

ppOpen-AT における演算精度と消費電力を考慮した 自動チューニング方式の提案

山梨祥平¹ 八代尚² 片桐孝洋³ 永井亨³ 大島聡史³

概要: ポストムーア時代に向けて、低精度演算を部分的に利用することで計算時間や消費電力の削減を目指す混合精度演算の研究が盛んに行われている。しかし、低精度演算の性能はアーキテクチャや問題規模などによって影響を受けるため、混合精度演算を活用時の最適化コストは膨大になると予想される。そこで本稿では、自動チューニング言語 ppOpen-AT の新機能として、プログラム上の演算精度を部分的に変化させることで演算精度と消費電力を考慮した最適実装を探索する自動チューニング方式を提案する。また全球雲解像度大気モデル NICAM に対して実験を行った。実験から以下の結果を得た：①提案する変数／配列方式では、精度を 1.E-13 に維持しつつ 1.04 倍の高速化と 1.06 倍の電力削減ができる；②提案するブロック方式では、精度を 3.E-04 まで許容するならば 1.12 倍の高速化と 1.06 倍の電力削減ができる。このように、提案する AT 方式により、全て単精度演算の場合と比較して精度の劣化を抑えつつ、速度向上が見込める事例があることを明らかにした。

キーワード: ソフトウェア自動チューニング, ppOpen-AT, 混合精度演算

1. はじめに

2020 年代の後半には半導体技術の進歩が停滞し、単位電力あたりの演算性能が限界に到達するポストムーア時代が到来すると予想されている。この状況に対抗するため技術の多様化が進む。そこで現在、CPU のマルチコア化やメモリ階層の深化といったアーキテクチャの多様化が進んでいる。

その一環として低精度演算への対応も進んでいる。例えば名古屋大学情報基盤センターに設置されているスーパーコンピュータ「不老」Type I サブシステムでは倍精度、単精度に加えて半精度演算にも対応しており、半精度の FLOPS 値は単精度の約 2 倍、倍精度の約 4 倍となっている。低精度演算は倍精度などの高精度演算と比較してビット幅が狭いことから、メモリ移動コストの削減や演算密度の向上が期待されている。しかし一方で、低精度演算は誤差を増大させ、アプリケーションの信頼性の低下を招く恐れがある。そこで、全ての演算を低精度演算に置き換えるのではなく、演算の一部に低精度演算を活用することにより、演算精度を損ねることなく計算することを目指す混合精度演算の研究が現在盛んに行われている。

ただし混合精度演算をソフトウェアに対して適用しようとした場合、低精度化の適用箇所や適用方法は多岐にわたる。低精度演算の性能は実行するアーキテクチャや問題規模、疎行列格納方式などによって影響を受けることが明らかになっており[1]、混合精度演算活用時に性能に影響を与えるパラメタの組み合わせは膨大になることが予想される。したがって、混合精度演算を用いた最適化を自動化することによって上記の問題を解決することができるので

はないかと考えた。

以上の背景から、本研究では混合精度演算の最適化に際し、ソフトウェア自動チューニング (AT) 技術[2]を用いることで最適化を自動化することを目的としている。本稿において提案する手法では、混合精度演算の性能パラメタを①変数／配列、②ブロック、③関数／サブルーチンの 3 種類に限定し、それぞれのパラメタによる AT 方式を AT 言語 ppOpen-AT[3]の新機能として提案する。

本稿の構成は以下のとおりである。2 章で AT 言語である ppOpen-AT を紹介する。3 章は本稿で提案する ppOpen-AT の新 AT 機能の説明を行う。4 章で性能評価を示す。5 章で関連研究を示し、最後の 6 章でまとめと今後の課題について述べる。

2. AT 基盤 ppOpen-AT

数値計算ソフトウェアをスパコン等の計算機環境で実行する場合、たいていはそのままでは計算性能が出ないため、その計算機環境に適するようにソフトウェアをチューニングする必要がある。しかし、チューニングには専門知識を要するため、近年のアーキテクチャの多様化も影響し、そのチューニングコストは増加している。このことから、ソフトウェアのチューニングを自動で行うソフトウェア自動チューニング (AT) が提案されている。

ppOpen-AT は数値計算ミドルウェア ppOpen-HPC において、プログラムに AT 機能を提供する目的で開発された AT 言語である。FIBER (Framework of Install-time, Before Execute-time, and Run-time auto-tuning) 方式[2][4]による AT 言語 ABCLibScript[4][5]の機能を引き継いでいる。FIBER と

1 名古屋大学 大学院情報学研究科
Graduate School of Informatics, Nagoya University
2 国立環境研究所
National Institute for Environmental Studies

3 名古屋大学 情報基盤センター
Information Technology Center, Nagoya University

は汎用的な AT 機能の付加を支援する AT フレームワークのことで、インストール時、実行起動前、実行時のタイミングで AT を適用する方式である。

次に ppOpen-AT を使用する際の流れを説明する。ppOpen-AT はユーザーとして主にソフトウェア開発者とエンドユーザーの 2 種類のユーザーを想定している。ソフトウェア開発者は自身が開発したソフトウェア上で AT を適用したい箇所に ppOpen-AT のディレクティブを挿入する。そして、プログラムに ppOpen-AT のプリプロセッサを適用することで①複数の最適化候補コード；②最適なコードを探索する AT 機能付きのプログラムコード；が自動生成される。エンドユーザーは自身の計算機環境においてソフトウェアの AT 機能を実行することで、ソフトウェアが自ら現計算環境に最適になるように自動でチューニングを行う。これによってエンドユーザーにチューニングに関する知識が十分に備わっていない場合であっても、計算機環境に最適化されたソフトウェアを利用することが可能になる。

以上が ppOpen-AT を用いた AT の流れである。

3. ppOpen-AT における演算精度と消費電力を考慮した自動チューニング機能

本稿において提案する AT 方式は ppOpen-AT の新機能として提供する。提案する AT 方式では、①変数／配列、②ブロック、③関数／サブルーチンを最適化対象とする。これらを性能パラメータとし、性能パラメータの精度を変更することによって、演算精度・実行時間・消費電力量の観点から対象ソフトウェアが最適になるような AT を行う。

図-1 に提案する AT 方式の流れを示す。以降では図-1 の各ステップについて詳細に説明する。

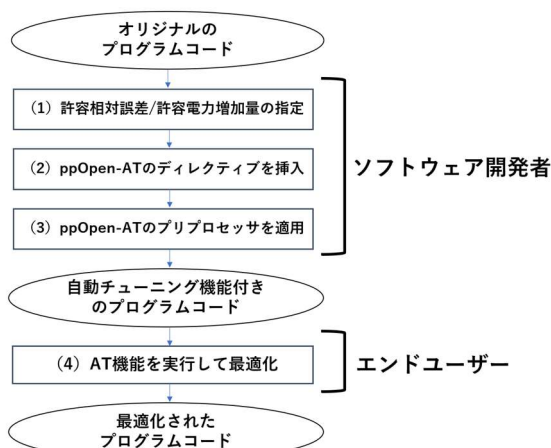


図-1. 提案する AT 方式の流れ

3.1 許容相対誤差／許容電力増加量の指定

本稿で提案する AT 方式では、ソフトウェア開発者は事前に基準となる演算に対する相対的な演算精度劣化の許容

値（許容相対誤差）を指定する。例えばある精度の演算に対して許容相対誤差を $1e-7$ 以下と指定した場合、本 AT 機能は指定された許容相対誤差 $1e-7$ を満たした上で計算速度・消費電力量が最適となるように実装を探索する。

また同時にソフトウェア開発者は、ソフトウェアが実行時に消費する電力量についても、電力増加に対する許容値（許容電力増加量）を指定することができる。許容電力増加量を指定しなかった場合、デフォルトでは電力増加とならないように実装を探索する。

3.2 ppOpen-AT のディレクティブの挿入

本 AT 機能では性能パラメータとして①変数／配列、②ブロック、③関数／サブルーチンの 3 種類を指定する。ユーザーは混合精度演算を適用したい箇所に、以下に示す 3 種類の AT 方式のうち適するディレクティブを挿入する。

・変数／配列方式

```
!oat$ MixedPrecision variables, [clause . . . ]
    {structured-block}
!oat$ end MixedPrecision variables
```

以上では、ユーザーは混合精度演算を適用したい箇所を囲むように上記のディレクティブを挿入する。このとき、低精度化させたい変数／配列を clause として指定する（同時に複数の変数／配列を指定することが可能）。また、精度の変更方法（例：倍精度→単精度）なども clause として指定するようにする。

・ブロック方式

```
!oat$ MixedPrecision blocks, [clause . . . ]
{
!oat$ MixedPrecision block <1>
    {structured-block}
!oat$ end MixedPrecision block <1>

!oat$ MixedPrecision block <2>
    {structured-block}
!oat$ end MixedPrecision block <2>

. . .
}
!oat$ end MixedPrecision blocks
```

以上では、ユーザーは混合精度演算を適用したい箇所を囲むように、“!oat\$ MixedPrecision blocks”と“!oat\$ end MixedPrecision blocks”を挿入する。さらに、その内部を“!oat\$ MixedPrecision block

<num>”と“!oat\$ end MixedPrecision block
<num>”で囲む。このコード部分が“ブロック”となり、
ブロック単位で精度を変更することができる。ブロックの
番号は<num>で指定する。指定したブロック群をどのよう
に低精度化するかは clause として指定する。また精度の
変更方法も clause として指定するようにする。

・関数／サブルーチン方式

```
!oat$ MixedPrecision subprogram, [clause . . . ]  
  {structured-block}  
!oat$ end MixedPrecision subprogram
```

以上では、ユーザーは混合精度演算を適用したい関数
／サブルーチンとその呼び出し元を上記のディレクティブ
で囲む。呼び出し側には clause に caller(num), 呼び出
される側には clause に callee(num)を指定する。num は
呼び出し側と呼び出される側で同じ数字を指定する。これ
により、関数／サブルーチン単位で低精度化を行う。また
精度の変更方法も clause として指定するようにする。

3.3 ppOpen-AT のプリプロセッサを適用

ppOpen-AT のプリプロセッサを適用すると、3.2 節で挿
入したディレクティブに従って候補コードが生成される。
このとき、①関数／配列、②ブロック、③関数／サブルー
チンの各指定に対して生成される候補コード例をそれぞれ
図-2 から図-7 に示す。なお、3 方式全ての場合で倍精度演
算 (DP) から単精度演算 (SP) への精度変更を指定した
場合を想定している。

・変数／配列方式

変数／配列方式ではディレクティブで指定した変数／
配列単位で低精度化を行う。図-2 は、配列 B, 配列 C の
各要素に 2 を足した後、行列積 A を求める簡単なプログ
ラムにディレクティブを挿入した例である。このとき、プ
リプロセッサを適用して生成された候補コードが図-3 で
ある。なお、図-3 では配列 B, 配列 C を低精度化した場
合の候補コードを表示している。

変数／配列方式での候補コードは以下の手順によって
生成される。

- ① 指定した変数／配列について、ディレクティブで指定
した区間内に存在する DP 変数／DP 配列を新たに用意
した SP 変数／SP 配列に置き換える。
- ② 置き換えた SP 変数／SP 配列について宣言を新たに挿
入する。
- ③ 精度変更箇所の直前と直後で DP 変数／DP 配列と SP
変数／SP 配列で値を代入する。

```
!oat$ MixedPrecision variables, ¥  
ChangeVariables(B(:, :), C(:, :)), ¥  
ChangePrecision(DP, SP)  
do i = 1, n  
  do j = 1, n  
    do k = 1, n  
      B(i, k) = B(i, k) + 2.0_DP  
      C(k, j) = C(k, j) + 2.0_DP  
      A(i, j) = A(i, j) + B(i, k) * C(k, j)  
    enddo; enddo; enddo  
!oat$ end MixedPrecision variables
```

図-2. サンプルプログラム (変数／配列方式のディレクテ
ィブを挿入)

```
real(SP) :: B_SP(n,n)  
real(SP) :: C_SP(n,n)  
  
B_SP(:, :) = B(:, :)  
C_SP(:, :) = C(:, :)  
  
!oat$ MixedPrecision variables, ¥  
ChangeVariables(B(:, :), C(:, :)), ¥  
ChangePrecision(DP, SP)  
do i = 1, n  
  do j = 1, n  
    do k = 1, n  
      B_SP(i, k) = B_SP(i, k) + 2.0_DP  
      C_SP(k, j) = C_SP(k, j) + 2.0_DP  
      A(i, j) = A(i, j) + B_SP(i, k) * C_SP(k, j)  
    enddo; enddo; enddo  
!oat$ end MixedPrecision variables  
  
B(:, :) = B_SP(:, :)  
C(:, :) = C_SP(:, :)
```

図-3. 図-2 のサンプルプログラムから候補コードを生成す
る例 (配列 B と配列 C を単精度化した場合)

・ブロック方式

ブロック方式ではディレクティブで指定したブロック
単位で低精度化を行う。図-4 は図-2 と同様のプログラム
にブロック方式のディレクティブを挿入した例である。こ
のとき、プリプロセッサを適用して生成された候補コード
が図-5 である。なお、図-5 ではブロック 1 の精度を倍精
度から単精度に低精度化した場合の候補コードを表示して
いる。

ブロック単位での候補コードは以下の手順によって生
成される。

- ① ディレクティブで指定したブロック内に出現する全て
の DP 変数／DP 配列／DP 定数を新たに用意した SP
定数／SP 配列／SP 定数に置き換える。
- ② ①で置き換えた SP 変数／SP 配列について宣言を新た
に挿入する。
- ③ 指定ブロック内の DP 変数／DP 配列が指定ブロック
外にも存在している場合は指定ブロック外についても
SP 変数に置き換える。
- ④ 精度変更箇所の直前と直後で DP 変数／DP 配列と SP
変数／SP 配列で値を代入する。

```
!oat$ MixedPrecision blocks, ¥
ChangeBlocks(1),ChangePrecision(DP,SP)
do i = 1, n
  do j = 1, n
    do k = 1, n

      !oat$ MixedPrecision block <1>
      B(i, k) = B(i, k) + 2.0_DP
      C(k, j) = C(k, j) + 2.0_DP
      !oat$ end MixedPrecision block <1>

      !oat$ MixedPrecision block <2>
      A(i, j) = A(i, j) + B(i, k) * C(k, j)
      !oat$ end MixedPrecision block <2>

    enddo; enddo; enddo
  !oat$ end MixedPrecision blocks
```

図-4. サンプルプログラム（ブロック方式のディレクティブを挿入）

```
real(SP) :: B_SP(n,n)
real(SP) :: C_SP(n,n)

B_SP(:, :) = B(:, :)
C_SP(:, :) = C(:, :)

!oat$ MixedPrecision blocks, ¥
ChangeBlocks(1),ChangePrecision(DP,SP)
do i = 1, n
  do j = 1, n
    do k = 1, n

      !oat$ MixedPrecision block <1>
      B_SP(i, k) = B_SP(i, k) + 2.0_SP
      C_SP(k, j) = C_SP(k, j) + 2.0_SP
      !oat$ end MixedPrecision block <1>

      !oat$ MixedPrecision block <2>
      A(i, j) = A(i, j) + ¥
        B_SP(i, k) * C_SP(k, j)
      !oat$ end MixedPrecision block <2>

    enddo; enddo; enddo
  !oat$ end MixedPrecision blocks

B(:, :) = B_SP(:, :)
C(:, :) = C_SP(:, :)
```

図-5. 図-4 のサンプルプログラムから候補コードを生成する例（ブロック<1>を単精度化した場合）

・関数／サブルーチン方式

関数／サブルーチン方式ではディレクティブで指定した関数／サブルーチン単位で低精度化を行う。図-6 は図-2 と同様の計算を行うサブルーチンコードにディレクティブを挿入する例である。このとき、プリプロセッサを適用して生成された候補コードが図-7 である。なお、図-7 は指定したサブルーチン全体を単精度化した場合の候補コードを表示している。

関数／サブルーチン方式での候補コードは以下の手順によって生成される。

- ① 対象サブルーチンと同様のコードを持つ新しい SP サブルーチンを複製する。（宣言も挿入）
- ② SP サブルーチン内に存在する DP 変数／DP 配列／DP 定数を新たに用意した SP 変数／SP 配列／SP 定数に置き換える。
- ③ ②で置き換えた SP 変数／SP 配列を呼び出し側で新たに宣言する。
- ④ サブルーチンを呼び出す直前と直後で DP 変数／DP 配列と SP 変数／SP 配列で値を代入する。

! 呼び出し側(caller)

```
!oat$ MixedPrecision subprogram, ¥
caller(1), ChangePrecision(DP,SP)
call sample_subroutine(A(:, :), B(:, :), C(:, :))
!oat$ end MixedPrecision subprogram
```

! 呼び出される側(callee)

```
!oat$ MixedPrecision subprogram, ¥
callee(1), ChangePrecision(DP,SP)
subroutine sample_subroutine(A,B,C)

  implicit none
  real(DP), intent(inout) :: A(n,n)
  real(DP), intent(in) :: B(n,n)
  real(DP), intent(in) :: C(n,n)

  integer :: i,j,k

  do i = 1, n
    do j = 1, n
      do k = 1, n
        B(i, k) = B(i, k) + 2.0_DP
        C(k, j) = C(k, j) + 2.0_DP
        A(i, j) = A(i, j) + B(i, k) * C(k, j)
      enddo; enddo; enddo

  return
end subroutine sample_subroutine
!oat$ end MixedPrecision subprogram
```

図-6. サンプルプログラム（関数／サブルーチン方式のディレクティブを挿入）

! 呼び出し側(caller)

```
real(SP) :: A_SP(n,n)
real(SP) :: B_SP(n,n)
real(SP) :: C_SP(n,n)
```

```
A_SP(:, :) = A(:, :)
B_SP(:, :) = B(:, :)
C_SP(:, :) = C(:, :)
```

```
!oat$ MixedPrecision subprogram, ¥
caller(1), ChangePrecision(DP,SP)
call sample_subroutine_SP( ¥
  A_SP(:, :), B_SP(:, :), C_SP(:, :))
!oat$ end MixedPrecision subprogram
```

```
A(:, :) = A_SP(:, :)
B(:, :) = B_SP(:, :)
C(:, :) = C_SP(:, :)
```

図-7. 図-6 のサンプルプログラムから候補コードを生成する例（サブルーチン全体の単精度化した場合）

エンドユーザーはソフトウェア開発者が提供する自動チューニング付きのソフトウェアに対して AT 機能を実行することで、ソフトウェアを現実実行環境に最適化することができる。AT 機能では、3.3 節で生成した候補コードを全て実行し、その演算精度と実行時間、消費電力量を調べる。このとき、(1) で指定した許容相対誤差と許容電力増加量を満たす候補コードのうち、実行時間と消費電力量の観点から最適になる候補コードを選択することで、最適実行が決定される。

また、予備実験において測定する実行時間は対象 3 重ループの実行時間とする。演算精度は NICAM の出力 QV, QC, QR, QI, QS, QG の最大相対誤差として測定する。最大相対誤差の式は以下の式 (1) である。またこのとき、比較元となっている DP 出力結果とは NICAM の計算を全て倍精度演算で行った際の値である。DP 出力結果と精度変更後の出力結果の違いは、対象ループにおける精度の変更とそれに伴うプログラムの一部の変更のみであり、実行環境やコンパイラオプション等は同一の条件で測定を行った。

$$(\text{最大相対誤差}) = \max \left\{ \frac{|(\text{DP 出力結果}) - (\text{精度変更後の出力結果})|}{(\text{DP 出力結果})} \right\} \quad \dots (1)$$

消費電力量の測定には富士通のジョブ運用ソフトウェア Technical Computing Suite が提供する Power API[8]を用いた。対象 3 重ループの実測電力量を測定値とした。

本提案手法では 3 章で説明したように低精度化に伴い、DP 変数/DP 配列と SP 変数/SP 配列の間で値を交換する必要があるため代入が発生する。そこで今回の実験では代入時間を削減するために OpenMP の workshare により並列化を行っている。

4.4 実験環境

名古屋大学情報基盤センター設置のスーパーコンピュータ「不老」Type I サブシステムを使用した。詳細は以下の通りである。

- FUJITSU Supercomputer PRIMEHPC FX1000
 - プロセッサ名 : A64FX
 - 1 ノードあたりのプロセッサ数 : 1
 - 1 ノードあたりのコア数 : 48 コア + 2 アシスタントコア (I/O 兼計算ノードは 48 コア + 4 アシスタントコア)
 - 周波数 : 2.2GHz
 - 1 ノードあたりの理論演算性能 : 倍精度 3.3792 TFLOPS, 単精度 6.7584 TFLOPS, 半精度 13.5168 TFLOPS
 - コンパイラ : ftrpx: Fujitsu Fortran Compiler 4.5.0 tcsds-1.2.31
 - 実行時のコンパイルオプション : -Kfast,ocl,preex,noalias=s,mfunc=2 -Nl1st=i -Nl1st=t -X03 -Ncompdisp -Koptmsg=2 -Cpp -Kdynamic_iteration -Ksimd,openmp -Kauto,threadsafe -x- -Kprefetch_sequential=soft -Nclang -L /opt/FJSVtcs/pwrm/aarch64/lib64 -lpwr

4.5 実験結果 (変数/配列方式)

変数/配列方式によってグループ単位で単精度化した場合の実行時間の比較を図-8 に示す。横軸の“All Group SP”とは全てのグループが単精度 (SP) の場合を表し, “All Group DP”とは全てのグループが倍精度 (DP) であることを示している。なお、ここでいう All Group SP とはディレクティブで指定した全ての変数/配列グループが単精度化した場合であって、NICAM 全体が SP 演算となったわけではないことに注意しておく。All Group SP では、あくまで

ディレクティブで指定した範囲内において全ての変数/配列が単精度化するのであって、ディレクティブの範囲外については DP 演算となっている。

図-8 を見ると提案手法には無視できない代入時間が発生することが確認できる。

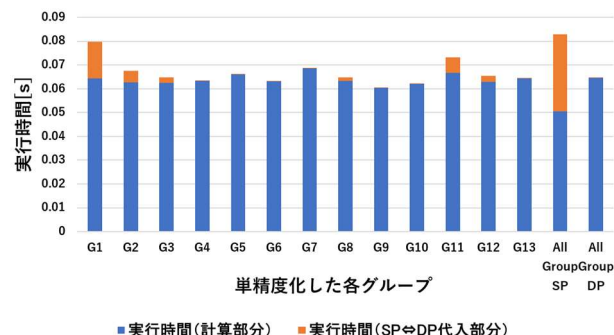


図-8. 各グループを単精度にした場合の実行時間の比較

図-8 の各グループの実行時間を All Group DP の実行時間と比較し、各グループの速度向上率を求めたものを図-9 に示す。図-9 によると、グループ 9 が最も速度向上率が大きく 1.07 倍、次にグループ 10 の 1.04 倍となった。また、All Group SP は代入時間のために 0.78 倍の速度低下となった。

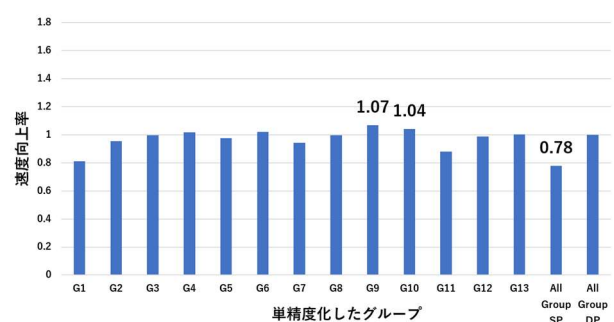


図-9. 各グループを単精度にした場合の速度向上率の比較

図-10 では各グループを単精度化した場合の速度向上率と最大相対誤差を比較している。最も精度が劣化する場合は All Group SP の場合で最大で 8.E-02 となった。また最も速度向上率が大きかったグループ 9 は NICAM の出力 QI, QC, QV, QS で精度の劣化が見られ、特に QI の最大相対誤差は 4.E-04 となった。一方、次に速度向上率が大きかったグループ 10 は NICAM 全ての出力で精度劣化が抑えられ、最大でも QI の 1.E-13 となった。

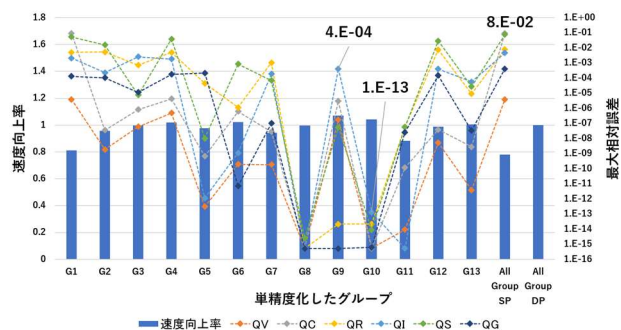


図-10. 各グループを単精度化した場合の速度向上率と最大相対誤差の比較

次に図-11 に各グループを単精度化した場合の消費電力量を示す。

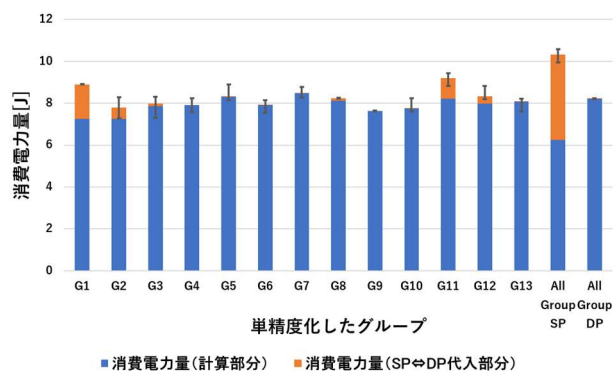


図-11. 各グループを単精度化した場合の消費電力量の比較

なおエラーバーは5回繰り返した結果の中の最大と最小を表している。測定値の誤差はあるものの、今回の実験ではグループ9において最も消費電力量が小さくなった。

図11をもとに電力削減率を求め、図-9の速度向上率と比較した結果を図-12に示す。図-12をみると、電力削減率と速度向上率の間に大きな傾向の違いは見られなかった。速度向上率同様、グループ9を単精度化した場合が最も電力削減率が大きく1.08倍の性能向上となった。

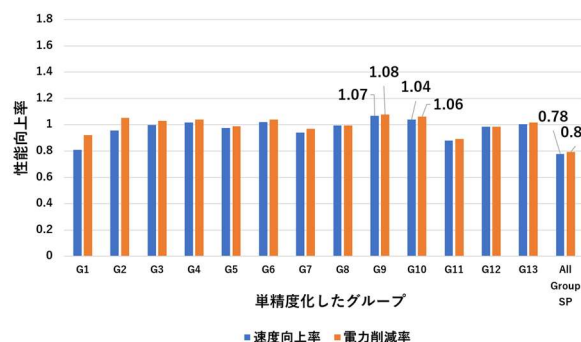


図-12. 各グループを単精度化した場合の速度向上率と電力削減率の比較

4.6 実験結果（ブロック方式）

ブロック方式によってブロック単位で単精度化した場合の実行時間の比較を図-13に示す。横軸の”All Block SP”とは全てのブロックが単精度(SP)の場合を示し,”All Block DP”は全てのブロックが倍精度(DP)の場合を示している。図-13をみると、All Block SPにおいて無視できない代入時間は発生しているものの、計算部分の実行時間が大きく削減していることが確認できる。

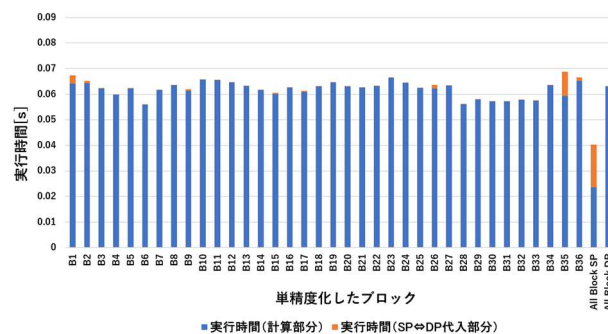


図-13. 各ブロックを単精度化した場合の実行時間の比較

図-13の各グループの実行時間をAll Block DPと比較し、各グループの速度向上率を求めたものを図-14に示す。図-14より、最も速度向上率が高いのはAll Block SPの場合で1.56倍であった。また次点はブロック6とブロック28の1.12倍であった。

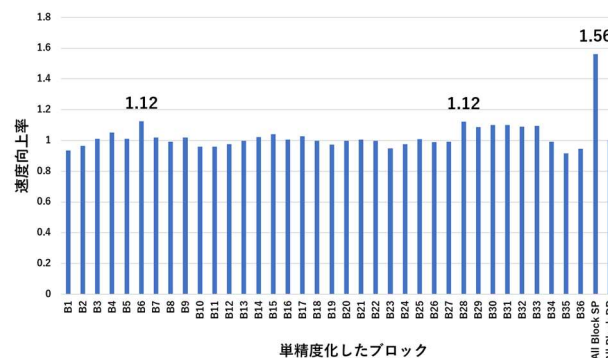


図-14. 各ブロックを単精度化した場合の速度向上率の比較

次に、図-14の各ブロックの速度向上率と最大相対誤差を比較したものを図-15に示す。図-15によると、速度向上率が1.56倍と最も大きいAll Block SPではNICAMの全ての出力で最大相対誤差が1.E-07を超えており、最大でQCが7.E-02の誤差となっている。All Block SPの次に速度向上率が高いのはブロック6とブロック28であり、速度向上率はどちらも1.12倍となっている。ただし最大相対誤差をみると、ブロック28は最大で3.E-02まで誤差が大き

くなるのに対し、ブロック 6 は最大で $3.E-04$ と、ブロック 28 よりも精度劣化を抑えられることが確認できる。

また図-10 の変数／配列方式と比較すると、ブロック方式では速度向上率は大きくなる傾向にあるが、一方で最大相対誤差も大きくなる傾向が見られる。特に QS, QR は全ての場合で最大相対誤差が $1.E-07$ 以上となっており、ブロック方式による精度劣化の影響を強く受けていると考えられる。

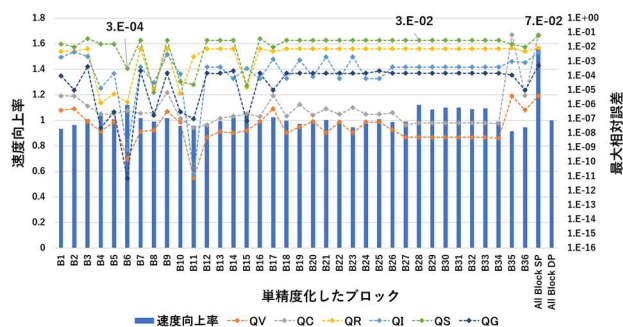


図-15. 各ブロックを単精度化した場合の速度向上率と最大相対誤差の比較

次に各ブロックを単精度化した場合の消費電力量の比較を図-16 に示す。また、図-16 をもとに各ブロックを単精度化したときの電力削減率を求め、図-14 の速度向上率と比較した結果を図-17 に示す。

図-17 をみると、電力削減率は変数／配列方式と同様に、速度向上率との間に大きな傾向の違いは見られなかった。電力削減率は All Block SP の場合に最も大きく、1.59 倍の性能向上となった。次にブロック 28 の場合で 1.08 倍、ブロック 6 の場合で 1.06 倍と続く。

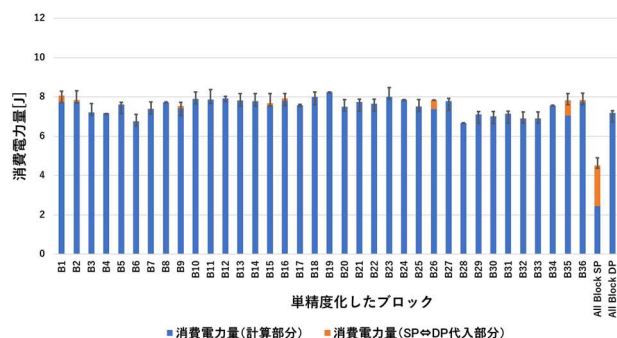


図-16. 各ブロックを単精度化した場合の消費電力量の比較

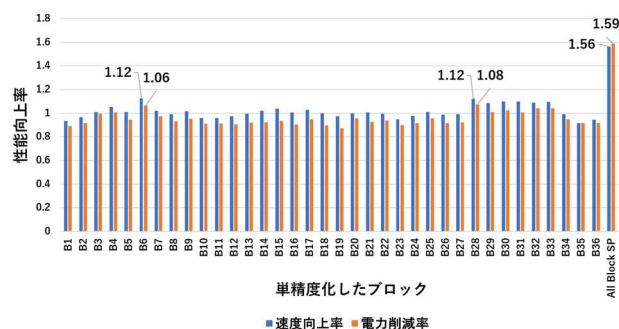


図 17. 各グループを単精度化した場合の速度向上率と電力削減率の比較

5. 関連研究

混合精度演算を活用した ppOpen-AT の AT 機能として、対象サブルーチンのローカル変数をグループ化し、グループごとに精度変更を行うことで高速化する AT 手法が櫻井らによって提案されている[7]。報告[7]は、ローカル変数を精度変更の対象としているのに対し、本 AT 手法では対象をグローバル変数、ブロック、関数／サブルーチンと広げており、混合精度演算における包括的な AT 方式の提案を目的としている。

また、GPU を用いた混合精度演算として、NVIDIA より Automatic Mixed Precision (AMP)[9]が提供されている。これは FP16 演算を行う Tensor コアを部分的に用いることで、行列の積和演算を精度の低下を抑えつつ高速に行う技術である。機械学習や数値計算に広く使われている。AMP はライブラリベースで提供されるのに対し、本稿の AT 手法はディレクティブベースで提供されるため、AMP と異なり任意のプログラムに対して幅広く適用可能である。

6. おわりに

本研究ではポストムーア時代に向けて低精度化する計算機環境に合わせた自動チューニング (AT) を行うため、従来の ppOpen-AT に実装されていない AT 機能を検討し、予備実験を行った。

特に本提案手法では、混合精度演算における性能パラメータとして①変数／配列、②ブロック、③関数／サブルーチンを用いることを提案し、今回の予備実験では①変数／配列方式と②ブロック方式について演算精度・実行時間・消費電力量の観点から評価を行った。

NICAM における予備実験の結果、①変数／配列方式では、グループ 9 を単精度化した場合が最も速度向上率が大きくなり、1.07 倍の速度向上、1.08 倍の電力削減率となったが、同時に最大相対誤差は $4.E-4$ まで悪化することを確認した。一方、グループ 10 を単精度化した場合は、最大相対誤差を $1.E-13$ に抑えた上で 1.04 倍の速度向上、1.06 倍

の電力削減率となった。②ブロック方式では、すべてのブロックを単精度化した場合が最も速度向上率が大きく 1.56 倍の速度向上、1.59 倍の電力削減率となったが、最大相対誤差は 7.E-02 まで悪化することを確認した。一方、ブロック 6 を単精度化した場合は、最大相対誤差を 3.E-04 に抑えた上で 1.12 倍の速度向上、1.06 倍の電力削減率となった。

以上の結果から、本 AT 手法によって倍精度演算と単精度演算を組み合わせることで、単精度のみの演算と比較して精度劣化を抑えつつ、倍精度演算と比較して速度向上を見込める事例があることを明らかにした。

今後の課題としては、本 AT 手法の NICAM 以外への適用が挙げられる。特にブロック方式は変数/配列方式と比較して精度劣化の影響が大きいいため、低精度化が計算結果に直接の影響を与えない前処理などへの適用を試みたい。

また、今回提案した③関数/サブルーチン方式についても予備実験を行い、その有用性について評価する必要がある。

さらに本提案は演算精度と消費電力の最適化を試みるのであるが、単精度化をすることでどのように演算精度が劣化するかを自動で調査するツールとみなすこともできる。このような演算精度の変化を見る工程を自動化するツールとしての有効性評価も、重要な今後の課題である。

謝辞 本研究は、科学技術研究費補助金、基盤研究 (S)、「(計算+データ+学習)融合によるエクサスケール時代の革新的シミュレーション手法」(課題番号: 19H05662) による。

参考文献

- [1] 中島研吾, 坂本龍一, 星野哲也, 有間英志, 埴敏博, 近藤正章, “低精度演算とアプリケーション性能”, 研究報告ハイパフォーマンズコンピューティング (HPC) ,2020-HPC-174(5),1-9 (2020-05-06) , 2188-8841
- [2] Takahiro Katagiri, Daisuke Takahashi, “Japanese Autotuning Research: Autotuning Languages and FFT”, Proceedings of the IEEE (Volume: 106, Issue: 11, Nov. 2018), pages 2056-2067, 2018.
- [3] 片桐孝洋, “ppOpen-AT: ポストペタスケール時代の数値シミュレーション基盤ソフト”, 数理解析研究所講究録 第 1791 巻, pages 107-111, 2012.
- [4] Takahiro Katagiri, Kenji Kise, Hiroki Honda, Toshiyuki Yuba, “Effect of auto-tuning with user's knowledge for numerical software”, Proceedings of ACM Computing Frontiers 04, pp.12-25, 2004.
- [5] Takahiro Katagiri, Kenji Kise, Hiroki Honda, and Toshitsugu Yuba, “ABCLibScript: A Directive to Support Specification of An Auto-tuning Facility for Numerical Software”, Parallel Computing, Vol. 32, No.1, pp.92-112, 2006.
- [6] Satoh, M., Tomita, H., Yashiro, H., et al., “The Non-hydrostatic Icosahedral Atmospheric Model: description and development”, Progress in Earth and Planetary Science 1, Article number 18, 2014.
- [7] 片桐孝洋, “演算精度と実行時間を考慮する自動チューニング方式と ppOpen-AT への実装について”, 計算工学講演会論文

集 Vol.25, 2020.

- [8] 富士通, “FUJITSU Software Technical Computing Suite V4.0L20 ジョブ運用ソフトウェア API ユーザーズガイド Power API 編”, J2UL-2462-02Z0(01) , 2020 年 6 月
- [9] NVIDIA Developer サイト, “Automatic Mixed Precision for Deep Learning”, <https://developer.nvidia.com/automatic-mixed-precision>, (参照 2021-11-02)

付録

A.1 グループの分け方

4.5 節におけるグループの分け方を、表 A-1 に示す。

表 A-1 4.5 節におけるグループの分け方

グループ番号	変数名/配列名
G1	rhog, rhogvx, rhogvy, rhogvz, rhogw, rhoge, rhogq, vx, vy, vz, w, UNCCN, rho, tem, pre, q, qd, precip, precip_rhoe, precip_lh_heat, precip_rhophi, precip_rhokin, gprec, rceff, rctop, rwtop, tctop, rceff_cld, rctop_cld, rwtop_cld, tctop_cld, gsgam2, gsgam2h, gam2, gam2h, ix, iy, iz, jx, jy, jz, z, dt
G2	drhogqv, drhogqc, drhogqi, drhogqr, drhogqs, drhogqg
G3	psatl, qsatl, psati, qsati, Nc
G4	dens, temp, qv, qc, qr, qi, qs, qg, qv_t, qc_t, qr_t, qi_t, qs_t, qg_t, Sliq, Sice, Rdens, rho_fact, temc
G5	RLMDr, RLMDr_2, RLMDr_3, RLMDs, RLMDs_2, RLMDs_3, RLMDg, RLMDg_2, RLMDg_3, RLMDr_1br, RLMDr_2br, RLMDr_3br, RLMDs_1bs, RLMDs_2bs, RLMDs_3bs, RLMDr_dr, RLMDr_3dr, RLMDr_5dr, RLMDs_ds, RLMDs_3ds, RLMDs_5ds, RLMDg_dg, RLMDg_3dg, RLMDg_5dg, RLMDr_7, RLMDr_6dr
G6	tems Xs2, MOMs_0, MOMs_1, MOMs_2, MOMs_0bs, MOMs_1bs, MOMs_2bs, MOMs_2ds, MOMs_5ds_h, RMOMs_Vt, coef_at, coef_bt, loga_b_nm
G7	Vti, Vtr, Vts, Vtg, Esi_mod, Egs_mod, rhoqc, Pracw_orig, Pracw_kk, Praut_berry, Praut_kk, Dc, betai, betas, Ka, Kd, Nu, Glv, Giv, Gil, ventr, vents, ventg, net, fac, fac_sw, zerosw, tmp
G8	sw_bergeron, a1, a2, ma2, dt1, Ni50
G9	sw, rhoqi, XNi, XMi, Di, Ni0, Qi0
G10	xf_qc, rf_qc, r2_qc, r2_qr, r3_qc, r3_qr, dgamma_a, GAM_dgam23, GAM_dgam, coef_dgam, coef_xf
G11	Vt, cva, rgs, rgsh
G12	wk, ml_Pconv, ml_Pconw, ml_Pconi
G13	UNDEF, EPS, PI, Rvap, LHV0, LHS0, LHF0, PRE00