計算機システムの性能評価とプログラムチューニング（後編）

筑波大学電子・情報工学系／計算物理学研究センター　高橋 大介
daisuke@is.tsukuba.ac.jp

アプリケーションの開発において，プロセッサの能力を最大限に引き出すように最適化を図るには，プログラムのチューニングは不可欠である．本稿では，プログラムのチューニング手法について解説する．

【 プログラムチューニングの意義 】

本稿では，計算機システムの性能評価とプログラムチューニングについて解説する．前編では，性能評価全般およびプログラムのチューニングの基本について述べたが，後編ではプログラムのチューニングに関して，より突っ込んだ内容について述べる．

ソフトウェア・アプリケーションにおけるパフォーマンスの重要性については誰もが認識しているが，パフォーマンスのチューニングに関していえば，ソフトウェア開発サイクルの中で後回しになりがちで，まったく考慮されない場合すらある．このような状況に陥っている要因として，コード生成ツールやコンパイラだけでアプリケーションを最適化できるという認識や，単に最新のプロセッサを使えばアプリケーション実行時に最高のパフォーマンスが得られるという過剰な期待が挙げられる．

しかしながら，実行に数カ月以上かかるような計算において，最適化を行うことにより，月のオーダーで実行時間を削減できるような場合であるとか，数値計算ライブラリのように，多くの人に使われるプログラムであれば，チューニングを行う価値は十分にある．

チューニングによってパフォーマンスが仮に3割向上したとすれば，それは，3割性能の高いマシンを使用しているのと同じことになる．

【 高度な最適化手法 】

高度な最適化手法のうち，コンパイラが行う最適化については，優れた著書[1)]があるので，ぜひ参考にしていただきたい．

高度な最適化手法を行ったプログラムを，高い最適化オプションを付けてコンパイルすると，コンパイラが余計な最適化を行い，逆に実行速度が遅くなってしまうことがあるので，注意が必要である．

たとえば筆者の場合，ユーザレベルでブロック化を行った多重ループをコンパイルした際に，あるベンダのコンパイラが勝手にループを入れ換えたことにより，実行速度が著しく低下したことがあった．コンパイラがループを入れ換えていたということは，コンパイラが出力したアセンブリリストを読んで初めて分かったのであるが，最近のコンパイラは高度な最適化を行うものが多く，コンパイラの振舞いがプログラマからは非常に見えにくくなっているものもあり，注意が必要である．

さらに，ある特定のコンパイラに特化した高度な最適化をプログラマが行っていると，同じプロセッサでも，他のコンパイラでコンパイルした場合に，性能が低下する場合があるので，注意が必要である．つまり，究極のチューニングを行う際には，それなりの副作用が生じる可能性があるということである．

演算器の有効利用およびロード・ストア回数の削減

最近のプロセッサでは，複数個の演算器を持つものが多く，これら複数個の演算器を並列に動かすことができる．

複数の演算器を有効に活かしたコーディングのための方針としては，*1*．代入文の右辺の演算項を多くする，*2*. 外側のループを*2*～*4*のステップ幅で回すようにする，等である．

行列積の例を**図-1**，**図-2**に示す．

ソフトウェア・パイプラインニング

ソフトウェア・パイプライニングは，ループからより多くの命令レベル並列性を引き出すための有力なスケジューリング方法である．

ソフトウェア・パイプライニング（モジュロ・スケジューリングとも呼ばれる）のアプローチは，複数の繰返しの実行をオーバーラップさせることで高い性能を実現する手法である．この手法はループの各繰返しに同じスケジュールを使用し，一定の速度で，つまり*1*つの開始インターバルクロックだけ離して各繰返しを開始する．この手法を使ってアルゴリズムを効果的にコーディングするためには，プログラマは以下の事柄を知っていなければならない．*1*．命令のレイテンシ，*2*．使用可能なリソースの数，*3*．適切なレジスタが使用可能かどうか．

ソフトウェア・パイプライニングの手法については，文献*2*），*3*）に詳しい．

【 マシンアーキテクチャに依存する最適化手法 】

積和演算命令

最近のプロセッサは積和演算命令を備えているものが多い．積和演算命令とは，*a*+*b* * *c*のような演算を*1*命令で行うものである．積和演算命令がない場合はまず*b* * *c*を計算して，次に*a*+ を行う必要があり，*2*命令が必要になる．

行列計算のように，積和演算が頻繁に出現するような計算では，積和演算命令を持つプロセッサは持たないプロセッサに比べて理想的には*2*倍程度の速度差が出ることになる．

なお，積和演算命令には大きく分けて*d*=*a* * *b*+*c*のように，*4*つのオペランドに対するものと，*c*=*a* * *b*+*c*のように，*3*つのオペランドに対するもの，そして，*c*=*a* * *b*，*d*=*d*+*e*のように*5*つのオペランドに対するものがある．ここで，*3*つのオペランドに対する積和演算命令は，むしろ「掃き出し」命令と呼ぶべきで，*4*つのオペランドに対する積和演算命令に比べて自由度が低く，最適化の余地が少なくなる傾向にある．

```
int i, j, k;
double A[N,N], B[N,N], C[N,N];
for（k = 0; k < N; k++）{
  for（j = 0; j < N; j++）{
    for（i = 0; i < N; i++）{
      A[j,i] += B[k,i] * C[j,k];
    }
  }
}
```

図-1　行列積の例

```
int i, j, k;
double A[N,N], B[N,N], C[N,N];
for（k = 0; k < N; k += 4）{
  for（j = 0; j < N; j++）{
    for（i = 0; i < N; i++）{
      A[j,i] += B[k,i] * C[j,k]
              + B[k+1,i] * C[j,k+1]
              + B[k+2,i] * C[j,k+2]
              + B[k+3,i] * C[j,k+3];
    }
  }
}
```

図-2　行列積を最適化した例

通常はコンパイラにより積和演算命令の適用が行われるが，ごく一部のアプリケーションではユーザにより積和演算命令向けにチューニングする必要がある．そのような例として，ここでは*FFT*（高速*Fourier*変換）を取り上げることにする．

従来の*2*基底*FFT*カーネルは以下のように表される．

$$u0 = X_R(0)$$
$$v0 = X_I(0)$$
$$r = X_R(1)$$
$$s = X_I(1)$$
$$u1 = r * wr_1 - s * wi_1$$
$$v1 = r * wi_1 + s * wr_1$$
$$Y_R(0) = u0 + u1$$
$$Y_I(0) = v0 + v1$$
$$Y_R(1) = u0 - u1$$
$$Y_I(1) = v0 - v1$$

この*FFT*カーネルを積和演算命令を持つプロセッサで実行する際には，*u1*および*v1*を計算するのに積和演算

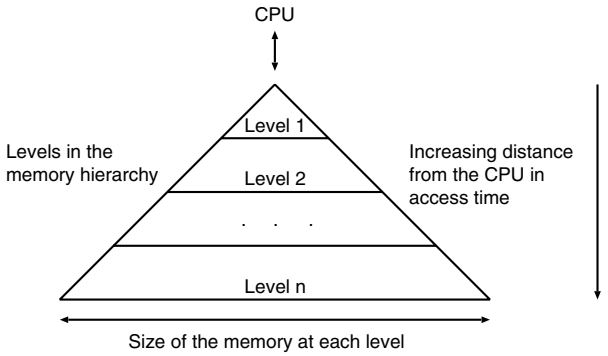

図-3　メモリ階層の例

が合計2回と乗算が合計2回必要であり，$Y_R(0)$，$Y_I(0)$，$Y_R(1)$，$Y_I(1)$ の計算において加算が合計4回必要である．つまり，このFFTカーネルの総演算命令数は8回となる．なお，$u1$や$v1$が

$u1 = 0 + r * wr_1$

$u1 = u1 - s * wi_1$

$v1 = 0 + r * wi_1$

$v1 = v1 - s * wr_1$

のように実行される場合においても，$u1$および$v1$を計算するのに積和演算が合計4回必要となり，$Y_R(0)$，$Y_I(0)$，$Y_R(1)$，$Y_I(1)$ の計算において加算が合計4回必要であるので，FFTカーネルの総演算命令数は8回となる．

Goedeckerの積和演算命令に向いた手法[4]では，$a \neq 0$のときに

$$ax+by \rightarrow a\,(x+\,(b/a)\,y) \qquad (1)$$

の変形が可能であることを利用し，積和演算命令を持つプロセッサにおいてFFTカーネルの演算命令数を削減している．

従来の2基底FFTカーネルでは$wr_1 \neq 0$であるので，式(1)の変形が可能であり，Goedeckerによる積和演算命令に向いた2基底FFTカーネルは次のようになる．

$wi_1 = wi_1/wr_1$

$u0 = X_R(0)$

$v0 = X_I(0)$

$r = X_R(1)$

$s = X_I(1)$

$u1 = r - s * wi_1$

$v1 = r * wi_1 + s$

$Y_R(0) = u0 + u1 * wr_1$

$Y_I(0) = v0 + v1 * wr_1$

$Y_R(1) = u0 - u1 * wr_1$

$Y_I(1) = v0 - v1 * wr_1$

このGoedeckerによる積和演算命令に向いた2基底FFTカーネルを積和演算命令を持つプロセッサで実行する際には，$u1$，$v1$，$Y_R(0)$，$Y_I(0)$，$Y_R(1)$，$Y_I(1)$ を計算するのに積和演算命令が合計6回必要である．つまり，このFFTカーネルの総演算命令数は6回で済むことが分かる．なお，$wi_1 = wi_1/wr_1$の値はあらかじめ計算しておくものとする．

メモリ階層

メモリ階層（memory hierarchy）の例を**図-3**に示す．メモリ階層は記憶域に対するアクセスパターンの局所性（locality）を前提に設計されている．局所性には時間的局所性と空間的局所性があり，前者は，ある一定のアドレスに対するアクセスは，比較的近い時間内に再発するという性質，後者は，ある一定時間内にアクセスされるデータは，比較的近いアドレスに分布するという性質である．

これらの傾向は，事務計算などの非数値計算プログラムには当てはまることが多いが，数値計算プログラムでは一般的ではない．特に大規模な科学技術計算においては，データ参照に時間的局所性がないことが多い．これが，科学技術計算でベクトル型スーパーコンピュータが有利であった大きな理由である．

ベクトル型スーパーコンピュータはデータに関してキャッシュを用いないが，RISCプロセッサのようなスカラプロセッサは性能をキャッシュに深く依存している．したがって，RISCプロセッサで高い性能を得るためにはブロック化を行うなどして，意図的にアクセスパターンに局所性を与えることが必要になる．

基本的なブロック化の手法については前編で述べたので，本稿では説明は省略する．

多くの計算機では，メモリとレジスタの間にキャッシュメモリが入っている．キャッシュメモリはメモリとレジスタの中間的な性質を持ち，メモリ内のデータや機械語命令のうち使用頻度の高い部分がキャッシュメモリに入り，さらに使用頻度の高い部分がレジスタに入る．

キャッシュメモリは通常データが入るデータキャッシュと機械語命令が入る命令キャッシュに分かれている．そのうちチューニングに特に関係があるのはデータキャッシュであるので，本稿ではデータキャッシュについて説明する．

キャッシュミス

メモリ階層のうち，上位階層の小容量高速メモリを，通常キャッシュメモリあるいはキャッシュと呼ぶ．キャッシュメモリの例を**図-4**に示す．

キャッシュについても，階層的に構成することができる．つまり，より高速な1次キャッシュ（Level1 Cache）

の下に，アクセス速度は1次キャッシュよりも遅いが，容量は1次キャッシュよりも大きい2次キャッシュ（Level2 Cache）を設けるのである．このようにすることにより，メモリ階層の局所性をより生かすことが可能になる．さらに3次キャッシュまで備えたプロセッサもある．

演算に必要なデータがキャッシュメモリにないため，メモリからいったんキャッシュメモリに転送せざるを得ないことをキャッシュミスという．キャッシュメモリよりもメモリの方が低速な半導体を使用しているため，データをメモリからキャッシュメモリに転送する時間はキャッシュメモリからレジスタに転送する時間の数倍かかる．したがってパフォーマンスを向上させるためにはキャッシュミスをできるだけ少なくする必要がある．

多階層のキャッシュを用いる場合には，1次キャッシュのヒット時の速度を高速化するようにすること，2次キャッシュのミスの割合を最小化するようにすること，が重要である．

【 計算機とプログラムの相性 】

あらゆるハードウェア，オペレーティングシステム，およびコンパイラに対し，同じように性能が一番よいようなプログラムを作ることが望ましいわけではあるが，これはかなり難しいことである．しかし，ハードウェアやコンパイラの特徴をつかんでアルゴリズムを考えると，かなり広範囲にわたってこれが可能となる．

ベクトルプロセッサ

ベクトルプロセッサとは，ベクトル（vector，1次元配列データ）に対する演算（ベクトル演算）機能を備えたマシンである．ベクトル演算とはたとえば，64要素の浮動小数点ベクトル2個を加算して，結果として64要素のベクトル1個を出力するようなものである．

ベクトルプロセッサにおいて高い性能を得るためには，1．ベクトル化率を高くする，2．ベクトル長を長くする，3．メモリ・アクセスの改善，4．演算器やベクトル・レジスタの有効利用，が重要である．

ハードウェアの限界まで効率向上を行う必要があるときは，ベクトル・レジスタの有効利用によるロード／ストア命令の削減，演算器の並列動作などに注意してプログラム作成を行う必要がある．

2重DOループの外側ループについては，いわゆるループ・アンローリング技法で繰返し回数を半減させて，ロード／ストア命令の数を削減し，また並列演算の数を増加させ効率向上を図ることができる．

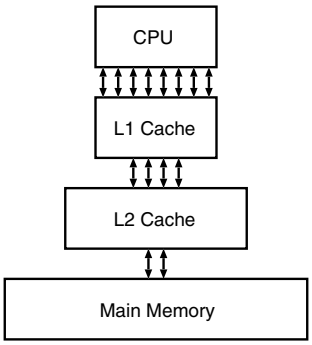

図-4　キャッシュメモリ

RISCプロセッサ

RISCプロセッサにおけるプログラムチューニングに関しては，優れた著書[3]があり，きわめて詳細にチューニング手法が記述されているので，改めてここでは詳しく触れない．

細かい点ではあるが，性能を向上させるいくつかのポイントについて述べる．アラインメントに合わないメモリアクセスを行うと，メモリのアクセス回数が倍増する．多くのプロセッサでは，アラインメントを8バイト境界に合わせることで，適切なメモリアクセスが行われる．もしコンパイラオプションにアラインメントに関するオプションがあれば，使ってみることをお勧めする．

SMP

SMP（Symmetric Multi-Processor）構成の並列計算機においては，バスの競合やロードアンバランスが性能向上を図る上でボトルネックとなる場合が多い．

SMPマシンについては，コンパイラにより自動並列化を行うことができる場合もあり，OpenMPのようにユーザが並列化を指示することができる場合もあるが，並列化ができたとしても性能がそれほど向上しない場合がある．

図-5のプログラムでは，最外側のループ*do j = 1, N2*が並列化されるが，配列Aのアクセスが*N1*要素とびのアクセスになってしまい，メモリアクセスの競合が発生して性能が低下したことがあった．

このように，SMPでは内側ループにおけるメモリアクセスパターンだけでなく，外側ループにおけるメモリアクセスパターンにも注意してコーディングする必要があるので，注意が必要である．

```
    parameter（N1=1024,N2=512）
    dimension A（N1,N2）, B（N1+7,N2）
!$omp parallel do
    do j = 1, N2
     do i = 1, N1
      B（i,j）= A（i,j）
     end do
    end do
```

図-5　メモリ転送の例

【 自動チューニングと人手チューニング 】

自動チューニングを行うライブラリとして，最近 *ATLAS*（*Automatically Tuned Linear Algebra Software*）や *FFTW*（*http://www.fftw.org*）などがある．

*ATLAS*では，オンチップの行列積においてブロック化を行う際のブロック幅の選択や，ループアンローリングの段数を最適化するといった，マシンに特化した部分を自動チューニングしている．

また*FFTW*では，*FFT*における基数（*radix*）の選択および組合せを自動チューニングしており，メモリアクセスの自動チューニングは行っていないのが特徴である．

自動チューニングと人手チューニングの使い分け

自動チューニングと人手チューニングの使い分けについて以下述べることにする．

*ATLAS*や*FFTW*のような自動チューニングを行っているライブラリの多くは，コードそのものを高速にするというよりも，多くのパラメータを変化させて，最適なポイントを見つけ出すという手法で，チューニングを行っている．したがって，いくら自動チューニングを行っているとはいえ，最後は人手チューニングを行う必要が出てくるケースもある．

その*1*つが，コンパイラやプロセッサアーキテクチャに依存した最適化である．*FFTW*では，特定のプロセッサアーキテクチャに特化したオプションを*configure*コマンドを実行する際に指定すると，コードの一部が人手チューニングしたものと置き換わるようになっており，これは自動チューニングと人手チューニングを相補っているよい例といえる．

ユーザレベルで，自動チューニングと人手チューニングを使い分ける際に注意する点としては，

1. コードに手を入れる前に，さまざまなコンパイラオプションを試してみる．たとえば，ターゲットプロセッサの指定などを行うことで，性能が大きく向上する場合がある．
2. コンパイラの最適化アルゴリズムが，ローカルな最適解にはまって抜け出せない場合，人手チューニングを加えることでグローバルな最適解にたどり着けることもある．ただし，逆のケースも考えられるので注意が必要である．
3. コンパイラは，静的な情報しか利用できない．したがって，できる最適化に制約がつく．たとえば，動的に割り当てられる配列については，安全な最適化しか行わない．また，ポインタによるデータ参照は，まったく最適化されない．プログラム中に，このような動的な部分があるならば，その部分は人手最適化のポイントになる．

上記に述べたように，自動チューニングは必ずしも万能ではなく，最終的には人手チューニングが必要になる場合もあることをユーザは認識する必要がある．

【 おわりに 】

プログラムのチューニングを行うにあたって，理想的な性能を得る上ではアセンブラによりプログラムを記述するのがベストであるが，手間が掛かる上に，コーディングミスの可能性も高くなる．さらに，最近のコンパイラの最適化技術の向上により，よほどの場合でない限り，アセンブラを使うことは現実的ではない．

したがって，コンパイラを「よりよいアセンブリコードを出力するためのトランスレータ」として使い，アセンブリ出力を見ながら，高級言語で書かれたプログラムを少しずつチューニングし，性能の向上をその都度確認するといった方法が現実的である．上記のような手法でチューニングを行った場合，コンパイラのバージョンが変わっただけで最適なコードが得られなくなってしまうことがあるので，注意が必要である．

最後に最適化に関して，非常に重要なことを述べることにする．いくら最適化を行っても，結果が不正になるような最適化は行ってはならない．特に最適化には副作用がつきものであるので，演算順序を変更するような最適化を行う際には細心の注意が必要である．つまり，正しい結果を得ることが一番重要であり，次に可読性が重要である．高速性は，その次に重要だということを認識する必要がある．

また，美しく，自然の摂理に逆らわないようにコーディングしたプログラムは速いことが多いというのが，筆者のこれまでの経験から得られた感想である．

参考文献

1) 中田育男: コンパイラの構成と最適化, 朝倉書店 (*1999*).
2) 中谷登志男: *VLIW*計算機のためのコンパイラ技術, 情報処理, *Vol.31, No.6, pp.763-772* (*June 1990*).
3) 寒川　光: *RISC*超高速化プログラミング技法, 共立出版 (*1995*).
4) *Goedecker, S.: Fast Radix 2, 3, 4, and 5 Kernels for Fast Fourier Transformations on Computers with Overlapping Multiply-Add Instructions, SIAM J. Sci. Comput., Vol.18, pp.1605-1611* (*1997*).

（平成*13*年*10*月*22*日受付）