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†

寒川 光††

スーパスカラ計算機では算術演算性能が従来のスカラ計算機に比して飛躍的に強化された．その結果データ移動(ロード／ストア命令)の計算時間に占める比率が増大した．計算速度を考慮するプログラムは計算順序を変更しデータ移動を削減する方法で，大きなチューニング効果をあげられる場合がある．この方法はFortranプログラムからは透過なレジスタへのロード命令の実行やキャッシュへのデータのステージングの回数を，媒介的な方法（計算密度，キャッシュ利用密度）で把え，計算機の個性に合わせた最適化を狙うものである．行列行列積和の例題では約3倍という大幅なチューニング効果を達成した．この方法をプログラミング技法の問題としてではなく，線形代数計算の問題として，ブロック化された定式化で記述すると，ベクトル計算機や階層型記憶装置をもつ計算機にも応用することができ，見通しが良くなる．

1. はじめに

数値計算プログラムにおいてデータの移動が高速計算を実現する上で非常に重要な項目になりつつある．初期の計算機では浮動小数点演算命令は他の命令に比べて時間のかかる命令だったため，浮動小数点演算回数を尺度に計算時間を見積もる方法が定着していた．しかし最近のRISCタイプのスーパスカラ計算機の中には，浮動小数点演算を他の演算と同等，あるいはそれ以上に速く実行する機種が存在する．このような新しいタイプの計算機に対して，これまでの方法で予測された計算時間が実際に計測された時間とかけ離れるケースが出てきている．この理由は主に，記憶域，キャッシュ，レジスタ，演算器間をデータ移動させる時間が演算時間そのものより長くなりつつあることに起因している．この計算機の変化を超LSI論理回路の発達の必然的な結果であると考えると，この傾向は今後さらに強くなると思われる．数値計算プログラムでデータの移動をどのように制御するかという問題は計算機システムと数値計算の両方の分野に跨がる問題なので，既存の数値計算アルゴリズム論とは異質な面を持つ．そこで論旨の焦点を明確にするために，論文の背景と目標をはじめに整理しておきたい．

1.1 背景

Fortranがモデルとした抽象的な計算機を状態遷移マシンと呼ぶ．ここでは計算の進行は記憶域の内容が代入文の左辺の変数の値が更新されてゆくことでとらえられる．プログラム(1)の内積を求める関数で考えると，変数Sの内容が更新されることでとらえられる(**図1**)．状態遷移マシンはひとつの平面的な記憶域しか持たないのに対し，現実の計算機は演算器と記憶域の間にレジスタやキャッシュという作業域を持っている（**図2**）．仮想記憶方式の計算機ではさらに記憶域が実記憶装置と外部の記憶装置（ページ・データセットや拡張記憶装置）に分割されている．このように論理的には1枚の記憶域が物理的には複数の階層から構成されており，代入文が実行されるとき右辺のデータがどの階層から参照されるかで計算時間に差がつく．Fortranプログラムからはこれは見えないのだが，何らかの方法でデータの所在／移動を炙り出して，できるだけ演算器に近い階層に残ったデータで計算が進められるような計算順序にプログラミングしたい．言い換えると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積極的に制御するようにプログラミングしたいのである．

1.2 目標

プログラムでデータの移動量を最適化する試みは，データが外部ファイル上に存在し，その一部分を入出力命令によって読みつつ計算を進めるようなプログラムでは一般的に行われている．このような陽な形での制御と，本論文で扱うFortranからは陰な形で制御する方法とはプログラム上は別のものであるが，プログラミング以前の定式化の考えには共通したものがある(行列データを扱う場合，行列を小行列にブロック化して，小行列単位に計算する考え方は共通している)．

現在，キャッシュを備えた仮想記憶方式の計算機に対して「データを連続的に参照すること」をプログラ

† Blocking Algorithm for Data Transfer Control in Numerical Linear Algebra Programming by HIKARU SAMUKAWA (IBM Japan, Ltd.).

†† 日本アイ・ビー・エム(株)

ミング上の制約として課する場合が多い．行列計算では（Fortran では左側の添字を先に回して2次元配列を1次元化するため）行添字を内側のループで回すことを意味している．これを守りさえすれば，実記憶装置と外部の記憶装置間，およびキャッシュと実記憶装置間の2か所のデータ移動を十分満足のいける性能で制御できる．このような簡便な方法で間に合う理由は，演算そのものがデータ移動量の差を蔽い隠すほどに遅いためと，2か所のデータ移動を制御するルールが同じもので済むからである．

RISC タイプのスーパスカラ計算機では演算が高速化され（しかもデータ移動のためのロード命令と演算命令が並列に動き），レジスタがひとつの記憶階層の役割を担えるだけの数用意されたため，

（1） 演算器とレジスタ間のデータの移動が計算時間の主要部分になる場合がある

（2） キャッシュミスの影響が相対的に大きく現れる

の2点が現象的に従来型スカラ計算機と異なって現れる．**図3**に両者の違いを表した．このような理由から，データの移動を上手に制御したプログラムがそうでないものに対し大きな性能差を示す例は少なくない．本論文の4章に示す行列行列積和の計算例では，従来型スカラ計算機用に（先に述べた連続的な参照を守った）最も標準的と考えられるプログラムを，計算量は変えずに，計算順序だけを変更して3倍以上のチューニング効果を得た．しかしあらためて両方のプログラムを従来型スカラ計算機で実行したところ，その性能差は 20% 以下に留まった．3倍以上の性能差がデータの移動を制御することの必要性を十分示していると思うが，この性能差を計算機価格に換算すると1桁高い価格のモデルと同等の性能を実現していると言える．

最後に，このようなデータ移動の最適化を誰（コンパイラかプログラマ／数値計算アルゴリズムか）が行うかという問題に触れる．おそらく多くの人がコンパイラによる自動的な最適化を期待するだろう．実際に命令の移動など局所的な最適化はかなり上手に処理されるが，大域的な最適化は自動的には処理されにくい．コンパイラより賢いはずのわれわれも，書かれたプログラムから計算手順を分析して計算順序の変更（ネストしたループを分割／入替え）を行うことを難

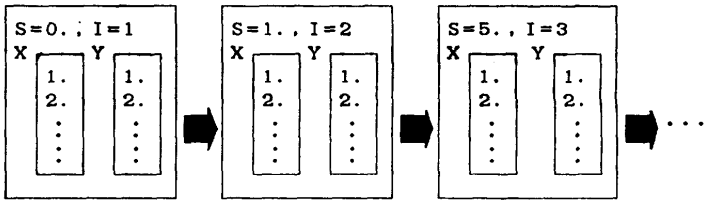


図 1 概念的な計算機
Fig. 1 Conceptual computer.

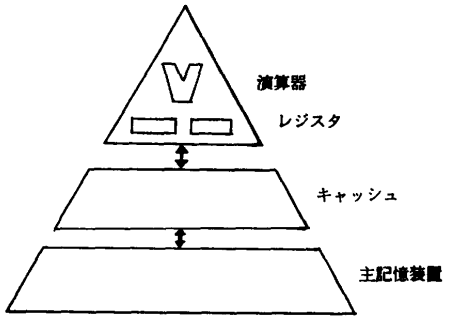


図 2 階層型記憶装置をもつ計算機
Fig. 2 Computer with storage hierarchies.

```
Program (1) DDOT function

      FUNCTION DDOT (N,X,INCX,Y,INCY)
      IMPLICIT DOUBLE PRECISION (A-H,O-Z)
      DIMENSION X(INCX,N),Y(INCY,N)
      S=0.DO
       DO I=1,N
        S=S+X(1,I)*Y(1,I)
       ENDDO
      DDOT=S
      RETURN
      END
```

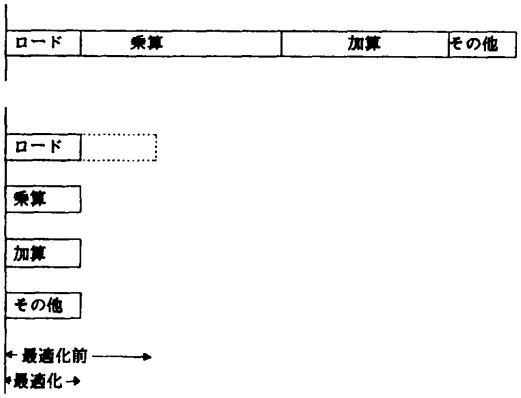


図 3 従来型スカラ計算機とスーパスカラ計算機での計算時間の比
Fig. 3 Computational time on conventional scalar processor and superscalar processor.

しく感じているほどである．筆者はこの問題の解決の最短距離にいるのは数値計算アルゴリズムであると考えている．計算手順は，書かれたプログラムから分析

すべきではなく，初めから種々の条件に合わせて変更／再構成可能な定式化で出発したほうがよい．どのような目標に向けて計算順序を最適化するかというと，「記憶域と演算器の間に一定の容量の作業域を設けた時，記憶域と作業域間のデータ移動量を最小化する」と記述できる．このような計算量とデータ移動量の観点が，数値計算アルゴリズムに求められているのである．

本論文は次章にスーパスカラ計算機の特徴を算術演算性能とデータ移動性能の比に着目して述べる．3章でプロセッサ内部でのデータ移動を制御する（Fortran プログラムから見えないレジスタを見る）手段を紹介する．4章でこの考えをキャッシュ，主記憶域間のデータ移動の制御にも用いる．5章ではデータ移動を制御するプログラミングをブロック化アルゴリズムの観点からまとめる．

2. スーパスカラ計算機

スーパスカラ計算機として IBM の RISC System/6000* (RS/6000) をとりあげ，System/370* (システム/370) の命令セット[1]で動く汎用プロセッサ 3090 と比較してスーパスカラの計算方式の特徴を述べる．

2.1 命令セット

プログラム(1)の内積計算を例に 3090 での計算のダイヤグラムを**図4**に示す．図では先行制御部分（命令の取出し，解読，オペランド取出しなど）は他の命令の実行のサイクルに並行して処理されるものとして点線で示した．したがって命令の実行サイクルのみを計算時間としている．ループを構成する6命令とそれぞれの命令の行う仕事は次のように表すことができる．

ロード命令	LD	F0, Xix	FX
乗算	MD	F0⇐F0＊Yiy	＊ | FY
加算	ADR	F6⇐F6＋F0	＋
添字更新	AR	iX⇐iX＋incx	iX
添字更新	AR	iY⇐iY＋incy	iY
条件分岐	BXLE	i=i+1, 分岐	i=i+1 | 分岐

これに対し RS/6000 では処理を整数系の演算，浮動小数点算術演算，条件分岐の3者に大別し，3者を並行処理しやすい命令セットを用いる．システム/370

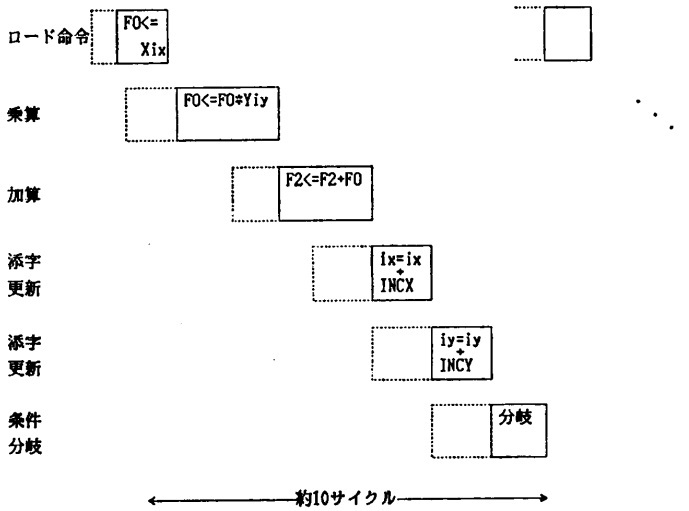


図4 3090 による DDOT 計算のダイヤグラム
Fig. 4 DDOT diagram by 3090 processor.

と比較すると次のような特徴を持つ[2),3)]．

（1） レジスタ同士の演算：記憶域オペランドを算術演算命令に使用しない．
（2） 4オペランド：乗加算 (fpt⇐fpb＋fpa＊fpc) 命令
（3） 32個の浮動小数点レジスタ
（4） 浮動小数点レジスタへのロード命令を整数系の命令とする．またオペランドを指す汎用レジスタの内容を同時更新する (lfdux 命令)．
（5） 命令長の統一：命令は1語（4バイト）長に揃える．

これらの特徴は一度レジスタにロードしたデータを保存し，後続の計算で利用可能にすること，また命令の高速取出しに効果を発揮する．いずれもデータ移動性能が隘路となりやすい現代の計算機を配慮している．内積計算のループは次の4命令で実現される．

ロードと更新	lfdux	fp0, r4, r5	FX | iX
ロードと更新	lfdux	fp2, r6, r7	FY | iY
乗加算	fma	fp1＝fp1＋fp0＊fp2	＊ | ＋
条件分岐	bc	−12	i=i+1 | 分岐

2.2 スカラ・パイプライン

RS/6000 のプロセッサは4つのコンポーネントからなる (**図5**)[4)]．ICU (Instruction Cache Unit) は条件分岐命令を，FXU (Fixed-point Unit) は整数系の演算を，FPU (Floating-point Unit) は浮動小数点演算を処理する．3者は互いに同期をとりながら命令を並行処理する．内積計算ループはデータが DCU (Data

* RISC System/6000, System/370 は IBM Corp. (米国) の商標です．

Cache Unit) に存在すれば2サイクルで実行できる (**図 6**). 複数の命令が同時に実行サイクルをもつのでスーパスカラと呼ばれる.

2.3 乗加算命令

FPU は独立した2つの乗加算命令の乗算と加算を並行処理する[5),6)]. 行列と行列の積和

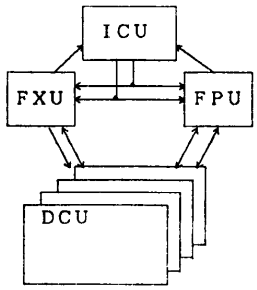
$$c_{ij} \leftarrow c_{ij} + \sum_{k=1}^{m} a_{ik} b_{kj}, \quad i=1 \sim l, \ j=1 \sim n \qquad (1)$$

を計算するプログラム(2)と, それを 2×2 重にアンローリングしたプログラム(3)を示す. (2)の最内側ループは内積計算なので図6と同様に乗加算1回を2サイクルで実行するが, (3)は乗加算4回を4サイクルで実行する (サイクル・タイムが24ナノ秒の RS/6000 の 550 型で 74 mflops を実測した, **図 7**).

2.4 演算・移動性能比

RS/6000 のプロセッサは見かけ上1サイクルに1回乗加算命令を処理できるが, データ供給速度も1サイクルにロード命令1回の性能である. 演算性能を乗加算 (2 flop) を計算する速度で, データ移動性能を 64 ビットのデータをロードまたはストアする速度で測った時, 両者の比, 演算・移動性能比は RS/6000 は 1.0 である. また演算と移動は並行処理されるので, プログラム(3)のように乗加算命令とロード命令の発行速度が一致すると, 一方の計算時間が他方の裏に隠れる形となる. これに対し 3090 など既存のスカラ・プロセッサの多くは 0.1～0.3 の演算・移動性能比であるため, 算術演算回数だけから計算時間を見積もってもあまり大きな誤差は生じなかった.

RS/6000 の計算速度は FPU そのものの速さと, FPU, FXU, ICU の並行処理によって飛躍的に向上したが, それを支えるデータ移動性能そのものはそれほど向上していない. そのためデータ・キャッシュと命令キャッシュを分離する, ロード命令の実行を削減する (豊富なレジスタ, 4オペランド演算命令, コンパイラの最適化) などの方法で, データ移動性能の実質的な強化を計っている. しかし, 線形代数計算のカーネル・ループは一般に演算命令に比べてロード命令の多いものが多く, 行列行列積和のようにロード命令の実行回数を減らすことで大きなチューニング効果をあげられる例も少なくない. この点がスーパスカラの既存のスカラ・プロセッサと性格の異なる点であ



ICU：命令キャッシュ・ユニット, 命令キャッシュを持ち、条件分岐命令を処理
FXU：固定小数点ユニット, 固定小数点(整数)演算命令、ロード、ストア命令を処理
FPU：浮動小数点ユニット, 浮動小数点算術演算命令を処理
DCU：データ・キャッシュ・ユニット

図 5 RS/6000 のプロセッサ・コンポーネント
Fig. 5 RS/6000 processor components.

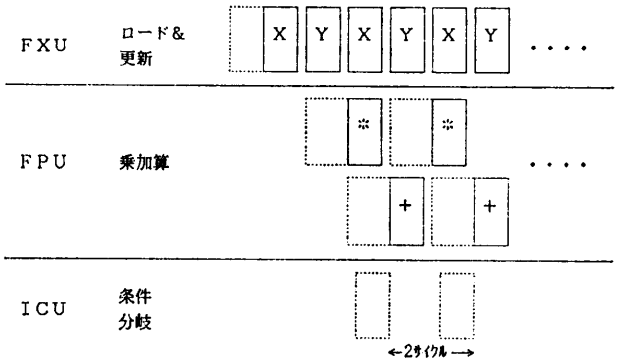


図 6 RS/6000 による DDOT 計算のダイヤグラム
Fig. 6 DDOT diagram by RS/6000 processor.

```
Program (2) JIK-formulation of matrix-matrix multiply-add

      DO J=1,N
        DO I=1,L
          DO K=1,M
           C(I,J)=C(I,J)+A(I,K)*B(K,J)
          ENDDO
        ENDDO
      ENDDO


Program (3) 2*2-unrolling for JIK-formulation

      SUBROUTINE MPYAD (A,LDA,B,LDB,C,LDC,L,M,N)
      IMPLICIT DOUBLE PRECISION (A-H,O-Z)
      DIMENSION A(LDA,M),B(LDB,N),C(LDC,N)
       DO J=1,N-1,2
         DO I=1,L-1,2
          S00=0.D0
          S01=0.D0
          S10=0.D0
          S11=0.D0
           DO K=1,M
            S00=S00+A(I,K)*B(K,J)
            S01=S01+A(I,K)*B(K,J+1)
            S10=S10+A(I+1,K)*B(K,J)
            S11=S11+A(I+1,K)*B(K,J+1)
           ENDDO
          C(I,J)=C(I,J)+S00
          C(I,J+1)=C(I,J+1)+S01
          C(I+1,J)=C(I+1,J)+S10
          C(I+1,J+1)=C(I+1,J+1)+S11
         ENDDO
        .................
       ENDDO
       .................
```

る．

3. スーパスカラ計算機での計算時間

RS/6000 ではプログラムの計算時間はロード命令の実行回数や命令順序に敏感である．この章では演算命令とロード／ストア命令からなるループ・プログラムの計算時間の予測法について述べる．なお，データはキャッシュに存在する（primed cache）ものと仮定する．

3.1 性能評価式

ロード命令が1回のループ計算で何回実行されるか（# LOAD）は，「実行文の右辺の変数で，ループ添字を配列の添字とする変数の重複しない数」として知ることができる．浮動小数点演算命令の数（#FP）は（除算は考えないものとして）乗・加減算の数であるが，乗算に引き続いて行われる加減算は乗加算として1回に数える．ループ計算に必要なサイクル数（#CYCLE）は #LOAD，#FP の大きなものより小さくなることはない[7]．

(#CYCLE)≧MAX(#LOAD, #FP)

この式を性能評価式と呼ぶことにする．右辺の第1項は FXU がデータ供給する性能を，第2項は FPU が演算を行う性能を表し，'MAX' は両者の並行処理を意味している．プログラム(2)と(3)には次のように適用することで，(3)が(2)の2倍の mflops 性能値になることが示せる（下線の項がロード命令を要する）．

```
DO K
  C(I, J)=C(I, J)+A(I, K)*B(K, J)
#LOAD=2, #FP=1
                → #CYCLE≧MAX(2, 1)=2
```

```
DO K
  S00=S00+A(I, K)*B(K, J)
  S01=S01+A(I, K)*B(K, J+1)
  S10=S10+A(I+1, K)*B(K, J)
  S11=S11+A(I+1, K)*B(K, J+1)
#LOAD=4, #FP=4
                → #CYCLE≧MAX(4, 4)=4
```

この方法は Fortran プログラムからレジスタの存在を浮き上がらせて，記憶階層間（キャッシュ→レジスタ）のデータ移動量を数えることを可能にする．その結果，アセンブラやダイヤグラムを考えることなく，単純なプログラムのループ性能を予測することができる．

ストア命令や整数演算も含めた性能評価式は次の形である．

(#CYCLE)≧MAX((#LOAD+#STORE+#FX), (#FP+#STORE))　　(2)

RS/6000 では浮動小数点ストア命令は FPU で解読（デコード）サイクルを持つため[4]，#STORE は両方の項に現れる．この式で if 文の入らないループを，レジスタが不足をきたさない範囲で予測することができる．

3.2 従属性の深さ

性能評価式が不等式である理由は，種々の理由で実行サイクルが延びるケースであるためである．倍精度データが倍語境界に合っていない時とか，演算が 'denormalized 数' を扱う時などはこの例であるが，このようなデータやオペランド番地の特異性によるケース以外に従属性によるケースがある．

（1） 算術演算命令の従属性

浮動小数点演算命令は FPU 内でパイプライン処理されるため，命令（fma 命令）が先行する命令の出力を入力とすると待たされるケースがある[6]．命令列に

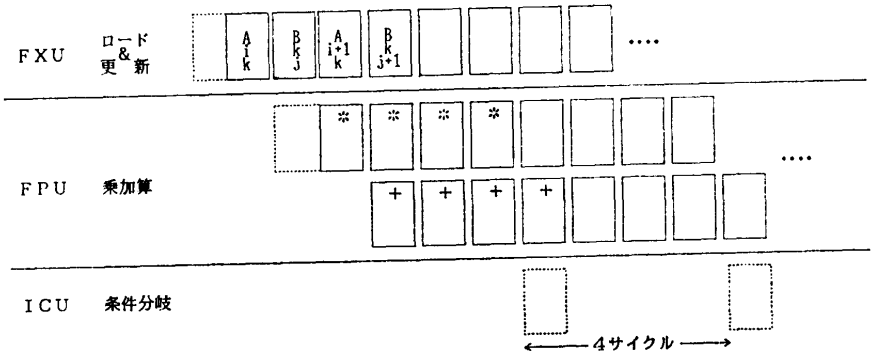


図 7　行列乗算（アンローリング）のダイヤグラム
Fig. 7　Matrix multiplication (unrolling) diagram.

よって待ちサイクルを生ずるケースを示す.

命令例	オペランド fpt＝fpb＋fpa＊fpc	待ちサイクル	パイプライン処理	従属性の深さ
fma fma	fp1, —, —, — —, fp1, —, —	+1	DMAW DDMAW	1
fma fma	fp1, —, —, — —, —, fp1, —	+2	DMAW DDD MAW	2
fma fma fma	fp1, —, —, — —, —, —, — —, —, fp1, —	+1	DMAW DMAW DDMAW	
fma fma	fp1, —, —, — —, —, —, fp1	+1	DMAW DDMAW	1

D：解読，M：乗算，A：加算，W：レジスタへの書込み

実行に影響を及ぼす命令間の距離を従属性の深さと呼ぶことにする．連続する2つのfma命令で，前のfmaの出力が次のfmaのfpaオペランドに用いられると2サイクル，fpbやfpcオペランドに用いられると1サイクル待たされる（fpa側にBoothのリコーダを持つ）．通常のプログラムはコンパイラによってfpaに先行する命令の出力が用いられないように配慮されるため，従属性の深さは1を考慮すればよい．

（2） ストア命令の従属性

命令例	オペランド	待ちサイクル	パイプライン処理	従属性の深さ
fma fma stfdu stfdu	fp1, —, —, — fp2, —, —, — fp1 fp2	+1	DMAW DMAW D DD	1

ストア命令が連続し，ストアすべきデータがパイプラインで計算中の時，後続のストア命令は先行する演算命令が出力をストア・キューに置くまで待たされる[7]．2次元グラフィックスの座標変換プログラムの例が文献[3),4)]に記載されている（プログラム（4-1））．このループは性能評価式からは＃FP＝4，＃LOAD＝2，＃STORE＝2なので6サイクルで実行されそうであるが，7サイクルを要する．fma命令の従属性はコンパイラの計算順序の最適化（B2＋A21＊X(I)を＋A12＊Y(I)より先に計算）によって回避されるが，ストア命令の従属性が残るためである．従属性の深さはパイプラインの段数によるものである．fpaオペランドのケースを除けば深さ1なので，チューニング作業では並列度をひとつ高くすることで解消される[7]．プログラム（4-1）はアンローリングによって座標変換1回あたりの実行サイクルを7から6に短縮できる（プログラム（4-2））．

```
Program (4-1) 2D graphics transformation

      DO I=1,N
       XX(I)=B1+A11*X(I)+A12*Y(I)
       YY(I)=B2+A21*X(I)+A22*Y(I)
      ENDDO

Program (4-2) 2-way unrolling

      DO I=1,N-1,2
       XX(I  )=B1+A11*X(I  )+A12*Y(I  )
       XX(I+1)=B1+A11*X(I+1)+A12*Y(I+1)
       YY(I  )=B2+A21*X(I  )+A22*Y(I  )
       YY(I+1)=B2+A21*X(I+1)+A22*Y(I+1)
      ENDDO
      IF(MOD(N,2).EQ.1) THEN
       XX(N)=B1+A11*X(N)+A12*Y(N)
       YY(N)=B2+A21*X(N)+A22*Y(N)
      ENDIF
```

3.3 計算密度

演算・移動性能比はプロセッサのハードウェアの性格を表す値である．ループ・プログラムの性格を表す値として計算密度（浮動小数点演算数と記憶域アクセス回数の比）を考える．本論文では性能評価式で用いた変数を用いて次の形で定義する．

（計算密度）＝(＃FP)/(＃LOAD＋＃STORE)

＃FPを用いる理由は，乗加算命令やベクトル計算機で行われる乗算と加算のチェイニング機能を生かせるプログラムをアルゴリズムの特徴として評価したいからである．この定義に基づくと内積計算は0.5であり，ベクトル・スカラ積和（$y \Leftarrow \alpha \cdot x + y$）は0.33である．同じ内積でも複素数では1.0となる．計算量が同じなら，計算密度が高いプログラムのほうが，データ移動量が少ない．

3.4 計算密度と演算・移動性能比

計算時間そのものの予測は不要で，アンローリングの効果を知りたいだけの場合，性能評価式によらず，計算密度を検討するだけで済ませられる．計算密度が演算・移動性能比よりも低い時に外側ループ・アンローリングが効果を発揮する．行列行列積和のプログラム（2）は計算密度が0.5で，アンローリングした（3）は1.0である（アンローリングをこれ以上深くする必要はない）．

演算・移動性能比が大きなプロセッサを想定する．そのプロセッサの高い演算性能を引き出すプログラムの形を考えてみたい．式（1）の行列積ではアンローリングを深くすることで計算密度を上げられる．3×3重では1.5（＝(3･3)/(3＋3)）であり$p \times q$重では$(p \cdot q)/(p+q)$となる．このプログラムが性能評価式どおりのサイクル数で計算できるためには，少なくとも$p+q$

個の浮動小数点レジスタが必要である.

3.5 アンローリングの定式化

$p\times q$ 重のアンローリングを施されたプログラムの計算順序を正しく表現する式は，行列 C の p 行 q 列の小行列 C_{IJ} に関する式で記述できる．A_I を A の p 行 m 列，B_J を B の m 行 q 列の小行列とする.

$$C_{IJ}=C_{IJ}+A_I\cdot B_J,\quad I=1\sim l/p,\quad J=1\sim n/q$$

ただし，$A_I\cdot B_J$ は $\sum_{k=1}^{m} a_{ik}\cdot b_{kj}$，$i=1\sim p$，$j=1\sim q$ (3)

(大文字は小行列とその添字，小文字は行列要素と添字を表すものとする).

プログラム(3)では p,q ともに2として外側の2重ループが式(3)の小行列の式（I と J のループ）を，内側のループが要素ごとの式（m のループ）を計算している.

小行列に関するブロック化された式を書く狙いのひとつは，計算機ごとにチューニングを施されて曖昧になりがちな計算順序の問題を，定式化の段階でプログラムと一致させる点にある.

4. 記憶階層とブロック化アルゴリズム

ブロック化アルゴリズムによってデータ移動を制御する考え方は，階層型の記憶域を備える計算機やローカル・メモリーを備えた並列計算機に応用できる．この章では階層型の記憶域(キャッシュ)を持つ RS/6000 においてこの方法の利用例を示す.

4.1 RS/6000 のデータ・キャッシュ

RS/6000 の DCU は容量 64 K バイトで，キャッシュ構成は 128 バイト (16 倍語) をキャッシュ行とするセット・アソシエーティブ方式をとる[8]．キャッシュと主記憶間のデータ移動速度は 128 ビット／サイクルである．主記憶とキャッシュ間のデータのステージング，デステージングは LRU (Least Recently Used) アルゴリズムによるストア・バック方式である．キャッシュミスを発生するとその語を含む1キャッシュ行がキャッシュへ送られるが，その語はパケットの先頭に置かれ，到着した時点で待ち状態は解除される（パケットの残り（トレーリング・エッジ）の移動完了を待たない)．したがってキャッシュミスを発生させた語の位置 (番地) はアクセス時間に影響しない.

なお，RS/6000の下位モデルである320型はキャッシュ容量 32 K バイトで 64 バイトのキャッシュ行の構成をとる.

4.2 行列行列積和におけるキャッシュミスとブロック化アルゴリズム

次数 n の正方行列の行列行列積和（式(1)で $l=m=n$) を考える．プログラム (5-1) に添字 I,J,K を式の添字と一致させてとり，ループ構成を外側から J,K,I の順に書いた．この構成では行列 C の1列を計算するたびに行列 A 全体を1回参照する．A がキャッシュに収まりきれば主記憶からキャッシュへのデータ移動(ステージング)は1回で済むが，溢れると n 回データ移動が繰り返される (図 8)．この点に着目して A をキャッシュに収まりきる p 行 q 列の小行列に分割する（プログラム (5-2))．点線内の3重ループが図 9 の斜線で示した小行列の乗加算を行う．C の小行列

Program (5-1) JKI-formulation

```
DO J=1,N
  DO K=1,M
    DO I=1,L
     C(I,J)=C(I,J)+A(I,K)*B(K,J)
    ENDDO
  ENDDO
ENDDO
```

Program (5-2) blocking

```
data p/50/,q/50/
do Is=1,L-1,p
  do Ks=1,M-1,q
    DO J=1,N
      DO K=Ks,min(M,Ks+q-1)
        DO I=Is,min(L,Is+p-1)
          C(I,J)=C(I,J)+A(I,K)*B(K,J)
        ENDDO
      ENDDO
    ENDDO
  enddo
enddo
```

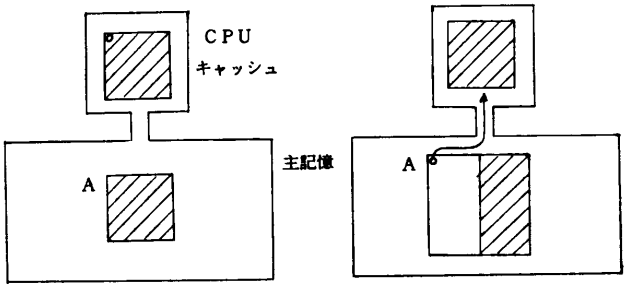


図 8 行列Aの大きさに依存するキャッシュミス
Fig. 8 Cache-miss depending on the size of matrix A.

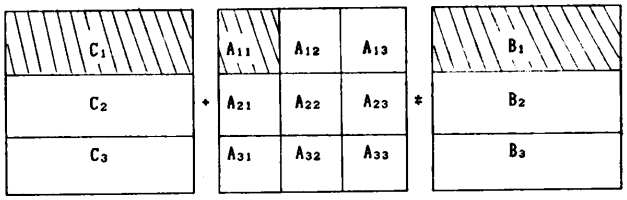


図 9 キャッシュ・ブロック化
Fig. 9 Cache blocking.

の 2〜n 列目の計算中 A に対するキャッシュミスは発生しない．したがって A に対してはセット・アソシエーティブに伴う誤差を無視すれば1回だけのデータ移動になる．RS/6000 の 540 型 (30MHz) で実行すると $n=500$ では 16.2 mflops と 17.9 mflops が計測され，約 10% ブロック化の効果がでた．ブロック化したものは乗加算1回を約 3.35 サイクルで計算しているのに対し，非ブロック化は約 3.70 サイクルを要している．キャッシュ行が16倍語なのでループ16回に1回の割合でキャッシュミスを発生していると考えると，キャッシュミス1回につき 5.6 サイクル (3.70×16−3.35×16) のアクセス時間と見積もることができる．

計算密度でプログラムのレジスタ利用効率を判定したように，キャッシュ利用密度でキャッシュの利用効率を判定したい．

$$\text{(キャッシュ利用密度)} = \frac{\text{(アクセスしたデータ量)}}{\text{(キャッシュと主記憶間のデータ移動量)}}$$

非ブロック化プログラムは，A の要素を乗加算に用いるたびに主記憶域にアクセスするのでキャッシュ利用密度は1である．ブロック化すると n になる．A 以外の行列も含めキャッシュと主記憶域間のデータ移動量を比較する．

	A	B	C	計
非ブロック化	n^3	n^2	$2n^2$	n^3+3n^2
ブロック化	n^2	n^3/q	$2n^2$	$n^3/q+3n^2$

ブロック化によりデータ移動量を q 分の1に削減できたことがわかる．

4.3 JIK 型行列行列積

同様の比較をループ構成を JIK 型として行う．元のプログラム(2)をブロック化したもの（プログラム (6-1)），2×2 重アンローリングを行ったもの（プログラム (3)），両方適用したもの（プログラム (6-2)）に対し，行列次数を 50 から 500 まで測定する．ブロック化は p,q ともに 50 とし RS/6000 の 540 型で実行した．結果を**図 10** に示す．ブロック化をしてないプログラムは $n=250$ から 300 で性能が大きく低下する．250 では行列 A の各列の2キャッシュ行 (500 キャッシュ行) がキャッシュ容量に収まりきるので，一度ステージングしたデータを消さずに計算できる．n が 300 になると K のループの最終段階で，最初にステージングしたデータ ($A(I,1)$ 等) を消さなくてはならない．そのため A の要素は参照されるたびにキャッシュミスを伴う．キャッシュ利用密度は $n=250$ で

```
Program (6-1) blocking

      do Is=1,L-1,p
        do Ks=1,M-1,q
          DO J=1,N
            DO I=Is,min(L,Is+p-1)
              DO K=Ks,min(M,Ks+q-1)
               C(I,J)=C(I,J)+A(I,K)*B(K,J)
              ENDDO
            ENDDO
          ENDDO
        enddo
      enddo


Program (6-2) blocking and unrolling

      do Is=1,L-1,p
       LL=min(L-Is+1,p)
        do Ks=1,M-1,q
         MM=min(M-Ks+1,q)
         call MPYAD(A(Is,Ks),LDA,B(Ks,1),LDB,C(Is,1),LDC,LL,MM,N)
        enddo
      enddo

      (subrooutine MPYAD is in program (3).)

Program (6-3) data compaction

      common/system/leng,ncache
      p=......
      q=......
       do Is=1,L-1,p
        LL=min(L-Is+1,p)
         do Ks=1,M-1,q
          MM=min(M-Ks+1,q)
          ID=0
           do K=Ks,Ks+MM-1
             do I=Is,Is+LL-1
              D(ID+I-Is)=A(I,K)
             enddo
            ID=ID+LL
           enddo
         call MPYAD(D,LL,B(Ks,1),LDB,C(Is,1),LDC,LL,MM,N)
        enddo
      enddo

      (subrooutine MPYAD is in program (3).)
```

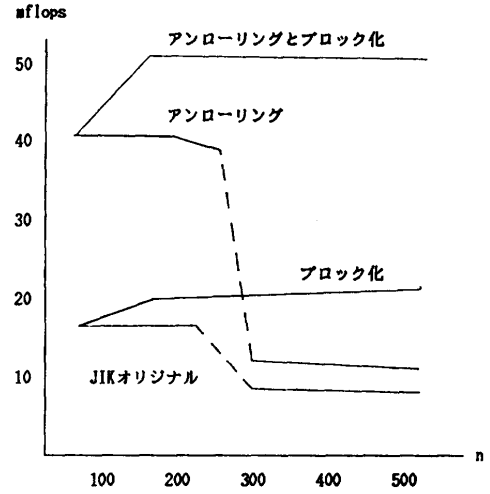


図 10 JIK 型行列積
Fig. 10 Matrix multiplication JIK-formulation.

は1であるが，n=300では1/16である（16倍語移動して1倍語しか使わない）．ブロック化とアンローリングを共に適用すると乗加算1回を約1.2サイクルで計算するが，アンローリングだけでは6サイクル以上費やしている．前節と同様の分析を行うとキャッシュミス1回につき10サイクル程度の待ち時間と見積もれる．行列を列方向に参照すると，キャッシュミス回数が増えるばかりでなく，キャッシュミス1回あたりのペナルティも大きくなる．この理由はアクセス時間以外に，トレーリング・エッジの移動完了まで次の要求が受け付けられない（パケットの先頭が到着してから最後尾の移動が完了するまで8サイクルかかる）ためと考えられる．

RS/6000ではストア命令は性能評価式の両方の項に現れるため，プロセッサ内でのデータ移動を考えると内積型のカーネル・ループが最適である．行列行列積和に対し内積型（Kのループを最内側）にすると行列Aを列方向に参照するため，プロセッサと主記憶域との間のデータ移動は最悪となる．このように最適な計算順序がプロセッサ内部とプロセッサ記憶域間の2か所でくい違うときブロック化は必須となる．キャッシュ・ブロック化されたサブプログラム（MPYAD）では行列を列方向に参照しても性能は低下しない．

行列行列積和の標準的なプログラムをJKI型と考えると，データ移動を最適化する方針でチューニングすると3倍以上（16→50 mflops）速くなる．同じチューニングを従来型スカラ計算機3090で行ってもチューニング効果は20%以下に留まる．これは演算・移動性能比の大きなプロセッサと階層型記憶機構（キャッシュ）を組み合わせた計算機には，データ移動を制御することの重要性を物語っている．

5. ブロック化アルゴリズム

数値計算プログラムの計算速度がデータ移動性能によって制限される現象を，ハードウェア技術の進歩に伴う必然的な傾向と考えると，プログラムでデータ移動を制御する必要性は今後さらに高まるものと考えられる．データ移動はファイル入出力も含めれば計算機では種々の形でプログラミングされている．本論文で扱ったデータ移動はFortranプログラムからは透過なもので，この点がファイル入出力と異なるが，線形代数の観点では類似した定式化がなされる．この章ではブロック化アルゴリズムの観点から本論文をまとめてみたい．

5.1 定式化とプログラミング

（1） ハードウェア依存部分のアルゴリズムからの分離

ベクトル・レジスタ長，キャッシュ容量，演算・移動性能比など，計算機のハードウェアに依存する箇所をサブルーチンやコモン変数として分離すると，プログラムの移植性を高くできる．プログラム(6-2)の場合，小行列のサイズp, qをベクトル・レジスタ長やキャッシュ容量から決めること，またサブルーチンMPYADは計算機（プロセッサ）ごとに性能評価式に合わせたチューニングを行うことができる．

（2） データ圧縮

キャッシュミスに伴う計算速度の変動には複雑な現象が多く，単純な方法で予測しきれない．特にキャッシュ合同（cache congruence）と呼ばれる，行列の先導次元（leading dimension）が2のべき乗の大きな値に一致した時，実質的に利用可能なキャッシュ容量が小さくなる現象が制御しづらい[9]．行列行列積和のようにキャッシュ利用密度がnの計算では，ブロック化したデータを連続的な配列にデータ圧縮（data compaction）することでキャッシュ合同を避けることができる（プログラム(6-3)）[10]．

（3） 記憶階層の数

記憶階層が多くなった場合はブロック化を繰り返し適用することで，各階層ごとにデータ転送を制御することができる．

5.2 アルゴリズムの分類

従来型のスカラ計算機に対してアルゴリズムを計算量で分類することは意味があった．しかし，演算・移動性能比の大きな計算機に対してはデータ移動制御の可能性を加味して考えるべきである．

（1） レベル2性能のアルゴリズム

スーパスカラ計算機で外側ループ・アンローリングによって計算密度を上げられるアルゴリズムをレベル2性能のアルゴリズムと呼ぶことにする（'2'は計算量が$O(n^2)$）．これはベクトル計算機ではベクトル・レジスタに計算結果を累加できるアルゴリズムでもある．行列ベクトル積和がレベル2性能のアルゴリズムの代表である．行列とベクトルの演算でも改訂シュミット直交化（Modified Gram-Schmidt Orthogonalization）のように計算順序の制約（内積を完了しないとベクトルの更新ができない）から，レベル1性能のアルゴリズムの結合でしかないものもある．

（2） レベル3性能のアルゴリズム

ブロック化によりキャッシュ利用密度を上げられるアルゴリズムをレベル3性能のアルゴリズムと呼ぶことにする．行列行列積和や密行列のLU分解は一般にレベル3性能である．LU 分解も軸選択のルールを完全軸選択 (complete pivoting) にするとレベル2性能になる．

このようにアルゴリズムをデータ移動制御の観点から分類すると，計算機とアルゴリズムの相性が整理しやすい．計算機設計者にとっては，ターゲットとするアルゴリズムに対し，記憶階層や演算・移動性能比などの設計値を決定する際の指針として用いることができ，科学技術計算ユーザにとっては，自分の使用する主たるアルゴリズムの性能レベルを知ることで，計算機選択の指針とすることができる．

6. おわりに

数値計算プログラムの高速化を「ハードウェア技術の進歩を自分のプログラムに生かすこと」と捉えると，両者の間には計算機構造（ハードウェアと計算方式），コンパイラ技術，アルゴリズムの3つのハードルが見えてくる．3つを全く改良せずに得られる高速化はサイクル・タイムの短縮の程度に留まるだろう．もし，3つのハードルを上手に越えられれば，サイクル・タイムの短縮比を大きく上回る高速化が達成されるかもしれない．

スーパスカラ計算機では計算機構造の改良については，2章に紹介したような大きな成果をあげることができた．コンパイラ技術もこの新しいタイプの計算機の特性を汲み取る最適化機能を備えつつある．3つのハードルの中で最も対応が遅れているのが数値計算アルゴリズムとプログラミング技法といった利用技術面にある．利用技術の改良の鍵は，計算時間の評価を計算量（浮動小数点乗算回数）中心主義から，「計算量とデータ移動量」に視点変換することにある．過剰なデータ移動を伴うアルゴリズムは，優れたベクトル化や並列化のアイデアをも台無しにしてしまうことが多い．Fortran プログラムからは見えないデータ移動を炙り出してチューニングするプログラミング技法や，アルゴリズム面から組織的にデータ移動を制御するブロック化アルゴリズムを3，4章に紹介した．本論文が新しいタイプの計算機の持つ高いポテンシャルを引き出すことの一助になれば幸いである．

謝辞 RS/6000 プロセッサの詳しい情報を提供してくれた Brett Olsson 氏，Steve White 氏，Richard Fry 氏 (IBM Corp.) に感謝する．

参考文献

1) ESA/370 解説書，N：SA 22-7200 および ESA/370 & S/370 VECTOR OPERATIONS, SA 22-7125, 日本アイ・ビー・エム(株) (1989).
2) 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，N：SA 23-2643, 日本アイ・ビー・エム(株) (1990).
3) Oehler, R. R. and Groves, R. D.: IBM RISC System/6000 Processor Architecture, *IBM J. Res. Dev.*, Vol. 34, No. 1, pp. 23-36 (1990).
4) Grohoski, G. F.: Machine Organization of IBM RISC System/6000 Processor, *IBM J. Res. Dev.*, Vol. 34, No. 1, pp. 37-58 (1990).
5) Montoye, R. K., Hokenek, E. and Runyon, S. L.: Design of the IBM RISC System/6000 Floating-point Execution Unit, *IBM J. Res. Dev.*, Vol. 34, No. 1, pp. 59-70 (1990).
6) Olsson, B., Montoye, R., Markstein, P. and NguyenPhu, M.: RISC System/6000 Floating-point Unit, IBM RISC System/6000 Technology, SA 23-2619, pp. 34-42, IBM Corp. (1990).
7) 寒川 光：IBM RS/6000 のスーパスカラに適したプログラミング，情報処理学会研究会報告，数値解析研究報告，No. 38 (1991).
8) Bakoglu, H. B., Grohoski, G. F. and Montoye, R. K.: The IBM RISC System/6000 Processor: Hardware Overview, *IBM J. Res. Dev.*, Vol. 34, No. 1, pp. 12-22 (1990).
9) 寒川 光：行列計算の計算速度分析に基づく IBM 3090 VF の内部設計に関する考察，SE 論文・1989年，pp. 337-353，日本アイ・ビー・エム(株) (1989).
10) Liu, B. and Strother, N.: Programming in VS Fortran on the IBM 3090 for Maximum Vector Performance, *IEEE Comput.*, pp. 63-76 (June 1988).

(平成4年2月14日受付)
(平成4年7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．