平成16年度　修士論文

FMAを利用した行列積の高精度演算

平成16年2月2日

指導教授：大石　進一　教授

早稲田大学大学院　理工学研究科

情報・ネットワーク専攻

3603U029-2

大山 博毅

目 次

第 1 章

序論

1.1 背景

数値計算は数学または応用数学の一分野である. 数学には, 代数学, 幾何学, 解析学, 微分方程式などさまざまな分野があり, 人類の豊かな知識体系の一部をなしている. 数学は科学, 工学に現れる現象, 関係, 設計過程などをモデル化するのに威力を発揮している. そして, それを解くことで現象を解析し, 工学的な製品を設計することが多い. したがって微分方程式を解いたり, 積分を計算したりといったように連続数学を解くことが多くなる. しかし, 連続数学を解くことは難しい場合が多い. そこで, 計算機を利用して連続数学の問題を解こうとすることから数値計算の研究が進み, 理論が作られるようになったといっても過言ではない.

しかし, 計算機を用いた数値計算での時間, 桁, 次元は有限であるため, 実数演算を浮動小数点演算で近似するため, 丸め誤差が生じたり, 微分方程式を有限次元方程式に近似するために生じる打ち切り誤差などさまざまな誤差が含まれる.

特に、丸め誤差においては 1 回の演算で生じる誤差はたかだか有限桁の最下位桁の単位程度であるが, 莫大なステップの各ステップにおいて生じるため, 数値計算の誤差を厳密にかつシャープに評価することは非常に困難であると考えられていた.

これに対し, 真の値を含む区間を数とみなす区間解析の概念が 1958 年須永氏により導入され, 丸め誤差が存在する浮動小数点演算を用いても, 数学的に正しい結果を数値計算により導く原理が示された. この区間解析の概念は R.E.Moore の学位論文で取り上げられるといったように, 活発に研究された. その結果, いろいろな成果が得られたが, 一方では, 単に浮動小数点演算を区間演算に置き換えると, 多くの場合区間の幅が大幅に増大し, 意味ある結論が得られないことがわかった.

この困難を解消する方法が Kulisch,Krawczyk などにより示されたが. それは, 平均値形式などの関数の評価に微分係数の区間評価を用いる方法や, 近似解が得られた後にその周りで Newton 反復作用素に対して縮小写像原理が成り立つかどうかにより, 区間演算で数値的に検証する方法である. その結果, 真の解を含む区間を縮小させることも可能となり, 数値計算の誤差を厳密にかつシャープに評価することが原理的に可能であることがさまざまな例から明らかにされてきた.

この区間解析や, 近年能力が大幅に向上してきた計算機を利用することにより, 連続数学の問題に対しても演算結果が誤差を含んでいても, 数学的に正しいことを保証されているという結論を導くための理論と技術の開発を, 精度保証付き数値計算 (numerical method with guaranteed accuracy) という.

浮動小数点数演算における丸め誤差を含む演算結果の保証が理論的にも実用的にも高い精度で効率よく実現できることが明らかになり, 計算の信頼性という問題は, 数学としての数値解析の観点からようやく取り上げられることとなった. これは単に丸め誤差を厳密に評価するということだけにとどまらず, 科学技術計算の元となった数値解析アルゴリズムそのものに影響を与え, さまざまな数理科学上にあらわれる問題の解を, 数学的な厳密さで検証する方向にまで展開しつつある.

このように, 精度保証付き数値計算は計算の信頼性の立場から見た今後の科学技術の計算法のあるべき 1 つの方向として考えられようとしている.

1.2 本論文の目的

精度保証付き数値計算が重要であることは,1.1 節に述べた内容などからも明らかである. しかし, 精度保証を行うための時間が近似解を求めるときと比べ何百倍もかかるのでは実用性に乏しい. そのため, 先に述べた区間解析などを用いた高速精度保証法などが研究され, 実績を上げているが, 本論文では別のアプローチから精度を保証することを考える.

Dekker,Knuth の定理を利用し, 実行環境として Intel より提供されている Itanium2 プロセッサが持つ FMA(Fused Mutiply-Add) を利用することにより, 数値演算の結果を高速かつ高精度に求める. 本論文では数値計算の分野で頻出する行列の積を取り上げることにする.

1.3 本論文の構成

本論文の構成は以下の通りである.

第 2 章では, 精度保証つき数値計算の準備として, 精度保証の概念, 浮動小数点数システ

ムについて述べる.

第 3 章では, 高精度演算のアルゴリズムについて述べる.

第 4 章では,Intel Itanium2 プロセッサ, 高速化について述べる.

第 5 章では, 数値例に対する, 実行結果について検証を行う.

第 6 章では, 総括と結論について述べる.

第 2 章

準備

2.1 はじめに

数値計算法とは, 連続数学の問題を計算機上で四則演算に還元して解く手法のことを言う. 従来の数値計算では, 四則演算の結果がそのつど四捨五入によって丸められ, かつ極値を含む無限演算は, 全て有限演算で近似して行われてきたため正確な結果が得られなかった. ところが, 近年コンピュータ技術の発展とアルゴリズムの研究により, 計算時間の高速化が急速に進んできた. そのため, 数値計算にさらに, 真の解の存在を証明したり, 真の解との誤差範囲が保証された精度の良い近似解を得ることも重要視されつつある. これを「精度保証付き数値計算」という.

本章では, この精度保証付き数値計算の原理となる浮動小数点数と浮動小数点数への丸めの指定, 四則演算について述べる. さらに, 本論文での精度保証法と従来の精度保証法の違いを明確にするため, 現在主流となっている区間演算について述べる.

2.2 浮動小数点数

現代の数値計算では, 浮動小数点数が標準として用いられている. 浮動小数点数は現在では標準化が進められ, ほとんどのパソコンやワークステーション, ベクトル計算機, 並列計算機などで同じ形式の浮動小数点体系が用いられている. この体系に基づいて, 高速に浮動小数点演算が実行される回路がコンピュータに実装されているため, 数値計算は浮動小数点数体系上で行われることがほとんどである. そこで, このような浮動小数点数の体系の上での数値計算の精度を保証することが重要である.

そのために, まず浮動小数点数について説明する. 浮動小数点数については IEEE 標準 754 (IEEE standard 754 , 以下 IEEE 754 と略記する.) がパソコンやワークステーションなどをはじめとして, 多くのコンピュータで標準的に用いられている. 本論文では, 以下 IEEE 754 に基づく 2 進数浮動小数点数システムを考えることにする.

2.2.1 浮動小数点数

IEEE 754 では 4 つのタイプの数が用意されている. それは規格化 2 進浮動小数点数, 零, 非規格化 2 進浮動小数点数, NaN (Not a Number , 非数) である.

2 進規格化浮動小数点数

2 進規格化浮動小数点数とは

$$a = \pm\left(\frac{1}{2} + \frac{d_2}{2^2} + \frac{d_3}{2^3} + \cdots + \frac{d_N}{2^N}\right) \times 2^e, \quad (d_i = 0 \text{ または } 1). \tag{2.1}$$

と書ける数をいう. $e_{\min}$ を負の整数, $e_{\max}$ を正の整数として, e は $e_{\min} \leq e \leq e_{\max}$ となる整数である.

$$m = \frac{1}{2} + \frac{d_2}{2^2} + \frac{d_3}{2^3} + \cdots + \frac{d_N}{2^N}. \tag{2.2}$$

を符合付き仮数 (signed mantissa) といい, e を指数 (exponent) という. 指数 e も 2 進数で表される. 通常, 単精度, 倍精度 (8 byte = 64 bit) , 拡張倍精度 (10 byte = 80 bit) の浮動小数点システムがあるが, それぞれつぎのような浮動小数点システムである.

$$N = 24, \quad (-126 \leq e \leq 127),$$
$$N = 53, \quad (-1022 \leq e \leq 1023),$$
$$N = 64, \quad (-16382 \leq e \leq 16383).$$

規格化 2 進浮動小数点システムにおいて表される数の絶対値の最大値は

$$x_{\max} = \left(\frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^N}\right) 2^{e_{\max}}. \tag{2.3}$$

であり, その最小値は

$$x_{\min} = \frac{1}{2} 2^{e_{\min}}. \tag{2.4}$$

である. 倍精度数では (2.3),(2.4) はそれぞれ

$$(2 - 2^{52}) \times 2^{1023} = 1.7976931348623157 \cdots \times 10^{308}, \tag{2.5}$$

$$2^{-1023} = 2.22507358507201 \cdots \times 10^{-308}. \tag{2.6}$$

である. $|x| > x_{\max}$ のときにオーバーフロー (overflow) が生じたという.

倍精度浮動小数点数においては, 仮数部が 53bit である.

$$2^{-53} = 1.1102230246\cdots \times 10^{-16}. \tag{2.7}$$

より, 倍精度浮動小数点数は 10 進数で約 16 桁の精度がある.

零

零は規格化されて

$$+\left(\frac{0}{2} + \frac{0}{2^2} + \frac{0}{2^3} + \cdots + \frac{0}{2^N}\right) 2^{e_{\min}}. \tag{2.8}$$

と表される.

非規格化 2 進浮動小数点数

IEEE 754 では浮動小数点数は指数部が $e_{\min}$ となったとき, 仮数部の最初の桁が 1 より小さい数を表すために, デフォルトで最初の桁を 1 とすることをやめ, ここが 0 となる数を置くことを許す規格となっている. これを非規格化数 (denormalized number) という. 非規格化数の範囲に数が入ることを, 漸近アンダーフロー (gradual underflow) という.

このような非規格化浮動小数点数の正で最も小さな数は

$$2^{-1074} = 4.94065645841246544\cdots \times 10^{-324}. \tag{2.9}$$

である. これ以下の数になると, アンダーフロー (underflow) が生じたという.

NaN

このほかに, IEEE 754 ではつぎのような特別な数が用意されている.

(a) NaN(Not a Number) は $\sqrt{-5}, \dfrac{\infty}{\infty}, +\infty + (-\infty)$ など不当な演算の結果として得られる.

(b) $\pm\infty$ はオーバーフローの結果や零で割った結果として得られる.

(c) ± 0 はアンダーフローか $\pm\infty$ での割り算の結果として得られる.

2.2.2 浮動小数点数と丸め

IEEE 754 では, つぎの 4 つの丸めのモードが指定できる. c を実数 ($c \in R$) とする.

上向きの丸め (round upward)

c 以上の浮動小数点数の中で最も小さい数に丸める. これを $\triangle : R \to F$ と表す. アルゴリズムでの表記は UP とする.

下向きの丸め (round downward)

c 以下の浮動小数点数の中で最も大きい数に丸める. これを $\nabla : R \to F$ と表す. アルゴリズムでの表記は DOWN とする.

最近点への丸め (round to nearest)

c に最も近い浮動小数点数に丸める. これを $\square : R \to F$ と表す. アルゴリズムでの表記は NEAR とする. もし, このような点が 2 点ある場合には, 仮数部の最後のビットが偶数である浮動小数点数に丸める. これを偶数丸め方式 (round to even) という.

切り捨て (round toward 0)

絶対値が c 以下の浮動小数点数の中で, c に最も近いものに丸める. アルゴリズムでの表記は ZERO とする.

2.2.3 浮動小数点数の性質

丸めの演算を写像として $\bigcirc : R \to F$ と書く. すなわち, $\bigcirc$ は $\triangle, \nabla, \square$ のいずれかと考える. IEEE 754 では, 丸めの演算はつぎの条件を満たす.

$\bigcirc x = x$,(任意の $x \in F$ について),

$x \leq y \to \bigcirc x \leq \bigcirc y$, (任意の $x, y \in R$ について).

また, $x \in F$ のとき, 符号を変えることにより, $-x$ や $|x|$ が得られるので, これらは正確に計算される.

IEEE 754 では, つぎの性質が成立する.

$\square(-x) = -\square x$, (任意の $x \in R$ について),

$\triangle(-x) = -\triangle x$, (任意の $x \in R$ について),

$\nabla(-x) = -\triangle x$, (任意の $x \in R$ について).

IEEE 754 では, 浮動小数点数演算 (F 上での四則演算) は丸めとの関係により, つぎのように定義されている. $\cdot \in \{+, -, \times, /\}$, $\bigcirc \in \{\triangle, \nabla, \square\}$ のとき

$$x \bigodot y = \bigcirc(x \cdot y), \tag{2.10}$$

(任意の $x, y \in R$ について). この式は, 左辺の浮動小数点数の四則演算の結果 $x \bigodot y$ は, 右辺の数学的に正しい (実数としての) 四則演算の結果 $x \cdot y$ を指定された丸めを行って得られた数 $\bigcirc(x \cdot y)$ に一致するように計算することを表している.

また, 平方根も $(\sqrt{x})_{fp} = \bigcirc(\sqrt{x})$, (任意の $x \in F$ について).

と, 浮動小数点数演算によって計算された平方根 $(\sqrt{x})_{fp}$ は, 正確な実数演算で計算された平方根 $\sqrt{x}$ を指定された丸めの方向へ丸めた数となる. 注意すべきことは, 指数関数や三角関数などはこのような規格を満たしていない. つまり, 初等関数の値を精度保証付きで求めるためには工夫が必要である. そこでこれまでは区間演算などの研究が進められてきた.

2.3 むすび

本章では, 精度保証付き数値計算の原理となる浮動小数点数と浮動小数点数への丸めの指定, 四則演算について述べた. 精度保証の方法として区間演算の研究が進められていると述べたが, 区間演算を用いた精度保証法では, 区間の幅の爆発や丸めモードの切り換えが頻繁に起こるなど, 様々な問題が生じてくる. こうした問題に影響されることなく精度保証をできることが, 本論文における精度保証法の大きな特徴の一つである.

次章では,Dekker,Knuth の定理を利用した高精度演算について述べる.

第 3 章

高精度演算

3.1 はじめに

これまでの議論は区間演算をもとにした精度保証法であったが, 本章では別のアプローチによる高精度演算法について述べる. 数値計算を行うにあたり, 丸め誤差を減らす根本的な解決策としては, アルゴリズムを改良する, もしくは丸め誤差の最小単位を小さくする, といった方法が考えられるが, 後者の方が前者に比べて簡単なのは明らかである. 後者を実現するには多倍長演算を行うわけであるが, 前章でも述べたが, 通常の PC においては IEEE754 によって定められた通り, 単精度に対して倍精度のように, 仮数部の桁数が固定された浮動小数点数を扱うことができる. これらはハードウェア (CPU) で直接処理できるものであり, 高速な演算を行うことができる. つまり, 単精度で行われる演算の丸め誤差をなくすには, 倍精度変数としてそれぞれの値を格納し, 各演算処理を倍精度で実行することで容易にできる. しかしながら, 計算がすでに倍精度で行われている場合に, 精度を倍加するのはそう簡単ではない. 現在では多倍長演算を行う数値計算ツールなど存在するが, いずれもソフトウェア的な処理を行うため IEEE754 倍精度浮動小数点数を用いた演算よりも速度は非常に遅くなる.

そこで, 本論文では Dekker と Knuth の定理を用いることにする. これらの定理を利用すると倍精度演算のほぼ 2 倍の精度で演算することができる.

3.2 エラーフリーな演算

エラーフリーな演算とは, 先に述べた Dekker,Knuth の定理を用いた演算のことを指す. 以降, 断りがなければエラーフリーとは以下のような IEEE754 倍精度の浮動小数点演算が数学的に成り立つことをいう.

$a, b, x, y \in F$

$\circ \in +, -, \times$

$a \circ b = x + y \qquad \leftarrow x = fl(a \circ b)$

ここで,$y = \delta$ (δ は $fl(a \circ b)$ の丸め誤差) である.

3.2.1 エラーフリーな和, 差

Dekker が最初に提唱し, 続いて Knuth が提唱した.
Dekker が提唱したものは first-two-sum と呼ばれ,3 回の浮動小数点演算と 1 回の条件分岐で行う. 一方 Knuth の提唱したものは 6 回の浮動小数点演算が必要だが, 条件分岐を必要としない. 以降 n 回の浮動小数点演算にかかるコストを $n$flops として考える.

Dekker のアルゴリズム Fast-TwoSum

事前に x, y の絶対値の大小を比較することにより,3 回の浮動小数点演算でエラーフリーな加算を行うことができる.

$x, y \in F \qquad (|x| \geq |y|)$
$Fast - TwoSum(x, y) = s + \delta$

$s = fl(x + y);$
$y_a = fl(s - x);$
$\delta = fl(y - y_a);$

コスト:3 flops,1 回の条件分岐

Knuth のアルゴリズム TwoSum

fast-twosum に比べ演算回数は多いが条件分岐を行うことなく, エラーフリーに加算を行うことができる.
一般的に, 条件分岐のあるプログラムは実行速度が遅いため, 本論文では加減算では twosum を採用する.

$x, y, s, \delta \in F$

$TwoSum(x, y) = s + \delta$

$s = fl(x + y);$

$y_a = fl(s - x);$

$x_a = fl(s - y_a);$

$\delta_y = fl(y - y_a);$

$\delta_x = fl(x - x_a);$

$\delta = fl(\delta_x + \delta_y);$

コスト:6 flops

3.2.2 エラーフリーな積

Dekker の以下の 2 つのアルゴリズムを利用し,$a \times b$ をエラーフリーに演算することができる.

Dekker のアルゴリズム Split

$a \in F$ を $a = a_H + a_L$ に分解する. 但し $|a_H| \geq |a_L|$.
t は浮動小数点の bit 数.IEEE754 の double 型は 53bit.

$Split(a) = a_H + a_L$

$N = 2^{\frac{t}{2}} + 1$ ← 事前に計算

$c = fl(N \times a);$

$d = fl(c - a);$

$a_H = fl(c - d)$;

$a_L = fl(a - a_H)$;

コスト:4 flops

Dekker のアルゴリズム TwoProduct

Split を利用して $a \times b$ をエラーフリーに演算.

$a, b, x, \delta \in F$

$TwoProduct(a, b) = a + \delta$

$x = fl(a \times b)$;

$[a_H, a_L] = Split(a)$;

$[b_H, b_L] = Split(b)$;

$\delta_1 = fl(x - a_H \times b_H)$;

$\delta_2 = fl(\delta_1 - aL \times b_H)$;

$\delta_3 = fl(\delta_2 - aH \times b_L)$;

$\delta = fl(a_L \times b_L - \delta_3)$;

コスト:17 flops

3.3 高精度ベクトル内積演算

行列の内積演算を行う準備として, ベクトルの内積演算を行う DotProduct を用意する. 具体的にはベクトル $a = [a_1, a_2, \ldots, a_n]$, $b = [b_1, b_2, \ldots, b_n]$ として,

$$\vec{a} \odot \vec{b} = \sum_{k=1}^{n} a_k \odot b_k$$

を計算することである. 以降, 各演算は倍精度の浮動小数点演算とする.

3.3.1 Dot1

これは通常のベクトルの内積演算である.

$a, b, x \in F^n$

$[x] = Dot1[a, b]$

$x = a_1 \times b_1;$

$for \quad i = 2 : n$

$\quad x = x + a_i \times b_i;$

end

$\vec{a} \odot \vec{b} = \vec{x}$

コスト:$2n$ flops

3.3.2 Dot2

ベクトルの内積演算をエラーフリーに計算する.

$a, b, x, y \in F^n$

$[x, y] = Dot2[a, b]$

$[x_1, y_1] = TwoProduct(a_1, b_1);$

$for \quad i = 2 : n$

$[h, r] = TwoProduct(a_i, b_i);$

$[x_i, q] = TwoSum(x_i, h);$

$y_i = y_i + (q + r);$

end

$\vec{a} \odot \vec{b} = \vec{x} + \vec{y}$

コスト:$25n$ flops

この演算を行うと通常のベクトルの内積演算に比べ 12.5 倍の計算コストで高精度の演算ができることになる. 実際にソースコードとして書く場合には高速化のため,TwoProduct や TwoSum は関数としては書かずにインラインで書く. プログラム自体は長くなり読みづらくなるが, インライン化することで高速化することはよく知られている. これ以降もアルゴリズムは関数を用いた表記にするが, 数値実験では全てインライン化したソースを用いている.

3.4 高精度行列ベクトル積

3.3 のアルゴリズムをそれぞれ拡張し, 行列ベクトル積を演算する.

$A \in F^{n \times n}$,$v, b \in F^n$ として,

$$A \odot \vec{v} = \vec{b}$$

を計算する.

3.4.1 MVDot1

3.3.1 のアルゴリズムを行列ベクトル積に拡張したもの.

$A \in F^{n\times n}$, $v \in F^n$

$[x] = MVDot1[A, v]$

$for i = 1 : n$

$\quad x_i = A_{i1} \times v_1;$

$\quad for \quad j = 2 : n$

$\quad\quad x_i = x_i + A_{ij} \times v_j;$

end

$A \odot \vec{v} = \vec{b}$

コスト:$2n^2$ flops

3.4.2 MVDot2

3.3.2 のアルゴリズムを行列ベクトル積に拡張したもの.

$A \in F^{n\times n}$, $v, x, y \in F^n$

$[x, y] = MVDot2[A, v]$

$for i = 1 : n$

$\quad [x_i, y_i] = TwoProduct(A_{i1}, v_{i1});$

$\quad for \quad j = 2 : n$

$\quad\quad [h, r] = TwoProduct(a_{ij}, v_{ij});$

$[x_i, q] = TwoSum(x_i, h);$

$y_i = y_i + (q + r);$

end

$A \odot \vec{v} = \vec{x} + \vec{y}$

コスト:$25n^2$ flops

3.5 高精度行列積演算

3.4 のアルゴリズムをそれぞれ拡張し, 行列積を演算する.

$A, B, X, Y \in F^{n \times n}$ として,

$$AB = X + Y$$

を計算する.

3.5.1 MMDot1

3.4.1 のアルゴリズムを行列積に拡張したもの.

$A, B, X \in F^{n \times n}$

$[X] = MMDot1[A, B]$

$for \quad j = 1 : n$

$for \quad i = 1 : n$

$X_{i,j} = A_{i,1} \times B_{1,j};$

$for \quad k = 2 : n$

$X_{i,j} = X_{i,j} + A_{i,k} \times B_{k,j};$

end

end

end

$AB = X$

コスト:$2n^3$ flops

3.5.2 MMDot2

3.4.2 のアルゴリズムを行列積に拡張したもの.

$A, B \in F^{n \times n}$

$[X, Y] = MMDot2[A, B]$

$for \quad j = 1 : n$

$for \quad i = 1 : n$

$[X_{i,j}, Y_{i,j}] = TwoProduct(A_{i,1}, B_{1,j});$

$for \quad k = 2 : n$

$[h, r] = TwoProduct(A_{i,k}, B_{k,j});$

$[X_{i,j}, q] = TwoSum(A_{i,j}, h);$

$Y_{i,j} = fl(Y_{i,j} + (q + r));$

end

end

end

$AB = X + Y$

コスト:$25n^3$ flops

3.6 むすび

本章では高精度演算に関するアルゴリズム, 及びそれらを利用した各種数値計算のアルゴリズムについて述べた.

第 3.2.1 節, 第 3.2.2 節でわかるように, このアルゴリズムでは加算, 乗算のみしか行わないので, 従来の多倍長演算と比較して非常に高速である.

また, 必要とするものは Dekker,Knuth のアルゴリズムのみなので扱いやすく, また計算量も把握しやすいのが特徴である. 例を挙げると,3.5.1 と 3.5.2 のアルゴリズムでは, 計算量は 12.5 倍となっており, アルゴリズムに特別な条件分岐などを含まないので, 計算時間もおよそ 12.5 倍であると予測することができる.

次章では,Intel Itanium2 プロセッサ, 高精度演算の高速化について述べる.

第 4 章

高速化

4.1 はじめに

前章では高精度演算のアルゴリズムについて述べた. また, これらのアルゴリズムを利用することでベクトルの内積演算, 行列ベクトル積, 行列積などを高精度に演算できることを示した. 本章では行列積を取り上げ, これの高速化について述べる.

4.2 Itanium2プロセッサ

Itanium2 プロセッサとは Intel より提供されているプロセッサで,IA-64 と呼ばれることもある. 従来 32 ビットの CPU が主流であったのに対し 64 ビットを持ち, 優れた演算性能を発揮する. また, 本論文で重要となる機能である FMA(Fused Multiply-add) 命令を持っている.

4.2.1 FMA(Fused Multiply-Add)

Itanium2 の加算, 乗算などの浮動小数点演算は基本的にすべて FMA 命令で行われている. FMA とは以下のような 1 回の乗算と 1 回の加算を,1 回の浮動小数点演算と 1 回の最近点への丸めの時間で行うことができる.

$FMA(a, b, c) = a \times b + c$

コスト：1 flops

本論文に合わせて表記するとこの計算を 1flops で行うことができることになる. また, 加減算, 乗算は FMA の特殊な例として以下のように計算される.

加算 $a + b = a \times 1 + b = FMA(a, 1, b)$

減算 $a - b = a \times 1 + (-b) = FMA(a, 1, (-b))$

乗算 $a * b = a \times b + 0 = FMA(a, b, 0)$

コスト：それぞれ 1 flops

FMA 命令を持つプロセッサは Itanium2 の他に,Power4 などがある.

4.2.2 FMA を利用した TwoProductTwoProductFMA

FMA 命令を用いた TwoProduct の拡張について述べる. 3.2.2 で示した TwoProduct は 4.2.1 の FMA を用いることにより, 以下のように変換できることが知られている.

$a, b, x, \delta \in F$
$TwoProductFMA(a, b) = x + \delta$

$x = fl(a \times b);$
$\delta = FMA(a, b, (-x));$

コスト:2 flops

このように,FMA を利用することで 17 flops かかる乗算を 2 flops で行うことができる.

4.2.3 FMA を利用した行列ベクトル積： MVDot2_FMA

3.4.2 で示した MVDot2 を FMA を用いた MVDot2_FMA に拡張する.

$A \in F^{n \times n}$, $v, x, y \in F^n$
$[x, y] = MVDot2\_FMA[A, v]$

$fori = 1 : n$
$[x_i, y_i] = TwoProductFMA(A_{i1}, v_{i1});$

$for \quad j = 2 : n$

$[h, r] = TwoProductFMA(a_{ij}, v_{ij});$

$[x_i, q] = TwoSum(x_i, h);$

$y_i = y_i + (q + r);$

end

$A \odot \vec{v} = \vec{x} + \vec{y}$

コスト:$10n^2$ flops

MVDot2 では $25n^2$flops かかる計算を $10n^2$flops で行うことができる.

4.2.4 FMA を利用した行列積： MMDot2_FMA

3.5.2 で示した MMDot2 を FMA を用いた MMDot2_FMA に拡張する.

$A, B \in F^{n \times n}$

$[X, Y] = MMDot2\_FMA[A, B]$

$for \quad j = 1 : n$

$for \quad i = 1 : n$

$[X_{ij}, Y_{ij}] = TwoProductFMA(A_{i1}, B_{1j});$

$for \quad k = 2 : n$

$[h, r] = TwoProductFMA(A_{ik}, B_{kj});$

$[X_{ij}, q] = TwoSum(A_{ij}, h);$

$Y_{ij} = fl(Y_{ij} + (q + r));$

end

end

end

$AB = X + Y$
コスト:$10n^3$ flops

MMDot2 では $25n^3$flops かかる計算を $10n^3$flops で行うことができる.

4.3 ループ変換： MMDot2_FMA_Loop

行列積を行うにあたり, 高速化の手法としてループ変換がある. ループ変換には交換 (Loop InterChange), 融合 (Fusion), アンローリングなどがあるが, ここでは 4.2.4 の MMDot2_FMA を拡張し, 最も高速化が期待できるループ交換を行う MMDot2_FMA_Loop を作る. 本論文で扱うプログラムは C 言語で書かれている.C 言語で行列の宣言をした場合, 各要素は行方向へ連続して格納されるので, 列方向へメモリアクセスを行うとアクセスが非連続となってしまう. そこで, メモリアクセスが連続になるようにループの入れ替えを行い, 最適化することで高速化を試みる. このループ交換はコンパイラにより自動的に行われることもあるが, 明示的にループ交換を行うことでさらなる高速化を望める. ループ交換を行った際に演算結果が正しくなるよう, 注意が必要である. 本論文ではループ交換を行い, 以下のようにした.

$A, B \in F^{n \times n}$
$[X, Y] = MMDot2\_FMA\_Loop[A, B]$

$for \quad i = 1 : n$
  $for \quad j = 1 : n$
    $[X_{ij}, Y_{ij}] = TwoProductFMA(A_{i1}, B_{1j});$
$for \quad i = 1 : n$
  $for \quad k = 2 : n$

$for \quad j = 1 : n$

$[h, r] = TwoProductFMA(A_{ik}, B_{kj});$

$[X_{ij}, q] = TwoSum(A_{ij}, h);$

$Y_{ij} = fl(Y_{ij} + (q + r));$

end

end

end

$AB = X + Y$

コスト:$10n^3$ flops

4.4 キャッシュブロック化

すべてのプログラムに通じることであるが, プロセッサの持つキャッシュを効率的に利用することでプログラムの高速化を図ることができる. 行列積の演算においてもキャッシュを利用することは有益である. 行列の一部を読み込み, キャッシュ内でそれらの値を演算することにより, 実行速度を上げる. これをキャッシュブロック化と呼ぶ. ブロック化もコンパイラによって最適化が行われることがあるが, ここでも明示的にソースレベルでブロック化を行う. 本論文では以下の 2 通りのパターンでブロック化した.

4.4.1 ブロック化パターン 1 : MMDot2_FMA_Loop_B1

4.3 の MMDot2_FMA_Loop をブロック化した MMDot2_FMA_Loop_B1 を作る. ここでは入力となる 2 つの行列の一部分を正方のブロックとして区切り, そのサイズをキャッシュ内に収めることにより高速化を図る. ここでは見やすさのため, 一部 C 言語の表記を使う. ここでの BLCK はキャッシュサイズに対して最適なブロックサイズとする.

$A, B \in F^{n \times n}$

$[X, Y] = MMDot2\_FMA\_Loop\_B1[A, B]$

$for(i = 0;\ i < n;\ i++)\{$

$\quad for(j = 0;\ j < n;\ j++)$

$\quad\quad [X_{ij}, Y_{ij}] = TwoProdcut(A_{i1}, B_{1j});$

$for(bi = 0;\ bi < n;\ bi+ = BLCK)\{$

$\quad si = (((BLCK) < (n - bi))?(BLCK) : (n - bi));$

$\quad for(bk = 1;\ bk < n;\ bk+ = BLCK)\{$

$\quad\quad sk = (((BLCK) < (n - bk)) : (BLCK) : (n - bk);$

$\quad\quad for(bj = 0;\ bj < n;\ bj+ = BLCK);$

$\quad\quad\quad sj = (((BLCK) < (n - bj))?(BLCK) : (n - bj);$

$\quad\quad\quad for(i = bi;\ i < bi + si;\ i++)\{$

$\quad\quad\quad\quad for(k = bk;\ k < bk + sk;\ k++)\{$

$\quad\quad\quad\quad\quad for(j = bj;\ j < bj + sj;\ j++)\{$

$\quad\quad\quad\quad\quad\quad [h, r] = TwoProductFMA(A_{ik}, B_{kj});$

$\quad\quad\quad\quad\quad\quad [X_{ij}, q] = TwoSum(A_{ij}, h);$

$\quad\quad\quad\quad\quad\quad Y_{ij} = fl(Y_{ij} + (q + r));$

$\quad\quad\quad\quad\quad \}$

$\quad\quad\quad\quad \}$

$\quad\quad\quad \}$

$\quad\quad \}$

$\quad \}$

$\}$

ブロック化することにより演算量は若干増えるが, キャッシュブロック化による最適化がそ

れを上回ることができれば, 全体としては高速化できると考えられる.

4.4.2 ブロック化パターン2 : MMDot2_FMA_Loop_B2

4.3 の MMDot2_FMA_Loop をブロック化した MMDot2_FMA_Loop_B2 を作る. ここでは入力の行列の次元数に対してプロセッサのキャッシュサイズがある程度大きいとして, メモリアクセスの連続化とキャッシュブロック化の両方を考慮したパターンを示す.

$A, B \in F^{n \times n}$

$[X, Y] = MMDot2\_FMA\_Loop\_B1[A, B]$

$for(i = 0;\ i < n;\ i++)\{$

$\quad for(j = 0;\ j < n;\ j++)$

$\quad\quad [X_{ij}, Y_{ij}] = TwoProdcut(A_{i1}, B_{1j});$

$for(bi = 0;\ bi < n;\ bi+ = BLCK)\{$

$\quad si = (((BLCK) < (bi - n))?(BLCK) : (bi - n));$

$\quad for(bk = 1;\ bk < n;\ bk+ = BLCK)\{$

$\quad\quad sk = (((BLCK) < (bk - n))?(BLCK) : (bk - n));$

$\quad\quad for(i = bi;\ i < bi + si;\ i++)\{$

$\quad\quad\quad for(k = bk;\ k < bk + sk;\ k++)\{$

$\quad\quad\quad\quad for(j = 0;\ j < n;\ j++)\{$

$\quad\quad\quad\quad\quad [h, r] = TwoProductFMA(A_{ik}, B_{kj});$

$\quad\quad\quad\quad\quad [X_{ij}, q] = TwoSum(A_{ij}, h);$

$\quad\quad\quad\quad\quad Y_{i,j} = fl(Y_{ij} + (q + r));$

$\quad\quad\quad\quad \}$

$\quad\quad\quad \}$

```
    }
  }
}
```

パターン 1 に比べ, ループを 1 つ減らすことができた. 極端に次元数の大きい行列が入力された場合, キャッシュが効かなくなる場合が想定されるが, 現在のコンピュータではメモリとキャッシュメモリのサイズの関係上これは起こりえない.

4.5 むすび

本章では Itanium2 による高速化, 及びソースのチューニングによる高速化について述べた. FMA を利用できる環境で演算を行うことにより内積演算の演算量をかなり減らすことができる. 行列ベクトル積, 行列積についても同様である. また, 高精度演算に FMA を適用することにより全体として演算量を減らすことができる.

次章では, 実行環境と結果について述べる.

第 5 章

実行環境と結果

本章では本論文での実行環境とその準備, および数値実験の結果について述べる.

5.1 実行環境

実行環境は表 5.1 の通りである.

表 5.1: 実行環境

OS	Red Hat Linux 7.2
Linux version	2.4.9
CPU	Intel Itanium2 1.4GHz
Memory	PC/2100 DDR SDRAM 1024MB
Cashe	1.5MB L3 Cathe
Compiler	Intel C++ Compiler for linux 8.0
Library	Intel Math Kernel Library 7.2

5.2 インストール

今回,Intel 製のプロセッサを利用するにあたり,Linux 版のコンパイラ及びライブラリが Intel から提供されているのでそれを利用する.
”http://www.intel.co.jp/” より日本語版がダウンロードできるが, 英語版の方がスムーズにダウンロード&インストールできる事が多いのでそちらを利用した.
メールアドレスを登録してダウンロードすると,Lisence ファイルとダウンロード先の URL が送られてくるので,Lisence ファイルは覚えやすい所にコピー, ダウンロードしたものは展開し, その中にある install.sh を root 権限で実行する.

```
# sh install.sh
```

後は指示に従って行くだけだが, 途中 Lisence ファイルの場所を聞かれるので先ほどコピーした場所を指定する. インストール後に*.lib ファイルが生成されているのでそれらを PATH の通っているディレクトリに移すと使いやすい.
コンパイラも Math Kernel Library もほぼ同様の手順でインストールできる.

5.3 コンパイルオプション

Intel C++ Compiler を利用するときは以下のように”icc”コマンドを使う.

```
$ icc sample.c
```

本研究では FMA の利用, 及び IEEE754 に準拠した浮動小数点演算が必要であるので, それに対応するコンパイルオプションを利用する.

```
$ icc -O3 -mp -IPF_fltacc -IPF_fma sample.c
```

各オプションの意味は表 5.2 の通りである.

5.4 Intel Math Kernel Library

今回, 行列の内積演算を行うにあたり, 比較対照として Intel 製の Math Kernel Library を選んだ. このライブラリは BLAS,LAPACK といった線形代数ライブラリを含んでおり, 行列積を行う DGEMM を備えている. また, このライブラリ自体が Itanium 2 プロセッサに最適化されているので比較対照としては十分であると考えられる.
このライブラリを利用する場合, 以下のようなコンパイルオプションが必要となる.

表 5.2: コンパイルオプション

-O3	プログラム全体の最適化. 指定なしでは-O2 がデフォルトとされているが, 本論文では高速化のため,-O2 を超える強い最適化がかかる-O3 を使う.
-mp	浮動小数点演算の精度を完全に保つ (精度が変わる最適化を防止)
-IPC_fltacc	期待されている浮動小数点演算の精度に違反しないように, 浮動小数点コードの最適化を制御する.-mp とほぼ同様のオプションであるが, これには FMA 命令とその変化形も含まれる.4.2.2 の TwoProductFMA を実行するにはこのオプションが必要である.
-IPF_fma	FMA 命令を利用する.-O2 以上ではデフォルトで ON になっている.

$ icc -lmkl_lapack -lmkl_ipf -lguide -lpthread sample.c

各ライブラリは事前にPATHを通しておくか, ライブラリの保存場所を指定する必要がある.

5.5 実行結果

これまでのアルゴリズムを表5.1の環境の下で実装し, 実行した結果を示す.

5.5.1 行列ベクトル積

行列ベクトル積を次元数を変えながら実行時間を計る. 入力する行列, およびベクトルの各要素は $[-1, 1]$ の乱数である. アルゴリズムは,MVDot1, MVDot1_FMA, MVDot2, MVDot2_FMA の4通りで行い, 結果を表5.3に示す.

表 5.3: 計算時間 (sec)

次元数	MVDot1	MVDot1_FMA	MVDot2	MVDot2_FMA
1000	0.003	0.003	0.018	0.015
2000	0.012	0.012	0.068	0.059
3000	0.028	0.027	0.153	0.130
4000	0.049	0.049	0.272	0.231
5000	0.077	0.075	0.425	0.361
6000	0.109	0.107	0.695	0.520
7000	0.149	0.145	0.832	0.709
8000	0.193	0.193	1.097	0.925
9000	0.256	0.241	1.375	1.171
10000	0.302	0.299	1.698	1.446

次に, 計算時間の比率を表 5.4 示す. また, アルゴリズムの flops 数で考えると,MVDot1 の計算量を 1 とすると, 理論値は表 5.5 のようになるはずである.

表 5.4: 計算時間 (比率)

次元数	MVDot1	MVDot1_FMA	MVDot2	MVDot2_FMA
1000	1.00	1.00	6.00	5.00
2000	1.00	1.00	5.83	5.00
3000	1.00	0.97	5.41	4.59
4000	1.00	1.00	5.58	4.74
5000	1.00	0.97	5.51	4.68
6000	1.00	0.98	6.36	4.76
7000	1.00	0.97	5.57	4.75
8000	1.00	1.00	5.68	4.79
9000	1.00	0.94	5.38	4.58
10000	1.00	0.99	5.63	4.80

表 5.5: アルゴリズムから考えられる演算量の比率 (MVDot1 を基準とする)

MVDot1	MVDot1_FMA	MVDot2	MVDot2_FMA
1.0	0.5	12.5	5.0

ここで, 理論値と実際の計算時間の比率に差異が見られるが, これは MVDot1 もコンパイラによって FMA による最適化が行われているためと考えられる. そうすると,MVDot1 の計算時間が MVDot1_FMA とほぼ同じになる説明がつく.Itanium2 プロセッサ上で数値計算を行う場合は FMA による影響を考慮必要があると言える.

次に, 精度について結果を表 5.6, 表 5.7, 表 5.8, 表 5.9, 表 5.10 に示す.

表 5.6: MVDot1 と MVDot1_FMA における計算結果の一部

次元数	MVDot1	MVDot1_FMA
1000	2.4501091148374849e+02	2.4501091148374849e+02
2000	4.8767219984618561e+02	4.8767219984618561e+02
5000	1.2571663205051757e+03	1.2571663205051757e+03
10000	2.5035895657839133e+03	2.5035895657839133e+03

表 5.7: MVDot2 と MVDot2_FMA の 1000 次元における計算結果の一部

MVDot2	2.4501091148374849e+02 + 1.0168077798705645e-13
MVDot2_FMA	2.4501091148374849e+02 + 1.0168077798705645e-13

表 5.8: MVDot2 と MVDot2_FMA の 2000 次元における計算結果の一部

MVDot2	4.8767219984618555e+02 + -8.3659705273500565e-13
MVDot2_FMA	4.8767219984618555e+02 + -8.3659705273500565e-13

表 5.9: MVDot2 と MVDot2_FMA の 5000 次元における計算結果の一部

MVDot2	1.2571663205051755e+03 + 1.3057800626097052e-12
MVDot2_FMA	1.2571663205051755e+03 + 1.3057800626097052e-12

表 5.10: MVDot2 と MVDot2_FMA の 10000 次元における計算結果の一部

MVDot2	2.5035895657839133e+03 + 9.7895440102413145e-12
MVDot2_FMA	2.5035895657839133e+03 + 9.7895440102413145e-12

結果をみてわかるように,MVDot2,MVDot2_FMA では通常の演算結果に加えて誤差も倍精度で拾えていることがわかる. また,FMA を利用した MVDot2_FMA でも正確に誤差を計算できていることがわかる. 次元数が増えても入力要素のほぼ倍の精度で演算することが可能である.

5.5.2 行列積

行列積を次元数を変えながら実行し, 計算時間を計る. 入力する条件は行列ベクトル積と同様である. 計算の性質上, 精度については行列ベクトル積と同様の結果が得られることは明らかであるので掲載は省略する. また,Dot1 と Dot1_FMA ではほぼ同様の結果が得られることが 5.5.1 でわかったので, 以降の計算では Dot1 の計算は省略する. まず,MMDot1_FMA,MMDot2,MMDot2_FMA の実行時間を表 5.11 に示す.

表 5.11: 各アルゴリズムでの計算時間 (sec)

次元数	Dot1_FMA	Dot2	Dot2_FMA
1000	10.6	15.5	11.6
2000	152.2	185.5	156.4
3000	598.0	820.3	660.5

FMA を利用することで高速化が図れることはわかるが実用的な実行時間ではない. そこで Dot1_FMA と Dot2_FMA のアルゴリズムに 4.3 で述べたループ交換を適用する. 結果を表 5.12 に示す.

表 5.12: ループ交換を行った計算時間 (sec)

次元数	Dot1_FMA	Dot1_FMA_Loop	Dot2_FMA	Dot2_FMA_Loop
1000	10.6	1.5	11.6	4.1
2000	152.2	12.1	156.4	32.4
3000	598.0	40.2	660.5	109.1

コンパイラの最適化オプションでもループ変換は行われているはずだが, 明示的に最適なループ交換をすることでかなりの高速化ができることがわかる. 次に Dot2_FMA_Loop に 4.4 で述べた 2 通りのキャッシュブロック化を適用する. ある程度次元数の大きいものを用いないとキャッシュブロック化による効果が見えにくいと考えられるため, 次元数を 3000 とする. 最適なブロックサイズをあらかじめ調べ, それを適用したものを表 5.13 に示す.

キャッシュの最適化もコンパイラの最適化によって行われている. この結果を見ると, 明示的にキャッシュブロック化を行うことにより若干の高速化はできるが, 下手にブロック分割すると逆に遅くなってしまうという結果がでた. 次に,Dot2_FMA_Loop_B2 を 5.4 で述べた数値計算ライブラリによる行列積関数 DGEMM と比較する. 結果を表 5.14 に示す.

また,5.14 を実行性能として MFLOPS で表すと表 5.15 のようになる.

表 5.13: キャッシュブロック化を行った結果 (sec)

次元数	Dot2_FMA_Loop	Dot2_FMA_Loop_B1	Dot2_FMA_Loop_B2
3000	109.212448	116.390928	108.674672

表 5.14: 実行時間,() 内は DGEMM を 1 とした時の時間比率

次元数	DGEMM	Dot2_FMA_Loop	Dot2_FMA_Loop_B2
1000	0.44408 [1.0]	4.080656 [9.19]	4.029904 [9.07]
2000	3.48432 [1.0]	32.418816 [9.30]	32.303648 [9.27]
3000	11.773488 [1.0]	109.212448 [9.28]	108.674672 [9.23]

表 5.15: 実行性能 (MFLOPS)

次元数	DGEMM	Dot2_FMA_Loop	Dot2_FMA_Loop_B2
1000	2251.8	2450.6	2481.4
2000	2296.0	2467.7	2476.5
3000	2293.3	2472.2	2484.5

表 5.14, 表 5.15 の結果を見てわかるように, ある程度の次元数以上であれば最適なブロックサイズを指定することにより若干の高速化を行うことができる.

5.6 むすび

本章では前章までの理論, アルゴリズムを用いた数値実験を行い, その結果を示した.

次章では, 本論文の総括を行う.

第 6 章

総括

6.1 はじめに

本章では, 本論文の総括と結論について述べる.

6.2 総括

第 2 章では, 準備として浮動小数点のシステム, 精度保証付き数値計算の概念について述べた.

第 3 章では,Dekker,Knuth の定理を利用した高精度演算について述べた.

第 4 章では,Intel Itanium2 プロセッサ, 高速化について述べた.

第 5 章では, 数値例に対する, 実行結果について検証を行った.

本論文の構成は以上のようであった.

6.3 結論

第 3 章, および第 4 章で述べた通り,Dekker,Knuth の定理を利用して演算を行うことにより, 内積演算, 行列ベクトル積, 行列積に対して高精度に演算できることがわかった. 精度については倍精度演算のほぼ 2 倍の精度を得ることができる. また, これらの演算は基本的には加算, 乗算しか行っておらず, 実行時間が予測しやすいため非常に扱いやすい. 第 5 章の結果では, コンパイラの最適化の他に明示的にコードを最適化することにより高速化できることがわかった. 特にループ交換を行った際の高速化は著しく, 必須であると言える. 行列積の MMDot2_FMA_Loop_B2 の実行時間を見ると,DGEMM に対して 10 倍弱となっている.DGEMM も Intel Itanium2 に最適化されていることから, 計算量は Dot1_FMA と同じ n^3flops であると考えるとこの 10 倍弱という時間は理論値以上の性能を引き出している.DGEMM は行列積の演算では高い評価を得ており,-O2,-O3 などの最適化オプションによらずほぼ同等の実行時間のコードが生成される. つまり,DGEMM の高速化はソフトウェア的にすでに Itanium2 のピーク性能に近い値を出せると考えられる. この DGEMM に対

して理論値以上の結果がでているということは, 逆に言えばこれ以上の高速化は非常に困難であると思われる. 現に, 第 5 章のキャッシュブロック化による高速化を試みた数値実験では, 若干の高速化はできるものの, チューニングの労力に対する成果は得られていない. ただ, キャッシュブロック化によって若干でも高速化ができるということは,-O3 の最適化オプション化でもまだ高速化の余地が残されているとも考えられる. これを行うには, やはりソフトウェア的にパイプライニングを行ったり,L3 キャッシュの有効利用を意識したチューニングが必要がある. 行列積は数値計算の分野において頻出することから, さらなるチューニングを行うことは有益であると考えられる. またこの高精度演算が他の精度保証法などにおいても役立つことを期待したい. 行列積の高精度演算に関しては期待通りの精度で結果を得られ, また計算時間もプロセッサの性能を十分に引き出せたという結果をもって本論文のむすびとする.

謝辞

本研究を進めるに当たり, 終始丁寧な御指導及び御激励を賜り, その他多くの面でも色々と御面倒を見て下さり御助言を与えて下さいました大石進一教授に深く感謝いたします.

また, 終始丁寧な御指導と御教示をして下さいました丸山 晃佐氏, 荻田 武史氏, 尾崎 克久氏に大いに感謝いたします.

最後に, 研究だけでなく, 日常生活においても色々とお世話になりました大石研究室の皆様に深く感謝いたします.

参考文献

[1] 大石進一 『精度保証付き数値計算』 コロナ社.

[2] 大石進一 『Linux 数値計算ツール』 コロナ社.

[3] ANSI/IEEE, *IEEE Standard for Binary Floating Point Arithmetic*, Std 754–1985 edition, IEEE, New York, 1985.

[4] T. J. Dekker. A floating-point technique for extending the available precision. *Numer. Math.*, 18: 224–242, 1971.

[5] D. E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*. volume 2, Addison-Wesley, Reading, Massachusetts, 1969.

[6] T. Ogita, S. M. Rump, and S. Oishi. Accurate sum and dot product. *SIAM J. Sci. Comput.*, to appear.