富士通 PRIMEHPC FX10 チューニング連載講座

３．単体ノード性能チューニング

片桐　孝洋

東京大学情報基盤センター　准教授

１．はじめに

本稿では、富士通 PRIMEHPC FX10（以降、FX10 と記載）で有効な単体性能チューニング技法を紹介します。逐次プログラムの最適化と、ノード内でのスレッド並列の最適化について、主に Fortran 言語での実装の観点から解説します。

紙面の都合から網羅的なチューニング手法の説明は割愛します。チューニング技法について網羅的に知りたい場合は、富士通社が東大ユーザ向けに公開しているチューニングマニュアル[1]がオンラインマニュアルにありますので、そちらをご覧ください。

また FX10 のハードウェア概要については、記事[2]で紹介されています。この記事は、東京大学情報基盤センタースーパーコンピューティング部門の HP に掲載されていますので､そちらもご覧ください。

本稿では、FX10 のハードウェア性能を最大限に引き出すチューニング手法のうち、SIMD 化、ソフトウェアパイプライニング、セクタキャッシュ、および、ループ分割とループ融合を中心に説明します。

２．FX10 のハードウェア特徴：プログラミングの観点から

２．１　SIMD (Single Instruction Multiple Data)

FX10 の CPU ハードウェアの特徴は、SPARC64 アーキテクチャを拡張した SPARC64 IXfx アーキテクチャ[2]にあります。SPARC64 IXfx に限定したことではありませんが、近年の CPU、特にスカラ計算機の特徴は、多数の SIMD (Single Instruction Multiple Data)演算器を搭載することで高速化を達成しています。SIMD とは、1 命令を実行する時、複数データの演算を同時に実行する方式です。

SPARC64 IXfx における SIMD の特徴[1]を以下に示します。

- 1 命令で 2 演算を実行
- 積和演算をサポート
- 1 つのコアで 2 つの SIMD 命令を同時に実行（単精度演算、および、倍精度演算）

ここで積和演算とは積と和からなる演算で、ａ＋ｂ＊ｃの形式の演算です。

つまり FX10 では、1 コアで 8 個（2［演算／命令］＊ 2［命令/SIMD 演算］＊ 2[SIMD 演算/コア]）の演算を同時実行することができます。1 チップ（ノード）には 16 個のコアがありますので、1 ノードでは 128 個の演算が同時実行できることになります。

SIMD 演算を実装したハードウェアの中には、データが固定境界、たとえば、16 バイト境界上に無いとデータが一度に CPU に取り込めず、2 回ロード命令を発行するものがあります。ところが FX10 では、倍精度 SIMD のロード命令は 8 バイト境界でも一度に取り込めます。また、非

連続なメモリ空間から 1 個ずつデータを取り込み SIMD 演算ができます。

２．２　マスク付き SIMD 化

数値計算プログラムの中には、do ループ中に IF 文があるものがあります。この IF 文を含む do ループを SIMD 化する場合、**マスク付き SIMD 化**を行うことで、専用のハードウェア命令を発行し処理を高速化します。

たとえば、以下のプログラム例[1]を考えます。

```
do i = 1, n
  if ( x(i) .gt. 0.0 ) then
    a(i) = b(i) * c(i)
  else
    a(i) = b(i) / c(i)
  endif
enddo
```

通常は、上記のループ中の IF 文の成立を判断するため、配列 x(i)のデータをロードし、値 0.0 と比較したうえで、IF 文中の式の実行を行います。この一連の処理を、条件分岐命令で行います。ところが、マスク付き命令を使うことで、条件分岐命令を削減して高速化できます。

マスク付き命令では、以下の一連の処理を行います[1]。

1. FP レジスタを比較し、結果を FP レジスタに書き込む（マスクを作る）
2. 演算を行う
3. FP レジスタの値に基づいて、選択的に FP レジスタの値をメモリにストアする

以上の処理を行う専用のハードウェア命令を行うことで処理が高速化されることに加えて、ループの回転をまたいで命令スケジューリングを行うことで、命令レベルの並列性を高める最適化を行います。これを、**ソフトウェアパイプライニングによる最適化**と呼びます。ソフトウェアパイプライニングも、FX10 で必要とされる最適化です。

なお上記のマスク付き SIMD は、コンパイラのデフォルトオプションではなされません。

マスク付き SIMD 命令を使用するには
-Ksimd=2
オプションの追加が必須です（Fortran、C、C++、共通事項）

２．３　ソフトウェアパイプライニング

ソフトウェアパイプライニングは、FX10 に限らず、スカラ計算機で必須の最適化手法です。ソフトウェアパイプライニングを行うには、コンパイラに処理を指示する方法（コンパイラオプションやディレクティブの挿入）、に加えて、場合により、コンパイラがソフトウェアパイプライニングを適用できるようにコードを書きかえることが必要になります。これらは、事例ごとに異なります。

ソフトウェアパイプライニングの基本的な原理は、ループにおいて次のループの演算に必要な処理を重ね合わせることで、処理の実行効率を高めることにあります。ここでいう処理とは、

（1）データロード、（2）演算、（3）データストア、の３つになります。

概念の説明のため、以下のハードウェア値[1]を想定します。データロードには 3 サイクル、データ演算（加算）には 3 サイクル、データストアに 1 サイクル必要だとします。

このとき、以下の演算を考えます。

```
do i = 1 , n
    a(i) = b(i) + c(i)
enddo
```

ループ中の演算をするには、b(i)と c(i)のデータロードが必要です。このとき、b(i)と c(i)のロードが同時にできるとすると、ロードで 3 サイクル必要です。その後、b(i)+c(i)の加算をします。これは、3 サイクル必要です。その後、加算データを a(i)にストアします。これは 1 サイクル必要です。つまり 1 回の演算で、合計で 3+3+1=7 サイクル必要となります。

以上をソフトウェアパイプライン化します。最初の 1 回(i=1)の演算は 7 サイクルかかることは変わりません。ただし、i=2,…N-1 の演算については、ロード、加算、ストアが同時に利用できる機構があれば、十分に時間がたつと同時に演算を実行している状況になります。

ここでハードウェア実装上、ロード、ストアは制限なく同時に実行できないです。ロード、ストアが同時に 3 つまでそれぞれ発行できるとすると、上記のループでは十分な反復時間が経つと、演算が 1 サイクルで実行できるようになります。(詳しくは、[1]を参照ください)。

以上の原理で、一次キャッシュからデータの読み出しと、一次キャッシュへのデータの書き出し、および演算の見た目の時間を隠蔽することで処理の高速化を狙う技術が、ソフトウェアパイプライニングです。

２．４　インダイレクトアクセスのプリフェッチ

数値計算処理の中には、以下のように演算に必要な配列のアクセスのために、間接的（インダイレクト）なアクセスを必要にする例が多数あります。以下の例を考えます。

```
do i = 1 , n
    do j=ind_a(i), ind_a(i+1)
      indx = ind_x(j)
      y(i) = y(i) + a(j) * x(indx)
    enddo
enddo
```

以上の例では、配列 x(indx)のデータをアクセスするためには、インデックスを収納した配列 ind_x(j)を先にアクセスし、x(indx)の参照インデックス indx を取得する必要があります。

以上のようなインダイレクトアクセスを高速化するには、ind_x のデータロードをソフトウェアパイプライニングなどにより高速化し、かつそのデータの indx を x(indx)のメモリアクセスに即座に利用できることが必要です。それが出来ないと、ソフトウェアパイプライニング等の命令スケジューリングも期待どおりの効果が得られない場合があります。このようなデータロードの高速化のため、事前にキャッシュなどの高速メモリにデータを取り込んでおく技術を**プリフェッチ**と呼びます。

上記のインダイレクトアクセスに対するプリフェッチが有効かどうかは、ループの長さに依存します。ここでは、j-ループの長さです。このループの長さは利用するアプリケーションごとに異なります。たとえば、5〜10 の長さのアプリケーションもあります。また状況によっては、100 以上の長さを持つものもあります。

一般に、プリフェッチが有効なループ長は、ハードウェア性能とループの構成に依存します。FX10 では、数百以上の長さが必要であることが、経験的にわかっております。

以上のインダイレクトアクセスのプリフェッチを行うことは、コンパイラのデフォルトで設定されていません。

インダイレクトアクセスのプリフェッチを使用するには **-Kprefetch_indirect** オプションの追加が必須です（Fortran、C、C++、共通事項）

再度ですが、上記のインダイレクトアクセスのプリフェッチを指定しても高速化されるかどうかは、ループ長などの状況に依存します。

２．５　セクタキャッシュ

FX10 のハードウェアの最大の特徴として、ソフトウェアから制御できるキャッシュを持っていることがあげられます。このソフトウェアから制御可能なキャッシュ機構のことを、**セクタキャッシュ**と呼びます。

FX10 のセクタキャッシュは、キャッシュを２つのセクタに分け、データを用途ごとに分けてユーザ自身が制御することで、キャッシュの利用率を向上させ高速化できる機構[1]です。

具体的に、以下の例[1]を示します。

```
!ocl cache_sector_size(6,18)
!ocl cache_subsector_assign(a)
do k=1,n
  do j=1,n
    do i=1,n
      a(i) = a(i) + b(i,j,k)*c(i,j,k)
   enddo
  enddo
enddo
!ocl end_cache_subsector
!ocl end_cache_sector_size
```

以上の例のうち、“!ocl”で記載される行が、富士通コンパイラへのユーザ最適化指示行（ディレクティブ）です。そのうち、“cache_”で指示されるディレクティブが、セクタキャッシュへの指示になります。

“!ocl cache_subsector_assign(a)”の()中に、セクタキャッシュのセクタ１に載せたい配列を記載します。この場合は、配列 a になります。この場合では、配列 b と配列 c は、セクタ 0 に載ります。これにより、セクタ１に載る配列 a は、配列 b、c のアクセスにより、キャッシ

ュから追い出されることがなくなります。“cache_sector_size(6,18)”は、二次キャッシュのセクタ 0 とセクタ 1 に割り当てるウェイ数を記載します。この場合は、配列 b、c のウェイ数を 6 とし、配列 a のウェイ数を 18 とします。なお、一次キャッシュに対するセクタ 0 とセクタ 1 のウェイ数の設定も、“cache_sector_size()”で設定できます。詳しくは、Fortran 利用手引書をご覧ください。

FX10 では、一次キャッシュのウェイ数は 2、二次キャッシュのウェイ数は 24 です。

２．６　高速ストア（XFILL）

FX10 で特徴的なハードウェア機構として、高速ストア(XFILL)があります。

XFILL[1]は、データをメモリからロードすることなく、二次キャッシュ上に書き込み用のキャッシュラインを確保することで、ストア命令時にロードがキャッシュヒットします。このため、メモリスループットがネックになっているプログラムでの性能改善が期待できます。

XFILL の動作条件は、イタレーション間の依存が無いこと、連続アクセスであること、および、定義した配列の参照が無いこと、です。

以上の XFILL の利用については、コンパイラのデフォルトで利用が設定されていません。

XFILL を使用するには **-KXFILL** もしくは **-KXFILL=*N***（*N*は 1～100） オプションの追加が必須です（Fortran、C、C++、共通事項）

なお、-KXFILL=*N*　指定時の *N*は、*N*キャッシュライン先のデータを XFILL 命令の対象とすることを示します。

また、以下の制約があります[1]。

- 同一ループ内に参照がある配列、非連続アクセスされる配列、または IF 構文配下でストアされる配列に対しては出力されません。
- 確保したキャッシュラインは、必ずストアするようなループ変形を行うため、ループアンローリング、ループストライピング、が適用できません。
- ループ長が小さいループでは、性能が低下することがあります。

以上から、XFILL が適用されたとしても、必ずしも高速化されるわけではありません。

２．７　コンパイラによるコード最適化の確認方法

コンパイラによる最適化がソースコードにどのように適用されたかどうか知ることは重要です。そのために最適化情報を、プログラムのファイルごとに、別ファイルとして出力するコンパイラオプションがあります。

コード最適化に関する情報を出力するには **-Qt** オプションの追加が必須です（Fortran のみ）

上記オプションを指定すると、<ファイル名>.lst というファイルが生成されます。このファ

イル中に、最適化情報が、基のソースコードとともに記載されて出力されます。

なおC、C++言語には、上記に相当するオプションが現在ありません。同様の機能を提供するように準備中です。

２．８　最適化事例

以上に説明した最適化の効果を示すため、事例を紹介します。以下のプログラムを考えます。

```
      program sample
      implicit real*8(a-h,o-z)
      include "omp_lib.h"
      integer N
      parameter (N=10000)
      double precision a(N),b(N),c(N),d(N)
      double precision t1, t2, tw
      double precision EPS
      parameter (EPS=1.0e-0)
      CALL RANDOM_NUMBER(a)
      CALL RANDOM_NUMBER(b)
      CALL RANDOM_NUMBER(c)
      icnt = 0
      t1 = omp_get_wtime()
      do i=1, N
        d(i) = a(i)*a(i)+b(i)*b(i)+c(i)*c(i)
      enddo
      do i=1, N
        do j=i, N
          if (dabs(d(i)-d(j)) .lt. EPS) then
            icnt = icnt + 1
          endif
        enddo
      enddo
      t2 = omp_get_wtime()
      tw = t2-t1
      print *, "N=",N
      print *, "dcnt=",icnt
      print *, "time=",tw
      stop
      end
```

デフォルトパラメタによる実行

以上のプログラムを、コンパイルオプション“-Kfast,openmp -Qt”でコンパイルして、１コ

アで実行すると、以下の結果が出力されます。

```
N= 10000
dcnt= 41491448
time= 0.2658829689025879
```

ここで、上記のループには IF 文があるので、マスク付き SIMD 化の対象となります。しかし、デフォルトのコンパイラオプションでは適用がなされません。また、マスク付き SIMD 化の性能は、IF 文の成立率(Ture になる回数)に依存します。

以上の例の場合の IF 文の成立率は、41491448/(10000*10000)/2*100=82.9%となります。

最適化の結果を出力のためのオプション“-Qt”を指定しているので、最適化情報を参照できます。これを、以下に示します。

```
                    <<< Loop-information Start >>>
                    <<<  [OPTIMIZATION]
                    <<<    SIMD
                    <<<    SOFTWARE PIPELINING
                    <<< Loop-information  End >>>
38     1       8v             do i=1, N
39     1       8v               d(i) = a(i)*a(i)+b(i)*b(i)+c(i)*c(i)
40     1       8v             enddo
41     1                      do i=1, N
42     2       8s               do j=i, N
43     3       8m                 if (dabs(d(i)-d(j)) .lt. EPS) then
44     3       8s                   icnt = icnt + 1
45     3       8v                 endif
46     2       8v               enddo
47     1                      enddo
```

以上のうち、最初の<<<…>>>の部分には、当該ループに対して適用した最適化が示されています。

最初のループでは、SIMD 化とソフトウェアパイプライニングが施されていることがわかります。一方、IF 文を含む次のループには、何も施されていないことがわかります。

上記のうち、最初の数字は基のプログラムの行数を示します。2 番目の数字は、ループのネストレベルを示します。また 3 番目の数字とアルファベットは、最初の数字がループ展開段数、次の数字が各行の SIMD 化情報になります。do のある行に付与されている記号から、そのループが SIMD 化されたかどうかを判断できます。具体的には、以下になります。

- v ：SIMD 化された
- m ：SIMD 化された部分と SIMD 化されなかった部分を含む
- s ：SIMD 化されなかった
- 空白：SIMD 化対象でない

以上から、たとえば“38　1　8v”は、基のソースコードの 38 行目において、ループネスト

レベル 1、8 段の展開を行い SIMD 化されている、ということを意味しています。
IF 文を含む do ループは“42　2　8s”なので、そのループが SIMD 化されなかった事を示します。-Ksimd=2 のコンパイルオプションや、!OCL SIMD のディレクティブ指示により高速化される可能性があります。

マスク付き SIMD 化の指定による実行

マスク付き SIMD 化をするためのオプションを追加し、“-Kfast,openmp -Ksimd=2 -Qt”でコンパイルして実行した結果を示します。

```
N= 10000
dcnt= 41491448
time= 0.1693410873413086
```

以上から、0.265[秒]/0.169[秒]*100 = 1.56 倍の速度向上が達成できました。このときの、最適化情報を見てみましょう。

```
                    <<< Loop-information Start >>>
                    <<<  [OPTIMIZATION]
                    <<<    SIMD
                    <<<    SOFTWARE PIPELINING
                    <<< Loop-information  End >>>
38      1       8v            do i=1, N
39      1       8v              d(i) = a(i)*a(i)+b(i)*b(i)+c(i)*c(i)
40      1       8v            enddo
41      1                     do i=1, N
                    <<< Loop-information Start >>>
                    <<<  [OPTIMIZATION]
                    <<<    SIMD
                    <<<    SOFTWARE PIPELINING
                    <<< Loop-information  End >>>
42      2       6v              do j=i, N
43      3       6v                if (dabs(d(i)-d(j)) .lt. EPS) then
44      3       6v                  icnt = icnt + 1
45      3       6v                endif
46      2       6v              enddo
47      1                     enddo
```

以上より、2 番目の IF 文を含むループにも、SIMD 化とソフトウェアパイプライニングが適用されたことがわかります。また、IF 文の行は“6v”になっており、SIMD 化ができていることがわかります。

XFILL の適用

最後に、XFILL の適用をしてみましょう。XFILL を適用するためのオプションを追加し、

“-Kfast,openmp -Ksimd=2 -XFILL -Qt”でコンパイルして実行した結果を示します。

```
N= 10000
dcnt= 41491448
time= 0.1693398952484131
```

以上から、マスク付きSIMD化に対して、実行時間はほとんど変化していません。
最適化情報を以下に示します。

```
                    <<< Loop-information Start >>>
                    <<<  [OPTIMIZATION]
                    <<<    SIMD
                    <<<    SOFTWARE PIPELINING
                    <<<    PREFETCH       : 2
                    <<<      d: 2
                    <<<    XFILL          : 2
                    <<<      d: 2
                    <<< Loop-information  End >>>
38     1        v             do i=1, N
39     1        v               d(i) = a(i)*a(i)+b(i)*b(i)+c(i)*c(i)
40     1        v             enddo
41     1                      do i=1, N
                    <<< Loop-information Start >>>
                    <<<  [OPTIMIZATION]
                    <<<    SIMD
                    <<<    SOFTWARE PIPELINING
                    <<< Loop-information  End >>>
42     2       6v               do j=i, N
43     3       6v                 if (dabs(d(i)-d(j)) .lt. EPS) then
44     3       6v                   icnt = icnt + 1
45     3       6v                 endif
46     2       6v               enddo
47     1                      enddo
```

以上から、最初のループに対して、XFILLが適用されていることがわかります。

XFILL の効果があまりなかった理由として、最初のループの実行時間の占める割合が少なかったことに加え、XFILLはメモリースループットネックの場合に効果があるので、今回の1スレッド実行では効果が得られなかったからと推測されます。

３．ループ分割とループ融合

FX10は1ノードあたり16コアある計算機です。したがって、OpenMPのスレッド並列化やコンパイラでの自動スレッド並列化にかかわらず、スレッド並列化が可能な逐次プログラムの書

き方でないと高性能化は望めません。

並列化の効率についてはコード構成に依存しますが、多くの場合、8 コア実行（8 スレッド実行）以上になるとスレッド実行効率が劣化し、何等かのチューニングが必要になります。

そこで本節では、ループ分割とループ融合について紹介することにします。

FX10 はベクトル計算機ではなく、スカラ計算機（キャッシュ計算機）に分類されます。したがって、単にベクトル長を長くする書き方では、一般のキャッシュ計算機と同様に、高速化に限度があります。したがって、キャッシュ計算機に向くブロック化手法を適用する必要があります。

この一方で、ベクトル長が短いと、1 スレッド実行時においても、ソフトウェアパイプライニングやプリフェッチの効果がなくなるため、実行効率が上がらないことが知られています。

既に記述していますが、ループの構成に依存しますが、FX10 では数百ループ長が高い実行効率を得るために必要ということが経験的に知られています。また、1 スレッド実行で数百ループ長が必要であるということは、16 スレッド実行時では全体として、数千ループ長が必要ということになります。したがって、スレッド実行時においては、スレッド実行の対象のループ長は長くする必要がある、とも言えます。

以上のように、キャッシュのためにはループ長を短くする必要があり、一方で、ソフトウェアパイプライニングのためにはループ長を（ある程度）長くする必要があるという矛盾を解決しないといけません。ですので、適切に長いループ長を維持する必要があります。

この節では以降、実例によりループ分割とループ融合の最適化技法について説明します。なお、この例題および最適化技法については、著者による HITACHI SR16000/M1 のチューニング技法入門の記事[3]と同一ですので、併せてご参照ください。

３．１　ループ分割

初めに、ループ分割について説明します。このチューニング手法は、ループ中に多数の配列アクセスや式が書かれているとき、レジスタがあふれることにより、メインメモリにデータを書き戻すコード（スピルコード）が生成されるのを防ぐことによる高速化手法です。レジスタ上のデータの有効活用により、高速化が達成されることを目指すチューニング技法です。

ここでは、以下のコードを考えます。

```
DO K = 1, NZ
  DO J = 1, NY
    DO I = 1, NX
      SXX (I,J,K) = ( SXX (I,J,K) + (RL(I) + RM(I)*DX(I,J,K))*DT )*QG(I)
      SYY (I,J,K) = ( SYY (I,J,K) + (RL(I) + RM(I)*DY(I,J,K))*DT )*QG(I)
      SZZ (I,J,K) = ( SZZ (I,J,K) + (RL(I) + RM(I)*DZ(I,J,K))*DT )*QG(I)
      SXY (I,J,K) = ( SXY (I,J,K) + (RM(I)*(DDX(I,J,K)+DDY(I,J,K)))*DT )*QG(I)
      SXZ (I,J,K) = ( SXZ (I,J,K) + (RM(I)*(DDX(I,J,K)+DDZ(I,J,K)))*DT )*QG(I)
      SYZ (I,J,K) = ( SYZ (I,J,K) + (RM(I)*(DDY(I,J,K)+DDZ(I,J,K)))*DT )*QG(I)
    ENDDO
  ENDDO
ENDDO
```

上記コードでは、ループ中に配列アクセスが多いため、スピルコードが生成される可能性があります。ここで、このコードでは最内側のＩループについて式を２分割しても、演算結果に違いがない性質を利用し、以下のように書きなおすことができます。これを、**Ｉループに対するループ分割**と呼びます。

```
DO K = 1, NZ
  DO J = 1, NY
    DO I = 1, NX
        SXX (I,J,K) = ( SXX (I,J,K) + (RL(I) + RM(I)*DX(I,J,K))*DT )*QG(I)
        SYY (I,J,K) = ( SYY (I,J,K) + (RL(I) + RM(I)*DY(I,J,K))*DT )*QG(I)
        SZZ (I,J,K) = ( SZZ (I,J,K) + (RL(I) + RM(I)*DZ(I,J,K))*DT )*QG(I)
    ENDDO
    DO I = 1, NX
        SXY (I,J,K) = ( SXY (I,J,K) + (RM(I)*(DDX(I,J,K)+DDY(I,J,K)))*DT )*QG(I)
        SXZ (I,J,K) = ( SXZ (I,J,K) + (RM(I)*(DDX(I,J,K)+DDZ(I,J,K)))*DT )*QG(I)
        SYZ (I,J,K) = ( SYZ (I,J,K) + (RM(I)*(DDY(I,J,K)+DDZ(I,J,K)))*DT )*QG(I)
    ENDDO
  ENDDO
ENDDO
```

一方、このループでは、Ｊループ、およびＫループに対しても、それぞれ独立にループ分割できます。特に、Ｋ ループに対してループ分割すると、２つの独立した３重ループが形成されます。これを以下に示します。

```
DO K = 1, NZ
  DO J = 1, NY
    DO I = 1, NX
        SXX (I,J,K) = ( SXX (I,J,K) + (RL(I) + RM(I)*DX(I,J,K))*DT )*QG(I)
        SYY (I,J,K) = ( SYY (I,J,K) + (RL(I) + RM(I)*DY(I,J,K))*DT )*QG(I)
        SZZ (I,J,K) = ( SZZ (I,J,K) + (RL(I) + RM(I)*DZ(I,J,K))*DT )*QG(I)
    ENDDO
  ENDDO
ENDDO
DO K = 1, NZ
  DO J = 1, NY
    DO I = 1, NX
        SXY (I,J,K) = ( SXY (I,J,K) + (RM(I)*(DDX(I,J,K)+DDY(I,J,K)))*DT )*QG(I)
        SXZ (I,J,K) = ( SXZ (I,J,K) + (RM(I)*(DDX(I,J,K)+DDZ(I,J,K)))*DT )*QG(I)
        SYZ (I,J,K) = ( SYZ (I,J,K) + (RM(I)*(DDY(I,J,K)+DDZ(I,J,K)))*DT )*QG(I)
    ENDDO
  ENDDO
ENDDO
```

このように完全にループが分割されると、場合により、コンパイラの各種最適化が適用され

るようになって、高速化される可能性があります[1]。

３．２　ループ融合

前節のループ分割は主に、レジスタへのデータ移動を最小にすることで高速化を狙うチューニング技法でした。ここでは、外側のループ長を長くすることで、高スレッド並列化（OpenMP並列化やコンパイラによる自動スレッド並列化）環境での高速化を狙うチューニング技法であるループ融合を紹介します。

前節と同じ以下のコードに、ループ融合を施すことを考えます。

```
DO K = 1, NZ
  DO J = 1, NY
    DO I = 1, NX
      SXX (I,J,K) = ( SXX (I,J,K) + (RL(I) + RM(I)*DX(I,J,K))*DT )*QG(I)
      SYY (I,J,K) = ( SYY (I,J,K) + (RL(I) + RM(I)*DY(I,J,K))*DT )*QG(I)
      SZZ (I,J,K) = ( SZZ (I,J,K) + (RL(I) + RM(I)*DZ(I,J,K))*DT )*QG(I)
      SXY (I,J,K) = ( SXY (I,J,K) + (RM(I)*(DDX(I,J,K)+DDY(I,J,K)))*DT )*QG(I)
      SXZ (I,J,K) = ( SXZ (I,J,K) + (RM(I)*(DDX(I,J,K)+DDZ(I,J,K)))*DT )*QG(I)
      SYZ (I,J,K) = ( SYZ (I,J,K) + (RM(I)*(DDY(I,J,K)+DDZ(I,J,K)))*DT )*QG(I)
    ENDDO
  ENDDO
ENDDO
```

上記ループは３重ループなので、ループ融合の可能性は、（１）K、J、I ループすべてをまとめて１重ループ化する方法；（２）J、I ループをまとめて２重ループ化する方法、の２つがあります。

まず、（１）の K、J、I ループすべてをまとめて１重ループ化するループ融合を示します。これは、以下になります。

```
DO KK = 1, NZ*NY*NX
      K = (KK-1)/(NY*NX) + 1
      J = mod((KK-1)/NX,NY) + 1
      I = mod(KK-1,NX) + 1
      SXX (I,J,K) = ( SXX (I,J,K) + (RL(I) + RM(I)*DX(I,J,K))*DT )*QG(I)
      SYY (I,J,K) = ( SYY (I,J,K) + (RL(I) + RM(I)*DY(I,J,K))*DT )*QG(I)
      SZZ (I,J,K) = ( SZZ (I,J,K) + (RL(I) + RM(I)*DZ(I,J,K))*DT )*QG(I)
      SXY (I,J,K) = ( SXY (I,J,K) + (RM(I)*(DDX(I,J,K)+DDY(I,J,K)))*DT )*QG(I)
      SXZ (I,J,K) = ( SXZ (I,J,K) + (RM(I)*(DDX(I,J,K)+DDZ(I,J,K)))*DT )*QG(I)
      SYZ (I,J,K) = ( SYZ (I,J,K) + (RM(I)*(DDY(I,J,K)+DDZ(I,J,K)))*DT )*QG(I)
ENDDO
```

[1] この例題のように簡単な式が書かれているループの場合は、コンパイラが自動でループ分割を行い最適化することが多いと思います。しかし、ループ中の式が多くなり複雑になると、コンパイラによるデータ依存解析ができなくなって、ループ分割がされなくなりコンパイラによる最適化も限定されます。つまるところ逐次コードもシンプルに記載するのがよく、これがコンパイラによる高速化の鍵にもなります。

上記ループの特徴は、ループ融合され、新しく作られたKKループ長はNZ＊NY＊NXの長さとなり、もとのKループ長であるNZより格段に長くなることです。これは、OpenMPなどでの高スレッド並列化する場合、大変有利となります。

たとえば、NX=XY=NZ＝10のときを考えます。このとき、外側ループをOpenMPでスレッド並列化すると、元のループではループ長が10しかないので、10スレッドまでしか並列化できません。一方、1重ループ化したときのループ長は1000ですので、FX10で16スレッド並列化を行っても、性能向上が期待できます。

このループの欠点は、K、J、Iの変数の値が関数（mod関数）で写像されるため、K、J、Iで参照される配列のメインメモリからの呼び出しに関し、コンパイラによるデータの読み出し最適化（データプリフェッチ、もしくは、ソフトウェアパイプライニング）が阻害されることです。その結果、データ読み出し時間の増大を招き、結果として、実行時間が遅くなります。

まとめると､データ読み出し時間は増えるが､スレッド並列化による速度向上が望める場合、上記のコードは高速になります。

次に、（2）のJ、Iループをまとめて2重ループ化するループ融合の例を紹介します。これは、以下になります。

```
DO KK = 1, NZ*NY
    K = (KK-1)/NY + 1
    J = mod(KK-1,NY) + 1
    DO I = 1, NX
      SXX (I,J,K) = ( SXX (I,J,K) + (RL(I) + RM(I)*DX(I,J,K))*DT )*QG(I)
      SYY (I,J,K) = ( SYY (I,J,K) + (RL(I) + RM(I)*DY(I,J,K))*DT )*QG(I)
      SZZ (I,J,K) = ( SZZ (I,J,K) + (RL(I) + RM(I)*DZ(I,J,K))*DT )*QG(I)
      SXY (I,J,K) = ( SXY (I,J,K) + (RM(I)*(DDX(I,J,K)+DDY(I,J,K)))*DT )*QG(I)
      SXZ (I,J,K) = ( SXZ (I,J,K) + (RM(I)*(DDX(I,J,K)+DDZ(I,J,K)))*DT )*QG(I)
      SYZ (I,J,K) = ( SYZ (I,J,K) + (RM(I)*(DDY(I,J,K)+DDZ(I,J,K)))*DT )*QG(I)
    ENDDO
ENDDO
```

上記コードでは、最外側のKKループの長さがNZ＊NYの長さになるので、高並列スレッド実行に向いたコードになります。かつ、最内のIループは連続アクセスなため、コンパイラによるデータプリフェッチも阻害しません。結果として、ループ長を長く保つスレッド並列化に向いたループになり、かつ、コンパイラによるデータ読み出し最適化も阻害しないコードとなります。結果として、効果的なループになる可能性があります。

３．３　ループ融合およびループ分割のコードチューニング事例

ここでは、有限差分法(Finite Difference Method, FDM)の実シミュレーションコードに現れる主要カーネルの１つに、ループ分割とループ融合を施した場合の性能評価結果を載せます。

このFDMカーネルは、東京大学で開発している数値計算ミドルウェアppOpen-HPC[4]で採用されているコード(ppOpen-APPL/FDM)の主カーネルの１つです。

このプログラムでは、最外ループをOpenMPによりスレッド並列化しています。以下に、元のプログラムを以下に載せます。

```
!$omp parallel do private (J, I, STMP1, STMP2, STMP3, STMP4, RL, RM, RM2,RMAXY, RMAXZ,
RMAYZ, RLTHETA, QG)
DO K = 1, NZ
  DO J = 1, NY
    DO I = 1, NX
      RL=LAM(I,J,K); RM=RIG(I,J,K); RM2=RM+RM;
      RLTHETA=(DXVX(I,J,K)+DYVY(I,J,K)+DZVZ(I,J,K))*RL
```

<分割後コピーが必要な式の始まり>

```
      QG = ABSX(I)*ABSY(J)*ABSZ(K)*Q(I,J,K)
```

<分割後コピーが必要な式の終わり>

```
      SXX(I,J,K) = ( SXX (I,J,K) + (RLTHETA + RM2 * DXVX(I,J,K)) * DT ) * QG
      SYY(I,J,K) = ( SYY (I,J,K) + (RLTHETA + RM2 * DYVY(I,J,K)) * DT ) * QG
      SZZ(I,J,K) = ( SZZ (I,J,K) + (RLTHETA + RM2 * DZVZ(I,J,K))*DT )*QG
```

<ループ分割ポイント>

```
      STMP1 = 1.0/RIG(I,J,K); STMP2 = 1.0/RIG(I+1,J,K);
      STMP4 = 1.0/RIG(I,J,K+1); STMP3 = STMP1 + STMP2;
      RMAXY = 4.0/(STMP3 + 1.0/RIG(I,J+1,K) + 1.0/RIG(I+1,J+1,K))
      RMAXZ = 4.0/(STMP3 + STMP4 + 1.0/RIG(I+1,J,K+1))
      RMAYZ = 4.0/(STMP3 + STMP4 + 1.0/RIG(I,J+1,K+1))
```

<分割後コピーが必要な式の挿入場所>

```
      SXY(I,J,K) = ( SXY (I,J,K) + (RMAXY * (DXVY(I,J,K) + DYVX(I,J,K))) * DT ) * QG
      SXZ(I,J,K) = ( SXZ (I,J,K) + (RMAXZ * (DXVZ(I,J,K)+DZVX(I,J,K))) * DT ) * QG
      SYZ(I,J,K) = ( SYZ (I,J,K) + (RMAYZ * (DYVZ(I,J,K)+DZVY(I,J,K))) * DT ) * QG
    END DO
  END DO;
END DO
!$omp end parallel do
```

上記のプログラムにおいて、**<ループ分割ポイント>**とあるのは、この分割ポイントの組合せでループ分割を行うことを意味しています。ただし、上記の分割点は最適な点ではなく、ユーザが直観的に有効と思える分割点を示しています。

また、**<分割後コピーが必要な式の始まり>～<分割後コピーが必要な式の終わり>**は、ループ分割により再計算が必要となる式の範囲の指定です。**<分割後コピーが必要な式の挿入場所>**は、実際にループ分割後に、＜コピーが必要な式＞をコピーする場所です。

この基プログラムに対して、ループ分割とループ融合を行った以下の 7 種類のコードの実行時間を、FX10 の 1 ノード（16 スレッド実行）を用いて性能評価しました。

以下に、7 種類のプログラムの概略を載せます。

- #1： 元の 3 重ループ（**基本性能**）
- #2: ループ分割を I-ループで行ったもの
- #3: ループ分割を J-ループで行ったもの
- #4: ループ分割を K-ループで行ったもの（独立した 3 重ループが 2 つ生成される）

- #5: ループ融合を#2 に対して行ったもの
- #6: ループ融合を#1 に対して行ったもの（ネスト無、1 重ループ）
- #7: ループ融合を#1 に対して行ったもの（２重ループ）

問題サイズは、nz = 257, ny = 256, nz = 128 です。また、対象ループに対して 10 回反復した時間を計測します。コンパイラオプションは、“-Kfast,openmp”です。

ループ分割およびループ融合による最適化効果

表 1 に、結果を示します。

表１　各カーネルコードの実行時間[秒]

スレッド数	カーネル#1（基本性能）	カーネル#2	カーネル#3	カーネル#4	カーネル#5	カーネル#6	カーネル#7
1	17.75	17.75	17.75	11.51	17.70	46.78	17.70
2	8.777	8.782	8.777	5.703	8.754	23.19	8.751
4	4.322	4.330	4.324	2.827	4.317	11.55	4.316
8	2.157	2.162	2.157	1.414	2.156	5.774	2.154
16	1.087	1.089	1.087	0.713	1.086	2.915	1.086

表１より、#4 のカーネルが、すべてのスレッド数での実行において高速となりました。また、#1 のカーネルに対する速度向上は 1.5 倍といえます。

次に、“-Qt”オプションにより最適化の状況を確認します。

#6 のカーネルが、どのスレッド数の実行においても低速になっています。最適化情報を参照すると、当該ループが 1 重ループ化することでソフトウェアパイプライニングが適用されなくなっていました。したがって、ソフトウェアパイプライニングの阻害が理由の 1 つとして考えられます。

また、#4 のカーネルが最高速になった理由を考えます。#1 のカーネル、および、#4 のカーネルともに、プリフェッチ、SIMD 化、および、ソフトウェアパイプライニングが対象ループに適用されていました。違いは、#4 のカーネルにおいて 2 つある 3 重ループのうち、前半のカーネルに 4 段のアンローリングが適用され、かつ SIMD 化されていました。この理由から、前半のカーネルが、基の#1 カーネルより効率よく実行されるようになったことが理由と考えられます。

一方、コードレベルで分割したループによるカーネル、#2、#3、#5 については、コンパイラ最適化により融合され、基のループ構成に戻っていました。したがって、これらのカーネルの実行時間が、基の#1 カーネルとほぼ同じになっているのは驚くべきことではありません。

以上のように、ループ分割とループ融合の効果は、コンパイラが行う最適化（ループ融合、ループ分割、ソフトウェアパイプライニングなど）に影響されます。どのようなコードの書き方が良いのかは、ハードウェアだけではなく、コンパイラ最適化の影響も考慮する必要があります。

セクタキャッシュによる最適化効果

次に、最高速であった#4 カーネルに対して、セクタキャッシュを利用し速度向上を試みます。

この例では、セクタ 1 に残したい配列は、データ更新部分の 3 配列となります。それ以外の配列は参照のみで、参照の配列数はそれぞれ 12 個、13 個です。また、FX10 の L2 キャッシュのライン数は 24 ラインです。

以上から、セクタ 1 に残す配列数は 3 に固定し、それ以外のセクタ 0 に置く配列数は 21 にします。この場合のディレクティブの記載は以下になります。

前半部分：

```
!ocl cache_sectior_size(21,3)
!ocl cache_subsector_assaign(SXX,SYY,SZZ)
```

後半部分：

```
!ocl cache_sectior_size(21,3)
!ocl cache_subsector_assaign(SXY,SXZ,SYZ)
```

以上のセクタキャッシュを設定し、性能評価を行った結果を表 2 に示します。

表 2　#4 コードの実行時間[秒]

セクタキャッシュ	1 スレッド実行	2 スレッド実行	4 スレッド実行	8 スレッド実行	16 スレッド実行
なし	11.51	5.703	2.827	1.414	0.713
あり	11.50	5.698	2.825	1.413	0.712

表 2 から、この例の場合は、セクタキャッシュを利用した効果はほとんどないといえます。理由は、このコードでは、そもそもキャッシュヒット率が高い、もしくは、参照配列数が多いのでセクタキャッシュの効果が少ない、ことが考えられます。しかしながら、詳細な原因解析には性能プロファイルを行うことが必要です。

６．おわりに

本稿では、FX10 で必要となるコードチューニング技術のうち、単体ノードでの性能チューニングで必要となる、SIMD 化、ソフトウェアパイプライニング、セクタキャッシュ、ループ分割、ループ融合、の技術を中心に解説しました。

FX10 特有であるセクタキャッシュなどのチューニング技術はありますが、チューニング技法の多くは、FX10 に限定しないスカラ計算機向きの最適化技法といえます。したがって、新しいアーキテクチャ向きの技術を勉強することよりも、従来のチューニング技術の基礎を身に着けることが先といえます。その後、FX10 特有のチューニング技術を学ぶことが、本質的な性能チューニングの技術を理解するための近道と考えます。

本解説では、FX10 で提供されている性能プロファイルツールとその利用方法について、紙面の都合から解説できませんでした。FX10 では、さまざまな性能プロファイルツールが提供されています。これらは、単体の性能チューニングに大変有効です。次回以降の記事で、性能プロファイルツールを紹介する予定です。

最後になりますが、本稿が少しでも皆様のプログラムの性能向上の参考になるのであれば、まことに幸甚です。

参 考 文 献

[1]富士通株式会社：FX10 チューニングガイド～Fortran 編～ (2012)
https://oakleaf-www.cc.u-tokyo.ac.jp/cgi-bin/hpcportal.ja/index.cgi

[2]大島聡史：富士通 PRIMEHPC FX10 チューニング連載講座, 1. ハードウェア概要, スーパーコンピューティングニュース,東京大学情報基盤センタースーパーコンピューティング部門, Vol.14, No.2 (2012 年 3 月)
http://www.cc.u-tokyo.ac.jp/support/press/news/

[3]片桐孝洋:HITACHI SR16000/M1 チューニング連載講座, 2. 単体ノード性能チューニング, スーパーコンピューティングニュース, 東京大学情報基盤センタースーパーコンピューティング部門, Vol.14, No.1 (2012 年 1 月)
http://www.cc.u-tokyo.ac.jp/support/press/news/

[4]ppOpen-HPC プロジェクト HP
http://ppopenhpc.cc.u-tokyo.ac.jp/

以上