数値解析 43−1
(1992. 10. 15)

LU分解のブロック化と計算順序

寒川 光
日本アイ・ビー・エム(株)

数値解析の高速計算法は計算機システムの個性に依存する面が非常に強い。そこでこの問題を数値計算の分野だけからでなく、数値計算と計算機システム(コンパイラーと計算機構造)の両分野に跨がった視野から眺めてみたい。計算機ハードウェアの演算回路は年々高速化の一途を辿っているが、計算機の内部でのデータ移動にかかる時間は相対的に長くなっている。この計算機システムの動向を考えると、LU分解のような古典的なアルゴリズムに対しても、データ移動量を制御しやすい形のアルゴリズムに変形しておくことが重要である。データ移動量の制御は計算順序を変えることで達成されるので、LU分解においてどのような計算順序が可能かという問題を考える。

Blocking algorithms and computational ordering for LU decomposition

Hikaru Samukawa
IBM Japan, Ltd.
18-24, Tsukiji 7-chome, Chuo-ku, Tokyo 104, Japan

Since the methods to perform numerical analysis in high-speed depend on the characteristics of computer systems, a discussion on this matter should be discussed not only from the view of numerical computation field but from the combined view of numerical computation field and computer systems field (compiler, architecture, hardware technology). Logic circuit becomes faster. On the other hand, time required to move data inside the computer becomes relatively slower than arithmetic computations. Considering this hardware trend, it is important to modify LU decomposition algorithms so that data transfer is easier to be controled. Since the data transfer control is achieved by selecting computational ordering, the possibilities of changing computational ordering are discussed.

1．はじめに

数値計算の分野では新しいタイプの計算機が登場すると、既存のプログラムやアルゴリズムをその計算機の高速性能がひき出せる形に改訂する作業が行われる。たとえばベクトル計算機が登場すると、連立1次方程式の直接解法のベクトル化アルゴリズムが研究される。並列計算機しかり，スーパースカラー計算機しかりである。計算機の種類は多く、しかもそれぞれが個性豊かなので、ガウスの消去法と呼ばれる古典的なアルゴリズムに対しても数多くのバリエーションが生み出されてゆくことになる。この現象を数値計算法の観点だけから眺めると何とも鬱陶しく感じられるかもしれない。問題の本質は、数値計算の分野と計算機システム（計算機構造とコンパイラー）の分野の間から出発している。そこでこの問題を両分野を合わせた視野で眺め直し、将来登場するであろう新しいタイプの計算機の搭載する技術を、数値計算アルゴリズムが受け入れるための技術を模索する立場で考えてみたい。

新しいタイプの計算機のために生み出された数値計算アルゴリズムのバリエーションはいづれも、

- プログラムの文脈に「並行性」（演算を同時に実行しても良い性質）を追加する
- 計算機内部で発生する「データ移動量」を削減する

の2つの目標の一方（または両方）を狙っている。数値計算プログラムの計算時間は乗算回数を計算量の尺度とする方法が一般的であるが、その背景には初期の計算機では乗算命令の実行時間が他の命令の実行時間に比べて非常に長かった、という経緯がある。とくに乗算回路に Wallace のアルゴリズムが導入される以前には、乗算命令の実行時間はビット長に比例して長くなった。この時代の計算機においては、記憶域のデータを参照するための時間は、倍精度の浮動小数点乗算に比べれば無視しうるものだった。ガウスの消去法の計算量が $n^3/3$ 回の乗加算であり，計算時間が n^3 に比例すると見積ることはごく自然だったのである。しかし最新の計算機ではこの事情はかなり異なっている。乗算回路はサブミクロンのスケールで微細加工された論理回路によって高速処理されるからである。しかもこの回路はパイプライン化され多重化される。一方、記憶域とプロセッサーの間は、依然としてピンのついた銅線で結合されている。この物理的な大きさの変化が、演算速度とデータ移動速度のバランスを変えてしまったのである。この計算機ハードウェアの変化が、高速計算のためのアルゴリズムに改訂を強いるのである。すなわち、

ハードウェア		プログラミング技法
パイプライン化，多重化	⟶	「並行性」の追加
演算速度とデータ移動速度の比	⟶	「データ移動量」の削減

となる。高速計算のために昔は計算量を削減したものであるが、現在は並行性を付与することとデータ移動量を削減することを狙う場合の方が多くなっている。簡単な例としてリバモアループを示す。このループは 960 回の乗算と 960 回の加算を行うが、共通な演算をまとめることで、729 回の乗算と 726 回の加算に減らすことができる。

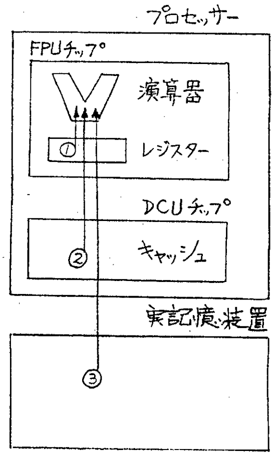
```
do i=1,120
 x(i)=u(i)+r*(z(i)+r*y(i))
     +t*(u(i+3)+r*(u(i+2)+r*u(i+1)))
     +t*(u(i+6)+r*(u(i+5)+r*u(i+4))))
enddo
```

⇨

```
do i=2,124
 a(i)=r*(u(i+1)+r*u(i))
enddo
do i=2,124
 b(i)=t*(u(i+1)+a(i))
enddo
do i=1,120
 x(i)=u(i)+r*(z(i)+r*y(i))
     +b(i+1)+t*b(i+4)
enddo
```

この変形は古典的な計算機では効果をあげるが、RISCタイプのスーパースカラー計算機であるIBMのRISC System/6000*（RS/6000）では逆効果である（相当に遅くなる）。この理由は、データ移動量がストア命令の増加で増えることと、乗算と加算のチェイニングが断ち切られて単独で実行しなくてはならない乗算と加算が現われる（並行性を失なう）ことにある。

並行性はFortranプログラムからも読みとることができるが、データ移動量はレジスターとかキャッシュがFortranプログラム（またはアーキテクチャー）から透過なので把えにくい。右図にRS/6000のハードウェア構成の一部を、データの流通の観点から示した。RS/6000で用いられるPOWERアーキテクチャーは、算術演算命令のオペランドはすべてレジスターとしている。したがってレジスターに存在するデータに対しては演算命令だけ①で計算できるが、レジスターになければロード命令と演算命令②を実行しなくてはならない。ロード命令の実行に際しては、アクセスされるデータがキャッシュに残っていないとキャッシュミスを発生して、実記憶装置からキャッシュへのステージングと呼ばれる複雑な処理を伴なってレジスターへロードされる③。このように参照されるデータがどこにあるかということがデータ移動量を左右する。同じ計算量であっても、演算器に近いところに残っているデータで計算が進められるような計算順序にすることで計算時間を短縮できる。行列計算のように少数の比較的単純な実行文でループが構成されるプログラムでは、レジスターへのデータ移動量はループの構成順序で、キャッシュへのデータ移動量はキャッシュブロック化と呼ばれる小行列を単位とする計算方法で制御することができる[1]。どちらの方法も「記憶域と演算器の間に一定の容量の作業域を設けたとき、作業域へのデータ移動量を最小化するような計算順序を選ぶ」ことと表現できる。なお、POWERアーキテクチャーは4オペランドの乗加算命令と32個の浮動小数点レジスターを定義しているので、レジスター群をひとつの作業域と考えやすい。



* RISC System/6000はIBM Corp.（米国）の商標である。

2．LU分解におけるループの順序

```
Generic matrix-matrix multiplication
do __=1,__
 do __=1,__
  do __=1,__
   C(i,j)=C(i,j)+a(i,k)*b(k,j)
```

行列行列積のプログラムでは i, j, k の3つの添字に関するループをどのような順序ででも構成可能である。すなわち，右の下線部には i, j, k の6つの組合せによる6通りのプログラムが可能である。i と j は行列Cの要素の位置（行と列）を示すが、kは左辺には現われない。本報告ではこれをテンソル数学にならってダミー添字(dummy index)と呼ぶ。スカラー計算機でレジスターへのデータ移動量を少なくするのは，ダミー添字が最内側に用いられた時である。これはレジスターへの累加(accumulation)が達成されることで中間結果のストアが省けるからである。ベクトル・レジスター型のベクトル計算機においても、ベクトル命令の出力がベクトル・レジスターに累加できるループ構成（外側ループ・ストリップマイニング）とした時、ベクトル・レジスターへのデータ移動量が少なくなる。

2.1　LU分解の式と計算順序の制約

$$\begin{cases} u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj} & \cdots\cdots(i \leq j \text{の要素}) \\ l_{ij} = \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj}\right) / u_{jj} & \cdots(i > j \text{の要素}) \end{cases}$$

$$\left\{\begin{array}{l} \text{do } k=1, n-1 \\ a_{ij}^{(k)} = a_{ij}^{(k-1)} - l_{ik} u_{kj} \quad \left(= a_{ij}^{(k-1)} - a_{ik}^{(k-1)} a_{kj}^{(k-1)}\right), \\ \qquad \text{上三角}(i \leq j)\text{の要素} \quad u_{ij} = a_{ij}^{(i-1)} \\ \qquad \text{下三角}(i > j)\text{の要素} \quad l_{ij} = a_{ij}^{(j-1)} / u_{jj} \\ \quad \text{ここで } a_{ij}^{(0)} = a_{ij} \text{ である。} \end{array}\right.$$

LU分解の式を2通り示す。上はこの式に従ってUの1行、Lの1列，Uの2行，‥‥の順に交互に計算すればクラウト分解になる式である。下は中間結果を意識した式で、kのループを外側に置いたまま内側に i と j のループを書けばガウスの消去法になる。

3つの添字 i, j, k に関して、行列行列積のようなループ構成が可能かどうかを考えてみたい。行列行列積ではコンパイラーが6つの組合せから最適なものを選んでくれそうなぐらい単純であったが，LU分解では、計算された要素（u_{ij} や l_{ij}）がその後のステップで乗数として用いられるので複雑である。たとえば2行目の要素は

$$(u_{2j} =)\ a_{2j}^{(1)} = a_{2j}^{(0)} + r_{21} \cdot a_{1j}^{(0)} \qquad \left(\text{ただし } r_{21} = -\frac{a_{21}^{(0)}}{a_{11}^{(0)}}\right)$$

として計算されるが，次のステップ（$k=2$）では

$$a_{ij}^{(2)} = a_{ij}^{(1)} + r_{i2} \cdot \underline{a_{2j}^{(1)}} \quad \left(= a_{ij}^{(1)} + r_{i2} \cdot \left(a_{2j}^{(0)} + r_{21} \cdot a_{1j}^{(0)}\right)\right)$$

となる。下線の項 $a_{2j}^{(1)}$ が $a_{2j}^{(0)}$ であってはならないのであるが、これは「下三角の j 列の要素を計算する時は、上三角の j 列の要素の計算が完了していなくてはならない」ことを意味している。上三角の i 列の要素を計算する時も同様に、下三角の i 列の計算が完了していなくてはならない。クラウト分解はこの計算順序の制約を端的に表わした計算順序になっている。

クラウト分解

```
do ij = 1,n
  i = ij
  do j = i,n
    do k = 1,i-1
      a(i,j) = a(i,j) - a(i,k)*a(k,j)
  ......
  j = ij
  do i = j,n
    do k = 1,j-1
      a(i,j) = a(i,j) - a(i,k)*a(k,j)
```

ガウス消去法

```
do k = 1,n-1
  .....
  do j = k+1,n
    do i = k+1,n
      a(i,j) = a(i,j) - a(i,k)*a(k,j)
```

Generic Gaussian elimination

```
do _ = _,_
  do _ = _,_
    do _ = _,_
      a(i,j) = a(i,j) - (a(i,k)*a(k,j))/a(k,k)
```

2.2　ループの順序（loop ordering）

LU分解のループの順序を論文[2)]では"Generic Gaussian elimination algorithm"と呼んで右のように記述し論じている。ここでは行列行列積における6通りのループ構成と同様の組合せが論じられている。しかしクラウト分解のように上三角の要素と下三角の要素を計算する時でループの順序が異なるものは、行列行列積のような6通りには入ってこない。そこで上三角のループの順序と下三角のループの順序を独立に扱えるようにしたい。クラウト分解を ijk/jik型と、'/'の前に上三角のループの順序を外側から、後に下三角のループの順序を外側から記述する。ガウスの消去法は kji/kji型となる。このように上下を独立に構成すると 6×6＝36通りの組合せが候補に上ってくる。

i) 外側ループが上下で一致するもの

i-i) 上三角、下三角が同じもの：6通り

i-ii) 上三角、下三角が異なるもの：6通り

ixx/ixx，jxx/jxx，kxx/kxx

ii) 外側ループが上下で異なるもの：4×6＝24通り

ixx/jxx，ixx/kxx，kxx/jxx，jxx/ixx，<u>jxx/kxx</u>，<u>kxx/ixx</u>

この中で前節で述べた計算順序の制約から、正しく計算できないものは下線を引いた8通りである。上下独立にループ構成すると28通りのプログラムができる。

Genericな表現だけ比べると行列行列積とLU分解は一見よく似ているので、言語仕様に行列がうまく取入れられればコンパイラーがあとはうまく最適化してくれるのではないか、との期待を抱きそうになる。しかしコンパイラーから見ると、行列行列積では両辺に現われる変数（C）は被加数でしかないが、LU分解では乗数、被乗数としても現われるし、またループ添字の動く範囲が外側のループ添字で記述されるという点で相当複雑になっている。このような理由で、たとえ行列（三角行列）がうまく言語仕様に組入れられても、計算順序の制約を適切な方法でコンパイラーに告げて、コンパイラーはそれから28通りのループの順序の組合せを比べて、対象となる計算機に最適なものを選び出すのは難かしいように思う。

2．3　基本変換(elementary transformation)と計算順序

$$\left\lceil\begin{array}{l}\text{do } k=1,n-1\\ A^{(k)}=R_k A^{(k-1)}\end{array}\right.$$

ただし

$$R_k=\begin{bmatrix}1 & & & & & \\ & \ddots & & & & \\ & & 1 & & & \\ & & r_{k+1,k} & \ddots & & \\ & & r_{k+2,k} & & \ddots & \\ & & \vdots & & & \\ & & r_{nk} & & & 1\end{bmatrix},\ r_{ik}=\frac{-a_{ik}^{(k-1)}}{a_{kk}^{(k-1)}}$$

LU分解の複雑な消去のステップを基本変換行列R_k（対角要素が1.で、才k列の$k+1\sim n$行がr_{ik}，それ以外の非対角要素が0.）を用いて表わす。上三角行列Uは$\underline{A^{(n-1)}=R_{n-1}R_{n-2}\cdots R_2R_1A}$である。ダミー添字kは行列行列積は総和の添字であるが、LU分解では（2つのGenericな表現が類似しているにもかかわらず）、基本変換行列による累積の式により、kは累積(production accumulation)の添字であることがわかる。2.1節で例示した計算順序の制約はR_1より先にR_2Aを計算するものであった。

R_kの逆行列をL_kとする。L_kはR_kの非対角項の符号を反転して得られる。L_kには、添字kの小さいものを左側から掛けても、積の行列の個々の要素には積が現われない、という性質がある。これにより、累積の式でいったん断たれたかの如く見えた計算順序の変更の可能性が復活する。

$$L_1L_2\cdots L_{n-1}=\begin{bmatrix}1 & & & & \\ -r_{21} & 1 & & & \\ -r_{31} & -r_{32} & 1 & & \\ \vdots & \vdots & & \ddots & \\ -r_{n1} & -r_{n2} & \cdots & \cdots & 1\end{bmatrix}=L$$

Lをm行m列で4つの小行列に分割し右肩に小行列の位置（11，21，22）を、右下に小行列のサイズ（行数×列数）を付して表わす。

$$\begin{bmatrix}L^{11}_{m\times m} & 0\\ L^{21}_{(n-m)\times m} & L^{22}_{(n-m)\times(n-m)}\end{bmatrix}$$

さらに行列A，L，Uの特定の行ベクトル、列ベクトルを表わす記号を決める。Aのk行目のベクトルを$\bar{a}_{r=k}$，k列目のベクトルを$\bar{a}_{c=k}$のように記すことにする（小文字と'⁻'でベクトル，'c='は列，'r='は行を表わす）。前節で挙げた36通りのループ構成を最外側ループの添字（内側2重ループのオペレーション）に着目して分類，再考察したい。

(1)　先行消去型（front elimination form）

kを最外側に用いた構成はループ添字の進行に対し消去計算が最も進行した（R_kが決定した時点でR_kを用いる計算をすべて完了する）形になる。内側2重ループはLのk列ベクトル（$\bar{l}_{c=k}$）とUのk行ベクトル（$\bar{u}_{r=k}$）を用いて右下小行列を消去するので、先行消去型と呼ぶことにする。

先行消去型

$$\left\lceil\begin{array}{l}\text{do } k=1,n-1\\ A^{(k)}=A^{(k-1)}-\bar{l}_{c=k}\cdot\bar{u}_{r=k}\end{array}\right.$$

(2)　前進代入型（forward substitution form）

ループ添字の進行に対して消去計算が最も遅れるのが、上三角のループの順序にjを最外側，下三角のループの順序にiを最外側に選んだ構成である。上三角でjを最外側に用いた場合（jxx/…型）を右に示した。内側2重ループはLの$(j-1)\times(j-1)$の主小行列による前進代入になるので前進代入型と呼ぶことにする。下三角でi最外側（…/ixx型）とすると、$U_{(i-1)\times(i-1)}$による代入計算になる。

前進代入型

$$\left\lceil\begin{array}{l}\text{do } j=2,n\\ \bar{u}_{c=j-1}=\left(L^{11}_{(j-1)\times(j-1)}\right)^{-1}\cdot\bar{a}^{(0)}_{c=j-1}\end{array}\right.$$

ただし$\bar{a}^{(0)}_{c=j-1}$はAのj列ベクトルの上三角の要素（1～j行）

(3)　行列ベクトル積型（matrix-vector product form）

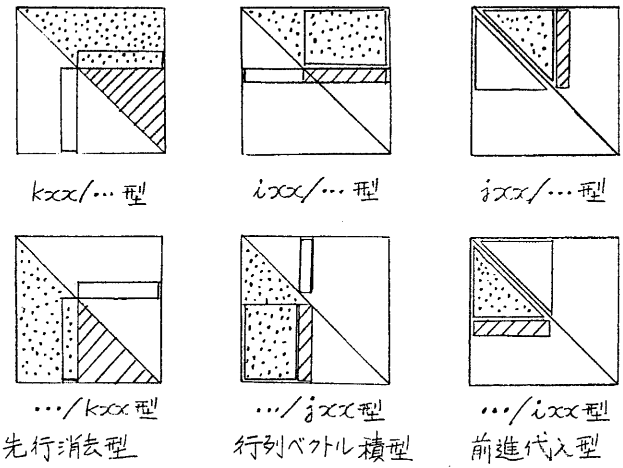
ループ添字の進行が(1)と(2)の中間になるのが、下

行列ベクトル積型

$$\left\lceil\begin{array}{l}\text{do } j=2,n\\ \bar{l}_{c=j}=\bar{a}^{(0)}_{c=j}-L^{21}_{(n-j-1)\times(j-1)}\cdot\bar{u}_{c=j}\end{array}\right.$$

ただし$\bar{a}^{(0)}_{c=j}$はAのj列ベクトルの下三角の要素（j+1～n行）

三角のループの順序にjを最外側（…/jxx型）、上三角のループの順序にiを最外側（ixx/…型）に選んだ構成である。内側2重ループは行列ベクトル積和になるので行列ベクトル積型と呼ぶことにする。下三角でj最外側を図示した。上三角でi最外側とすると転置行列ベクトル積になる。



左に3つの型を示した。内側2重ループで更新する部分（斜線）、参照する部分（枠）、分解が完了した部分（点）を示した。先行消去型で斜線部分は中間結果$a_{ij}^{(k)}$であるが、他の型では分解が完了している。iとjは行列要素の位置に対応する添字なので、前進代入型と行列ベクトル積型の違いは解の得られる領域の違いになる（三角形の辺上か台形の辺上に解が得られる）。計算順序の制約は「先行消去型と前進代入型を組合せられない」と言い換えられるが、これは両者（たとえばjxx/…型の図と…/kxx型の図）を重ねると消去型に必要な行ベクトル$\bar{u}_{c=k}$が未計算であることからわかる。

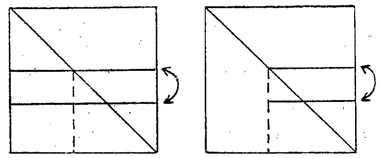
現在LU分解アルゴリズムのバリエーションが種々の名称で呼ばれているが、筆者はこれらの名称が必ずしも適切ではないと感じている。たとえばjki/jki型を"内積型"と呼ぶ場合があるが、プログラムはkが最内側にないので最深のカーネル・ループは内積を計算するものではない。コンパイル技術の観点からも内積型はk最内積の構成を指すようである[3]。また、対称行列のLU分解（LDL^tまたはコレスキー分解）を考える時、上三角の行（i）と下三角の列（j）、上三角の列と下三角の行が対称の関係にあるのでijk/jik型のように上下対称の型の変型とするのが考え易い。現在流行している呼称からはこれが"内積型"の変形なのかクラウト分解の変形なのか判然としない。上三角と下三角を分離して考えることと、数多くの計算順序のバリエーションに適切な名称を与えることは大切なことと考える。

2．4　軸選択と行要素交換（pivot selection and swapping）

部分軸選択（partial pivoting）はk段の消去で中間結果$A^{(k-1)}$の第k列のk〜n行要素の中から絶対値最大の要素を選ぶ。したがって下三角のループの順序にiを最外側（…/ixx型）に用いると、比較される要素が$a_{ik}^{(0)}$のままなので軸選択できない。2.2節の分類でこれは8通り存在する。完全軸選択（complete pivoting）では上三角、下三角ともに先行消去型（ガウス消去法）にしなくてはならない。

部分軸選択に伴なう行交換に2通りの方式がある。k段の消去で選択された行とk行の要素を交換する時，全列要素を交換する方式を全列交換（FCS：Full Column

Swap)、K〜n列の要素だけを交換する方式を部分列交換(PCS：Partial Column Swap)と呼ぶことにする。2つの方式の違いは連立1次方程式 $Ax=b$ の解を求めるための代入計算($y=L^{-1}b$, $x=U^{-1}y$)でのベクトル要素の交換のタイミングの違いに関係する。FCSでは代入に先立って右辺ベクトルの要素交換をまとめて行えるが、PCSでは前進代入のループの中に折り込まなくてはならない。



全列交換(FCS)　　部分列交換(PCS)

3. LU分解のブロック化アルゴリズム

$$\begin{cases} U_{IJ}=(L_{IJ})^{-1}\left(A_{IJ}-\sum_{K=1}^{I-1}L_{IK}U_{KJ}\right) \\ L_{IJ}=(U_{JJ})^{-1}\left(A_{IJ}-\sum_{K=1}^{J-1}L_{IK}U_{KJ}\right) \end{cases}$$

```
do K=1,N-1
  ...
  do J=K+1,N
    ...
    do I=K+1,N
      A(K)_IJ = A(K-1) - L_IK U_KJ
```

次数 n の行列Aを $N\times N$ 個の小行列 $\{A_{IJ}\}$ に分割する(小行列の大きさは同じである必要はないが、対角部分の小行列は正方とする)。ブロック化されたLU分解の式を2.1節のLU分解の2つの式に対応させて書くと右のようになる。ブロック化しないLU分解もサイズ1×1でブロック化されたと考えると、$L_{II}=1$. なので、2.1節の式はブロック化の特殊な場合と見做すこともできる。小行列に関する計算順序も、前章で述べたルールが当て嵌まり、I,J,Kに関する28通り(部分軸選択の場合20通り)が可容である。ブロック化の目的は記憶域→キャッシュへのデータ移動量を減らすことができる点にある。n が m で割り切れたとして、m 行 m 列の小行列に分割された場合、この移動量は $1/m$ になる(jki/jki型とKJI/KJI型での比較)。

$$\frac{1}{m} \doteqdot \left\{\frac{1}{2}\sum_{k=1}^{n-1}(n-1+n-k-1)\cdot k\right\} \Big/ \left\{\frac{1}{2}\sum_{K=1}^{n/m}(n-1+n-Km)\cdot Km\right\}$$

キャッシュを考えたブロック化には馴染みが無くても、行列が外部ファイルにある時は行列を何らかの形でブロック化して入出力量を減らす工夫がなされる。原理的に両者は同じである。なお、部分軸選択を行う場合、ブロック内部ではFCSとする。PCSでは代入時($U_{IJ}=(L_{IJ})^{-1}(A_{IJ}\cdots)$)に行交換をループ内に入れなくてはならなくなるからである。

ブロック化の式は用途が広く、たとえばアンローリングによって2行2列を同時に消去する場合、$m=2$ としてブロック化の式を書くと計算順序を明確にできる。

4. おわりに

高速計算のアルゴリズムでは計算順序が重要である。計算順序を変更して並行性を豊かにし、データ移動量を少なくするのがチューニングの主たる部分だからである。したがってコンパイルされ実行される段階迄計算順序を追求すべきである。

―――参考文献―――

1). 寒川光：数値計算プログラミングにおけるデータ移動制御のためのブロック化アルゴリズム，情報処理学会論文誌，Vol.33 No.10 (1992.予定)

2). Dongarra, J.J., et al. : Implementing Linear Algebra Algorithms …, SIAM Review, Vol.26, No.1 (1984)

3). 田中義一，岩澤京子：ベクトル計算機のためのコンパイル技術，情報処理学会誌，Vol.31, No.6 (1990)