Vol.2020-HPC-177 No.21
2020/12/22

FP21 及び FP41 を使用した不完全コレスキー分解前処理

河合 直聡[1,a)]　中島 研吾[1,2,b)]

概要：科学技術計算では倍精度浮動小数点演算（FP64）が広く使用されてきたが，近年，計算量・メモリアクセス量・消費電力削減の観点から，単精度（FP32），半精度（FP16）などの低精度演算の適用が盛んに実施されるようになっている。FP32 によって広範囲な問題を解けることが示されているが，FP16 は有効桁が 3 桁程度であり，用途は限定されている。山口等によって提案された FP21 は FP32 と FP16 の中間であり，GPU 向けに実装され，地震シミュレーションにおいてはその有効性が示されている。本研究では，この FP21 を汎用 CPU 向けに実装し、構造解析アプリケーションを解くための不完全コレスキー分解前処理に適用、その効果を検証した。構造解析のアプリケーションはポアソン比などの変更で問題の条件数が変わるため、より実用的に FP21 の効果を評価可能である。また、先述のような理由から FP21、32、64 の適用によって収束性が変化するため、FP21 に加えて FP42 の実装も行い、同時に評価を行った。

Incomplete Cholesky Preconditioner with FP21 and FP42

Masatoshi Kawai[1,a)]　Kengo Nakajima[1,2,b)]

1. はじめに

近年の科学技術分野では FP64 以外に FP32 や FP16 の利用が再検討され、計算時間短縮、消費電力削減を目的として複数の精度を混在させる研究が行われている。現在の数値シミュレーションなど分野では暗黙的に倍精度浮動小数点演算（FP64）が利用されているが、深層学習の分野を中心に単精度 (FP32) や半精度 (FP16) の使用が広く議論、実用化されている [1], [2]。実際に、近年の GPU では半精度演算がサポートされ、また 2020 年春から試験稼働を開始している富岳で採用されている CPU(ARM64Fx) でも同様にサポートされている。加えて、第 3 世代の Intel Xeon Scalable processor でも FP16 のサポートがアナウンスされている。このような背景を受け、計算科学の分野でも単精度や半精度の利用が研究されており、FP32 がアプリケーションによっては実用的な精度であると報告されている [3], [4], [5]。一方で、FP16 は有効桁数が 3 桁程度と少なく、指数部の最大値も 10^5 と小さいために、用途が限定的である。そこで、FP16 と FP32 の間の精度を持つ FP21 が山口等によって提案、GPU 向けに実装され、地震シミュレーションにおいてその有効性が示されている [6]。

数値シミュレーションで広く用いられているクリロフ部分空間法は FP64 の利用が一般的と考えられている典型的な例である。これは、クリロフ部分空間法は収束性が計算精度に強く影響を受けるためである。しかし、クリロフ部分空間法で併用される前処理部分に着目すると、ここでは簡易的に対象の方程式が解ければよいという性質から、低精度でもある程度の効果が得られることが分かってきている。特に、不完全コレスキー分解前処理は、簡易的なコレスキー分解であるため、低い精度でも十分な収束性改善が確認されており、実際に計算時間の短縮に寄与している [7]。

本研究ではこれまでに提案された FP21 を GeoFEM ベンチマーク [8], [9] の不完全コレスキー分解前処理に適用し、FP32 と比較した。加えて、FP32 と 64 の間である FP42 を新たに実装し、同様に評価を行った。不完全コレスキー分解は前処理付きクリロフ部分空間法の計算時間の半分以上を締めているため、この前処理への FP21 の適用によりさらなる時間短縮が期待できる。ただし、問題の条件数と

[1] 東京大学 情報基盤センター
ITC, The Uiveristy of Tokyo
[2] 理化学研究所 計算科学研究センター
R-CCS, RIKEN
[a)] kawai@cc.u-tokyo.ac.jp
[b)] nakajima@cc.u-tokyo.ac.jp

ⓒ 2020 Information Processing Society of Japan

前処理の低精度化によっては収束性が大きく悪化し、FP21 の優位性がなくなる場合あがる。よって、本研究では計算時間と収束性に着目し、FP21 の優位性を確認した。同様に、収束性の観点から FP32 よりも FP64 のほうが優位な結果も想定でき、この場合には FP42 の優位性が出てくる可能性がある。よって、新たに FP42 提案、実装も行い評価を行った。

2. FP21、FP42 型浮動小数

本節では FP21 および FP42 の格納形式、表現能力を既存のデータ型である FP16、FP32、FP64 との比較しつつ述べる。また、FP21、42 はコンパイラー、CPU のサポートがないため、使用のためには型キャストが必須となる。本節ではこれを実現する実装についても述べる。

2.1 FP21、FP42 の概要

本節では FP21 および FP42 の形式および表現能力について述べる。

計算機で扱う浮動小数表現はいくつか存在し、2008 年に改定された IEEE754[10] では半精度 (FP16)、単精度 (FP32)、倍精度 (FP64)、四倍精度 (FP128) が基本形式として定められている。いずれの形式でも、符号部、指数部、仮数部をもっており、それぞれで指数部と仮数部の長さが異なる。図 1 に FP16、FP32、FP64 の符号部、指数部、仮数部の長さを示す。高精度な形式ほど長い指数部および仮数部を持っている。

FP21 は、計算時の精度が FP16 では不足であるが、FP32 は十分であるというアプリケーションにおいて、データ転送量削減を目的に提案されている。FP21 の指数部は FP32 と同じであるが、仮数部が 23bit から 12bit に削減されている (図 1)。結果、データ転送量はおおよそ 2/3 に削減されるため、メモリ律速なアプリケーションで、計算時間短縮が期待できる。本稿ではこの FP21 の CPU 上での実用性を評価する。

本稿では FP32 と FP64 の間の表現をもつ FP42 に関しても提案、実装、評価を行う。FP42 は FP64 の仮数部を 52bit から 30bit に削減している。FP21 と同様に FP32 では表現力が足りないが、FP64 では十分なアプリケーションで 2/3 へのデータ転送量削減が期待できる。

表 1 には各精度の表現能力を示す指標として、仮数部の 10 進数での表現可能桁数と指数部の最大冪指数を示す。仮

表 1　各浮動小数表現の表現力

データ型	仮数部：表現可能桁数	指数部：最大冪指数
FP16	3.31	5
FP21	3.91	38
FP32	7.22	38
FP42	9.33	308
FP64	15.95	308

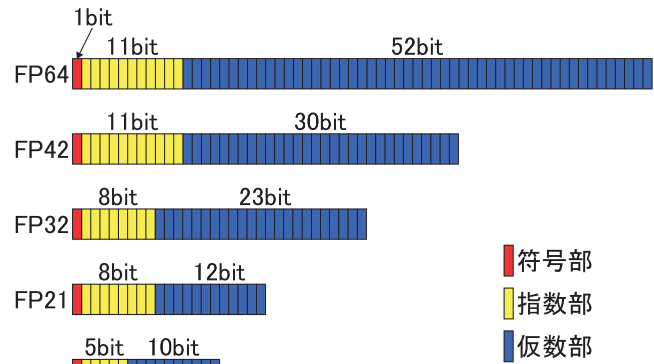

図 1　FP64,42,32,21,16 の比較

数部の 10 進数での表現可能桁数 y は、各精度の仮数部のビット数を x として、

$$10^y = 2^{(x+1)} \tag{1}$$

$$y = (x+1)\log_{10} 2 \tag{2}$$

のように算出している。なお、式 (1) の右辺、2 進数での表現可能桁数を示す冪指数を $x+1$ としているのは、浮動小数表現での仮数部が全て 0 で、かつ指数部がすべて 0 でない場合、仮数部の最上位ビットのさらに上に 1 が付与されているとして (Hidden Bit)、浮動小数形式を扱うためである。本表から、10 進数での表現能力を基準として考えた場合、FP21 と FP16 の仮数部の表現能力は同程度であり、そこから高精度な形式になるほど表現能力が高くなっている。

2.2 先行研究

複数の浮動小数表現を使った計算手法は深層学習の分野で広く用いられている [11], [12]。また、いくつかのシミュレーションでもすでに研究されており [13], [14]、複数の報告がある。このような背景を受け、近年では反復法での研究も行われるようになっている [15], [16]。

ただし、アプリケーションによっては FP16 や FP32、FP64 の間がちょうどよい場合があり、そのような状況に対して、IEEE754 で定義されたデータ型以外の研究が行われるようになっている。代表的な物は、Google の深層学習ライブラリで実際に使用されている bfloat[17] などがあるが、山口等による論文 [6] では FP21 が提案されており、これを適用した地震シミュレーションが GPU で評価されている。同論文内ではベクトルの AXPY、内積および行列ベクトル積が FP21 で実装されており、全体で 10%程度の性能向上が確認されている。

これらの先行研究に対して、本稿では、FP21 の CPU 上での効果的な実装の提案、評価を行っている。加えて、新たなデータ型である FP42 の提案も行っている。FP21 は CPU およびコンパイラによるサポートがないため、計算時にはより高精度な FP32 への型キャストが必要となる。

ⓒ 2020 Information Processing Society of Japan

これらの型キャストは頻繁に必要となるため、型キャストによるオーバーヘッドを小さくしなければならない。山口等の論文 [6] では C 言語を用いて実装されており、inline 指示文の挿入でオーバーヘッドの最小化が図られている。本稿での評価には、Fortran で記述されたアプリケーションを想定しており、C 言語での実装以上に注意しなければならず、これらの点を考慮して実装の提案、およびその評価を行っている。加えて、IC 前処理付き CG 法では FP32 の表現力でも足りない場合が想定されるため、新たなデータ型として FP32 と 64 の間の精度をもつ FP42 の提案、評価も同様に行っている。

2.3 FP21、FP42 の型変換

本節では FP21 と FP32 データ型間、および FP42 と FP64 データ型間の型キャストの実装について述べる。

山口等によって提案された FP21 の型キャストの実装では、FP21 の 3 つを C 言語の unsigned long int 1 つに格納する手法を取っている。本稿で対象とするアプリケーションは Fortran であるため、同様の実装を Fortran で行った。ただし、Fortran は unsigned 型の扱いがないため、unsigned long int 型を integer(8) 型に置き換えている。サンプルコード 1 に FP21 と FP32 間での型キャスト関数を示す。本サンプルコードでの “fp32x3_to_fp21x3_f” は FP32 から FP21 への、“fp21x3_to_fp32x3_f” は FP21 から FP32 への型キャスト関数である。FP32 から FP21 へ型キャストは、以下の手順で行う。なお、FP21 から FP32 への変換は逆の操作を行う。

(1) “cast_fp32_to_fp21x3” 関数を使用して、“FP32(real(4))“型のデータを”fp21x3(integer(8))“型に変換（型キャストではなく、単純なデータ型の変換）
(2) 変換後のデータから 21bit 分の値を論理積を使って抽出
(3) shiftr または shiftl 関数を使用して、論理シフトを適用 (shiftr および shiftl 関数は Fortran2008 準拠)
(4) 論理和を使用して対象の変数に代入 (もともとのデータを保持するために論理和を使用)

ここで、“cast_fp32_to_fp21x3” 関数は、FP32 型のデータを fp21x3 型に内部的なビットの情報を変更することなく変換するための関数である。一般的な型キャストは数字的な意味が変わらないように内部的なビット情報が更新される。FP21 と FP32 間の型キャストでは、この更新を行わずに、言語上で扱う型のみを変更する必要があるため、このような関数を用いている。C 言語ではポインタのやり取りでこれを実現できるが、Fortran では同じことができないため、“cast_fp32_to_fp21x3” 関数では real(4) 型のデータを暗黙的に integer(8) 型のデータとして受け取ることで、目的の動作を実現している。

なお、山口等の実装したコードは github にて公開されており [18]、実装の一案として、公開されたコードを Fortran で呼び出すための interface を作成する方法がある。ただし、ここでの型変換は頻繁に呼び出されるため、レイテンシが重要となる。もしプログラムのコンパイル時に型キャスト関数がインライン展開される実装ができていれば、レイテンシの最小化が期待できる。インライン展開されることでコンパイル時や実行時のアウト・オブ・オーダーエンジンによる命令の並び替えが行われ、型キャストがそれ以外の計算やメモリのロードストアでオーバーラップされる。実際に、山口等によって公開されているコードでは inline 指示文が挿入されている。一方で、Fortran の intarface を用いて C 言語を呼び出す方法ではインライン展開が行わ

サンプルコード 1 FP32 → FP21、FP21 → 32 型キャスト関数

```
#define fp21x3 integer(8)

function fp32x3_to_fp21x3_f(a1, a2, a3) result(b)
  implicit none
  real(4), intent(in) :: a1, a2, a3
  fp21x3 :: b
  fp21x3 c

  call cast_fp32_to_fp21x3(a1, c)
  b = shiftr(and(c, int(Z'00000000ffff800', 8)), 11)
  call cast_fp32_to_fp21x3(a2, c)
  b = or(b, shiftl(and(c, int(Z'00000000ffff800', 8)), 10))
  call cast_fp32_to_fp21x3(a3, c)
  b = or(b, shiftl(and(c, int(Z'00000000ffff800', 8)), 31))

end function fp32x3_to_fp21x3_f

subroutine fp21x3_to_fp32x3_f(a, b1, b2, b3)
  implicit none
  fp21x3, intent(in) :: a
  real(4), intent(out) :: b1, b2, b3

  call cast_fp21x3_to_fp32(shiftl(and(a,                    &
                  int(Z'00000000001ffff', 8)), 11), b1)
  call cast_fp21x3_to_fp32(shiftr(and(a,                    &
                  int(Z'000003fffe00000', 8)), 10), b2)
  call cast_fp21x3_to_fp32(shiftr(and(a,                    &
                  int(Z'7ffffc0000000000', 8)), 31), b3)

end subroutine fp21x3_to_fp32x3_f

subroutine cast_fp32_to_fp21x3(a, b)
  implicit none
  fp21x3, intent(in)  :: a
  fp21x3, intent(out) :: b

  b = a

end subroutine cast_fp32_to_fp21x3

subroutine cast_fp21x3_to_fp32(a, b)
  implicit none
  real(4), intent(in)  :: a
  real(4), intent(out) :: b

  b = a

end subroutine cast_fp21x3_to_fp32
```

サンプルコード **2**　FP64 → FP42、FP42 → 64 型キャスト関数

```
#define fp42x3 integer(8)

function fp64x3_to_fp42x3_f(a1, a2, a3) result(b)
  implicit none
  real(8), intent(in) :: a1, a2, a3
  fp42x3 :: b(2)
  fp42x3 c

  call cast_fp64_to_fp42x3(a1, c)
  b(1) = shiftr(and(c, int(Z'ffffffffffc00000', 8)), 22)
  call cast_fp64_to_fp42x3(a2, c)
  b(1) = or(b(1), shiftl(and(c,                                  &
                         int(Z'00000fffffc00000', 8)), 20))
  b(2) = shiftr(and(c, int(Z'fffff00000000000', 8)), 44)
  call cast_fp64_to_fp42x3(a3, c)
  b(2) = or(b(2), and(c, int(Z'ffffffffffc00000', 8)))

end function fp64x3_to_fp42x3_f

subroutine fp42x3_to_fp64x3_f(a, b1, b2, b3)
  implicit none
  fp42x3, intent(in) :: a(2)
  real(8), intent(out) :: b1, b2, b3
  fp42x3 right, left

  call cast_fp42x3_to_fp64(shiftl(and(a(1),                       &
                         int(Z'000003ffffffffff', 8)), 22), b1)
  right = shiftr(and(a(1), int(Z'fffffc0000000000', 8)), 20)
  left  = shiftl(and(a(2), int(Z'00000000000fffff', 8)), 44)
  call cast_fp42x3_to_fp64(or(left, right), b2)
  call cast_fp42x3_to_fp64(and(a(2),                              &
                         int(Z'ffffffffffc00000', 8)), b3)

end subroutine fp42x3_to_fp64x3_f

subroutine cast_fp64_to_fp42x3(a, b)
  implicit none
  fp42x3, intent(in)  :: a
  fp42x3, intent(out) :: b

  b = a

end subroutine cast_fp64_to_fp42x3

subroutine cast_fp42x3_to_fp64(a, b)
  use iso_c_binding
  implicit none
  real(8), intent(in)  :: a
  real(8), intent(out) :: b

  b = a

end subroutine cast_fp42x3_to_fp64
```

れないため、型キャストによるオーバーヘッドが大きくなる可能性がある。よって本研究では同様の動作を Fortran で実装することとした。なお、Fortran には言語で標準化されたインライン展開を促す指示文などは存在しないが、インライン展開は一般的な最適化手法であり、必要な最適化オプションを付与すれば、サンプルコード 1 で示した程度の規模であれば、ほとんどのコンパイラはインライン展開を行う。著者らが確認したところ、GNU コンパイラのバージョン 9.3.1 で”-O3 -flto” オプションをつけた場合と、Intel コンパイラのバージョン 19.1.1.217 で”-O3 -ipo” オプションをつけた場合で、型キャスト関数とそれを参照するプログラムが分割コンパイルされても、インライン展開がされていることを確認した。なお、インライン展開が行われたかどうかは各コンパイラの最適化レポート (GNU では-fopt-info-inline、Intel では-qopt-report) で確認している。加えて、Intel コンパイラではインライン展開を促すディレクティブとして、
”!DIR$ ATTRIBUTE FORCEINLINE :: *function*“
が用意されており、これを呼び出し元に挿入すれば、インライン展開が明示的に行われる。

次に、FP64 から FP42、FP42 から FP64 への変換について述べる。FP21 の実装を FP42 にそのまま拡張するためには、integer(16) がコンパイラでサポートされている必要がある。しかし、Intel コンパイラのバージョン 19.1.1.217(2020 update2) ではサポートされていなかかった (GNU コンパイラのバージョン 9.3.1 ではサポート)。そこで、FP42 型のデータ 3 つを integer(8) 型のデータ 2 つで格納する形式を採用している。サンプルコード 2 に FP42 と FP64 間での型キャスト関数を示す。

前述のようにここでの型キャストのオーバーヘッドは性能に大きく影響する。本稿の数値実験による評価ではこのオーバーヘッドの計測も同時に行う。

3. 評価対象のアプリケーション

本研究では FP21,FP42 の効果を検証するために、GeoFEM ベンチマークの不完全コレスキー分解にこれらの精度を適用し、評価を行う。GeoFEM ベンチマークでは構造解析から導出される連立一次方程式を不完全コレスキー分解前処理つき共役勾配 (ICCG) 法で解いている。構造解析の問題はその性質から FP21、FP42 でデータを扱うのに向いている。

3.1 構造解析問題

本節では GeoFEM ベンチマークで扱っている連立一次方程式が構造解析の問題から導出されるまでを扱う。

本稿で扱う問題は 3 次元の構造解析問題であり、ある荷重を物体にかけた場合にその物体の変形 (変位) を平衡方程式を用いて表す。3 次元の問題では、各接点毎に x、y、z 方向の変位 3 つを持つ。FP21 および FP42 の型キャストでは、データ 3 つを 1 つの単位として扱うため、評価する上で相性のよいアプリケーションである。

平衡方程式では、物体の変位とそこにかかる荷重の関係が剛性行列を用いて表現される。ここで使用するメッシュを 6 面体のソリッド要素とし、ある要素の接点にかかる力 $\boldsymbol{f}$ は、同要素の全て接点の変位 $\boldsymbol{u}$ と、弾性行列 $\boldsymbol{E}$、変位ー歪の関係を示す行列 $\boldsymbol{B}$ および要素の体積 V を用いて以下のように表される。

$$\boldsymbol{f} = \int_V \boldsymbol{B}^T \boldsymbol{E}\boldsymbol{B} dV\, \boldsymbol{u} \tag{3}$$

このときの、$\int_V \boldsymbol{B}^T \boldsymbol{E}\boldsymbol{B} dV$ が剛性行列 $\boldsymbol{K}$ と呼ばれる。モデル全体での関係式は、式 (3) の全要素文を加算した形となる。行列 $\boldsymbol{B}$ はソリッド要素の形によって決まり、行列 $\boldsymbol{E}$ はヤング率とポアソン比によって決まる。ここでの $\boldsymbol{E}$ は、ヤング率を E、ポアソン比を ν として以下のように表される。

$$\boldsymbol{E} = \gamma \begin{pmatrix} 1 & \alpha & \alpha & 0 & 0 & 0 \\ \alpha & 1 & \alpha & 0 & 0 & 0 \\ \alpha & \alpha & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \beta & 0 & 0 \\ 0 & 0 & 0 & 0 & \beta & 0 \\ 0 & 0 & 0 & 0 & 0 & \beta \end{pmatrix} \tag{4}$$

$$\because \alpha = \frac{1}{1-\nu},\ \beta = \frac{1-2\nu}{2(1-\nu)},\ \gamma = \frac{E(1-\nu)}{(1-2\nu)(1+\nu)}$$

なお、ポアソン比は $-1 < \nu < 0.5$ である。

3.2 不完全コレスキー分解前処理つき共役勾配法

本稿では、GeoFEM ベンチマークのソルバ部について述べる。GeoFEM ベンチマークのソルバ部では、ICCG 法を採用しており、構造解析の問題から導出された連立一次方程式

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b} \tag{5}$$

を解いている。なお、$\boldsymbol{A}$ は剛性行列 $\boldsymbol{K}$ を全要素で加算し、算出された全体剛性行列 (係数行列) であり、解ベクトル $\boldsymbol{x}$ および右辺ベクトル $\boldsymbol{b}$ は全接点での変位 $\boldsymbol{u}$ および荷重 $\boldsymbol{f}$ を並べたベクトルである。対象の問題の接点数を N とすると、解くべき方程式??e:SLE) の未知変数は $3 \times N$ となる。

図 2 に ICCG 法の計算手順を示す。本図での k は反復回数、$\boldsymbol{x}^k$ は k 反復目の近似解ベクトル、$\boldsymbol{r}^k$ は残差ベクトル、$\boldsymbol{p}^k$ は探索ベクトル、$\boldsymbol{q}$ は前処理の結果を格納するベクトル、$IC()$ は不完全コレスキー分解 (IC) を示す。

本研究で実装した IC 前処理は、Additive Schwartz 型であり、メモリ分散環境などでも比較的実装しやすい。IC 前処理前処理では、反復前に係数行列 $\boldsymbol{A}$ から対角行列 $\overline{\boldsymbol{D}}$ および上三角行列 $\overline{\boldsymbol{U}}$ を以下のように計算する。

$$\overline{d_{i,i}} = a_{i,i} - \sum_{k=1}^{i-1} \overline{u_{k,i}}\ \overline{d_{i,i}}\ \overline{u_{k,i}} \tag{6}$$

$$\overline{u_{i,j}} = \begin{cases} \frac{1}{d_{i,i}} \left(a_{i,j} - \sum_{k=1}^{i-1} \overline{u_{k,i}}\ \overline{d_{i,i}}\ \overline{u_{k,j}} \right), & a_{i,j} \neq 0 \\ 0, & a_{i,j} = 0 \end{cases} \tag{7}$$

これを用いて、Additive Schwartz 型の IC 前処理では以下のように $\boldsymbol{r}$ から $\boldsymbol{q}$ を計算する。

$$\boldsymbol{q} = \left(\boldsymbol{U}^T\right)^{-1} \boldsymbol{r}$$
$$\boldsymbol{q} = \boldsymbol{U}^{-1} \boldsymbol{q}$$
通信による $\boldsymbol{q}$ の更新
$$\boldsymbol{q}' = \left(\boldsymbol{U}^T\right)^{-1} \boldsymbol{q}$$
$$\boldsymbol{q}' = \boldsymbol{U}^{-1} \boldsymbol{q}'$$
$$\boldsymbol{q}' = \left(\boldsymbol{U}^T\right)^{-1} \boldsymbol{q}'$$
$$\boldsymbol{q}' = \boldsymbol{U}^{-1} \boldsymbol{q}'$$
$$\boldsymbol{q} = \boldsymbol{q} + \boldsymbol{q}' \tag{8}$$

次に、ICCG 法への FP21 および FP42 を含めた低精度演算の適用範囲について述べる。式 (4) に示すように、構造解析の問題では、係数 γ が

$$\lim_{\nu \to 0.5} \gamma = \lim_{\nu \to 0.5} \frac{E(1-\nu)}{(1-2\nu)(1+\nu)} = \inf \tag{9}$$

という性質を持つため、ポアソン比によっては係数行列 $\boldsymbol{A}$ の精度が重要になり、$\boldsymbol{A}$ を低精度で格納する場合は導かれた近似解が正しいかの検証が必要になる。また、近似解ベクトル $\boldsymbol{x}^k$ や残差ベクトル $\boldsymbol{r}^k$、探索ベクトル $\boldsymbol{p}^k$ を低精度化した場合、内積の精度が落ちるため、CG 法そのものが破綻する可能性がある。一方で、IC 前処理は低精度の適用によって収束性悪化の可能性はあるが、式 8 に示すようにその計算量は CG 法内の行列ベクトル積の 3 倍以上に相当し、ICCG 法全体の大半を占めるため、1 反復辺りの計算時間短縮の効果が大きく、収束性悪化の影響を上回ることが期待できる。よって本研究では IC 前処理に低精度演算を適用することを考える。

IC 前処理へ低精度演算を適用する場合、行列 $\overline{\boldsymbol{D}}$ および $\overline{\boldsymbol{U}}$ のみに適用する場合と、ベクトル $\boldsymbol{q}$ および $\boldsymbol{q}'$ にも適用する場合が考えられる。行列へのアクセスに要するデータ転送料はベクトルへのアクセスよりも 3 倍程度あり、行列だけの低精度化でもその効果は十分に得られると期待できる。ベクトルも含めて適用する場合、1 反復あたりのさらなる計算時間短縮が期待できるが、前進後退代入による演算精度低下が蓄積されるため、行列の低精度化以上に前処

```
do  k = 1,  !until converge
    α = (r^k, p^k) / (p^k, Ap^k)
    x^{k+1} = x^k + αp^k
    r^{k+1} = r^k − αAp^k
    q = IC(r^{k+1})        !不完全コレスキー分解前処理
    β = −(q, Ap^k) / (p^k, Ap^k)
    p^{k+1} = q + βp^k
enddo
```

図 2 不完全コレスキー分解前処理つき共役勾配法の計算手順

ⓒ 2020 Information Processing Society of Japan

表 2 Oakbridge-CX の仕様

CPU	モデル名	Xeon Gold Platinum 8280 (Cascade Lake)
	コア数	56 (2 ソケット)
	動作クロック	2.7GHz
	キャッシュサイズ	38.5Mbyte/Socket
Memory	規格	DDR4
	サイズ	192Gbyte

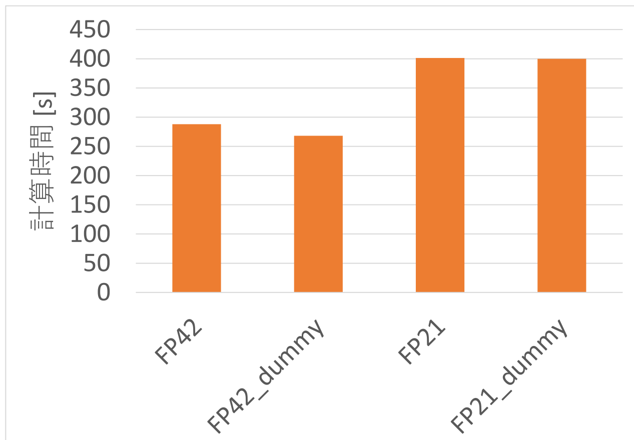

図 3 FP21 および FP42 の型キャストによるオーバーヘッド

理の効果が低下する可能性が発生する。本研究では、前処理行列のみに低精度演算を適用してこれを評価する。

4. 評価

本節では FP21 および FP42 の効果を FP32、FP64 と比較して示す。

4.1 評価環境

評価に使用した構造解析問題では構造格子で離散化されたモデルを使用し、格子点数は 256^3 とした。このときに導出される連立一次方程式の自由度は 3^3 でおおよそ 5 千万となっている。必要なメモリ量はベクトル 1 つ辺りで 400MByte 以上が必要となり、計算がメモリ律速となる条件である。評価では、ポアソン比 ν を 0.30〜0.49 まで変化させ、FP21 や FP32、FP42、FP64 の使用による収束性の変化も含めて確認する。なお、係数行列の格納形式には CRS 形式を用いている。

評価で使用した計算機は東京大学情報基盤センターがサービスを行っている Oakbridge-CX2 である。ここでは 1 ノードを使用し、1 ノードあたり 4 プロセス、プロセスあたり 14 スレッドとした。コンパイラは Intel コンパイラのバージョン 19.1.1.217 を使用し、コンパイルオプションは"-O3 -xHost -qopenmp -ipo" を付与した。

4.2 評価結果

はじめに、FP21-FP32 間および FP42-FP64 間の型キャストのオーバーヘッドの計測結果を示す。

計測にあたって、ソースコード内の型キャスト部を削除し、FP21 および FP42 を使用した場合と同じメモリ転送量になるように、単純な FP32 または FP64 のデータ参照に置き換えたソースコードを新たに用意した。この際、計算結果は型キャストの削除前と同じにならず、ICCG 法は収束しなくなるが、反復回数の上限を正しい結果と同じになるようにしている。これらの条件での比較により、型キャストを行う場合と行わない場合の計算時間の差が算出され

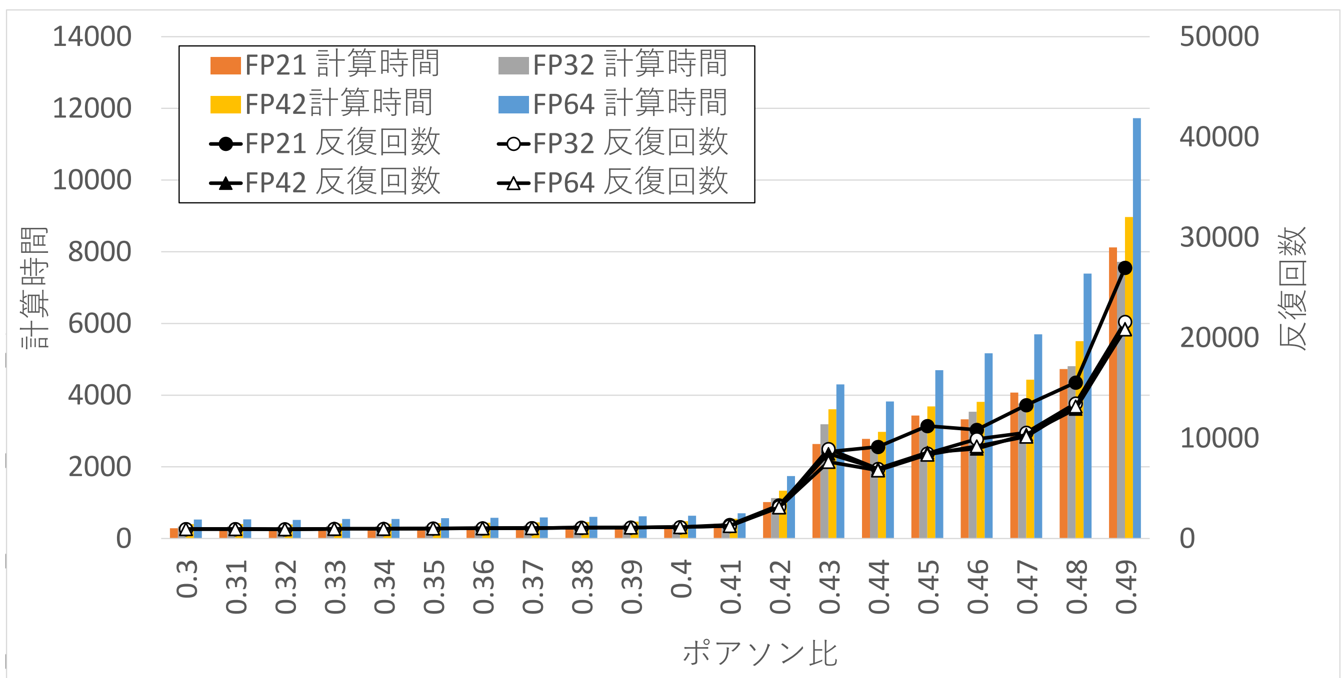

図 4 各ポアソンに対して FP21,32,42,64 を使用した場合の計算時間および反復回数

ⓒ 2020 Information Processing Society of Japan

る。図 3 に FP21 または FP42 を使用した場合と、型キャストを取り除いた場合 (FP21_dummy、FP42_dummy) での前処理部のみの計算時間を示す。本図の結果から FP21 での型キャストのオーバーヘッドは 6.9%、FP42 でのオーバーヘッドは 0.4%と算出される。FP21 と FP42 の型キャストによるオーバーヘッドは同程度が妥当な結果であるが、ここでの計測結果では FP21 の型キャストによるオーバーヘッドが FP42 の 10 倍以上という結果になった。これは、行列の格納形式が CRS であり、FP21 では十分にメモリバンド幅を使い切れておらず、メモリ転送で型キャストによるオーバーヘッドを隠せていないためである。よって、本評価での型キャストによるオーバーヘッドは FP42 での結果である 0.4%と結論付ける。このレイテンシは十分に小さく、実用に耐えうる値である。

図 4 にポアソン比を変更した場合の各精度での計算時間および反復回数を示す。図 5 はポアソン比 0.30〜0.41 の範囲を拡大したものである。これらの図の棒グラフは計算時間を、折れ線グラフは反復回数を示す。これらの図から、いずれの精度のデータ形式でもポアソン比の増大とともに収束性の悪化が確認できるが、いずれのポアソン比でも FP64 より低精度なデータ形式で十分な収束性が確認でき、より短い計算時間で近似解を得られている。また、ポアソン比が 0.43 以下の範囲では精度毎の収束性の差が小さく、FP21 が最も効果的であった。一方で、ポアソン比が 0.44 以上の条件では、FP21 での収束性の悪化が目立つ場合が多く、6 つのポアソン比のうち 4 つで FP32 のほうが計算時間が短くなった。本結果では FP21 または FP32 で十分であるという結果を得たが、FP42 の FP64 に対する優位性も十分に確認できており、非構造格子での問題など、FP32 でも収束性が悪化するような問題では FP42 の優位性がある。具体的な数値では、ポアソン比 0.3〜0.43 の範囲で、FP21 の使用により FP32 を使用した場合に対して平均で 14.2%、FP64 に対して 44.8%の計算時間短縮を、FP42 は FP64 に対して 23.2%の計算時間短縮を達成した。

5. まとめ

本稿では GPU で既に効果が確認されている FP21 および実用性が期待できる FP42 を CPU 用に実装、評価を行った。これらの型はコンパイラおよび CPU の演算器によるサポートがないため、型キャストが必要となるが、本稿で提示した Fortran による型キャストの実装は十分にオーバーヘッド小さいことを数値実験の結果から確認した。

これらの FP21 および FP42 を GeoFEM ベンチマークの不完全コレスキー分解前処理に適用し、ポアソン比を変更してその収束性および計算時間の評価を行った。結果、FP21 および FP42 の使用による収束性の悪化は十分に小さく、実用的であることを確認した。具体的にはポアソン

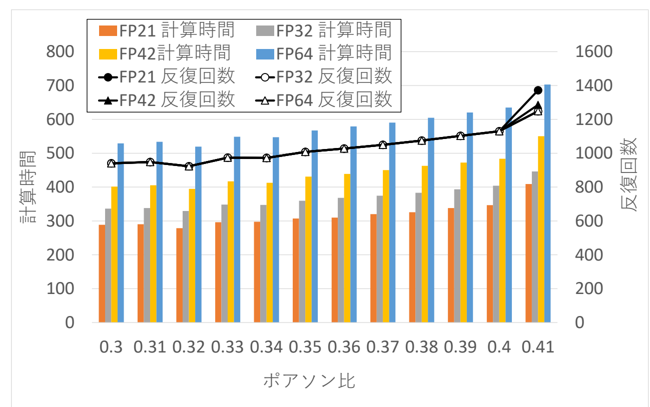

図 5 図 4 のポアソン比 0.3〜0.41 の範囲の拡大図

比が 0.30〜0.43 の範囲で FP21 の使用により FP32 を使用した場合に対して平均で 14.2%、FP64 に対して 44.8%の計算時間短縮を、FP42 は FP64 に対して 23.2%の計算時間短縮を達成した。

今後、より悪条件な非構造問題での評価、構造解析問題以外への適用の検討や、アプリケーション毎に適した精度を自動的に選ぶためのオートチューニング手法の開発などを行っていく予定である。

ⓒ 2020 Information Processing Society of Japan

謝辞 本研究の遂行に関しては、本研究は JSPS 科研費 JP12345678 の助成および、学際大規模情報基盤共同利用・共同研究拠点、および、革新的ハイパフォーマンス・コンピューティング・インフラからの計算資源の支援を受けている (課題番号: jh200037-NAH)。この場を借りて感謝の意を表する。

参考文献

[1] Haidar, A., Tomov, S., Dongarra, J. and Higham, N. J.: Harnessing GPU Tensor Cores for Fast FP16 Arithmetic to Speed up Mixed-Precision Iterative Refinement Solvers, *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 603–613 (2018).

[2] Lee, J., Lee, J., Han, D., Lee, J., Park, G. and Yoo, H.: 7.7 LNPU: A 25.3TFLOPS/W Sparse Deep-Neural-Network Learning Processor with Fine-Grained Mixed Precision of FP8-FP16, *2019 IEEE International Solid- State Circuits Conference - (ISSCC)*, pp. 142–144 (2019).

[3] Matsuoka, N., Qiu, L., Li, X., Omori, T. and ya Hashimoto, K.: Applicability of single precision graphics processing unit for fast simulation of 2D surface acoustic wave devices using an hierarchical cascading technique, *Japanese Journal of Applied Physics*, Vol. 58, No. SG, p. SGGC11 (2019).

[4] Fujita, K., Horikoshi, M., Ichimura, T., Meadows, L., Nakajima, K., Hori, M. and Maddegedara, L.: Development of element-by-element kernel algorithms in unstructured finite-element solvers for many-core wide-SIMD CPUs: Application to earthquake simulation, *Journal of Computational Science*, Vol. 45, p. 101174 (2020).

[5] Hess, B.: P-LINCS : A Parallel Linear Constraint Solver for Molecular Simulation, *Journal of Chemical Theory and Computation*, Vol. 4, No. 1, pp. 116–122 (2008).

[6] Yamaguchi, T., Fujita, K., Ichimura, T., Naruse, A., Lalith, M. and Hori, M.: GPU implementation of a sophisticated implicit low-order finite element solver with FP21-32-64 computation using OpenACC, *Lecture Notes in Computer Science, WACCPD 2019*, Vol. 12017 (2019).

[7] Sakamoto, R., Kondo, M., Fujita, K., Ichimura, T. and Nakajima, K.: The Effectiveness of Low-Precision Floating Arithmetic on Numerical Codes: A Case Study on Power Consumption, HPCAsia2020, p. 199–206 (2020).

[8] Nakajima, K.: Parallel Iterative Solvers of GeoFEM with Selective Blocking Preconditioning for Nonlinear Contact Problems on the Earth Simulator, *SC '03: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, pp. 13–13 (2003).

[9] 研吾中島，孝洋片桐：マルチコアプロセッサにおけるリオーダリング付き非構造格子向け前処理付反復法の性能，技術報告 6，東京大学情報基盤センター／科学技術振興機構戦略的創造研究推進事業（CREST），東京大学情報基盤センター (2009).

[10] : IEEE Standard for Floating-Point Arithmetic, *IEEE Std 754-2008*, pp. 1–70 (2008).

[11] Nandakumar, S. R., Le Gallo, M., Piveteau, C., Joshi, V., Mariani, G., Boybat, I., Karunaratne, G., Khaddam-Aljameh, R., Egger, U., Petropoulos, A., Antonakopoulos, T., Rajendran, B., Sebastian, A. and Eleftheriou, E.: Mixed-Precision Deep Learning Based on Computational Memory, *Frontiers in Neuroscience*, Vol. 14, p. 406 (2020).

[12] Jiang, W., Song, Z., Zhan, J., He, Z., Wen, X. and Jiang, K.: Optimized co-scheduling of mixed-precision neural network accelerator for real-time multitasking applications, *Journal of Systems Architecture*, Vol. 110, p. 101775 (2020).

[13] Clark, M., Babich, R., Barros, K., Brower, R. and Rebbi, C.: Solving lattice QCD systems of equations using mixed precision solvers on GPUs, *Computer Physics Communications*, Vol. 181, No. 9, pp. 1517 – 1528 (2010).

[14] Le Grand, S., Götz, A. W. and Walker, R. C.: SPFP: Speed without compromise—A mixed precision model for GPU accelerated molecular dynamics simulations, *Computer Physics Communications*, Vol. 184, No. 2, pp. 374 – 380 (2013).

[15] Walden, A., Nielsen, E., Diskin, B. and Zubair, M.: A Mixed Precision Multicolor Point-Implicit Solver for Unstructured Grids on GPUs, *2019 IEEE/ACM 9th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, pp. 23–30 (2019).

[16] Ooi, R., Iwashita, T., Fukaya, T., Ida, A. and Yokota, R.: Effect of Mixed Precision Computing on H-Matrix Vector Multiplication in BEM Analysis, *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, HPCAsia2020, New York, NY, USA, Association for Computing Machinery, p. 92–101 (2020).

[17] : TensorFlow bfloat, `https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/framework/bfloat16.h`.

[18] Yamaguchi, T.: FP21AXPY, `https://github.com/y-mag-chi/fp21axpy`.