マルチコア計算機による高精度行列 - 行列積アルゴリズムの性能評価

市村 駿太郎†，片桐 孝洋††，尾崎 克久†3, 荻田 武史†4, 永井 亨††，荻野 正雄††

BLAS（Basic Linear Subprograms）は多くの線形計算で必須のものであるが，現在，計算結果の正確性の考慮がほとんどなされていない．一方，倍精度演算による精度を保証する高精度行列-行列積アルゴリズムが知られているが，先進計算機環境での性能評価が不十分である．そこで本発表では，高精度行列-行列積アルゴリズムを複数の実装方式でスレッド並列化し，実行速度と精度の観点から性能評価した結果を報告する．特に，演算の途中で密行列から疎行列になる特性を利用した「疎行列-密行列」方式について名古屋大学に設置された FX100 システムを用いて性能評価を行った．性能評価の結果，無誤差変換により入力行列が多数疎行列になる場合において，CRS 形式による実装によるスレッド並列化方式の方が，従来の密行列演算による実装方式に対し，高精度行列－行列積ルーチン全体時間において最大で約 1.8 倍，カーネル時間で最大で約 22 倍の高速化が達成された．

1. はじめに

行列-行列積に代表される基本線形計算を集約したライブラリ BLAS (Basic Linear Algebra Subprograms)は，多くの線形計算で必須の処理である．一般に BLAS を含む従来の数値計算ライブラリでは演算速度は考慮しているが，計算結果の正確性の考慮が不十分なことが多い．解の精度保証が重要な課題となっている．一方で，BLAS を用いた汎用的な数値計算ライブラリ，たとえば，LAPACK において，精度保証をする研究や実装提案は，必ずしも多くはない．

BLAS を精度保証する研究が，早稲田大学の大石教授のグループにより進められている．本研究は大石グループで開発された高精度行列-行列演算（以降，尾崎の方法[1][2]と呼ぶ）を基本とし，その並列化を行ったものに対して評価を行う．

尾崎の手法に関して以下の実装がなされている．

i. 「疎行列 - 密行列積」, もしくは,「疎行列 - 疎行列」の実装方式（CRS 形式）

ii. 分散メモリ型計算機による並列化手法

本稿では，「疎行列 - 密行列積」，もしくは，「疎行列 - 疎行列」の実装方式に ELLPACK（ELL）形式を追加実装した尾崎の方法に対して，スレッド並列化を考慮し，演算速度と演算精度について報告するものである．

本稿は以下の構成からなる．まず 2 節で，高精度行列-行列積アルゴリズムについて説明する．3 節は，「疎行列 - 密行列積」，もしくは，「疎行列 - 疎行列」の実装方式，スレッド並列化の説明を記載する． 4 節は，尾崎の方法の性能を実行時間と演算精度の観点から評価する．5 節は，従来研究との比較である．最後にまとめを行う．

† 名古屋大学 大学院情報科学研究科
Graduate School of Information Science, Nagoya University
†† 名古屋大学 情報基盤センター
Information Technology Center, Nagoya University
†3 芝浦工業大学 システム理工学部
College of System Engineering and Science, Shibaura Institute of Technology.
†4 東京女子大学 現代教養学部
Division of Mathematical Science, Tokyo Woman's Christian University

2. 高精度行列 - 行列積アルゴリズム

2.1 概要

尾崎の方法は，入力行列に対して，以下の式（1）の無誤差変換を行う[1]：

I)行列 A と行列 B を下記のように分解する(インデックスが若いほうが高いビットを持つようにする)

$$A = A^{(1)}+A^{(2)}+A^{(3)}+\cdots+A^{(p)}$$
$$B = B^{(1)}+B^{(2)}+B^{(3)}+\cdots+B^{(q)} \quad \cdots(1)$$

II)行列積 AB を以下のように計算する（ $p\times q$ 個の行列積となる）

$$\begin{aligned} AB &= (A^{(1)}+A^{(2)}+\cdots+A^{(p)})\,(B^{(1)}+B^{(2)}+\cdots+B^{(q)}) \\ &= (A^{(1)}+A^{(2)}+\cdots+A^{(p)})\,(B^{(1)}+B^{(2)}+\cdots+B^{(q)}) \\ &= A^{(1)}\ B^{(1)}+\ A^{(1)}\ B^{(2)}+\ A^{(2)}\ B^{(1)}+\cdots+A^{(p)}\ B^{(q)} \end{aligned} \quad \cdots(2)$$

ここで，式(2)の分解された行列積どうしの加算には，高精度な和を行う演算で加算される[1]．

処理 I）の分解の仕方を工夫することで，処理 II)における行列積を無誤差の演算にすることができる[1]．行列サイズが大きくなるに従い，ほとんどの演算時間は行列積部分の時間となる．

また，分解数である p， q を限定すれば，部分的な多倍長化演算と同様の効果を得ることができる．したがって，演算時間を考慮して高精度化を図ることが可能である．

処理 I)の分解の過程で，入力行列の要素値の散らばり度合い（レンジ）に依存し，疎行列が生成される．このため密行列を入力としても，分解の過程で疎度が大きくなる場合は，密行列から疎行列化したほうがよい[1]．すなわち，「疎行列 - 密行列積」，および，「疎行列 - 疎行列積」の演算（SpMV）に切り替えるほうが，全体の計算量（実行時間）の縮減が期待できる．ただし，一般の数値計算ライブラリは，「疎行列 - 密行列積」，および，「疎行列 - 疎行列積」に特化した実装は提供されていないので，

新規開発が必要である．また，この並列化は単純でないことが予想されるので，コードの独自並列化が必要となる．

2.2 表記法

以下に本原稿で扱う表記法についてまとめる．

- fl（・）： 最近点への丸めで行う浮動小数点演算
- F：浮動小数点数の集合
- A：サイズが m 行 n 列の行列
- B：サイズが n 行 p 列の行列
- u：the unit round off
    - ➢ 2^{-24} ：IEEE 754 binary32 (single precision)
    - ➢ 2^{-53} ：IEEE 754 binary64 (double precision)

2.3 尾崎の方法の主演算

尾崎の方法では，図 1 が主演算をなす．ここで，式 (2) において，高性能和以外の部分である．また，行列 A の分解数を，n_A，行列 B の分解数を n_B とした．

```
Function EF = EFT_Mul(A, B)
    [A, n_A ] := Split_A;   [B, n_B ] := Split_B;
    k := 1;
    for i =1: n_A
        for j =1: n_B
            EF{ k } := A{ i } * B{ j };    k := k + 1;
end;  end;  end
```

図 1 $AB = \sum_{k=1}^{n_A n_B} EF^{(k)}$ の変形部分

図 1 の EF の和には，faithful と呼ばれる結果を得るアルゴリズム[3]を用いている．そのため演算結果は，ほぼ最良になることが知られている．

図 2 に行列 A の分解部分（無誤差変換）の詳細を載せる．行列 B の無誤差変換については，図 2 の行列 A の無誤差変換に対して転置した処理と同等なので説明を省略する．

3. 「疎行列 - 密行列積」,「疎行列 - 疎行列」の実装方式

3.1 概要

尾崎の方法に，「疎行列 - 密行列積」，および，「疎行列 - 疎行列積」を行うため必要である疎行列格納形式を説明する．ここでは，CRS 形式および ELL 形式について説明を行う．

3.2 CRS 形式

CRS（Compressed Row Storage）形式は，疎行列を行方向に順に走査し非零要素を格納する形式である．いま，$n \times n$ 行列 $A = (a_{ij})$ の非零要素数を nnz とする．このとき，CRS 形式では図 3 に示されるように，以下の 3 つの配列を使用する．

1) 非零要素の値を保持する長さ nnz の配列 val
2) 配列 val に対応する非零要素の列番号を格納した，長さ nnz の配列 colind
3) 配列 val, colind における各行の開始位置を記憶する $n+1$ の配列 rowptr

```
Function [D, n_A ] = Split_A(A)
    n_A := 0; n := size(A, 2)
    while (norm (A, inf )~=0)
        n_A := n_A + 1;
        μ := max(abs(A),[ ],2);
        τ := 2.^ceil ((log2(u^-1 ) + log2(n +1))/2);
        t := 2.^ceil((log2( μ )* τ ;
        δ_A^(nA) := repmat(t, 1, q);
        A[n_A ] := fl((A + δ_A^(nA) ) − δ_A^(nA) );
        A := fl(A - A[n_A ]);
```

図 2 行列 A の分解部分（無誤差変換）の詳細

$$\begin{bmatrix} a_1^1 & b_2^2 & c_3^3 & 0 \\ 0 & 0 & 0 & d_4^4 \\ e_1^5 & 0 & 0 & f_4^6 \\ 0 & 0 & g_3^7 & 0 \end{bmatrix}$$

$$\mathrm{val} = [a \quad b \quad c \quad d \quad e \quad f \quad g]$$
$$\mathrm{colind} = [1 \quad 2 \quad 3 \quad 4 \quad 1 \quad 4 \quad 3]$$
$$\mathrm{rowptr} = [1 \quad 4 \quad 8 \quad 7 \quad 8]$$

図 3 CRS 形式の例

3.3 ELL 形式

ELL（ELLPACK）形式は，各行における成分数を最大非零成分数 numcol に固定する方法である（図 4)．実際に非零成分が存在しない場合は 0 でパディングする．CRS と比較して高いメモリアクセス効率が得られる一方で，計算量，必要記憶容量が増加することが知られている．

$$\begin{bmatrix} a_1^1 & b_2^2 & c_3^3 & 0 \\ 0 & 0 & 0 & d_4^4 \\ e_1^5 & 0 & 0 & f_4^6 \\ 0 & 0 & g_3^7 & 0 \end{bmatrix} \Rightarrow \begin{bmatrix} a & b & c \\ d & 0 & 0 \\ e & f & 0 \\ g & 0 & 0 \end{bmatrix}$$

$$\mathrm{val} = [a \quad b \quad c \quad d \quad 0 \quad 0 \quad e \quad f \quad 0 \quad g \quad 0 \quad 0]$$
$$\mathrm{colind} = [1 \quad 2 \quad 3 \quad 4 \quad 0 \quad 0 \quad 1 \quad 4 \quad 0 \quad 3 \quad 0 \quad 0]$$
$$\mathrm{numcol} = 3$$

図 4　ELL 形式の例

3.4 スレッド並列化の実装方法

3.4.1 dgemm による実装

図 1 での各部分行列の行列-行列積をスレッド並列化する時，BLAS ルーチン dgemm を用いるとする．このとき，以下の２通りの実現方法が知られている[4]．

I. 一つの行列 - 行列積を行う BLAS 関数 dgemm をスレッド並列化する方法（**dgemm スレッド並列化**）

II. dgemm は逐次計算を行い，図 1 自体の並列性を抽出し，スレッド並列化する方法（**逐次 dgemm で問題レベル並列化**）

3.4.2 疎行列演算による実装

行列-行列積 $A\,B$ を行う場合，行列の要素値の幅が大きい場合，無誤差変換により，行列 A と行列 B が疎行列になることが予想される．その場合，密行列演算を行うと多くが 0 演算となるため，無駄に計算量を増加させることになる．そこで，無誤差変換時に疎行列と判定される場合において，密行列から疎行列へ変換する実装を考える．

一般に，行列 A と行列 B で，双方とも疎行列になる可能性がある．しかし，行列-行列積演算 $A\,B$ を考慮すると，行列 A の疎度を判定して疎行列化することとする．

行列 A を疎行列化する場合，行列 B は密行列として扱う．このとき，行列-行列積 $C = A\,B$ は以下のようになる．

$$c_i = \mathrm{Sp}(A)\; b_i \;,\; (i = 1,\ldots,n), \qquad \ldots(3)$$

ここで，行列 C の i 列目のベクトルを c_i，行列 B の i 列目のベクトルを b_i，および行列 A を疎行列化した疎行列を Sp(A)と記載した．ここで，式(3)の計算は，疎行列-ベクトル積（Sparse Matrix-Vector Multiplications, SpMV）である．

式(3)の計算をスレッド並列化する方法について，以下の 3 種類の実装方式が知られている[5].

1.内部並列化
SpMV 内でスレッド並列化する方法．SpMV の行単位の並列性を利用して，スレッド並列化する．（図 5）

2.外部並列化
SpMV 呼び出し部分でのスレッド並列性を使う（図 6）．

3.複数右辺による内部並列化
複数右辺専用の SpMV における，内部のスレッド並列性を使う（図 7）．なお図 7 は，同時に計算する複数右辺の数を m とすると，m 個の右辺ごとに計算することで，演算のブロック化を行う演算である．

```
for (i=0; i<n; i++) {
#pragma omp parallel for
  for (j=0; j<n; j++) {
    (ci )_j= Sp(A)_j bi
  }
}
```

図 5　内部並列化の OpenMP によるスレッド並列化のコード．ここで，(c_i)_j は，ベクトルの第 j 番の要素，Sp(A)_j は，疎行列 Sp(A)の j 列ベクトルである．

```
#pragma omp parallel for
for (i=0; i<n; i++) {
    ci = Sp(A) bi
}
```

図 6　外部並列化の OpenMP によるスレッド並列化のコード．

```
#pragma omp parallel for
for (j=0; j<n; j++) {
  for (i=0; i<n; i+=m) {
    (ci )_j= Sp(A)_j Bi:i+m-1
  }
}
```

図 7　複数右辺利用の内部並列化の OpenMP によるスレッド並列化のコード．ここで，$B_{i:i+m-1}$ = （b_i，$b_{i+1, \ldots,}$ b_{i+m-1}）からなる行列である．

図 5 の内部並列が通常 SpMV をスレッド並列化するときに行う実装方式である．本課題においては，SpMV で行列積を行っているため，必ず複数右辺に相当する行列 B の列からなるベクトル b_i が複数存在する．そのため，複数ベクトル b_i 単位の並列化である外部並列（図 6，この場合は SpMV 演算は逐次処理），と，従来の内部並列の行単位の並列性と複数ベクトル b_i 単位の並列性を利用した，複数右辺による内部並列化（図 7）が実装できる．

外部並列化に対する複数右辺による内部並列化の優位性は，複数右辺計算時に，疎行列の要素を再利用できることにある．そのため，複数右辺による内部並列化では，ブロック幅 m を適切に設定することで，演算効率の向上が見込める．

4. 性能評価

4.1 評価環境

ここでは尾崎の方法の演算精度評価を行う．

評価では以下の計算機を利用した．

1. Fujitsu PRIMEHPC FX100 (FX100)
   - 名古屋大学情報基盤センター設置
   - CPU：SPARC64 XIfx, 2.2 GHz 32(+2)コア
   - 記憶容量：32 GB
   - 理論ピーク性能（ノード）：1.1264 TFLOPS(倍精度)，2.2528 TFLOPS（単精度）
   - キャッシュ構成
     - L1:64KB（命令／データ分離，コア毎)，L2:24MB（共有）
     - 4 ウェイ
   - 1 ソケット当たり 16 コア，ノードあたり 2 ソケットの NUMA 構成
   - 富士通 MPI
   - コンパイラ:Fujitsu C/C++ Compiler Driver Version 2.0.0 P-id: T01776-01 (Jun 22 2016 14:52:00)
   - コンパイラオプション:
     - 疎行列カーネル部分：-Kfast –Kopenmp
     - それ以外：-O0 –Kopenmp
   - メモリアクセス性能（node あたり）：240 GB/秒 (入力／出力ごと)

尾崎の方法は C のコードで記述されたものを利用し，FX100 に任意精度である MPFR ライブラリを導入してこれを真値として相対誤差を調べた．

評価は 1 ノード 32 スレッドに固定した．

富士通ライブラリの FX100 向け BLAS（スレッド並列版，および逐次版の双方）を BLAS 演算部分には利用している．

4.2 入力行列

試験行列は以下のとおりである．

1. 行列 A，B の要素を 0~1 の範囲で生成.
2. 行列 A の要素を 0~1 の範囲で生成．B は A の逆行列.
3. 行列 A, B の要素を 0~1 の範囲で生成し，ある疎度分の要素に対し$\mathrm{pow}(10, \mathrm{rand}(\ )\%\Phi)$ で生成した値を挿入.
4. 行列 A を単位行列とある疎度分の要素に対して 0~1 の範囲で生成する．B は A の逆行列。

以上のΦの値が大きいと，行列要素の値の分散が大きくなる. この場合, 尾崎の方法による行列分割の回数が増え，行列 - 行列積の演算回数も増える．したがって，総合的な演算量（演算時間）が増加するので，並列処理の効果に影響する．

4.3 実験条件

- 行列サイズは N=200，および N=1000 とした．
- Φ=5 に固定した．
- 試験行列 3 については，入力行列に値を入れる際の疎度を 10%～90%まで，10 ポイントずつ変化させた．
- MPFR ライブラリの桁数は 2 進 212 桁とした．
- 誤差評価基準は以下の通りである．
  - MPFR による行列-行列積の結果をC^*とするとき，以下の相対精度を計算する：

$$\max_{1<i,j<n} \left|C_{i,j}^* - C_{i,j}\right| / \left|C_{i,j}^*\right|, \qquad \ldots(4)$$

ここで，$C_{i,j}^*$は，行列の i 行，j 列の要素を示している．

4.4 実装方式

以下の 11 種類を実装した．

1. dgemm による実装.（以降，dgemm と記載.）
2. 尾崎の方法で dgemm を用いる実装．ここで，dgemm 内でスレッド実行する方式である．(以降，尾崎(dgemm)と記載.）
3. 尾崎の方法において疎度 90%以上で CRS 形式による疎行列化を行う方式．ここで，実装方式は，内部並列化．(以降，尾崎（CRS，内部並列）と記載)．
4. 尾崎の方法において疎度 90%以上で CRS 形式による疎行列化を行う方式．ここで，実装方式は，外部並列化．(以降，尾崎（CRS，外部並列）と記載)．
5. 尾崎の方法において疎度 90%以上で CRS 形式による疎行列化を行う方式．ここで，実装方式は，複数右辺による内部並列化（以降，尾崎（CRS，複数右辺）と記載)．
6. 尾崎の方法において疎度 90%以上で CRS 形式による疎行列化を行う方式．ここで，実装方式は，複数右辺による内部並列化でブロック幅が 100 に固定.（以降，尾崎（CRS，複数右辺（100））と記載)．
7. 尾崎の方法において疎度 90%以上で ELL 形式による疎行列化を行う方式．ここで，実装方式は，内部並列化．(以降，尾崎（ELL，内部並列）と記載)．
8. 尾崎の方法において疎度 90%以上で ELL 形式による疎行列化を行う方式．ここで，実装方式は，外部並列化．(以降，尾崎（ELL，外部並列）と記載)．
9. 尾崎の方法において疎度 90%以上で ELL 形式による疎行列化を行う方式．ここで，実装方式は，複数右辺による内部並列化（以降，尾崎（ELL，複数右辺）と記載)．
10. 尾崎の方法において疎度 90%以上で ELL 形式による疎行列化を行う方式．ここで，実装方式は，複数右辺による内部並列化でブロック幅が 100 に固定.（以降，尾崎（ELL，複数右辺（100））と記載)．
11. 内積演算による高精度和の方式を利用して演算する方式 Dot2 [6]（以降，高精度内積と記載）.なお，行列は疎行列化を行わず密行列として取り扱う．

4.5 結果

ここでは，尾崎の方法の性能を実行時間と演算精度の観点から評価を行う。

4.5.1 無誤差変換における分割数

各問題における行列 A と行列 B に関する無誤差変換の分解数（それぞれ，Ak および Bk），そのうち疎行列化された数（Spm）は以下の通りであった．

- 問題 1
  - N=200：Ak=3，Bk=4， Spm = 1
  - N=1000: Ak=3，Bk=5， Spm = 1
- 問題 2
  - N=200：Ak=3，Bk=4， Spm = 0
  - N=1000: Ak=3，Bk=5， Spm = 0
- 問題 3
  - N=200：Ak=3-5，Bk=3-5， Spm = 0
  - N=1000: Ak=4-5，Bk=3-5， Spm = 1
- 問題 4
  - N=200：Ak=2，Bk=5， Spm = 2
  - N=1000: Ak=3，Bk=3， Spm = 2

4.5.2 演算精度と速度の比較（N=200）

図 8 に，入力行列 1 に対する結果を示す．

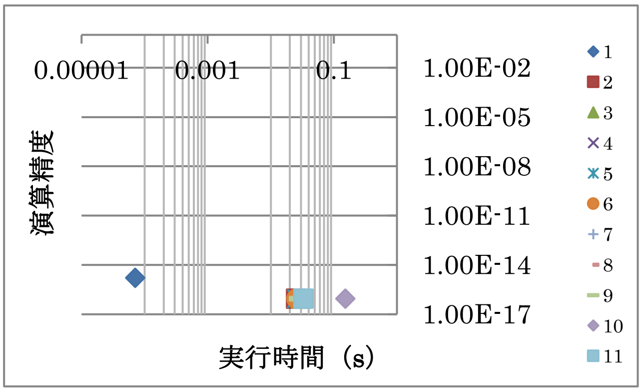

図 8 入力行列 1 に対する各実装方式の演算精度と速度の関係

図 8 では，dgemm の精度は 1.6e-15 であるが，尾崎の方法の精度は 8.9e-17 程度である．一方，dgemm に対する尾崎の方法の実行時間は約 100 倍である．

図 9 に，入力行列 2 に対する結果を示す．

図 9 では，dgemm の精度が 7.2e+3 と極めて悪く，演算精度の観点で破綻している．一方，尾崎の方法を適用することで 1.1e-17 の精度を保持している．そのため，図 9 の入力行列においては，精度の関連から尾崎の方法の導入の意義がある．

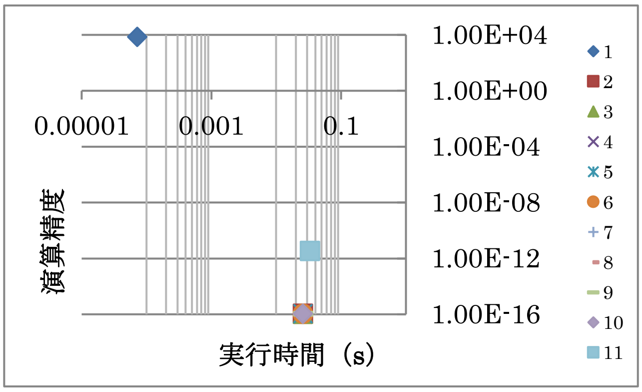

図 9 入力行列 2 に対する各実装方式の演算精度と速度の関係

図 10 に，入力行列 3 に対する結果を示す．

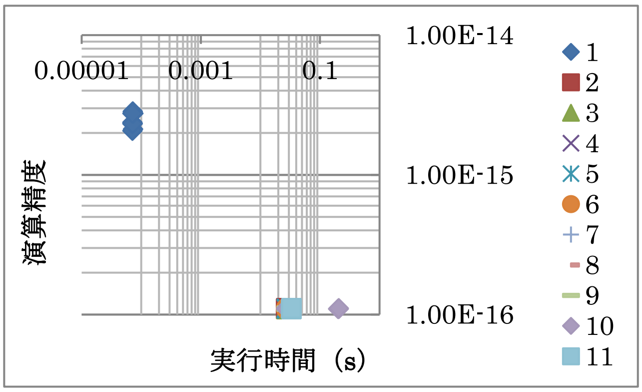

図 10 入力行列 3 に対する各実装方式の演算精度と速度の関係

図 10 では，入力行列に大きな値をランダムな位置に 10%~90%の割合で挿入するため，各実装手法の点が複数存在するが，全体の傾向は変わらず，dgemm では約 2e-15 ~ 3e-15 の精度であるが，尾崎の方法は 1.1e-16 の精度を維持している．

図 11 に，入力行列 4 に対する結果を示す．

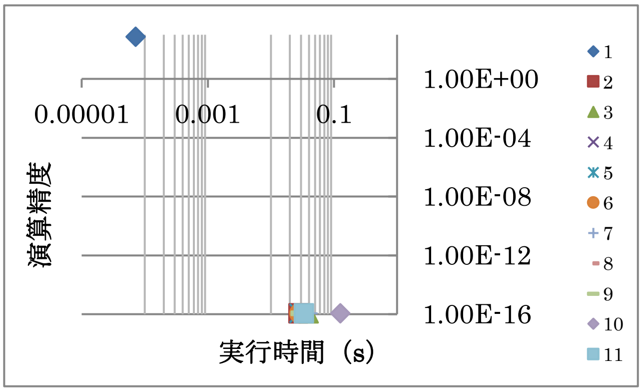

図 12 入力行列 4 に対する各実装方式の演算精度と速度の関係

図 12 では，同様に dgemm は 7.2e+2 の精度破綻しているのに対し，尾崎の方法は 1.1e-16 の精度を維持している．

4.5.3 実行時間の比較（N=1000）

本節では，比較的大規模な問題サイズに対して，尾崎の方法を評価する．ここでは，尾崎（dgemm）を基準として，各実装手法の時間を評価した．なお，高精度行列-行列積ルーチン全体の時間（無誤差変換などの時間を含む），およびカーネル時間（疎行列データ変換時間と行列－行列積時間）の 2 種を測定した．なお，実装方式 10 は，他の方式に比べ 20-30 倍の実行時間を要したため，時間は掲載していない．

図 13，図 14 に，入力行列 1 の各種時間を載せる．

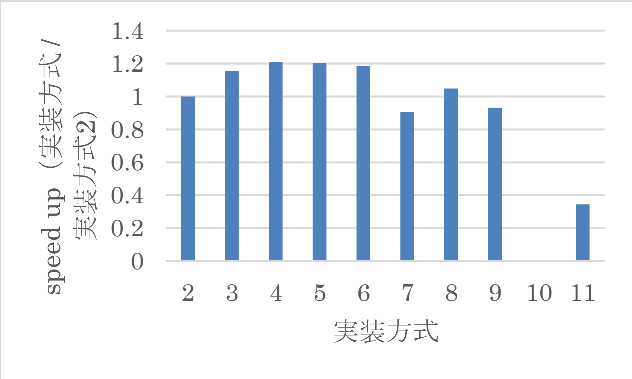

図 13　入力行列 1 に対して実装方式 2 を 1 とした場合の速度向上率（ルーチン全体時間）

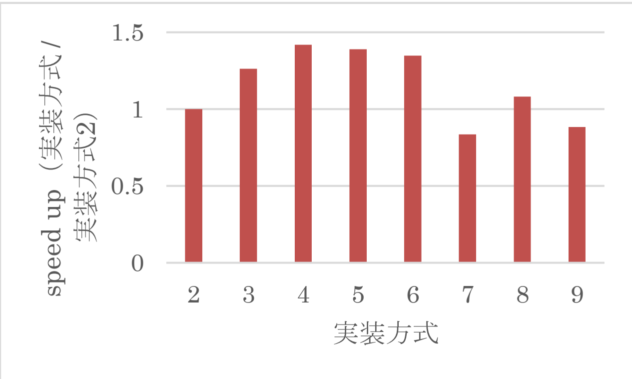

図 14　入力行列 1 に対して実装方式 2 を 1 とした場合の速度向上率（カーネル時間）

図 13，図 14 から，実装方式 4（尾崎（CRS，外部並列））が最も速く，尾崎（dgemm）に対して，ルーチン全体で約 1.2 倍，カーネル時間で 1.4 倍の速度向上を得た．

図 15，図 16 に，入力行列 2 の各種時間を載せる．

図 15，図 16 から，この行列の場合は，尾崎（dgemm）に対する速度向上があまりなく，カーネル時間で実装方式 5（尾崎（CRS，複数右辺）において約 1.08 倍の速度向上を得た．実装による差が無い理由は，無誤差変換後の行列において，疎行列化される行列が無かったためと推察される.なお，疎行列化が行われない場合は，実装方式 3～9 は，dgemm を呼ぶだけの実装なため，尾崎（dgemm）ほぼ同等の時間になると推察されるが，図 16 のカーネル時間で差が出る理由については，測定誤差を含め解析が必要である．

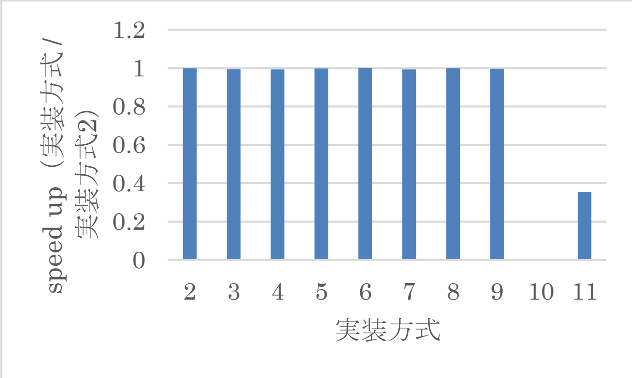

図 15　入力行列 2 に対して実装方式 2 を 1 とした場合の速度向上率（ルーチン全体時間）

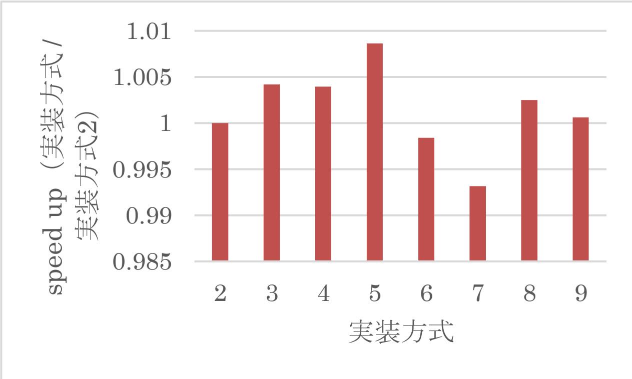

図 16　入力行列 2 に対して実装方式 2 を 1 とした場合の速度向上率（カーネル時間）

図 17，図 18 に，入力行列 3 の各種時間を載せる．

図 17，図 18 から，この行列の場合は，尾崎（dgemm）に対して，大きな値を入れ込む度合いは関係なく，実装方式 4（尾崎（CRS，外部並列））が最高速であり，尾崎（dgemm）に対して，ルーチン全体時間で約 1.1 倍，カーネル時間で約 1.3 倍の速度向上を得た．

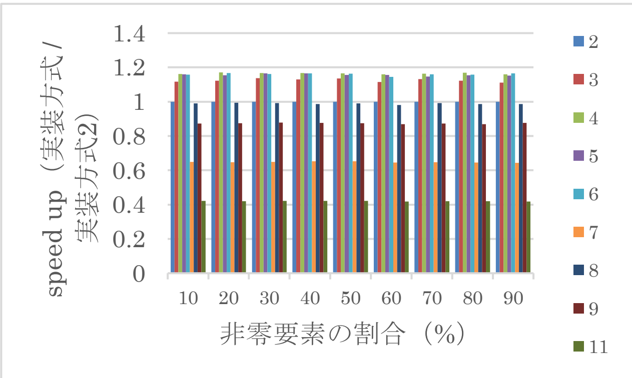

図 17　入力行列 3 に対して実装方式 2 を 1 とした場合の速度向上率（ルーチン全体時間）

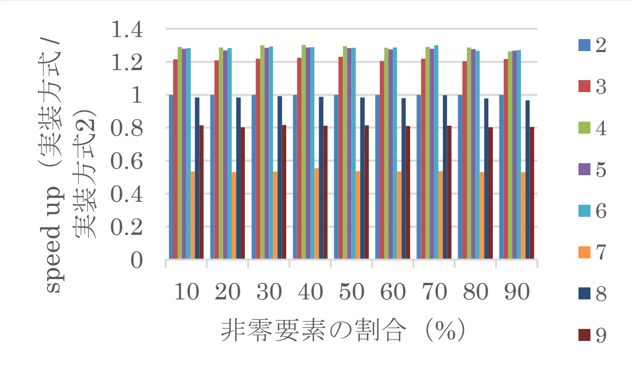

図 17　入力行列 3 に対して実装方式 2 を 1 とした場合の速度向上率(カーネル時間)

図 18，図 19 に，入力行列 4 の各種時間を載せる．

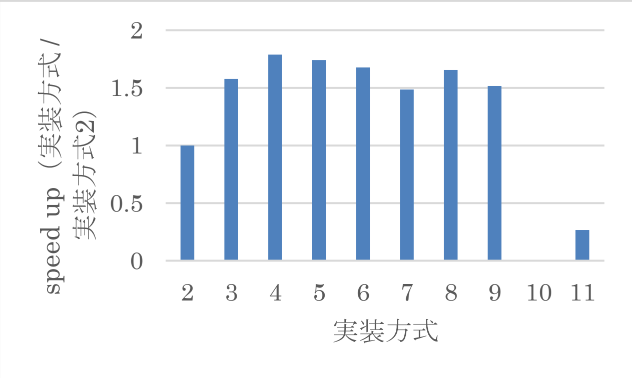

図 18　入力行列 4 に対して実装方式 2 を 1 とした場合の速度向上率（ルーチン全体時間）

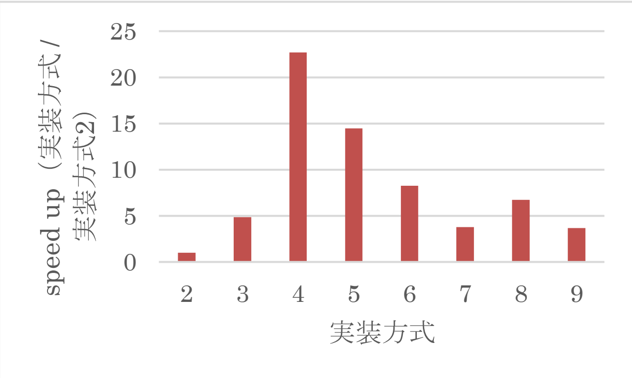

図 19　入力行列 4 に対して実装方式 2 を 1 とした場合の速度向上率(カーネル時間)

図 18，図 19 から，この行列の場合は，尾崎（dgemm）に対して，実装方式 4（尾崎（CRS，外部並列））が，ルーチン全体時間で約 1.8 倍，カーネル時間で約 22 倍も高速化されている．この理由は，入力行列 4 は入力行列が最初から疎行列のため，無誤差変換された行列も全て疎行列となり，演算量の観点で，密行列演算を利用している尾崎（dgemm）に対して有利になるからである．

4.6　考察

図 8〜図 12 の結果より尾崎の方法を利用することで,通常の倍精度演算である dgemm では実現できない高精度演算が可能になることがわかる。

一方，図 18，図 19 から，入力行列 5 のように最初から疎行列の場合には，導入した疎行列演算を用いることで，内積演算による高精度和の方式と比べても，極めて高い速度向上が達成される．

一方，高精度内積であるが，今回は疎行列化した尾崎の方法よりも約 4 倍実行時間は遅い．しかし今回の実験では密行列実装をしている．高精度内積にも疎行列化を適用することができるため，今回の試験行列では演算精度は尾崎の方法と同等のことを考慮すると，疎行列化により高精度内積が高速化できれば，有力な手法となるかもしれない．そのため，再評価が必要である．

また ELL においても，演算カーネルのチューニングが不十分である．さらに，無誤差変換により事前に疎行列となる行列がわかるため，その情報をもとに対象行列全てを ELL 形式に変換するなどの疎行列化時間を短縮すると，CRS 形式よりも有利な局面があると予想される．そのため，ELL 形式の性能の再評価も行う必要がある．

5.　関連研究

行列 - 行列積を含む，数値計算を高精度化（多倍長演算化）する研究は，多くの先行研究と開発ライブラリが知られている．

まず汎用的な多倍長演算ライブラリでは，GNU Multi-Precision Library (GMP) [7] や Extend precision floating-point arithmetic library (exflib) [8]が開発されており，多くの数値計算の事例で利用されている．

数値計算ライブラリ LAPACK を多倍長化したライブラリとしては，Multiple precision arithmetic BLAS (MBLAS) and LAPACK (MLAPACK) [9]がある．

BLAS レベルの多倍長（混合精度）演算化を目的にしているものでは，Extra Precise Basic Linear Algebra Subroutines (XBLAS) [10]が知られている．XBLAS は，拡張精度として，double-double precision (128-bit total, 106-bit significand)を用いて実装されており，double-double precision の基本演算は +, -, *, / が開発されている．また混合精度演算では，いくつかの入出力において，異なる型（real と complex の混合，もしくは，精度（single と double）の混合，が提供されている．

以上の研究と本研究のアプローチとの違いは，尾崎の方法では，入力データの行列要素の値を考慮し，必要な多倍長桁数が自動設定（動的設定）される方式になっている点である．また，対象の演算精度（ここでは倍精度計算）の丸め誤差の限界までの精度を保証することができる点にあ

る．上記のライブラリにおける桁数は，事前にユーザが設定（静的設定）する必要があるので，個別の演算に応じて，動的に多倍長演算の＜桁数＞を変更できない．すなわち，場合により無用に高精度演算がなされる可能性がある．
一方で尾崎の方法では，必要に応じて演算を疎行列化できることから，動的に計算量削減も実現できる点も，従来方式には無い特徴となる．

6. おわりに

本稿では，高精度行列 - 行列積演算を可能にする尾崎の方法について，スレッド並列化したものを FX100（名古屋大学情報基盤センター）の 1 ノード 32 スレッドを用いて評価した．性能評価の結果，入力行列が高い疎度を持つ場合に疎行列演算を行うことにより演算効率が大きく上昇することを確認した．具体的には，疎行列化の方式において CRS 形式では外部並列方式のスレッド並列化を行う場合，従来の高性能 BLAS による密行列演算 dgemm での尾崎の方法に対して，高精度行列－行列積ルーチン全体時間において最大で約 1.8 倍，カーネル時間で最大で約 22 倍高速化される事例があることが明らかになった．

ただしこの結果は，入力行列のサイズや入力値の分布，スレッド数，および疎行列演算 SpMV の性能に依存するものと考えられるため，さらに多様な実験環境のもとで性能評価を行う必要があると考える．

本研究で開発したプログラムには，幾つかのチューニングパラメタが存在する．例えば，疎行列と見なす疎度，CRS における SpMV において同時に計算する複数右辺の数，および，尾崎の方法の実装方式の選択（dgemm を使う実装方式，CRS を使う実装方式（4 種），ELL を使う実装方式（4 種））などである．これらは，対象となるハードウェア構成，入力行列の数値特性に依存し，推奨パラメタの設定が困難である．これらの性能に関するパラメタの自動性能チューニング[11]-[14]の適用は，重要な今後の課題である．

謝辞

本研究の一部は，文部科学省委託事業，ポスト「京」萌芽的課題アプリケーション開発，萌芽的課題 1，基礎科学のフロンティアー極限への挑戦「極限の探究に資する精度保証付き数値計算学の展開と超高性能計算環境の創成」，および，科学技術研究費補助金，挑戦的萌芽研究「精度保証のための高性能基盤技術の創成」の支援による．

参考文献

1) K. Ozaki, T. Ogita, S. Oishi, S.M. Rump: Error-Free Transformation of Matrix Multiplication by Using Fast Routines of Matrix Multiplication and its Applications, Numerical Algorithms, Vol. 59, No.1, pp.95-118, 2012.
2) 尾崎克久，荻田武史：浮動小数点数として最高の結果を返す行列積の計算法，京都大学 数理解析研究所 研究集会「科学技術計算における理論と応用の新展開」，2011 年
3) S.M. Rump, T. Ogita, and S. Oishi,: Accurate floating-point summation part I: Faithful rounding. SIAM J. Sci. Comput., Vol. 31, No.1, pp.189-224, 2008.
4) 片桐孝洋，尾崎克久,荻田武史,大石進一：高精度行列 - 行列積アルゴリズムのスレッド並列化と ABCLibScript への機能実装，情報処理学会研究報告，2012-HPC-133 (26), pp.1-8 (2012)
5) 片桐孝洋，尾崎克久，荻田武史，大石進一：高精度行列-行列積アルゴリズムの疎行列演算化による高速化，日本応用数理学会「行列・固有値問題の解法とその応用」研究部会第 15 回研究会，SWoPP2013, 2013 年
6) T. Ogita, S. M. Rump, S. Oishi, Accurate sum and dot product, SIAM Journal on Scientific Computing, vol. 26, no. 6, pp. 1955–1988, 2005.
7) The GNU Multi-Precision Arithmetic Library（GMP） http://gmplib.org/
8) Extend precision floating-point arithmetic library http://www-an.acs.i.kyoto-u.ac.jp/~fujiwara/exflib/
9) The MPACK; Multiple precision arithmetic BLAS (MBLAS) and LAPACK (MLAPACK) http://mplapack.sourceforge.net/
10) Extra Precise Basic Linear Algebra Subroutines (XBLAS) http://www.netlib.org/xblas/
11) T. Katagiri, K. Kise, H. Honda, T. Yuba, “ABCLibScript: a directive to support specification of an auto-tuning facility for numerical software,” Parallel Computing, Vol. 32, Issue 1, pp.92-112, 2006.
12) T. Katagiri, S. Ohshima, M. Matsumoto, “Directive-based auto-tuning for the finite difference method on the Xeon Phi,” Proceedings of IPDPSW2015, pp. 1221–1230, 2015.
13) T. Katagiri, M. Matsumoto, S. Ohshima, “Auto-tuning of Hybrid MPI/OpenMP Execution with Code Selection by ppOpen-AT,” Proceedings of IPDPSW2016, pp. 1488–1495, 2016.
14) T. Katagiri, S. Ohshima, M. Matsumoto, “Auto-tuning on NUMA and Many-core Environments with an FDM Code,” Proceedings of IPDPSW2017, 2017.