TX7/AzusA Fortran プログラムの高速化技法

日本電気株式会社
第一コンピュータソフトウェア事業部　　左近　彰一

概要

TX7/AzusA 上で動作する Fortran プログラムの実行性能向上（チューニング）のための高速化技法を紹介します。ハードウェアアーキテクチャの概要、プログラムのチューニングについて説明します。

1.　TX7/AzusA ハードウェア

TX7/AzusA は、Intel 社の 64 ビットプロセッサである Itanium(IA-64)を１６個搭載したシステムです。Itanium は、１クロックサイクルで最大 6 命令(内、浮動小数点命令は 2 命令)が実行可能であり、積和命令（乗算と加減算を１命令で行う）をサポートしていますので、１クロックサイクルで最大 4 つの倍精度浮動小数点演算が実行可能です。科学技術計算向きには、多数のレジスタ、レジスタローテーション、ループ専用ブランチ命令などハードウェア上の工夫が行われています。

コンパイラは、Itanium の性能を活かすために、ソフトウェアパイプライニングの技法（図１）を用いたループの最適化を行い、さらに命令並べかえ（スケジューリング）を行って、できるだけ少数の命令でかつ１サイクルでできるだけ多くの命令を同時に実行できるような高速のコードを生成します。ソフトウェアによるパイプライニングとは、ハードウェアによるパイプライン化と同じような形で、ループの繰り返しをオーバーラップさせことによって、依存関係のない命令を同時に実行するテクニックです。

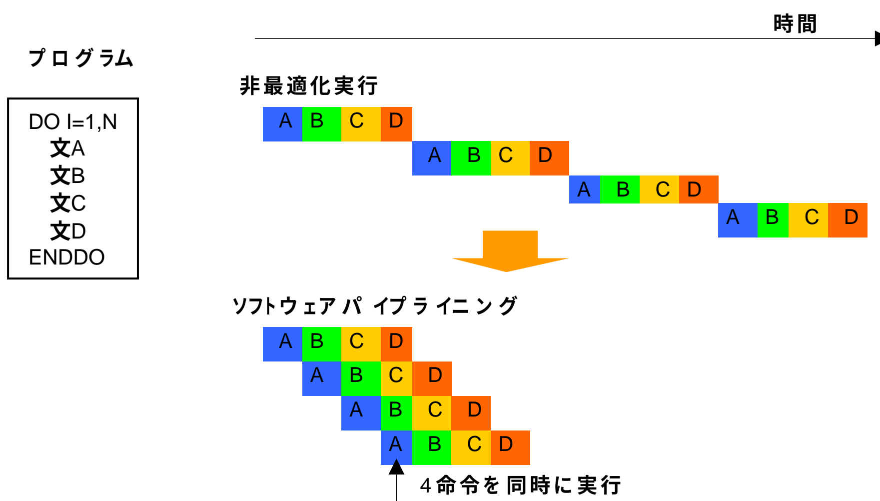

図１　ソフトウェアパイプライニング

また、キャッシュメモリは16KB(命令)＋16KB（データ）のL1キャッシュ、96KBのL2キャッシュ、4MBのL3キャッシュを備えています（浮動小数点データはL2とL3キャッシュのみ利用）。

TX7/AzusAの性能を引き出すには、以下の点がポイントになります。

– クロックあたり多くの命令を同時に実行すること。
– キャッシュを有効に利用し、メモリアクセスによる待ち時間を減らすこと。

2. 高速化のためのコンパイルオプション

2.1 最適化レベルの指定(-O0,-O1,-O2,-O3)

最適化レベルを指定するオプションには、-O0,-O1,-O2,-O3があり、既定値は-O2です。一般には最適化レベルが高いほどプログラムが高速に実行されますが、コンパイル時間は増大します。プログラムの実行速度を追求する場合には-O3の指定を推奨します。ただし、-O3では、除算の乗算化などによる誤差が生じることがあり、精度に敏感なプログラムでは注意が必要です。

表１ 最適化レベルの概要

オプション	最適化機能
-O0	最適化を行わない
-O1	基本的な最適化を行う – グローバルレジスタ割り当て – 命令スケジュール – 共通式削除 – 不要コードの削除 – ループ不変式のループ外移動 – 定数伝播
-O2	高度の最適化を行う（既定値） – ソフトウェアパイプライニング – 定数回ループの展開
-O3	最大限の最適化を行う – キャッシュプリフェッチ – スカラー置換 – 各種ループ変換（融合、入れ換え、アンロール等）

2.2 手続き間最適化（-ipo）

多数回呼び出されている小さな関数やサブルーチンを呼び出し元にインライン展開するオプションです。この機能により、命令列が一箇所にまとまるため、ハードウェアの命令キャッシュや命令バッファのヒット率が向上する等の効果があります。

プログラム

```
do i=1,n
    call sub(b(i),c(i))
    a(i)=b(i)
end do
       :
subroutine  sub(x,y)
    x=sin(y)
end
```

インライン展開後のイメージ

```
do i=1,n
    b(i)=sin(c(i))
    a(i)=b(i)
end do
```

2.3 プロファイル利用最適化(-prof_gen，-prof_use)

プログラムを実際に実行して分岐の方向や多く実行されている部分の情報（プロファイル情報）を採取し、最適化に役立てる機能です。-prof_gen で計測コード入りの実行コードを生成し、-prof_use で採取したプロファイル情報を使用して、実際のプログラムの動きを考慮したインライン展開、命令の並べかえや、ソフトウェアパイプライニングの促進の最適化などに利用します。IF 文を含むループや、goto 文等による複雑な分岐を含むプログラムの高速化に効果があります。

[例] prof_gen/prof_use の使用

```
% f95 -prof_gen p.f
% a.out                  （計測実行。*.dyn ファイルが生成される）
% f95 -prof_use p.f
% a.out                  (最適化実行)
```

3．性能分析

プログラムのチューニングのためには、時間のかかっている部分を見つけ出し、その部分を重点的に高速化するのが効率的です。コンパイラの簡易性能解析（ftrace）機能を用いれば、コンパイルおよびリンク時に-ftrace オプションを指定することによってサブルーチンの呼び出し回数と詳細なハードウェア性能モニタ情報を取り出すことができます（図２）。図で Frequency は実行回数、CPU_CYCLES は CPU クロック数、Routine はサブルーチン名であり、この例では比較的小さなサブルーチン(ran, daxpy 等)が多数呼ばれています。このような場合には、コンパイラの-ipo オプションを指定してインライン展開を行うことで、高速化が可能となります。

また、gprof を使えば、サブルーチン・関数単位の実行時間を表示することができます。

[例] ftrace の使用

```
% f95 -O3 -ftrace p.f
% setenv FTRACE_EVENTS CPU_CYCLES
% a.out
```

[例] gprof の使用

```
% f95 -O3  -p  p.f
% a.out
% gprof  -b  --no-graph
```

```
FLOW TRACE ( profiled by performance monitor ) Version: 1.02
Frequency       CPU_CYCLES Routine
--------- --------------- ---------------
        1          788568 main
       27        15708725 matgen_
   270000        28267808 ran_
       72       108570460 second_
       26         5701485 dgefa_
     2574         1214513 idamax_
     2574          773145 dscal_
   133874        49029978 daxpy_
       25          350901 dgesl_
        1           37215 dmxpy_
        1              56 epslon_
        1           15171 dgesl_
--------- --------------- ---------------
```

図２　ftrace の出力情報

4. プログラムのチューニング

4.1 ソフトウェアパイプライニングの促進

性能分析により時間のかかっているサブルーチンが見つかった場合、サブルーチン中の do ループがソフトウェアパイプライン(SWP)化されているかを調べます。ハードウェアの項で述べたように、SWP は命令の並列実行を促進するため、プログラムの高速化においては重要な要素です。コンパイル時に-opt_report オプションを指定することにより、ループが SWP 化されているか否か、また、SWP 化できない理由は何かという SWP レポートが表示されます。

SWP が行われない、または行われていても効果が出ていない原因には以下のようなものがあり、その対処方法は次のとおりです。

– ループの繰り返しをまたいだデータ依存関係がある。
  この場合には、SWP レポートには次のようなメッセージが表示されます。対処方法としては、依存関係が見かけ上のものかどうかを調べます。実際には依存関係が無い場合、指示行　!dir$ ivdep を指定して SWP 化を促進します。

```
Following are the loop-carried memory dependence edges:
Store  at line     ... --> Load   at line     ...
```

つぎのプログラム例では、コンパイラが配列 a に関してループの繰り返し間で依存関係があるかどうか分からないため、ループの繰り返しあたり 23 サイクル（SWP レポート中 Schedule II の値）かけてスケジュールしています。これは、こ

こで定義するaがつぎの繰り返しで参照されるかもしれないことを考慮してその間を離すためです。しかし、kの値がすべて異なるような場合この心配は無用です。そのような時には、ループの前にivdep指示行を指定することによって、コンパイラに繰り返し間のaの重なりがないことを教えます。この例では、繰り返しあたり3サイクルでスケジュールされるようになり、性能が向上します。

[例] SWPレポートとチューニング

(a) プログラム例

```
subroutine sub(a,b,c,k,n)
        :
do i=1,n
  a(k(i))=a(k(i))+b(i)+c(i)
enddo
        :
```

(b)コンパイルオプション

% f95 *-opt_report -opt_report_phaseecg_swp* a.f

(c)SWPレポート

```
Swp report for loop at line 5 in sub_ in file a.f

     Resource II   =       3
     Recurrence II =       22
     Minimum II    =       22
     Scheduled II  =       23

     Following are the loop-carried memory dependence edges:
     Store  at line     5 --> Load   at line     5
```

(d)指示行の利用

```
       :
!dir$ ivdep
   do i=1,n
     a(k(i))=a(k(i))+b(i)+c(i)
   enddo
       :
```

– ループ中に複雑なIF文がある。

この場合には、SWPレポートには次のようなメッセージが表示されます。コンパイラは、IF文を含むループのSWP化を行わない場合があります。この場合、2.3で述べたプロファイル利用最適化を行うとコンパイラが正確にIF文の分岐確率を把握できるので、IF文を含むループのSWP化が促進されます。

```
Loop has lopsided control flow. Either there are too many IF statements
within the loop, OR the THEN and the ELSE parts are highly unequal =>
loop cannot be pipelined
```

– ループ中に call 文または関数呼び出しがある。
この場合には、SWP レポートには次のようなメッセージが表示されます。可能なら call 文または関数呼び出しの前後でループを分割します。これにより、call 文または関数を含まないループは SWP 化の対象となります。

```
Loop body has a function call => cannot be pipelined
```

4.2 メモリアクセスの改善

メモリアクセスを削減したり、キャッシュメモリを有効に利用したりして、メモリアクセスによる待ち時間を解消します。以下のような技法が使用できます。

– ループのアンローリング
ループの回転数を 1/n にして、ループ内の文を n 倍にする変形です。ループ内の演算を増加させることにより命令の並べかえと並列実行の余地を増やすことが出来ます。また外側ループのアンローリングにより、配列要素のメモリへのロードストアの回数を削減出来る場合があります（比較的単純な形のループの場合、-O3 オプション指定時にコンパイラが自動的にアンローリングを行うことがあります）。つぎの例は、４段アンロールを行うときの例です。

[例] アンロール

アンロール前

```
do i=1,1000
 a(i)=b(i)+c(i)*r
end do
```

アンロール後

```
do i=1,1000,4
    a(i)=b(i)+c(i)*r
    a(i+1)=b(i+1)+c(i+1)*r
    a(i+2)=b(i+2)+c(i+2)*r
    a(i+3)=b(i+3)+c(i+3)*r
end do
```

– ループの入れ換え
多重ループにおいて配列データへのアクセスのアドレスが連続になるように、ループの入れ換えを行います。これによってキャッシュラインが有効に利用されます。つぎの例では、配列 a,b,c,d に関しアクセスが連続的でないため、右側のようにループを入れ換えて i のループを内側にします。

[例] ループの入れ換え

入れ換え前

```
do i=1,n
    do j=1,n
     a(i,j)=b(i,j)-c(i,j)*d(i,j)
    end do
end do
```

入れ換え後

```
do j=1,n
    do i=1,n
     a(i,j)=b(i,j)-c(i,j)*d(i,j)
    end do
end do
```

– 配列のパディング

配列宣言において、各次元のサイズに小さな値を足しこんで配列の非効率なアクセスを避ける方法です。特に、2 のベキ乗飛びのメモリアクセスがあるとき、キャッシュの特定の領域しか使用されなくなり性能が低下する場合があります。パディングを行うことで、これらのキャッシュ利用の非効率な利用を避けることができます。 つぎの例は、配列 a の一次元目に 1 を加えた宣言に変更しています。

[例] 配列のパディング

```
real a(1024,1024)   →   a(1025,1024)
```

5. おわりに

TX7/AzusA Fortran プログラムのチューニング方法およびプログラミング技法について説明いたしました。弊社では、今後性能向上をより容易に行えるようにコンパイラの最適化技術の追求に努めていく所存です。

・Intel および Itanium は、米国 Intel 社またはその子会社の米国および他の国における商標あるいは登録商標です。
・Linux は、Linus Torvalds の米国およびその他の国における商標あるいは登録商標です。