlauf können in allen Fällen leicht erkannt werden.

Addition und Subtraktion von SL-Zahlen

Für Addition und Subtraktion in logarithmischen Zahlensystemen existieren verschiedene alternative Ansätze:

1. Wertetabelle der Größe $l \times 2^{2 \times l}$ bit (mit $l = k + n$), nicht praktikabel für übliche Werte von l.
2. Die Operanden werden delogarithmiert und mit gewöhnlicher Addition behandelt, das Ergebnis wird logarithmiert (Rückgriff auf Wertetabellen der Größe $l \times 2^l$ bit, die für die Konvertierung in bzw. aus Standarddarstellungen ohnehin gebraucht werden).
3. Direkte Berechnung einer approximativen Summe oder Differenz (wegen kleinerer Wertetabellen der bevorzugte Ansatz).

Approximative Summen und Differenzen

Seien o. B. d. A. $X, Y > 0$.

$X > Y$:	$Z = X + Y = X \times (1 + \frac{Y}{X})$,		$Z = X - Y = X \times (1 - \frac{Y}{X})$,
		also	
	$L_Z = \log_2 X + \log_2(1 + \frac{Y}{X})$		$L_Z = \log_2 X + \log_2(1 - \frac{Y}{X})$
	$= L_X + \Phi^+(L_X - L_Y)$		$= L_X + \Phi^-(L_X - L_Y)$
		mit	
	$\Phi^+(h) = \log_2(1 + 2^{-h})$, $h > 0$		$\Phi^-(h) = \log_2(1 - 2^{-h})$, $h > 0$
$X < Y$:	$L_Z = L_Y + \Phi^+(L_Y - L_X)$		$L_Z = L_Y + \Phi^-(L_Y - L_X)$
$X = Y$:	$L_Z = L_X + 1$		$Z = 0$

Implementierung von Φ^+ und Φ^-

Φ^+ und Φ^- können durch Wertetabellen der Größe $l \times 2^l$ bit implementiert werden.
Um eine möglichst hohe Genauigkeit der Addition/Subtraktion zu garantieren, werden die exakten Werte von $\log_2(1 \pm 2^{-h})$ zum Eintrag in die Wertetabelle Round-to-nearest gerundet.

Wegen $0 < \Phi^+(h) < 1$ braucht bei der Addition nur ein Nachkommateil erzeugt werden.
Statt einer $(l \times 2^l)$-bit Wertetabelle genügt also eine $(n \times 2^l)$-bit Wertetabelle.

Für große h ist $\log_2(1 \pm 2^{-h}) \approx \pm 2^{-h}$, und damit fast Null; wegen der beschränkten Genauigkeit werden diese Werte durch die Rundung zu Null und es macht keinen Sinn, sie explizit zu speichern.

Die Wertetabellen können auch in eine Reihe kleinerer Tabellen zerlegt werden, in denen jeweils ein ähnlicher Effekt ausgenutzt werden kann.

Bei Kombination von Wertetabellen und Interpolation müssen weniger Einträge in der Wertetabelle gespeichert werden; hierzu gibt es ziemlich raffinierte, schnelle Verfahren.

Konversionen

Festkommaformat $\leftrightarrow$ logarithmische Darstellung:

$$v \times m = v \times R^L$$

Gleitkommaformat $\leftrightarrow$ logarithmische Darstellung:

$$v \times m \times R^e = v \times R^{L+e}$$

Es wird also im Prinzip nur eine Logarithmentafel für $1 \leq m < R$ und eine Antilogarithmentafel für $0 \leq L < 1$ benötigt.

Berechnung der Logarithmen

Sei $m = m_u m_{u-1} \ldots m_0 . m_{-1} m_{-2} \ldots m_{-w}$ und $t = \max\{i : m_i = 1\}$.

Es gilt $m = 2^t + \sum_{i=-w}^{t-1} 2^i \times m_i = 2^t \times \left(1 + \sum_{i=-w}^{t-1} 2^{i-t} \times m_i\right) = 2^t \times (1+h)$ mit $0 \leq h < 1$.

Also $\log_2 m = t + \log_2(1+h)$,
wobei t der ganzzahlige Anteil des Logarithmus ist, $\log_2(1+h)$ der Nachkommateil.

Meist wird $\log_2(1+h) \approx h$ ausgenutzt.

Eine Verbesserung ergibt sich durch geschickte Unterteilung des Intervalls $[0, 1)$ für h.

Die Implementierung erfolgt mittels eines Zählers und eines Schieberegisters.

Verallgemeinerung

Index Calculus Double-Base Number System (IDBNS), $y = v \times 2^a \times 3^b, a, b \in \mathbb{Z}$

Eigenschaft: $\forall \varepsilon > 0, x \geq 0\ \exists a, b \in \mathbb{Z} : |x - 2^a \times 3^b| < \varepsilon$

Andere Basen, mehr Ziffern (n digit two-dimensional logarithmic representation):

$$y = \sum_{i=1}^{n} v_i \times 2^{a_i} \times p^{b_i}, \quad p \text{ ungerade}$$

Beispiel: Darstellung mit Fehler $\leq 0{,}5ulp$

Standard-Darstellung: $x \in \mathrm{Int}_2(10)$ (10-Bit-Darstellung)

SL-Repräsentation erfordert 12 Bit Logarithmus und ein Vorzeichenbit

Repräsentation durch zweistelliges 2-D LNS ($n = 2, p = 47$): $a_i \in \mathrm{Int}_2(6)$, $b_i \in \mathrm{Int}_2(3)$
$334 \approx 2^9 \times 47^{-1} + 2^{25} \times 47^{-3} \approx 334{,}082429$

Ziel der Darstellung mit mehreren Basen: Verwendung kleinerer Tabellen

Arithmetik mit variierender relativer Genauigkeit

Zweck variierender relativer Genauigkeit:

- Verbesserung der *durchschnittlichen* relativen Genauigkeit.
- Vergrößerung des Zahlenbereichs zur Vermeidung von Überlauf oder Unterlauf.

Die durchschnittliche relative Genauigkeit lässt sich durch *Tapered-Floating-Point-Systeme* verbessern; als Nebeneffekt ergibt sich zusätzlich eine gewisse Vergrößerung des Zahlenbereichs.

Eine entscheidende Vergrößerung des Zahlenbereichs wird erreicht durch verschiedene Methoden des *Leveling*.

Tapered-Floating-Point-Systeme

Ansatz von Morris (1971), aufgegriffen von Iri und Matsui (1981).

Statt eines Gleitkommaformats mit Signifikanten- und Exponentenfeldern fester Länge besteht ein *Tapered-Floating-Point-Format* aus Signifikanten- und Exponentenfeldern variierender Länge.

Hinzu kommt ein *Pointerfeld* fester Länge, das die Anzahl der Ziffern im Exponentenfeld angibt.

Die Anzahl der Ziffern im Signifikantenfeld (einschließlich des Vorzeichens der Zahl) ergibt sich als Differenz der festen Gesamtlänge des Formats abzüglich der Längen des Exponenten- und Pointerfelds.

Tapered-Floating-Point-Systeme: Implementierung

Die Anzahl der Ziffern im Exponentenfeld sollte für jeden konkreten Exponenten minimal gewählt werden, um eine möglichst hohe Genauigkeit zu ermöglichen.

Liegt der Signifikant bzw. Exponent in Binärcodierung vor, so gilt:

- Die führende Ziffer des Betrags des Signifikanten bzw. Exponenten braucht nicht gespeichert zu werden, da sie stets den Wert 1 trägt (Hidden-Bit).
- Ein Signifikantenfeld bzw. Exponentenfeld mit nur einem Bit kann die Signifikanten bzw. Exponenten ± 1 darstellen (besteht nur aus dem Vorzeichen).
- Ein Signifikantenfeld der Länge Null zeigt die Zahl Null an.
- Ein Exponentenfeld der Länge Null codiert den Exponenten 0.

Eigenschaften von Tapered-Floating-Point-Formaten

Vorteile:

- Im gewöhnlichen Zahlenbereich von Gleitkommasystemen können Zahlen, die weder sehr groß noch sehr klein sind, mit erhöhter Genauigkeit gespeichert werden.
- Der Zahlenbereich kann durch geringere Genauigkeit für die hinzukommenden Zahlen wesentlich erweitert werden.

Nachteile:

- Die Implementierung ist aufwändiger als für gewöhnliche Gleitkommasysteme.
- Bei gleichem Speicheraufwand ist für manche Zahlen im gewöhnlichen Zahlenbereich entsprechender Gleitkommasysteme die Genauigkeit wegen des Pointerfelds geringer.

Praktische Implementierungen zeigen, dass bei gleichem Speicheraufwand durch Tapered-Floating-Point-Systeme in der Regel eine höhere durchschnittliche Genauigkeit erreicht wird.

Genauigkeit von Tapered-Floating-Point-Systemen

Betragsgroße und betragskleine Zahlen sind wegen des betragsgroßen Exponenten ungenauer als solche moderater Magnitude.

Beispiel

Länge 64 Bit, davon 6 Bit für das Pointerfeld: Zahlenbereich $\approx \pm 10^{\pm 4 \times 10^{16}}$

$\lvert x \rvert \approx$	relativer Fehler beschränkt durch	
1	2^{-57}	(höchste relative Genauigkeit)
$2^{\pm 16}$	2^{-52}	
$10^{\pm 8 \times 10^{7}}$	2^{-28}	
$10^{\pm 4 \times 10^{16}}$	100%	(soweit x im zulässigen Bereich)

Das Format *double* im Standard IEEE-754 besitzt einen Zahlenbereich von $\approx \pm 10^{\pm 308}$ bei gleichmäßigem maximalen relativen Fehler $\approx 2^{-52}$.

Wahrscheinlichkeit von Überlauf und Unterlauf

Bei Tapered-Floating-Point-Systemen sind Überlauf und Unterlauf wegen des erweiterten Exponentenbereichs recht unwahrscheinlich, können aber – insbesondere bei wiederholter Exponentiation – dennoch auftreten.

Mit *Leveling* erreicht man Größenordnungen, die in der Praxis nicht mehr vorkommen.
Damit ist zwar nicht ausgeschlossen, dass Überlauf oder Unterlauf auftritt, die Wahrscheinlichkeit ist aber so gut wie Null.

Zu beachten ist, dass die Operationen $+, -, \times, /$ für höhere Levels hochgradig ungenau ausfallen.

Leveling nach Iri und Matsui

Ist der Exponent einer darzustellenden Zahl betragsgrößer als der maximale Exponent eines Tapered-Floating-Point-Formates (Level 0), so wird der Betrag dieses Exponenten selbst in einem (etwas kleineren) Tapered-Floating-Point-Format (Level 1) repräsentiert.

Ein spezieller Wert des Pointerfelds zeigt an, wann der Exponent im Level 1 Format vorliegt.

Beispiel

Level 0: Länge 64 Bit, davon 6 Bit Pointerfeld; Zahlenbereich $\approx \pm 10^{\pm 4 \times 10^{16}}$

Level 1: Länge 56 Bit, davon 6 Bit Pointerfeld; Zahlenbereich $\approx \pm 10^{\pm 10^{10^{15}}}$

Dieses Prinzip kann über weitere Stufen (Level 2, Level 3, ...) fortgesetzt werden, bis die immer kleiner werdenden Exponentenfelder dem eine Grenze setzen.

Mit fortschreitendem Level wird die Darstellung aber immer ungenauer.

Level-Index-Arithmetik

Level-Index (LI)

Clenshaw und Olver (1984)

Zahlenformat: Vorzeichen $v \in \{0,1\}$ der Zahl, vorzeichenlose Festkommazahl f zur Codierung des Betrags.

Der ganzzahlige Anteil von f heißt *Level*, der Nachkommateil von f heißt *Index*.

$$x = (-1)^v \times \Phi(f)$$

$$\Phi(f) = \begin{cases} f & \text{falls } 0 \leq f \leq 1 \\ e^{\Phi(f-1)} & \text{sonst} \end{cases}$$

Symmetric Level-Index (SLI)

Clenshaw/Olver/Turner (1987)

Zahlenformat: Vorzeichen $v \in \{0,1\}$ der Zahl, Vorzeichen $u \in \{0,1\}$ des Exponenten, vorzeichenlose Festkommazahl f zur Codierung des Betrags.

$$x = (-1)^v \times \Psi(f)^{(-1)^u}$$

$$\Psi(f) = \begin{cases} e^f & \text{falls } 0 \leq f \leq 1 \\ e^{\Psi(f-1)} & \text{sonst} \end{cases}$$

Symmetric-Level-Index-Arithmetik (Beispiel)

Länge 64 Bit, davon 3 Bit Levelfeld: Zahlenbereich $\approx \pm 10^{\pm 10^{10^{10^{10}}}}$

$|x| \approx 1$: relativer Fehler $\approx 2^{-59}$ (höchste Genauigkeit)

$|x| \approx 1.73 \times 10^{13}$: relativer Fehler $\approx 2^{-52}$

$|x| \approx 10^{5 \times 10^6}$: relativer Fehler $\approx 2^{-30}$

$|x| \approx 10^{10^{15}}$: relativer Fehler ≈ 1

Für noch größere $|x|$ kann der relative Fehler unvorstellbar groß werden.

Dynamische Vereinbarung von Zahlformaten

Cohen/Hamacher/Hull (1981):
Clean Arithmetic with Decimal base And Controlled precision (CADAC)

Hull et al. (1985): NUMERICAL TURING

- Zahlenbasis 10
- Nachkommateile mit p Stellen
- Ganzzahlige Exponenten im Bereich $[-10 \times p, +10 \times p]$

An jeder Stelle des Programms kann eine *precision* p in Form eines dynamisch ausgewerteten Ausdrucks erklärt werden.

Eine *precision*-Definition legt für den Rest des Gültigkeitsbereiches der die Definition enthaltenden Kontrollstruktur bzw. bis zum nächsten Auftreten einer *precision*-Definition innerhalb derselben Kontrollstruktur die Genauigkeit p aller deklarierten Variablen und aller Gleitkommaoperationen fest.

Jede Variable kann alternativ auch direkt mit einem Ausdruck versehen werden, der ihre Genauigkeit p angibt.

Algorithmen in Numerical Turing

- Hohe Genauigkeit für den gesamten Algorithmus ist aufwändig.
- Hohe Genauigkeit wird meist nicht durchgängig benötigt (nur an kritischen Stellen).
- Der Grad an nötiger Genauigkeit hängt oft von den Daten ab, ist also zur Übersetzungszeit schwer abschätzbar.

Beispiel: Berechnung der Quadratwurzel nach dem Newton-Verfahren:

```
var p:= 3
const maxp := currentprecision+2
loop
   p := min(2*p-2,maxp)
            % p = 4, 6, 10, ..., maxp
   precision p
   approx := .5*(approx+f/approx)
   exit when p = maxp
end loop
```