Vol. 34 No. 3 情報処理学会論文誌 Mar. 1993

LU 分解のブロック化アルゴリズム

寒川 光†

新しいタイプの技術計算プロセッサでは算術演算能力が非常に強化されたため，データ移動に要する時間が相対的に増加した．ここで言うデータ移動は，レジスタやキャッシュへの移動を指すが，新しいタイプのプロセッサの中にはこのデータ移動を算術演算命令の実行と並行して処理するものもあるため，データ移動の時間が明確に現れることも多い．Fortran プログラムからはレジスタやキャッシュの存在は見えないため，媒介的な手段によってデータ移動量を把握したり，大域的なデータ移動を制御しやすいブロック化アルゴリズムによって問題（あるいはプログラム）を記述することが，大きな効果を生む．対象となるアルゴリズムが行列行列積のように計算順序に関する制約がないものであれば，ブロック化アルゴリズムへの変形はスムーズに行える．しかしLU分解では計算順序に制約があったり，軸選択を行ったりするために，ブロック化にも工夫が必要となる．本論文ではLU分解にブロック化を試みることを通して，データ移動を制御するプログラミング手法とその考え方を示す．

Blocked Algorithm for LU Decomposition

HIKARU SAMUKAWA†

The scientific processors has been enhanced its floating-point arithmetic performance. As a result, time required to move the data from one place to another becomes relatively larger than that in the conventional processors. The programming manner to maximize these enhanced capabilities is to consider not only the number of arithmetic computations but also the amount of data movement. To achieve this objective, blocking algorithm is efficient to control implicit data movement in matrix computations. This paper discusses the possible changeability of computation in the LU decomposition algorithm, then searches optimal order of computation with respect to a given machine organization. This approach can be said more basic than vectorization or parallelization, because of the mathematical characteristics.

1. はじめに

最新の技術計算用プロセッサ（ベクトル計算機やRISC タイプのスーパスカラ計算機）では算術演算能力が非常に強化されたため，データ移動に要する時間が演算時間に対して相対的に増加している．この結果，数値計算プログラムの書き方が，これまでのように計算量だけに着目した方法から，計算量とデータ移動量の両方に着目したものへ変えることが重要になりつつある．

計算機は Fortran プログラムからは単一の記憶域をもったシステムとして映るが，実際には演算器と記憶域の間に中間的な作業域を備えている．豊富なレジスタやキャッシュがプログラムからは一般には見えない作業域を形成している．プログラムの実行中にデータがこれらの作業域から参照できる時には，記憶域から参照された時より実行時間が短い．プログラムの書き方としては，できるだけ演算器に近い作業域に残ったデータを使って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．このような方法，すなわち「演算器と記憶域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」を，本論文では「データ移動に関する計算順序の最適化」と呼ぶことにする．この最適化の効果は，たとえばIBM の RISC System/6000☆(RS/6000) では約3倍の計算速度の向上を生み出す[1]．

データ移動に関する計算順序の最適化をコンパイラやプログラマが行うには，ループ順序の入替え，外側ループ・アンローリング，ストリップマイニングと呼ばれるプログラムの改変を適用する．これらの手法自身は機械的な改変であるが，書かれたプログラムを解析してこれらの改変が適用可能かどうかを判断することは意外に難しい．むしろプログラムを書く前に，小

† 日本アイ・ビー・エム(株)
IBM Japan, Ltd.

☆ RISC System/6000 は IBM Corp.(米国) の商標です．

行列による定式化によってアンローリングやストリップマイニングの適用可能性を確かめることが得策と思われる.

本論文では2章で行列積を例に最適化のアプローチを概説する. 3章でLU分解におけるループ順序の変更の可能性を考察し, 4章でこの結果を小行列による定式化(ブロック化アルゴリズム)に発展させる.

本論文に用いる表記法をここでまとめておく.

(1)行列, ベクトル, 行列要素

- 行列を大文字(行列A), 行列要素を小文字(a_{ij})
- 行列Aの第k列ベクトル (column vector) またはその部分を $a_{c=k}$, 第k行ベクトル (row vector) またはその部分を $\bar{a}_{r=k}$ で表す.

(2)行列の分割

- 行列Aを $N\times M$ 個の小行列に分割した時, 小行列とその添字も大文字, $A\Rightarrow\{A_{IJ}\}$, $I=1\sim N$, $J=1\sim M$
- 次数nの正方行列Aを, 主小行列の次数がmになるように4分割した時, 小行列のサイズを示す.

$$A\Rightarrow\begin{bmatrix} A_{m\times m} & A_{m\times(n-m)} \\ A_{(n-m)\times m} & A_{(n-m)\times(n-m)} \end{bmatrix}$$

(3)中間結果 (intermediate results) を右肩カッコ内に示す.

- $S=\sum_{k=1}^{n} a_k$ のとき $S^{(1)}=a_1$, $S^{(k)}=S^{(k-1)}+a_k$

2. データ移動制御のアプローチ

単純なアルゴリズム(行列積)を例に, 計算機内部でのデータ移動とその削減法を, Fortran プログラムでどのように捉え, さらに計算式の上では計算順序の変更がどのように記述されるかを示す.

データ移動は演算器やレジスタへのものと, キャッシュを対象にしたものの2通り存在する. 前者は演算に直結しており, 後者はバッファ領域へのデータ移動になるので, 制御の方法も異なる.

2.1 記憶域参照と最内側ループ

"キャッシュないし記憶域" と "演算器ないしレジスタ" 間のデータ移動は, コンパイルされたプログラムリストに現れるアセンブラ命令のロード/ストア命令や演算命令の**記憶域オペランド**に対応する. 単純なプログラムではFortranプログラムから記憶域参照を推定することもできる.

多重ループを持つプログラムではループの順序を入替えることで記憶域参照を少なくできる. 行列積

$$c_{ij}=\sum_{k=1}^{m} a_{ik}b_{kj},\quad i=1\sim l,\ j=1\sim n \qquad (2.1)$$

を例として次の形のプログラム[2]を考える.

```
プログラム(2.1)
  Generic matrix multiplication algorithm
  do_=1, _
   do_=1, _
    do_=1, _
     c(i, j)=c(i, j)+a(i, k)*b(k, j)
```

下線部には i, j, k の添字(と対応する l, n, m)が6通りの組合せで入れられる. 最内側ループと記憶域参照回数の対応は次のようになる.

最内側ループ添字	$c_{ij}=c_{ij}+a_{ik}*b_{kj}$	記憶域参照回数(ループあたり)
i	○ ○ ○	3
j	○ ○ ○	3
k	○ ○	2

この場合はk最内側が記憶域参照が少ない. 実行文の左辺の記憶域参照がストア命令である. チューニング作業のためにロードとストアは分けて考えることが実践的である. たとえばRS/6000ではロードは浮動小数点演算と並行処理されるため, 外側ループ・アンローリングによってロード命令を減らしてゆくと, ロード命令の実行時間を演算命令の実行時間の裏に隠すことができる. しかしストアは並行処理されないので, このような方法によるチューニングが期待できないからである[1]. このような事情から, 左辺に現れない添字(この例では i/j は左辺の行列の行/列位置を示すが k は左辺に現れない)を本論文ではテンソル数学にならって**ダミー添字** (dummy index) と呼び, 特別な注意を払う. スカラ計算機ではダミー添字を最内側ループに用いると**レジスタへの累加**が達成され, 中間結果のストアがなくなる.

計算順序を式(2.1)に忠実に従わせると, kを最内側とする内積を基本とする計算になる. ループ順序を入替えて k を最外側にすると, Aの第 k 列 ($\bar{a}_{c=k}$) と Bの第 k 行 ($\bar{b}_{r=k}$) による階数1更新 (rank-1 update) をm回繰り返す.

$$\begin{aligned} &\text{do } k=1, m \\ &\quad \boldsymbol{C}^{(k)}=\boldsymbol{C}^{(k-1)}+\bar{a}_{c=k}\cdot\bar{b}_{r=k} \end{aligned} \qquad (2.2)$$

ただし $C^{(0)}=0$ とする. 式(2.2)のように書けることが, ダミー添字kを最外側に置けることに繋がるが, 右肩のカッコ内の添字が0とm以外は中間結果であり式(2.1)の内積型の式には現れない.

2.2 記憶域参照の削減

内積型の行列積プログラムの外側の2つのループにアンローリングを適用する.

```
プログラム(2.2)  2×2 outer loop unrolling
 do j=1, n-1, 2
  do i=1, l-1, 2
   do k=1, m
    c(i, j)=c(i, j)+a(i, k)*b(k, j)
    c(i+1, j)=c(i+1, j)+a(i+1, k)*b(k, j)
    c(i, j+1)=c(i, j+1)+a(i, k)*b(k, j+1)
    c(i+1, j+1)=c(i+1, j+1)+a(i+1, k)*b(k, j+1)
```

記憶域参照回数は理想的には「ループ添字を配列の添字とする変数の重複しない数」に一致する. プログラム(2.2)はRS/6000では実線下線で示した4項となり, アンローリング前と比較すると(行列積全体では)半分に減る[1].

Fortran プログラムから記憶域参照回数を特定することが困難な場合もある. IBM の System/370☆(S/370) では浮動小数点レジスタは4倍語で, 演算命令は2オペランドであるため[3],

- 左辺の4変数のうち1つは記憶域にとられる
- レジスタにロードされた値は, 乗算結果を上書きされるため再利用できない

の理由から記憶域参照が冗長になる(コンパイル結果を調べると点線下線の6項が増え, さらに添字を更新計算するための記憶域参照も加わる).

RS/6000 では32倍語の浮動小数点レジスタと4オペランドの乗加算命令により[4], レジスタ群が1つの記憶階層の役割を果たせ, 記憶域参照が少なく抑えられる.

2.3 ベクトル計算機と外側ループ・ストリップマイニング

ベクトルレジスタ型のベクトル計算機(以下ベクトル計算機)では, ループ長がベクトルレジスタ長($l1$とする)よりも長いループを $l1$ 以下のベクトル長の計算を繰返すことで処理する. この処理をストリップマイニング(露天掘りの意)と呼ぶ. Fortran で表現すると次のように書ける.

```
do i=1, l                    do is=1, l-1, l1
 c(i)=c(i)+a(i)*b      ⇒     do i=is, min(is+l1-1, l)
                              c(i)=c(i)+a(i)*b
```

☆ System/370 は IBM Corp.(米国)の商標です.

点線で囲ったループ部分がループ構造を持たないベクトル命令列に置換えられる. S/370 のベクトル機構では3オペランドのベクトル乗加算命令が記憶域オペランドを持つ(VR←VR+FR*a(i))[5]ので, ベクトルロード, ベクトル乗加算, ベクトルストアの3つのベクトル命令になる.

多重ループを持つプログラムにおいて, ストリップマイニング後のループ構成でダミー添字が内側から2番目に用いられると, ベクトルレジスタへの累加が達成され, ベクトルストアが消える[6].

```
do j=1, n
 do is=1, l-1, l1
  do k=1, m                    ┐ロード   FR, b(k)
   do i=is, min(is+l1-1, l)    │乗加算   VR←VR+FR*a(i)
    c(i, j)=c(i, j)+a(i, k)*b(k, j)  ┘分岐
```

点線で囲った k のループ部分は, ベクトル命令は乗加算1つになる. この形はループ順序を外側から jik と構成した時, i のループをベクトル化して得られるので"外側ループ・ストリップマイニング"と呼ばれる. IBM の VS Fortran は jik または ijk の順で構成されたプログラムの i をベクトル化してこの形のコード生成をする(ベクトル機構で実行すると, jki と構成したプログラムの i をベクトル化した時に比べて2倍以上速くなる[6]).

最新のベクトル計算機では演算能力がデータ移動能力に比して高く設計されるようになったため, プログラマやコンパイラによる外側ループ・アンローリングによってデータ移動(記憶域参照)を減らすことが重要になった[7].

2.4 記憶域とキャッシュ間のデータ移動

記憶域とキャッシュ間のデータ移動を制御する方法をキャッシュブロック化(cache-blocking)と呼ぶ. これは3重ループを持つプログラムで, 配列要素が何度も参照されるとき効果を発揮する.

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

内側2重ループの添字を持つ変数 a に着目する. $a(i, k)$ の参照パターンは $a(1, 1)$ から $a(l, m)$ の連続参照を n 回(j のループの反復回数)繰返す. この計算順序では行列 A の大きさ $l\times m$ がキャッシュ容量より大

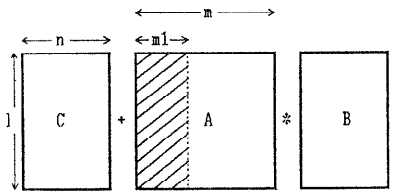


図 1 行列Aのキャッシュブロック化
Fig. 1 Cache-blocking for A matrix.

きいと，$a(i,k)$ は常に記憶域から参照される．すなわち定期的に発生するキャッシュミスが計算を妨げる．そこで添字 k の進行を $a(i,k)$ がキャッシュを満たす前に止め，外側ループの添字 j を増やしてキャッシュ上のデータを利用できる計算を先に完了させる（**図 1**）．プログラムでは $l\times m1$ を有効キャッシュ容量として次のように書くことができる．

```
do ks=1, m-1, m1
 do j=1, n
  do k=ks, min(ks+m1-1, m)
   do i=1, l
    c(i, j)=c(i, j)+a(i, k)*b(k, j)
```

これはサイズ $m1$ で k のループをストリップマイニングした形である．これにより A をキャッシュにステージングする回数を n 回から 1 回に減らせるが，c は 1 回から $m/m1$ 回に増える．l が大きく $m1$ をあまり大きく選べない時は i のループもストリップマイニングする．

プログラム (2.3)
Cache-blocked matrix multiplication

```
do is=1, l-1, l1
 do ks=1, m-1, m1
  do j=1, n
   do k=ks, min(ks+m1-1, m)
    do i=is, min(is+l1-1, l)
     c(i, j)=c(i, j)+a(i, k)*b(k, j)
```

このような改変そのものは機械的に行うことができ，Fortran のプリプロセッサの機能にも含まれている[8]．

2.5 小行列による定式化

外側ループ・アンローリングやストリップマイニングが適用可能かどうか調べるには，**計算順序を忠実にトレースできる小行列による定式化**を試みるのがよい．行列積のキャッシュブロック化では（l と m がそれぞれ $l1$ と $m1$ で割切れるとすると），行列 A を $(l/l1)\times(m/m1)$ 個のサイズ $l1\times m1$ の小行列 $\{A_{IJ}\}$ に，行列 C を $(l/l1)\times 1$ 個のサイズ $l1\times n$ の小行列 $\{C_{I1}\}$ に，行列 B を $(m/m1)\times 1$ 個のサイズ $m1\times n$ の小行列 $\{B_{K1}\}$ に分割して

$$C_{I1}=\sum_{K=1}^{m/m1} A_{IK}\cdot B_{K1},\quad I=1\sim l/l1$$

と書き表せることが，2 回のストリップマイニングが可能なことを示す．

コンパイラが 2 回のストリップマイニングを自動的に行うには，多重ループ・プログラムに対するデータ依存性を解析する必要がある．行列積のように単純なアルゴリズムに対してならば可能かもしれないが，LU 分解に対しては（次章に述べるが），データ依存性の解析は困難である．

3. LU 分解

LU 分解は正方行列 A を，A と同じサイズの単位下三角行列 L（対角項が 1. で，対角項より上はゼロ）と，上三角行列 U（対角項より下はゼロ）の積に分解する（$A\rightarrow LU$）．A から L と U を作るアルゴリズムは通常，領域節約の目的から A が置かれていた配列に L と U を上書きする形で記述される．前章で行列積に対して行ったのと同じアプローチでデータ移動制御（ループ順序の入替えとストリップマイニング）を考察する（第 3，4 章）．

3.1 ループ順序の入替え

LU 分解の式を行列積の式 (2.1) に対応する総和記号 Σ による内積を基本演算とする形で記述する．これはクラウト法と呼ばれる．

$$\begin{cases} u_{ij}=a_{ij}-\sum_{k=1}^{i-1} l_{ik}u_{kj} & (i\leqq j \text{ の要素}) \\ l_{ij}=\left(a_{ij}-\sum_{k=1}^{j-1} l_{ik}u_{kj}\right)\Big/u_{jj} & (i>j \text{ の要素}) \end{cases} \tag{3.1}$$

通常 2 つの式は交互に（U の第 1 行，L の第 1 列，U の第 2 行…の順で）計算する．行列積の式 (2.2) のように中間結果を左辺に明示する方法はガウスの消去法と呼ばれる．$a_{ij}^{(0)}=a_{ij}$ として，

```
do k=1, n-1
 a(i,j)(k)=a(i,j)(k-1)-l(i,k)u(k,j)  (=a(i,j)(k-1)-a(i,k)(k-1)a(k,j)(k-1))
        上三角 (i≦j) の要素  u(i,j)=a(i,j)(i-1)
        下三角 (i>j) の要素  l(i,j)=a(i,j)(j-1)/u(j,j)
```

(3.2)

と書かれる．

Dongarra らはガウス消去法の Generic なアルゴリズムによりループ順序を論じている．

プログラム(3.1)
Generic Gaussian elimination algorithm

```
do_=_, _
 do_=_, _
  do_=_, _
   a(i, j)=a(i, j)−(a(i, k)∗a(k, j))/a(k, k)
```

このプログラムの下線部には行列積同様 i, j, k の3つの添字の組合せが入り，6通りのループ順序が選択可能である[2]．しかし式(3.1)のクラウト法は通常 L と U の要素について異なったループ順序で計算するのでこの6通りに含まれない．そこで本論文では次の形のGeneric アルゴリズムを用いる．

プログラム(3.2)
Generic LU decomposition algorithm

```
do ijk=1, n                          (クラウト法の場合)
 _=ijk          ←————————(i=ijk)
 do_=_, _       ←————————(do j=i, n)
  do_=_, _      ←————————(do k=1, j−1)
   a(i, j)=a(i, j)−a(i, k)∗a(k, j)
 ......
 _=ijk          ←————————(j=ijk)
 do_=_, _       ←————————(do i=j, n)
  do_=_, _      ←————————(do k=1, j−1)
   a(i, j)=a(i, j)−a(i, k)∗a(k, j)
```

前半が U の要素，後半が L の要素を計算する．右のカッコに示したように U を ijk, L を jik の順でループ構成するとクラウト法になる．本論文ではこれを ijk/jik 型のように記すことにする．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

$ijk, ikj, jki, jik, kij, kji$ ……6通り

i-ii) 上三角と下三角が異なるもの

$ixy/iyx, jxy/jyx, kxy/kyx$

……2×3=6通り

ii) 外側ループが上下で異なるもの

$ixx/jxx, ixx/kxx, kxx/jxx,$
$jxx/ixx, \underline{jxx/kxx}, \underline{kxx/ixx}$

……4×6=24通り

3.2 計算順序の制約

行列積アルゴリズムには，計算の最小単位を1回の乗加算：$c_{ij}^{(x)}=c_{ij}^{(x-1)}+a_{ik}b_{kj}$ として捉えた時，全体を $l\times n\times m$ 個に分けられるが，これらをどれから先に計算してもかまわない（コンパイラから見るとデータ依存性がない）という特徴がある．これは両辺に現れる変数が被加数としてしか用いられず，しかも添字は両辺で同じ形をしている（$c(i, j)=c(i, j)+\cdots$）からである．これに対し，LU 分解アルゴリズムでは両辺に現れる変数が乗数，被乗数にも用いられ，しかも添字の動く範囲が外側のループ添字で記述される．ここでは，要素 a_{ij} を更新する時，「乗数，被乗数に用いられる a_{ik} と a_{kj} の計算が完了して，それぞれ l_{ik} と u_{kj} になっていなくてはならない」という計算順序の制約がある．たとえば kji/kji 型で，2行目の要素 a_{2j} に着目すると，$k=1$ の段で

$(u_{2j}=)a_{2j}^{(1)}=a_{2j}^{(0)}-l_{21}a_{1j}^{(0)}$ （ただし $l_{21}=a_{21}/a_{11}$）

として計算され，次（$k=2$）の段で乗数として

$a_{ij}^{(2)}=a_{ij}^{(1)}-l_{i2}\underline{a_{2j}^{(1)}}$ $(=a_{ij}^{(1)}-l_{i2}(a_{2j}^{(0)}-l_{21}a_{1j}^{(0)}))$

と使用される．後の式の乗加算が前の式の乗加算を追越すと下線部の項は $a_{2j}^{(0)}$ のままなので正しく計算されない．この制約をループ順序の選択のために言換えると，「下三角の j 列の要素（$a_{ij}, i>j$）を計算する時は，上三角の j 列の要素（$a_{1j}\sim a_{i-1,j}$）が計算完了して（$u_{1j}\sim u_{i-1,j}$ になって）いなくてはならない」．上三角の要素が下三角の要素から受ける計算順序の制約も同様の記述が可能である．これらは式(3.1)から直接言えることであるが，既に書かれたガウス消去法のプログラム（kji/kji 型）をデータ依存解析して同様のルールを見出すのは至難の技と思われる．

前節に挙げた Generic LU decomposition に対する36通りの組合せの中には，この制約条件を満たさないものが8通り（下線で示した jxx/kxx 型と kxx/ixx 型）ある．

3.3 基本変換とループ順序

基本変換行列を用いると複雑な消去のステップを見通しの良い形で記述できる．R_k を対角要素が1で，第 k 列の $k+1\sim n$ 行以外の非対角要素がゼロの基本変換行列とする．LU 分解の k 段のステップは次の形である．

$A^{(k)}=R_kA^{(k-1)}$

ただし

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

LU 分解は

do $k=1, n-1$
$A^{(k)}=R_kA^{(k-1)}$

すなわち

$$(U=)A^{(n-1)}=R_{n-1}R_{n-2}\cdots R_2R_1A \qquad (3.3)$$

となっている．ダミー添字 k は式(3.1)では総和添字のごとく見えるが，**消去の意味の上では累積の添字で**あり，このことが3.2節で述べた計算順序の制約に繋がっている．行列の累積であるため直感的には k を内側に回すようなループ順序の変更は行えない．しかし行列 R_k には次に述べる性質があって，計算順序変更の可能性が生まれる．

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．添字 k の小さいものを左側から掛けても，積の行列の**個々の要素には積は現れない**（計算せずに求められる）という性質である．

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

この性質により，LU 分解のループ順序変更によるアルゴリズムの変形が可能となる．

なお LU 分解 ($A=LU$) は式(3.3)の両辺に左から L を掛けた形で表される．

行列 L を4分割して，ループ順序の組合せによるアルゴリズムを分類する．

$$L\Rightarrow\begin{bmatrix} L_{m\times m} & 0 \\ L_{(n-m)\times m} & L_{(n-m)\times(n-m)} \end{bmatrix}\left(=\begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix}\right)$$

（1） 先行消去型 (front elimination form)

式(3.2)のダミー添字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進んだ（R_k が決定した時点でそれ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 列と U の k 行ベクトルで右下の小行列部分を消去するので，先行消去型と呼ぶことにする．

$$\begin{cases}\text{do } k \\ \quad A^{(k)}=A^{(k-1)}-\bar{l}_{c=k}\cdot\bar{u}_{r=k}\end{cases}$$

（2） 前進代入型 (forward substitution form)

ループ添字の進行に対し消去計算が最も遅れるのが，上三角に j，下三角に i を外側に用いる構成である．

$$\begin{cases}\text{do } j=2,n \\ \quad \bar{u}_{c=j-1}=(L_{(j-1)\times(j-1)})^{-1}\bar{a}^{(0)}_{c=j-1}\end{cases}$$

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．下三角に i を用いると $U_{(i-1)\times(i-1)}$ による代入になる．

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

ループ添字の進行に対し消去計算がこれらの2つの方法の中間になるのが，下三角に j，上三角に i を最外側に用いる構成である．

$$\begin{cases}\text{do } j=2,n \\ \quad \bar{l}_{c=j}=\bar{a}^{(0)}_{c=j}+L_{(n-j-1)\times(j-1)}\cdot\bar{u}_{c=j}\end{cases}$$

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．上三角に i を用いると転置行列ベクトル積になる．

図2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（枠で囲った），分解が完了した部分（点を打って示した）である．先行消去型では中間結果 $a^{(k)}_{ij}$ を扱うので式(3.2)で考える必要がある．他の型は式(3.1)が考えやすい．i と j は左辺の変数の行位置，列位置に対応する添字なので，最外側が i か j かの違いは解の得られる領域の形の違いになる．行列ベクトル積型では台形で，その短い方の底の辺上に解が得られ，前進代入型では三角形で，その辺上に解が得られてゆく．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．

なお，対称行列の LU 分解 (LDL^T またはコレスキー分解) を考える時は上三角の行と下三角の列，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形とみるのが考えやすい．

いずれの方法でも計算量は同じである．記憶域参照回数は，スカラ計算機では k を最内側，ベクトル計算

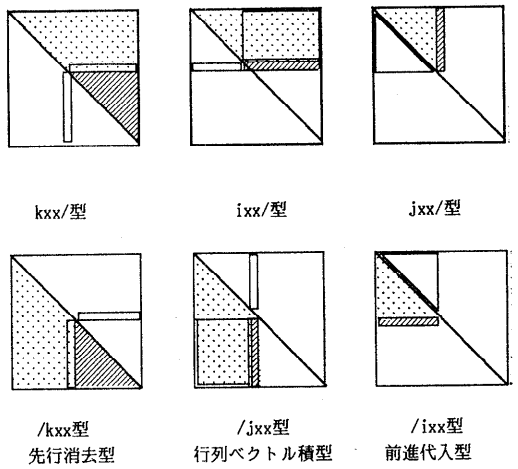


図 2 ループ順序の組合せ
Fig. 2 Loop ordering.

機ではストリップマイニング後のループ構成で k が内側から2番目に置かれた形が最小のとなる（行列積の場合と同様）．ただし前進代入でベクトルレジスタへの累加を達成するには，ベクトルレジスタ上の要素データをスカラレジスタへ抽出ロードする命令や，ベクトルを下から逆向きにした計算を用いなくてはならないので，コンパイラにとっては外側ループ・ストリップマイニング以上に難しい．したがってデータ移動量とループ構成の関係は，コンパイラの機能に影響されないところで（アセンブラで書くか，メーカ提供のライブラリーを基に）考えるべきである．

3.4 軸選択

部分軸選択（partial pivoting）は k 段の消去で中間結果の第 k 列，$a_{ik}^{(k-1)}$ の k〜n 行要素の中から絶対値最大の要素を選ぶ．したがって下三角を i を外側に置いたループ構成（ikx 型）では，比較対象位置にある要素が $a_{ik}^{(0)}$ のままなので軸選択ができない．3.1節の分類では下三角が前進代入型のものは8通り存在する．

完全軸選択（complete pivoting）では計算順序の制約はさらに強まり，上下ともに先行消去型にする必要がある．LU 分解では完全軸選択を用いることは稀なので，以下部分軸選択について述べる．

軸選択に伴う行交換に2通りの方式がある．本論文では交換すべき行の全列要素を交換する方式を全列交換（FCS: Full Column Swap），k 列以降のみを交換する方式を部分列交換（PCS: Partial Column Swap）と呼ぶ（**図3**）．行交換のための基本変換行列 P_k を用いて表すと式(3.3)の分解式が次のように書き改められる（$n=4$ とする）．

$$(U=)A^{(n-1)}=R_3P_3R_2P_2R_1P_1A$$

$R_k^{-1}=L_k$ として $A=LU$ の形に書き改める．なお，$P_k^{-1}=P_k$ である．

$$A=P_1L_1P_2L_2P_3L_3U \tag{3.4}$$

この式のとおりに計算すると L_k に後続の行交換（P_{k+1}〜）が掛けられるので，これを $\bar{L}_k$ を用いて表す．上式は

$$A=P_1(P_2P_3\underbrace{P_3P_2)L_1P_2(P_3P_3}_{\bar{L}_1})\underbrace{L_2P_3L_3}_{\bar{L}_2}\cdot U$$

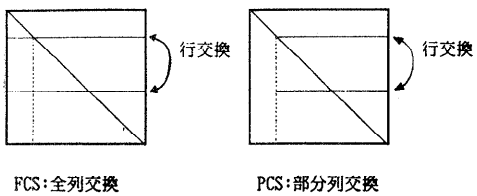


図3 全列変換と部分列変換
Fig. 3 Full column swap and partial column swap.

$$A=P_1P_2P_3\bar{L}_1\bar{L}_2L_3\cdot U$$

と変形できる．$\bar{L}_k=P_{n-1}\cdots P_{k+1}L_kP_{k+1}\cdots P_{n-1}$ は L_k の非対角要素を後続の P_{k+1}〜P_{n-1} によって行交換した行列である．FCSでは $\bar{L}=\bar{L}_1\bar{L}_2\cdots\bar{L}_{n-2}L_{n-1}$ が求められ，PCSでは $L=L_1L_2\cdots L_{n-2}L_{n-1}$ が求められる．連立1次方程式 $Ax=b$ を解く場合FCSでは $A=P_1P_2P_3\bar{L}U$ より

$$\bar{L}Ux=P_3P_2P_1b$$

を解けばよいので，最初にまとめて右辺ベクトルの行交換を済ませてから前進代入を行う．

これに対しPCSでは式(3.3)に従って記述すると，

$$A^{(0)}x=b$$
$$R_1P_1A^{(0)}x=R_1P_1b \quad :A^{(1)}x=b^{(1)}$$
$$R_2P_2A^{(1)}x=R_2P_2b^{(1)} \quad :A^{(2)}x=b^{(2)}$$
$$R_3P_3A^{(2)}x=R_3P_3b^{(2)} \quad :(Ux=)A^{(3)}x=b^{(3)}$$

とステップごとに変換と交換が対になって現れるため，右辺ベクトルも中間的な状態 $b^{(k)}$ を考えなくてはならない．プログラムでは両者は，右辺ベクトルの行交換を代入ループの外でまとめて行うか，代入ループの中で行うかの差になる．

(FCS)

```
do i=1,n-1
 (bの行交換)
do k=1,n-1
 do i=k+1,n
  x(i)=x(i)-l(i,k)*x(i-1)
```

(PCS)

```
do k=1,n
 (bの行交換)
 do i=k+1,n
  x(i)=…(同左)
```

PCSはLU分解時に行交換（データ移動）する要素数を半分にできる反面，代入時の右辺ベクトルの行交換をまとめて行えないのでブロック化には不適当である（後述）．

4. LU分解のブロック化

前章でLU分解の基本的な式をループ順序の入替えの観点からまとめたが，この章ではループの分割計算の観点からまとめ，記憶域とキャッシュ間のデータ移動制御の方法を述べる．

4.1 分解された三角行列の構造

ブロック化に先立って，$A^{(0)}$，L，U，$A^{(k)}$ などを分割して得られる小行列相互の関係式を導いておく．$A=LU$ は次のように m 行 m 列に分割される（$A^{(0)}=A$ である）．

$$\left[\begin{array}{c|c}A_{m\times m}^{(0)} & A_{m\times(n-m)}^{(0)}\\ \hline A_{(n-m)\times m}^{(0)} & A_{(n-m)\times(n-m)}^{(0)}\end{array}\right]=\left[\begin{array}{c|c}L_{m\times m} & 0\\ \hline L_{(n-m)\times m} & L_{(n-m)\times(n-m)}\end{array}\right]\cdot\left[\begin{array}{c|c}U_{m\times m} & U_{m\times(n-m)}\\ \hline 0 & U_{(n-m)\times(n-m)}\end{array}\right] \tag{4.1}$$

次にガウス消去法（*kji*/*kji* 型）でm段（$k=m$）まで消去された状態の $A^{(m)}=R_mR_{m-1}\cdots R_2R_1A^{(0)}$ を考える．

$$\left[\begin{array}{c|c}A^{(0)}_{m\times m} & A^{(0)}_{m\times(n-m)}\\ \hline A^{(0)}_{(n-m)\times m} & A^{(0)}_{(n-m)\times(n-m)}\end{array}\right]=\left[\begin{array}{c|c}L_{m\times m} & 0\\ \hline L_{(n-m)\times m} & I\end{array}\right]\cdot\left[\begin{array}{c|c}U_{m\times m} & A^{(m)}_{m\times(n-m)}\ (=U_{m\times(n-m)})\\ \hline 0 & A^{(m)}_{(n-m)\times(n-m)}\end{array}\right] \tag{4.2}$$

jki/*jki* 型でm列まで分解が完了した状態を考えると，式(4.1)の $L_{m\times m}$，$U_{m\times m}$，$L_{(n-m)\times m}$ が得られ，$m+1$〜n 列は $A^{(0)}$ のままである．これを $A^{(m)}$ に更新するには式(4.1)または式(4.2)の上の式より前進代入

$$(U_{m\times(n-m)}=)A^{(m)}_{m\times(n-m)}=(L_{m\times m})^{-1}A^{(0)}_{m\times(n-m)} \tag{4.3}$$

また，式(4.2)の下の式より行列行列積差

$$A^{(m)}_{(n-m)\times(n-m)}=A^{(0)}_{(n-m)\times(n-m)}-L_{(n-m)\times m}U_{m\times(n-m)} \tag{4.4}$$

を計算すればよいことがわかる．さらに式(4.1)，(4.2)の下の式を比較すれば，次数 $n-m$ の LU 分解

$$A^{(m)}_{(n-m)\times(n-m)}=L_{(n-m)\times(n-m)}U_{(n-m)\times(n-m)} \tag{4.5}$$

により，リカーシブな LU 分解の式が導かれる[9]．

4.2 小行列による定式化

行列 A をあらかじめ $N\times N$ 個の小行列 A_{IJ} に分割しておき（個々の小行列のサイズは任意），上記のステップを $N-1$ 回繰返すアルゴリズムを次のように記述できる．

```
DO K=1, N-1
  L_KK, U_KK, L_K+1,K〜L_NK を作成
  DO J=K+1, N
    U_KJ=(L_KK)^-1 A_KJ^(K-1)
    DO I=K+1, N
      A_IJ^(K)=A_IJ^(K-1)-L_IK U_KJ
```

(4.6)

これは式(3.2)と同じ形である．式(3.2)の $u_{ij}=a_{ij}^{(i-1)}$ では $l_{ii}=1.0$ なのでこれをサイズ1の小行列と考え，前進代入 $u_{ij}=(l_{ii})^{-1}a_{ij}^{(i-1)}$ を想定すれば，式(3.2)は式(4.6)の特別な場合と考えることもできる．

またひとつの小行列 A_{IJ} に着目すれば，上三角の位置にあるもの（$I\leqq J$），下三角の位置にあるもの（$I>J$）について，次のように書き直すことができる．

$$\begin{cases}U_{IJ}=(L_{II})^{-1}\left(A_{IJ}-\sum_{K=1}^{I-1}L_{IK}U_{KJ}\right)\\ L_{IJ}=(U_{JJ})^{-1}\left(A_{IJ}-\sum_{K=1}^{J-1}L_{IK}U_{KJ}\right)\end{cases} \tag{4.7}$$

これは式(3.1)と類似しており，小行列に関する計算順序も前章で導いたルールを適用できる．

このように同じ形の式を導く利点は，ブロック化されたアルゴリズムで最も複雑な部分である主小行列の LU 分解に，ブロック化する前のプログラムを流用できるところにある（軸選択を行う場合は $L_{K+1,K}$〜L_{NK} を合わせた長方形行列を扱うように変更する）．これ以外の部分は複数列の前進先入と行列行列積であるため，より単純である．

4.3 ブロック化 LU 分解の計算

行列を外部ファイルに置く場合は，小行列（あるいはいくつかの小行列）を入出力の単位（レコード）とするので，式(4.6)，(4.7)に基づく．例えば縦型ブロック・ガウス法[10]では列（J）ごとにN個の小行列をまとめて1レコードとして入出力を行い，ループ構成は *JKI*/*JKI* 型であると考えると解りやすい．同じレコード形式でも *KJI*/*KJI* 型とすると中間結果を書き戻すので入出力量は増加する．

キャッシュブロック化では入出力レコードのように小行列の形を固定する必要はない．式(4.6)は *KJI* 型に書かれているが，*KIJ* 型に変更し，かつ小行列内部の計算で j を最外側にすると，J と j をまとめられるのでループのネストをひとつ少なくすることができる．これが式(4.4)を計算する行列積となり，ループのネストは5重である．この状態で記憶域とキャッシュ間のデータ移動量を考察する．

```
do j
  do k
    do i
      a(i, j)=a(i, j)-l(i, k)*u(k, j)
```

2.4節の行列積の場合と同様に内側2重ループの添字を持つ変数 $l(i,k)$ をキャッシュに残すようにする．**図4**に示すように m 列のブロック化が施され，小行列 L_{IK} がキャッシュに入りきるものとする．この時 A_{IJ} の 2〜m 列の計算では $l(i,k)$ はキャッシュに残っている．ブロック化しなければこのデータは参照されるたびに記憶域から送り込まれなければならない．したがってブロック化により記憶域→キャッシュへのデータ移動量を $1/m$ に減らせる．

非ブロック化（*jki* 型）：$\frac{1}{2}\sum_{k=1}^{n-1}(n-1+n-k-1)\cdot k$ 項

ブロック化：$\frac{1}{2}\sum_{K=1}^{n/m}(n-1+n-Km)\cdot Km$ 項

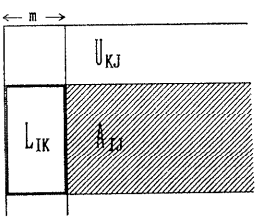
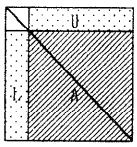
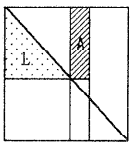


図4 ブロック化
Fig. 4 Blocking.

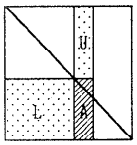


$A^{(K)}=A^{(K-1)}-LU$
(階数m更新)
(KXX/KXX型)

回帰型 (recursive formula, right-looking LU)

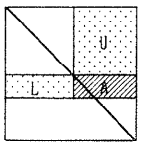


$U=L^{-1}A^{(0)}$
(前進代入)
(JXX/型)

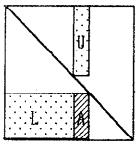


$A^{(K)}=A^{(0)}-LU$
(行列行列積)
(/JXX型)

後方参照型 (left-looking LU)



$A^{(K)}=A^{(0)}-LU$
(行列行列積)
(IXX/型)



$A^{(K)}=A^{(0)}-LU$
(行列行列積)
(/JXX型)

ブロック・クラウト型 (block Crout)

図 5 ブロック化アルゴリズム
Fig. 5 Blocked algorithms.

LU 分解もブロック化によって記憶域とキャッシュ間のデータ移動量を減らせるアルゴリズム（レベル3性能のアルゴリズム[1]）であることがわかる．

キャッシュブロック化でよく用いられる方法を示す．

（1） 回帰型（リカーシブ・フォーミュラ）

式(4.3)，(4.4)により次数をmずつ縮小してゆく方法で，ループ構成は *KXX/KXX* 型である．Right-looking LU と呼ぶこともある[11]（図5上）．

（2） 後方参照型

JXX/JXX 型に構成するもので，Left-looking LU と呼ぶことがある[11]ので，本論文でも後方参照型と呼ぶ（図5中）．

（3） ブロック・クラウト型

IXX/JXX 型で，式(4.7)をクラウト法のような順序で計算する（図5下）．

この3種類のほかにも，前章で述べた i, j, k に関する28通りの組合せ同様の組合せが I, J, K について可能である．

4.4 計算量の分布と標準化

ブロック化によっても計算量は変わらない．回帰型で考えると行列積式(4.4)で処理される階数 m 更新の計算量の全計算量に占める割合が算出しやすい．これは $2\sum_{K=1}^{N-1}K^2$ と $\frac{2}{3}N^3$ の比として

$$1-\frac{3}{2N}+\frac{1}{2N^2}$$

で与えられる．Nが10で($n=1000$を$m=100$でブロック化したような時) 85.5%である．20になると92.6%，50では97%に達する．通常キャッシュ容量はキロ・バイトのオーダであり，mは30〜60程度になる．$n=2000$ に対し $m=40$ で計算した場合が $N=50$ となる．したがって行列積サブルーチンをチューニングすることが非常に重要になる．他の型でもこの事情は同じで，結局行列積と複数ベクトル用前進代入に計算量のほとんどが集中する．これらのルーチン（特に行列積）は計算順序の制約が少ないので，計算機の特性に合わせてチューニングすれば，ハードウェアの持つポテンシャルに肉迫する性能を期待することもできる．ブロック化の利点はチューニングの対象となるカーネルサブルーチンを単純なアルゴリズムに追い込んでゆけるところにもある．一般にピーク性能値の高い計算機ほどチューニング効果が敏感に現れる傾向があるので，チューニング対象ルーチンの単純化と絞り込みはソフトウェアの移植性にとって非常に重要である．

BLAS (Basic Linear Algebra Subprograms) に集められたメニューもこの観点から選ばれた．複数ベクトル用代入ルーチンを TRSM (TRiangular system Solve with Multiple right-hand sides)，行列積を GEMM (GEneral Matrix Multiply) と呼び，どちらもレベル3性能のアルゴリズムなので，BLAS-3 で採用された[12]．

キャッシュブロック化のプログラミングでは，小行列の管理を上位ルーチンですべて行う必要はない．LU 分解の場合は最外側ループのみを上位ルーチンで管理する方法が便利である．後方参照型で示す．

```
DO J=1,(n-1)/m-1
  j1=(J-1)*m+1
  j2=min(n, j1+m-1)
  call TRSM…(U=L^-1 A^(0))
  call GEMM…(A^(J-1)=A^(0)-L・U)
  call XXX…}長方形行列の分解, j1～j2列)
```

この場合，TRSM や GEMM はそれぞれのルーチンの内部で自身のアルゴリズムに適したブロック・サイズを選択できるので，小行列のサイズは m とは限らない．特に TRSM では次数が大きくなると内部はさらに行列行列積で計算することになる．このように小行列のサイズを固定する必要がないところが，外部入出力に対するブロック化とキャッシュブロック化の異なる点である．

4.5 FCS と PCS

軸選択を行う場合，ブロック内部では FCS で行交換する必要がある（**図6**）．式(4.3)の前進代入を行交換をしていない $L_{m\times m}$ で行うと代入ルーチンのループの中に $A^{(0)}_{m\times(n-m)}$ の行交換を入れなくてはならない．$m\times m$ の小行列の中だけは FCS で行えば $\bar{L}_{m\times m}$ が得られるので，代入に先立って m 行分の行交換を済ませられる．BLAS の TRSM ルーチンのインタフェースもこの仕様になっている．

通常ピボット列を含むブロックの先頭の列以降を行交換（ブロック単位では PCS，ブロックの内部は FCS に）する．この方法で注意しなくてはならないことは，LU 分解した時のブロック・サイズを知らないと右辺の代入計算ができなくなることである．常に分解と代入をペアにして計算できれば良いが，キャッシュ容量を動的に調べてブロック・サイズを決めるようなプログラムが，ネットワーク上で複数の計算機を跨いで稼動するような環境では注意を要する．

4.6 帯行列のブロック化

帯行列のブロック化も式の上では式(4.6)，(4.7)を考えればよい．プログラミング上は**図7**(上)に示した論理的な行列が，図7(下)のようなバンド幅と次数のサイズの2次元配列に格納されているので，前進代入，行列積ともに複雑になる．特に小行列の形状が長方形でない（台形または5角形になる）ところは GEMM などの標準ルーチンでは計算できない．

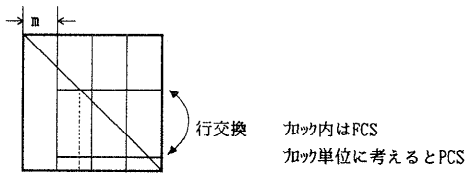
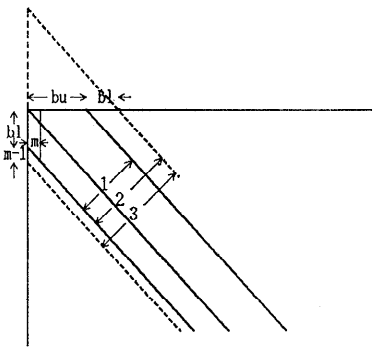


図 6 ブロック化における行変換
Fig. 6 Column swapping for blocked algorithm.

軸選択を行う場合は2次元配列の行数をあらかじめ下三角行列の帯幅（bl）だけ上三角を拡大しておく必要がある．PCS であればこれで済むが，ブロック化（列数 m）ではブロック内で FCS とするため下三角も軸選択の行交換で帯幅が拡大する．分解計算を回帰（KXX/KXX）型で行う場合は $bl\times m$ のサイズの作業域をもつことで計算できるが，後方参照（JXX/JXX）型では2次元配列そのものをさらに $m-1$ 拡げなくてはならない．ブロック化によって小行列に零要素が混入して計算量そのものを増加させるので，ブロック・サイズ m を大きく選ぶことは得策ではない．



1. A のバンド幅
2. 軸選択とPCSによるバンド幅
3. ブロック化をJXX/型で計算する時のバンド幅

bu,blは対角項を含まないセミバンド幅

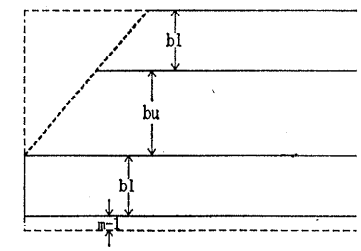


図 7 帯行列のブロック化
Fig. 7 Blocked algorithm for band matrix.

データ移動に関して計算順序を最適化しようとする試みは，行列積においては機械的に実現できるため，コンパイラによる自動化も期待できそうである．LU分解ではアルゴリズムのもつ計算順序の制約によって事情は大きく異なる．最適化における複雑さは軸選択とそれに伴う行交換の方法，さらに帯行列では軸選択した場合に計算順序が作業域にも影響を及ぼすことによって助長される．

5. おわりに

計算機構造，コンパイラ，アルゴリズムの3分野の技術が数値計算の高速化に貢献している．Fortranは計算機構造を演算器と記憶域によって構成される状態遷移マシンとして平面的に捉えている．この単純さはプログラミング時の思考には必須であり，またその形を変えないことはソフトウェア資産の蓄積に繋がった．一方現実の計算機構造はレジスタやキャッシュという作業域をFortranプログラムからは見えないところに持つことで，ノイマン隘路の問題を緩和してきた．最近の技術計算用プロセッサでは両者の隔たりは大きく，これらの作業域へのデータ移動時間が大きく姿を現す場合も多い．その結果，本来Fortranプログラムからは陽な形では見えないデータの動きを制御しようとする試みが生まれる．

データ移動を減らす努力は，計算機構造としてはレジスタと演算命令のオペランドを増やし，キャッシュ容量を大きくすることになる．コンパイラ技術やプログラム・チューニング技術としては，ループ順序の入替え，外側ループ・アンローリング，ストリップマイニングになる．数値計算法としては小行列による定式化になる．3つの分野に個別に実施されるこれらの方法論の意図を理解し，**これらを連係した形で適用するためには，3つの分野を統一的に眺められる視野**が必要である．このような視野を育成することは，将来の計算機の性能を十分に引き出す上で重要である．

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, pp. 1183-1192 (1992).
2) Dongarra, J. J., Gustavson, F. G. and Karp, A.: Implementing Linear Algebra Algorithms for Dense Matrices on a Vector Pipeline Machine, *SIAM Rev.*, Vol. 26, No. 1, pp. 91-112 (1984).
3) IBM Enterprise Systems Architecture/370, Principles of Operations, IBM Corp., 資料番号：SA 22-7200 (1988).
4) RISC システム/6000 パワーステーションおよびパワーサーバー，ハードウェア技術解説書 概説書，日本アイビー・エム(株)，資料番号：N-SA23-2643 (1990).
5) IBM Enterprise Systems Architecture/370 and System/370 Vector Operations, IBM Corp., 資料番号：SA 22-7125 (1988).
6) Samukawa, H.: Programming Style on the IBM 3090 Vector Facility Considering Both Performance and Flexibility, *IBM Syst. J.*, Vol. 27, No. 4, pp. 453-474 (1988).
7) 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，Vol. 31, No. 6, pp. 736-743 (1990).
8) AIX XL FORTRAN コンパイラー/6000 使用者の手引きバージョン 2.2，日本アイ・ビー・エム(株)，資料番号：N-SC 09-1354, pp. 96-102 (1992).
9) Wilkinson, J. H.: *The Algebraic Eigenvalue Problem*, pp. 201-202, Clarendon Press, Oxford (1965).
10) 小国 力(編)：行列計算ソフトウェア (WS, スーパーコン，並列計算機)，pp. 112-113, 丸善(株) (1991).
11) Dongarra, J. J., Mayes, P. and Radicati di Brokolo, G.: The IBM RISC System/6000 and Linear Algebra Operations, *Super Computer*, Vol. 8, No. 4, pp. 15-30 (1991).
12) Dongarra, J. J., Du Croz, J., Hammarling, S. and Duff, I.: A Set of Level 3 Basic Linear Algebra Subprograms, *ACM Trans. Math. Softw.*, Vol. 16, No. 1, pp. 1-17 (1990).

(平成4年7月29日受付)
(平成4年12月10日採録)

寒川 光（正会員）

1948年生．'72年早稲田大学理工学部機械工学科卒業．同年より日本ユニバック(株)応用ソフトウェア部に勤務．主にFEM構造解析ソフトウェアのサポートを行う．'84年に日本アイ・ビー・エム(株)に入社．IBM 3090ベクトル機構，RS/6000などの技術計算用プロセッサをアプリケーション・プログラムの面からサポートする仕事に従事，現在に至る．計算機構造，コンパイラ技術，数値計算アルゴリズムの3者を合わせた，数値計算プログラムの高速化に関心がある．日本応用数理学会会員．