並列コンピュータ AzusA の高速化技法

―― メモリアクセスの効率化 ――

左近　彰一
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

概要

並列コンピュータ TX7/AzusA（以下、AzusA という）は、１つのプログラムを最大 16 個の CPU で分担して実行する並列処理によって高速化を実現していますが、高速化のためには単一 CPU で実行するときの性能をできるかぎり向上させておくことが重要です。そこで、本稿では AzusA の単一 CPU 上での実行性能向上（チューニング）のための、主としてメモリアクセスの削減による高速化技法を紹介します。例題として、行列積のプログラムの高速化方法を説明します。

1. AzusA のメモリとキャッシュ

AzusA の CPU は、メモリとの間に L1 キャッシュ、L2 キャッシュ、L3 キャッシュの 3 階層のキャッシュを持っています（図１）。[1][2]

各々の容量と性能を表１に示します。浮動小数点データは L1 キャッシュを用いませんので、L2 キャッシュが最も CPU に近い高速なキャッシュとなります。

AzusA 1CPU の倍精度浮動小数点演算性能は、最大 3.2GFLOPS ですから、倍精度浮動小数点の計算が連続的に行われる場合、L2 キャッシュにデータがあれば、１つのデータに対して１演算を行うことによって、データ供給のバンド幅と演算の速度がつり合います（3.2G[Flops]*8B=25.6GB）。

しかし、同様の計算により、データが L3 キャッシュにある場合は、１つの倍精度要

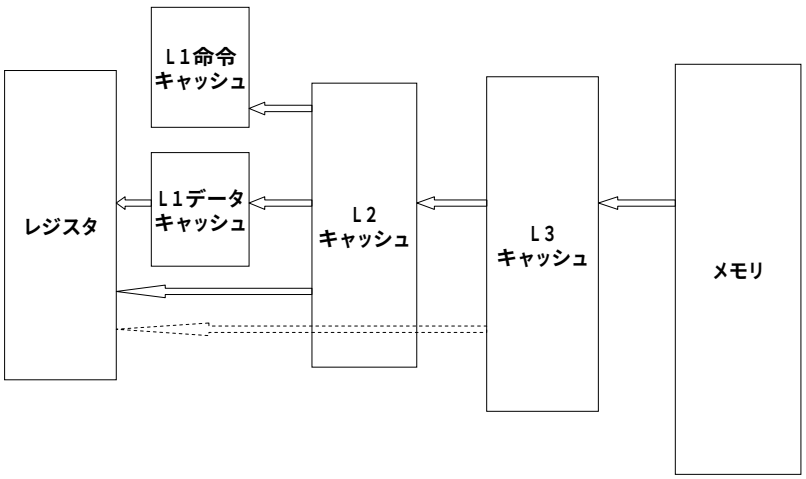

図１　キャッシュメモリの階層

素に対して 2 演算、メモリの場合は 12 演算を行わないと、データ供給スピードと演算性能がつり合いません。逆に言うと、L3 キャッシュやメモリにデータがある場合、1 データに対して上記の数程度の演算を行わない場合、データ供給ネックとなり、最大演算性能を出すことができません。またレーテンシ（要求してからデータが届くまでの時間）も L2→L3→メモリへ行くほど長くなっています。

表１　キャッシュとメモリの特性

	サイズ	レーテンシ (サイクル数)	バンド幅	バンド幅：倍精度演算性能比
L1 データ	16KB	2	12.8GB/s	-
L2	96KB	6-9	25.6GB/s	1 : 1
L3	4MB	21-24	12.8GB/s	1 : 2
メモリ		<160-240	2.1GB/s	1 : 12

図 2 は積和演算 a(i)=a(i)*x+y の性能をループ長を変えながら測定したものです。このループの外側にダミーのループを入れて計測し、キャッシュの効果を見ています。図で分かるように、データが L2 キャッシュに乗っている間は高い性能が出ていますが、L2 キャッシュを外れて L3 キャッシュの領域に入ると急に性能が低下し、さらに L3 キャッシュを外れると著しく性能が悪くなってしまいます。

```
do k=1,kk   !ダミーのループ
  do i=1,n   !計測ループ。n=ループ長
    a(i)= a(i) *x + y
  enddo
enddo
```

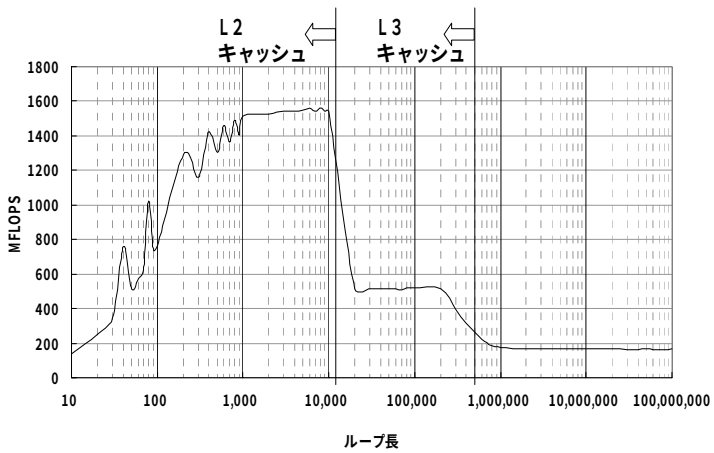

図２　積和演算の性能

これから分かるように、AzusA においては、できるだけメモリにデータを取りにいかずに、CPU のレジスタやキャッシュ上にデータを保持し、それを再利用するようにすることが、性能向上に重要です。

2. メモリ関連の高速化技法

ここでは、メモリ関連の高速化技法のいくつかを紹介します。ここでは一般的に性能向上に有効と考えられる手法について述べますが、ここで述べた方法も万能ではなく、場合によっては単純に適用すると、逆の効果が発生して性能が向上しないこともあります。実際のプログラムはさまざまであり、各プログラムの特性に応じて対応する必要があります。実際のプログラムに適用した場合の例は 3.行列積プログラムを見てください。

2.1. メモリアクセスそのものを減らす

高速化のために最も効果的なのは、メモリアクセスそのものを減らしてしまうことです。例えば、以下のようなプログラムでは、内側ループにある配列参照が、外側ループのインデックス変数に依存していないので、外側ループのアンローリングを行うことによって、B(J)をレジスタ中などに保持し、メモリからのロードを減らすことができます。

```
do i=1,n                          do i=1,n-3,4
 do j=1,m                          do j=1,m
   a(j,i)=a(j,i)*b(j)      ⇨        a(j,i)=a(j,i)*b(j)
   ...                               a(j,i+1)=a(j,i+1)*b(j)
                                     a(j,i+2)=a(j,i+2)*b(j)
                                     a(j,i+3)=a(j,i+3)*b(j)
                                     ...
```

上の例では４段のアンロールを行っています。アンロール段数を多くすれば、メモリアクセスは減らせますが、あまり多くするとレジスタが足りなくなってソフトウェアパイプライニングが出来なくなり逆に性能が低下してしまいます。したがって、性能が向上するアンロール段数には限界があります。手作業でアンローリングを行う場合は、コンパイラの最適化レポートを見ながら、ソフトウェアパイプライニングが行われる範囲内でアンロールする必要があります。

このプログラムの場合、以下のようにループを入れ換えることによってもメモリアクセスを減らすことができます。これは、ループ入れ換えにより B(J)がループ内で一定となり、ループの外で一度だけロードすれば良いようになるからです。

```
do j=1,m
 do i=1,n
    a(j,i)=a(j,i)*b(j)
```

...

しかし、この例の場合は、ループ入れ換えを行うと、配列ＡのＩに関するアクセスが連続でなくなるので、高速化の効果が減ってしまいます。従って、この場合はアンローリングを行うほうが有効と考えられます。

このような外側ループのアンローリングあるいはループの入れ換えは、ある程度はコンパイラが自動的に行います。[3] どのような変形を行ったかは、コンパイルオプション-opt_report を指定したときに出力される最適化リストに表示されます。コンパイラが判断できなかった場合や、人手によるチューニングが効果を発揮すると思われる場合に手作業によるチューニングを検討することになります。

2.2. データをキャッシュに置いて再利用する

プログラム中で同一のデータを何度も使う場合、それをキャッシュ上に保持しておくのは性能を出す上で重要です。キャッシュの大きさには限りがあるので、データをキャッシュサイズに収まるようにしなければなりません。そのため、例えば１つのループをキャッシュのサイズのループとそれ以外の部分の２重ループに分解し、さらに外側ループとの入れ換えを行うことによって、同一データを小さい範囲で再利用する方法があります。この技法をキャッシュブロッキングと言います。

```
DO I=1,N                          DO J2=1,M,JB
   DO J=1,M                          DO I=1,N
    ... = ... *B(J)                    DO J=J2,min(J2+JB-1,M)
                                       ... = ... *B(J)
```

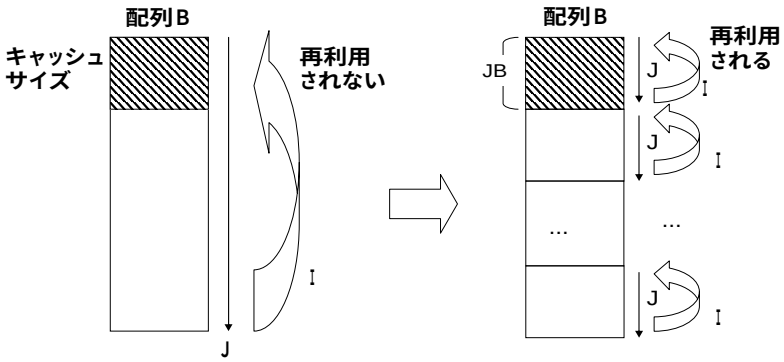

図３　キャッシュブロッキング

2.3. データを連続アドレスでアクセスする

Ｌ2キャッシュはラインサイズが64バイト(倍精度浮動小数点データ8要素)です。キャッシュとメモリのやり取りはこのライン単位で行われるため、配列データを連続アドレスでデータをアクセスすると、キャッシュ上のデータが有効に使われ、非常に

効率が良いです。逆に、配列のとびアクセスを行うと、キャッシュにロードされたデータの内、実際に使うのはそのうち一部のデータとなるため効率が悪くなります。

データを連続アドレスでアクセスするには、ループの入れ換えを行って、一次元目の添字で回るようにする方法があります。また、同一データを何度も参照する場合、作業配列を確保してそこに連続になるようにデータを詰めなおす方法があります。データの参照回数が多ければ詰めなおしのオーバーヘッドを考えても有利な場合があります。

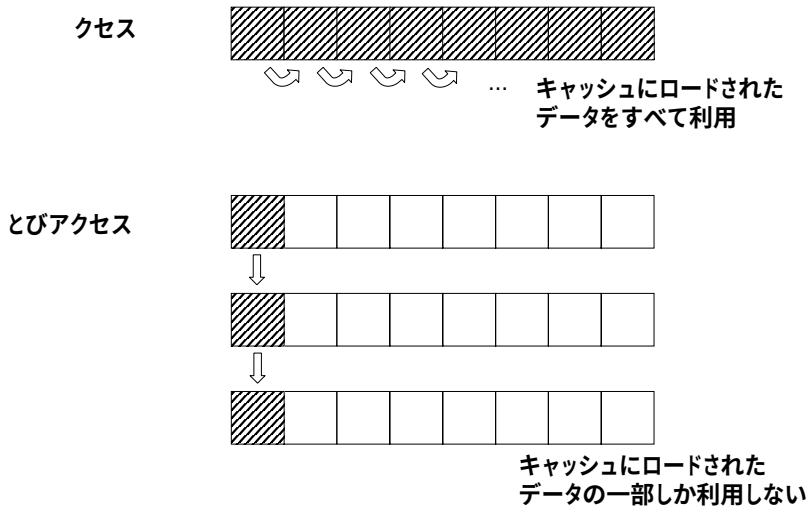

図４　連続アクセスと、とびアクセス

3. 行列積プログラム

次の行列積のプログラムを例題にいくつかのチューニングを適用していきたいと思います(配列 a,b は、一次元目を+3 するパディングを行っています[4])。

```
parameter ( n=2048 )
double precision a(n+3,n),b(n+3,n),c(n,n)
(配列の初期値の設定は省略)
do j=1,n
  do k=1,n
    do i=1,n
      a(i,j)=a(i,j)+b(i,k)*c(k,j)
    end do
  end do
end do
```

3.1. ループの入れ換え

このプログラムでは、ループをどのようにも入れ換えることができますが、性能上はどの順番にするのがいいのでしょうか？　メモリアクセスを数えてみると以下のようになります。

表２　最内側ループ変数の選択と配列アクセスの特性

最内側ループ	メモリへのストア	メモリからのロード	
i	a のストア（連続）	a のロード(連続)	b のロード(連続)
j	a のストア（とび）	a のロード(とび)	c のロード(とび)
k	―	b のロード(とび)	c のロード(連続)

i や j のループを内側にする（積和型）よりも、k のループを内側にする（内積型）の方がストアが不要であり、メモリアクセスが少ないことが分かります。従って k のループを内側にし、その外側を i のループにします。

このループでは、ストアするもの(a(i,j))と同じものをロードしているためその影響は顕著ではありませんが、AzusA では、ある要素をロードせずにストアする場合、キャッシュライン上にデータを持ってきてからストアを行うため、ストアは 2 回のメモリアクセスがあるとしなければなりません。

k のループを内側にすると、配列 b は 2 次元目でアクセスするため、連続ではなく、とびアクセスとなりキャッシュの利用面では不利になります。これは後で対策を考えます。

k のループを最内側にした結果は以下のようになります。

```
do j=1,n
  do i=1,n
    do k=1,n
      a(i,j)=a(i,j)+b(i,k)*c(k,j)
    end do
  end do
end do
```

3.2. 外側ループアンローリングによるメモリアクセスの削減

上記のループを観察すると、配列 c(k,j)は i に依存しないので、i のループをアンローリングすると c(k,j)のメモリアクセスが共通化され削減されることが分かります。また、同様に b(i,k)は j に依存しないので、j のループでアンローリングするとメモリアクセスが削減されます。

アンローリング段数をいくらにするかですが、段数を多くすればするほどメモリアクセスを削減できるのですが、先ほど述べたように、あまり多くしすぎると、ループ内の要素が増えるために、ソフトウェアパイプライニング(SWP)を行うためのレジス

タが足りなくなり、SWP ができなくなって、かえって遅くなってしまいます。コンパイラの SWP に関するレポート(-opt_report)を見ながら、SWP が行われる最も多い段数を実験的に調べると、このループでは i に関して 4 段、j に関して 5 段のアンローリングまで行えることがわかりました。この結果、以下のようなループとなりました。

```
do j=1,n-4,5
  do i=1,n-3,4
    do k=1,n
      a(i,j)=a(i,j)+b(i,k)*c(k,j)
      a(i+1,j)=a(i+1,j)+b(i+1,k)*c(k,j)
      a(i+2,j)=a(i+2,j)+b(i+2,k)*c(k,j)
      a(i+3,j)=a(i+3,j)+b(i+3,k)*c(k,j)
      a(i,j+1)=a(i,j+1)+b(i,k)*c(k,j+1)
      a(i+1,j+1)=a(i+1,j+1)+b(i+1,k)*c(k,j+1)
      a(i+2,j+1)=a(i+2,j+1)+b(i+2,k)*c(k,j+1)
      a(i+3,j+1)=a(i+3,j+1)+b(i+3,k)*c(k,j+1)
      a(i,j+2)=a(i,j+2)+b(i,k)*c(k,j+2)
      a(i+1,j+2)=a(i+1,j+2)+b(i+1,k)*c(k,j+2)
      a(i+2,j+2)=a(i+2,j+2)+b(i+2,k)*c(k,j+2)
      a(i+3,j+2)=a(i+3,j+2)+b(i+3,k)*c(k,j+2)
      a(i,j+3)=a(i,j+3)+b(i,k)*c(k,j+3)
      a(i+1,j+3)=a(i+1,j+3)+b(i+1,k)*c(k,j+3)
      a(i+2,j+3)=a(i+2,j+3)+b(i+2,k)*c(k,j+3)
      a(i+3,j+3)=a(i+3,j+3)+b(i+3,k)*c(k,j+3)
      a(i,j+4)=a(i,j+4)+b(i,k)*c(k,j+4)
      a(i+1,j+4)=a(i+1,j+4)+b(i+1,k)*c(k,j+4)
      a(i+2,j+4)=a(i+2,j+4)+b(i+2,k)*c(k,j+4)
      a(i+3,j+4)=a(i+3,j+4)+b(i+3,k)*c(k,j+4)
    end do
  end do
end do
```

(n を 4,5 で割った余りがある場合のループ処理は省略)

これによって、１メモリアクセス当たりの演算数は、2/2=１から 40/9=4.4 になりました。ただし、このプログラムでは-03 オプションでコンパイルすると、コンパイラが自動的に i に関して 4 段、j に関して 4 段のアンローリングを行ってくれます。上記の手動アンローリングはループが複雑でコンパイラが自動的にアンローリングできない場合向けです。

3.3. キャッシュブロッキング

本ループは、b(i,k)がjに依存しないため、i,kに関してブロッキングを行えば、b(i,k)をキャッシュに乗せることができます。ここではL2キャッシュに乗せるため、ブロッキングファクタをib=108, kb=108とします。（108*108*8バイト=93K<96Kバイト）。

```
do is=1,n,ib
  do ks=1,n,kb
   do j=1,n-4,5
    do i=is,min(is+ib-1,n)-3,4
      do k= ks,min(ks+kb-1,n)
        ループ内は前と同じ
      end do
    end do
  end do
 end do
end do
```

3.4. 配列アクセスの連続化

配列bはとびアクセスになっているため、これを連続アクセスになるようにします。作業配列bbを用意し（サイズはib*kb）、配列bを転置してコピーします。これによって、キャッシュへのロード時の性能がさらに改善されます。配列bbにいったんコピーするオーバーヘッドはnが大きければ無視できます。

```
 do is=1,n,ib
  do ks=1,n,kb
    do i=is,min(is+ib-1,n)
      do k= ks,min(ks+kb-1,n)
        bb(k-ks+1,i-is+1)=b(i,k)
      end do
     end do
   do j=1,n-4,5
    do i=is,min(is+ib-1,n)-3,4
      do k= ks,min(ks+kb-1,n)
        a(i,j)=a(i,j)+bb(k-ks+1,i-is+1)*c(k,j)
        ...
         (上記の文でiを4段、jを5段アンローリング)
      end do
    end do
```

```
    end do
  end do
end do
```

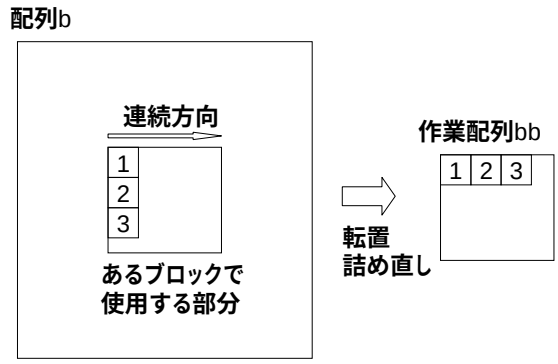

図５　配列の詰め直し

3.5. プリフェッチの最適化

AzusA では、メモリロードのレーテンシを隠すためにプリフェッチの生成が有効です。プリフェッチとは、実際のメモリアクセスの前にキャッシュにデータを持ってきておくことです。プリフェッチは、-03 オプション指定時にはコンパイラが自動的に行いますが、コンパイラはその配列が現れる内側ループ内にプリフェッチを行おうとします。しかし、たとえば配列 c のアクセスに関しては i に依存しないので、i のループの外側でプリフェッチを行うことによって、プリフェッチ命令実行のオーバーヘッドを削減できます。手動でプリフェッチを行うには、lfetch 組込みサブルーチンを呼び出します。

```
do is=1,n,ib
  do ks=1,n,kb
    do i=is,min(is+ib-1,n)
      do k= ks,min(ks+kb-1,n)
        bb(k-ks+1,i-is+1)=b(i,k)
      end do
    end do
   do j=1,n-4,5
      do k=ks,min(ks+kb-1,n),8
        call lfetch(c(k,j))
        call lfetch(c(k,j+1))
```

```
          call lfetch(c(k,j+2))
          call lfetch(c(k,j+3))
          call lfetch(c(k,j+4))
        end do
      do i=is,min(is+ib-1,n)-3,4
        call lfetch(a(i+8,j))
        call lfetch(a(i+8,j+1))
        call lfetch(a(i+8,j+2))
        call lfetch(a(i+8,j+3))
        call lfetch(a(i+8,j+4))
        do k= ks,min(ks+kb-1,n)
          a(i,j)=a(i,j)+bb(k-ks+1,i-is+1)*c(k,j)
          ...
            (上記の文を使って i を 4 段、j を 5 段アンローリング)
        end do
      end do
    end do
  end do
end do
```

c(k,j)のプリフェッチで k を 8 とびに回しているのは、L2 キャッシュのラインサイズが 64 バイトであるので、1 ライン当たり 1 要素のプリフェッチ（これで、そのライン全体が取ってこられる）で十分だからです。なお、手動でプリフェッチを行った場合、コンパイラがプリフェッチを余分に生成しないように-02 でコンパイルします。また、-02 でコンパイルすると、a に関しても、手動でプリフェッチを行う必要が出てくるので、そのための lfetch 呼び出しコードも入れておきます。

4. 性能測定結果

上記のチューニングを行った行列積プログラムの性能を AzusA で測定しました。行列のサイズは 2048*2048 とし、L2 および L3 キャッシュサイズよりかなり大きなものとしました。コンパイラの最適化レベルは既定値（-02）です。

オリジナルプログラムに対して、まずループ入れ換えを行うと、配列アクセスが連続でなくなるので性能が一時的に大きく低下します。しかし、その後、各種の最適化を適用することにより、1741MFLOPS を達成しました。これは、図 2 の積和演算のグラフからみても、データが L2 キャッシュに載った時の性能と同じレベルであり、最大性能を達成できていると言えます。

なお、オリジナルプログラムで-03 オプションを指定するとコンパイラによって外側ループのアンロールとプリフェッチが行われているので、約 1000MFLOPS という性

能が出ます。

表３　チューニング性能測定結果

	行ったチューニング	時間（秒）	性能（MFLOPS）
1	オリジナル	325.5	57
2	ループ入れ換え	497.1	34
3	2 + 外側ループアンロール	76.0	226
4	3 + キャッシュブロッキング	33.3	514
5	4 + 配列連続化	11.1	1542
6	5 + プリフェッチの最適化	9.8	1741
―	オリジナル(-03でコンパイル)	17.0	1009

5. おわりに

メモリアクセスの削減の観点からAzusA向けプログラムのチューニング方法を説明しました。AzusA ではコンパイラの最適化機能を利用して自動的に性能を出すことができますが、ここで説明したような技法を適用することにより、さらに性能向上できる場合があります。本稿がチューニングの参考になれば幸いです。その他さまざまなチューニング技法は、参考文献[5]に記述されています。

なお、今回はシングルCPUでのチューニングを説明いたしましたが、並列処理向けのチューニング技法については、別の機会にご紹介したいと思います。

参考文献

[1] 鈴木 重信, 高木 均, 横山 淳, “汎用コンピュータシステム「TX7/AzusA」”, SENAC Vol.34, No.3, 2001

[2] Intel株式会社, ”Itanium(R)プロセッサ・マイクロアーキテクチャ・リファレンス” 資料番号: 245473J-002, 2000
(ftp://download.intel.co.jp/jp/developer/jpdoc/24547302-s_j.pdf)

[3] 山本 秀喜, 左近 彰一,“TX7/AzusA Fortran”, SENAC Vol.34, No.3, 2001

[4] 左近 彰一,“TX7/AzusA Fortranプログラムの高速化技法”, SENAC Vol.35, No.2,

2002

[5] 寒川　光, “RISC超高速化プログラミング技法”, 共立出版, 1995 (ISBN4-320-02750-7)