修士論文概要書

2007 年 2 月提出　　学籍番号 3605U055-6

専攻名（専門分野）	情報・ネットワーク高度計算機構	氏名	栗原　崇	指導教員	大石　進一　印
研究指導名	情報数理工学				
研究題目	AccSum を用いた多倍長計算環境の実装と高速化				

概要

本研究では、実浮動小数点数ベクトルの総和の faithful rounding を返す AccSum アルゴリズムを用いて、多倍長計算環境を設計し、また高速化を検討した。具体的には、AccSum 内部の基本アルゴリズムである ExtractScalar の実行回数を、演算対象となる多倍長数の構成に依存せず、恒に加減算では $\frac{1}{2}(n(n-1)+m(m-1))$ 回、乗算では $mn(m+n-1)$ 回削減可能であることを導いた。また、AccSum の加算による桁上がりに着目し、多倍長数の性質を用いれば、桁上がりの上限を見積もることが可能となり、これからループの実行回数を削減可能であることも導いた。そして、数値実験を試みた結果、これら高速化手法が高倍長では有用であることが実証できた。

1　AccSum

AccSum は、与えられた実浮動小数点数ベクトル $p=(p_1,p_2,\cdots,p_n)$ の総和の faithful rounding を返すアルゴリズムである。faithful とは、真の解に対して最近の隣り合う上への丸め、下への丸め演算結果となる 2 つの浮動小数点数のどちらかになる性質のことである。この AccSum は、ExtractVector によって絶対値の大きな数値と和を行うことで意図的に”情報落ち”を発生させながら目的の総和を計上していくところに特徴がある。

アルゴリズム 1.1 浮動小数点数の上位部分抽出

$$
\begin{aligned}
&\text{function}\,[q,p'] = \mathbf{ExtractScalar}(\sigma,p)\\
&\quad q = \mathrm{fl}((\sigma+p)-\sigma)\\
&\quad p' = \mathrm{fl}(p-q)
\end{aligned}
$$

アルゴリズム 1.2 浮動小数点数ベクトルの上位部分抽出

$$
\begin{aligned}
&\text{function}\,[\tau,p'] = \mathbf{ExtractVector}(\sigma,p)\\
&\quad \tau = 0\\
&\quad \text{for } i = 1:n\\
&\qquad [q_i,p'_i] = \mathbf{ExtractScalar}(\sigma,p_i)\\
&\qquad \tau = \mathrm{fl}(\tau+q_i)\\
&\quad \text{end for}
\end{aligned}
$$

このアルゴリズム適用後、次の定理が成立することから無誤差変換となり、高精度な総和を算出可能となる。

定理 1.1 $\sigma = 2^k \in \mathbb{F}$, $k \in \mathbb{Z}$, $n+2 \le 2^M$, $\forall i\ |p_i| \le 2^{-M}\sigma$, $0 \le M \in \mathbb{N}$ が成立するとき、

$$
\sum_{i=1}^{n} p_i = \tau + \sum_{i=1}^{n} p'_i\ ,\ \max|p'_i| \le \mathbf{eps}\sigma\ ,\ |\tau| < \sigma \quad (1)
$$

また、次の特徴から非常に高速なアルゴリズムとなっている。

- ソートをしない
- 仮数や指数へのアクセスなど特殊命令を必要としない
- 内部ループに分岐が存在しない
- 扱っている精度で閉じている（余分な精度を必要としない）

2　多倍長計算環境の実装

現在の PC や WS の環境では、IEEE standard 754 に基づいた、仮数部の桁数が固定された浮動小数点数を扱うのが普通である。これらは CPU で直接処理できるものであり、高速な演算が期待できる。しかし、仮数部が固定されているが故に、例えば倍精度浮動小数点数での計算結果は 10 進に換算して最大 16 桁しか保証しない。それに対し、多倍長計算環境とは、計算結果に対し計算精度（有効桁数）が向上したものである。反面、多倍長計算を行うには、現在の所、ソフトウェアで多くの命令を組み合わせて実現するほかなく、後者に比べると計算時間はずっと大きくなる。

本研究では、1 つの数値を幾何学的に互いに重ならない複数の倍精度浮動小数点数の和として表現することで多倍長数計算環境を実現する。扱う多倍長数 A を図式化すると次のようになる。

図 1: 多倍長数 A

$$
A = a_1 + a_2 + ... + a_{m-1} + a_m \quad (2)
$$

$$
|a_i| \le \mathbf{eps}^{i-k}|a_k|\ ,\ \ 1 \le k \le i \le m\ ,\ \mathbf{eps} = 2^{-53} \quad (3)
$$

多倍長数同士の加減算を AccSum を用いて実現するために必要な操作はない。加算、減算の対象となる多倍長数を構成する各要素を、1 つのベクトルを構成する要素と見做して AccSum に適用すればよい。多倍長数同士の乗算を実現するためには、対象となる乗算をその浮動小数点数演算結果である積と誤差項の 2 つに無誤差変換する TwoProduct アルゴリズムを用いる [4]。乗算を和に変換することで、加減算と同様にして AccSum が適用可能となる。

3 高速化の検討

$A = (a_1, a_2, \cdots, a_{m-1}, a_m)$ を多倍長数としたとき、次の命題が成立する。

命題 3.1 **ExtractVector**(σ_h, A) によって a_i が初めて計上されたならば、**ExtractVector**(σ_h, A) によって a_{i+1} が計上されることはない

命題 3.2 **ExtractScalar**(σ_h, a_i) の実行により返された残部 a_i' は、少なくとも 2 回の ExtractScalar の実行 (**ExtractScalar**$(\sigma_{h+1}, a_i^{(1)})$, **ExtractScalar**$(\sigma_{h+2}, a_i^{(2)})$) で完全消滅する。

ExtractScalar の実行回数削減

命題 3.1 より、ループを繰り返し、σ の値を更新するごとに、**ExtractVector**(σ, p) に適用させる浮動小数点数ベクトル p を順次、$(a_1), (a_1, a_2), (a_1, a_2, a_3) \cdots$ と拡大していくアルゴリズムに改良しても、元のアルゴリズムと恒に解は等しい。アルゴリズムをこのように改良することで、加減算では ExtractScalar の実行を $\frac{1}{2}(n(n-1)+m(m-1))$ 回、乗算では ExtractScalar の実行を $mn(m+n-1)$ 回減らすことが可能となる。また、この操作にはカウンタ制御しか必要としないので分岐命令が含まれず高速に実現可能である。

2^Meps の抑えこみによる反復回数削減

命題 3.2 を利用することで、加減算による桁上がりは多倍長数を構成する要素数に依存せず 3 回以下、乗算による桁上がりは $\lceil \log_2 m(n+3) \rceil$ 回以下であることを示すことが可能となる。桁上がりの上限を見積もることで σ の更新差分量を大きくすることが可能となり、これによりループの回数を削減することができる。

4 実験検証

実行環境

- CPU : Celeron 2.00GHz
- RAM : 768MB
- OS : Windows XP
- Scilab-4.0

実験結果

- 高速化 1 : 2^M**eps** の抑えこみによる反復回数削減
- 高速化 2 : ExtractScalar の実行回数削減
- 高速化 3 : 高速化 1+高速化 2

n	高速化なし	高速化 1	高速化 2	高速化 3
3	0.004364	0.004137	0.004407	0.004186
5	0.005492	0.005054	0.005156	0.004920
10	0.010172	0.009702	0.008012	0.007076
20	0.026162	0.025828	0.018438	0.017082
30	0.058340	0.055998	0.036826	0.034894

表 1: 加減算（密な多倍長数）（一部抜粋）

n	高速化なし	高速化 1	高速化 2	高速化 3
3	0.014726	0.013445	0.016118	0.013437
5	0.018438	0.019351	0.018472	0.018178
7	0.023512	0.022908	0.022548	0.021099
10	0.031299	0.029291	0.028081	0.026911

表 2: 加減算（疎な多倍長数）（一部抜粋）

n	高速化なし	高速化 1	高速化 2	高速化 3
2	0.005005	0.005182	0.004991	0.004863
4	0.010398	0.010672	0.007386	0.007525
6	0.024932	0.025152	0.016315	0.014936
8	0.044255	0.044059	0.024081	0.022653
10	0.079330	0.079212	0.039822	0.035936

表 3: 乗算（密な多倍長数）（一部抜粋）

5 結論

数値実験より、密な多倍長数における加減算では 3 倍長まで、疎な多倍長数における加減算では 5 倍長まで、高速化処理を適用しない方が速いという結果が現れた。これは高速化処理に伴うオーバーヘッドの影響によるものだと考えられる。このオーバーヘッドの影響は、両条件とも倍長を増やすにつれ相対的に小さくなるため、高速化の効果は倍長に比例して実行時間に反映されていった。故に、高倍長では高速化処理は有用だと言える。密な多倍長数において加減算では最大約 1.7 倍程、乗算では最大約 2 倍程高速化が実現できた。

参考文献

[1] 大石 進一, Linux 数値計算ツール, コロナ社, (2000)

[2] 大石 進一, 精度保証付き数値計算, コロナ社, (2000)

[3] D. KNUTH, The Art of Computer Programming: Seminumerical Algorithms, vol. 2, Addison Wesley, Reading, Massachusetts, second edition, (1981)

[4] T. J. DEKKER, A Floating-Point Technique for Extending the Avaiable Precision, Numerische Mathematik, 18:224-242, (1971)

[5] T.OGITA, S. RUMP, AND S. OISHI, Accurate Sum and Dot Product, SIAM J. Sci. Comput., 26:1955-1988, (2005)

[6] etc... 論文を参照