計算機アーキテクチャ 137－4
ハイパフォーマンス 80－4
コンピューティング
（2000．3．2）

ショートベクトルプロセッサ向けループ並列化技法

岸 秀明[†] 古関 聰[††]
小松 秀昭[††] 深澤 良彰[†]

ループ中の配列をベクトルと見なして、配列に対する演算をベクトル実行することで、ループを高速に実行できる。近頃登場した安価なプロセッサにもベクトル演算装置が内蔵されている。このベクトル演算装置はベクトル長が高々数個と短い。配列要素同士の依存関係の複雑さは、ベクトル長が短いほど簡潔になる傾向がある。本研究で対象とするショートベクトルプロセッサは、依存関係による制約が柔軟である事から、ループのベクトル実行に向いていると思われる。本稿では、この性質に注目し、ショートベクトルプロセッサに適したループ並列化技法を提案する。この方法を実際のプログラムに適用し、その効果について示す。

A Loop Parallelization Technique for Short Vector Processors

HIDEAKI KISHI,[†] AKIRA KOSEKI,[††] HIDEAKI KOMATSU[††]
and YOSHIAKI FUKAZAWA [†]

Vector processing of an array in a loop contributes to reduce the execution time. The processor that is equipped with some vector processing units whose vector length is short has appeared in the market recently. The shorter the vector length becomes, the easier the compiler treats dependencies among array elements. It seems that the short vector processor is suited for vector processing, because restriction by dependence is not strict. In this paper, we propose a loop parallelization technique for the short vector processor that takes account of its characteristics. We apply it for some benchmark programs and show its effectiveness.

1. はじめに

大規模データの高速処理が求められている現在、コストをかけずに高性能な計算を行なうために、ループの並列化は不可欠である。ループ並列化手法はさまざまなものが提案されており、有効な手法も多い。

ベクトル演算装置は、SIMD(=Single Instruction / Multiple Data) を実現するプロセッサのハードウェア機構である。しかし、技術的にもニーズ的にもパーソナルコンピュータという世界から離れていたため、数年前まではスーパーコンピュータにしか搭載されていなかった。ところが近年のめざましいプロセッサ技術の発展と、大量のデータ処理を必要とするマルチメディアソフトの普及により、パーソナルコンピュータにもベクトル演算装置の組み込まれたプロセッサが使用されるようになった。

ループ内で繰り返し行われる配列の演算を、配列の要素をベクトルの要素とみなして、ベクトル演算装置で処理することにより、ループを高速に実行できる[1]。しかし、ベクトル実行と逐次実行では各命令の実行順序が入れ替わるために、プログラム中に存在する依存の性質によって、正しい計算結果が得られないことがある。このような場合、コンパイラはベクトル実行ではなく、逐次実行するコードを生成するため、ベクトル実行によるループの並列化を行うことができない。

依存の形を変化させる手法の一つとしてユニモジュラ変換[2] がある。ユニモジュラ変換は、プログラムにおける配列間の依存関係を依存ベクトルという依存距離を成分とするベクトル形式で表し、ループ内の依存ベクトル群に対して一次変換

† 早稲田大学理工学部
School of Science & Engineering, Waseda University
†† 日本 IBM（株）東京基礎研究所
Tokyo Research Laboratory, IBM Japan Ltd.

を施すことで、依存の性質を異なったものに変換する手法である。

本研究では、ショートベクトルプロセッサ向けのユニモジュラ変換のアルゴリズムを提案する。ループ中に含まれる配列間の依存[3]に着目し、コンパイル時に得られる情報から、ベクトル実行を阻害するような依存関係が発見された場合に、ユニモジュラ変換によってベクトル実行を阻害する依存を阻害しない依存に変換する。

本研究では、ユニモジュラ変換のためのコードはループを単位として挿入するものとする。また、ループ中にif文などの分岐を含まないものとする。

2. 依存ベクトル

ベクトル実行では、ループ内に配列の演算命令があった場合、要素数がループイタレーション数に等しいベクトル同士の演算命令と解釈される。従って、図1のようなプログラムがあった場合、逐次実行では、(i=1, S_1) → (i=1, S_2) → (i=2, S_1) … (i=100, S_2) という順番で命令が実行されるのに対して、ベクトル実行では、(i=1, S_1) … (i=100, S_1) → (i=1, S_2) … (i=100, S_2) という順番で命令が実行される。

```
For i=1 to 100
    S1 : A[i] = B[i] + 3
    S2 : B[i] = D[i+1] + F[i]
End For
```

図1 サンプルプログラム1

```
For i=1 to 100
  For j=1 to 100
    E[i-1][j] = A[i][j-1] - 4
    A[i][j] = B[i][j+3] + 3
    C[i][j+1] = A[i][j+1] + D[i][j]
  End For
End For
```

図2 サンプルプログラム2

図2のサンプルプログラム2は、逐次実行とベクトル実行で実行順序が入れ替わったしまうため、ベクトル実行不可能である。このように実行順序が入れ替わった場合に計算結果が正しいかを判定するための道具として、依存ベクトルを用いる。依存ベクトルとは、ループに含まれる配列に対して、ループのネストに等しい次元を持ち、配列に値が書き込まれてから、読み出されるまでにかかるループイタレーション数を成分とするベクトルである。図2のようなプログラムでは、依存ベクトルは

$$\vec{d} = (0,1), (0,-1)$$

と表される。

図2のサンプルプログラム2では、A[1][1], A[1][2] の配列値を読み書きする順序が入れ替わってしまう。これはベクトル実行した際に計算結果が正しくないことを表している。A[1][2] に対する読み書きの順序が入れ替わってしまう点は、依存が逆依存であるから、図2のサンプルプログラム2の3行目のA[i][j+1] をZ[i][j+1] のようにリネームしてやることで問題を回避できる。

図2のサンプルプログラム2の2行目の配列Aに対する書き込みと、3行目の配列Aに対する読み込みの間の依存関係は逆依存であり、ベクトル実行を阻害しない。このときの依存ベクトルは、$\vec{d} = (0,-1)$ であり、依存ベクトルの成分は負である。

逆に図2のサンプルプログラム2の2行目の配列Aに対する書き込みと、1行目の配列Aに対する読み込みの間の依存関係はイタレーションにまたがる真依存であり、ベクトル実行を阻害する。このときの依存ベクトルは、$\vec{d} = (0,1)$ であり、依存ベクトルの成分は正である。

依存がループ独立な依存である場合は、ベクトル実行を阻害しない。このときの依存ベクトルの成分は0である。

以上のことから、ループがベクトル実行されるためには、対象となるループを表す依存ベクトルの次元成分がすべて正でない事が必要である。

3. ベクトル実行のための手法

依存ベクトルによって、ループがベクトル実行できない形であると判断された場合でも、ループ内の一部の命令だけはベクトル実行可能な場合がある。

ループがベクトル実行できない形であるということは、ループ中にイタレーションにまたがる真

依存を持つ命令が存在し、イタレーションにまたがる真依存によって、真依存によるサイクルが形成されていることを示している。そのような場合でも、依存の性質によってはプログラムを変形させることにより、一部の命令をサイクルから切り離すことで、サイクルに含まれていない命令だけでもベクトル実行させることができる。

本手法では、ループ内の命令をショートベクトルプロセッサで、ベクトル実行可能な形に変形させるに当たって、効果的と思われるいくつかの変形手法を組み合わせて用いる。それらの変形手法をあげる。

- 命令の順序交換
  2つの隣接する命令の実行順序を交換する。
  交換対象となる2つの命令文の間にループ独立な依存が無ければ、交換可能である。
- ループ分割
  ループを命令文を単位として、複数のループに分割する。
  サイクルを形成する真依存で結ばれていない命令は分割可能である。
- ループ交換
  ネストされた隣接する2つのループの深さを交換する。
  多次元ループの場合は、ある深さのループがベクトル実行できない場合、そのループよりも浅いループがベクトル実行可能であっても、実行時にベクトル実行されないため、並列化の意味が無い。そこでベクトル実行可能なループは深い方へ、ベクトル実行不可能なループは浅い方へ深さを交換する。
  交換可能であるための十分条件は、以下の項目全てに該当する依存ベクトルが存在しないことである。
  (1) 交換対象となる2つのループよりも浅いループを表す、依存ベクトルの成分が全て0である。
  (2) 交換対象となる2つのループのうち、浅い側のループを表す、依存ベクトルの成分が負である。
  (3) 交換対象となる2つのループのうち、深い側のループを表す、依存ベクトルの成分が正である。
- ユニモジュラ変換
  変換行列 T を用いて、依存ベクトルに一次変換を施して、依存ベクトルの向きを変化させる。
  依存ベクトルの性質によっては、必ずしも最適な変換行列 T が存在するとは限らない。どのような依存ベクトルに対しても、ユニモジュラ変換を行うことができる。

4. ベクトル実行のための変換行列

4.1 ショートベクトルプロセッサの特性

本研究で対象とするのはベクトルプロセッサの中でも、ショートベクトルプロセッサと呼ばれる部類である。

ベクトルプロセッサと異なる部分は、スーパーコンピュータなどに搭載されているベクトルプロセッサはベクトル演算器によって数十ものベクトル要素を一度に計算できる。これに対して、ショートベクトルプロセッサは、32bitの演算器を8bitの演算器4つとみなして命令実行するPentium IIIプロセッサのように、一度に計算できる要素は高々数個である。この一度に計算できる要素数をブロック数と定義する。

3章において、イタレーションにまたがる真依存がベクトル実行を阻害すると述べたのは、ベクトルプロセッサが数十ものベクトル要素を一度に計算できるからである。依存ベクトルの成分は依存距離、すなわちある配列値が書き込まれてから、読み出されるまでのイタレーション数を表している。依存距離がブロック数以上の場合は、ループの変形によって真依存によって形成されるサイクルを断ち切ってベクトル実行させることができる。ショートベクトルプロセッサにはブロック数が小さいという特性から、依存ベクトルの成分が正であっても、ある程度の依存距離が確保できればベクトル実行可能であるという利点がある。

この利点をブロック数4として、サンプルプログラムを用いて説明する。

```
For i=8 to 100
    A[i] = A[i-8] + Const
End For
```

図3 サンプルプログラム3

図 3 のサンプルプログラム3のような場合は、依存距離が8とブロック数4より大きい。そのためベクトル実行した場合に一度目のベクトル実行で、A[8], A[9], A[10], A[11] が書き込まれ、三度目のベクトル実行で A[16], A[17], A[18], A[19] を計算する際に読み込まれる配列値が一度目の実行で計算されており、逐次実行と同じ計算結果を得ることができる。

```
For i=4 to 99
    B[i] = A[i-4] + Const
    A[i] = C[i+2] + D[i]
End For
```

図4 サンプルプログラム4

図 4 のサンプルプログラム4のような場合は、依存距離が4とブロック数4に等しく、配列Aに関する命令が複数行にまたがっているため、図 5 のサンプルプログラム4'のように変形することで、ベクトル実行した場合に逐次実行と同じ計算結果を得ることができる。サンプルプログラム4'における Block とは、ブロック数のことである。

```
For i=1 to 24
  For j=0 to Block-1
    B[4*i+j] = A[4*(i-1)+j] + Const
    A[4*i+j] = C[4*i+j+2] + D[4*i+j]
  End For
End For
```

図5 サンプルプログラム4'

次のセクションで、プロセッサの特性に基づいて、プログラムをショートベクトルプロセッサでベクトル実行可能な形へ変換するために必要な、変換行列の生成について述べる。

4.2 変換行列の生成

ショートベクトルプロセッサによる特性から、ループをベクトル実行させるためには、依存ベクトルの全成分が0以下であるか、ブロック数以上でなければならない。最内周ループがベクトル実行できなければ、ループ全体がベクトル実行できないので、最内周ループの依存を表すベクトルの成分を変換させる。n次の依存ベクトル $\vec{d}$ がm個あった場合の変換過程は、下の式で表される。

$$\begin{bmatrix} \vec{d}_1 \\ \vec{d}_2 \\ \vdots \\ \vec{d}_m \end{bmatrix} \begin{bmatrix} 1 & 0 & \dots & x_1 \\ 0 & 1 & \dots & x_2 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \dots & x_n \end{bmatrix} = \begin{bmatrix} \vec{d'}_1 \\ \vec{d'}_2 \\ \vdots \\ \vec{d'}_m \end{bmatrix}$$

これより、図 6 のn元一次連立不等式を満たすような $\vec{x} = (x_1, x_2, \dots, x_n)$ を求めてやればよい。式中の k はm個ある $\vec{d}$ を識別している。

$$\sum_{i=1}^{n} d_{ki} x_i \leq 0$$

または。

$$\sum_{i=1}^{n} d_{ki} x_i \geq Block$$

図6 n元連立一次不等式

変換された依存ベクトルの成分は、依存距離であるため整数でなければならない。n元連立一次不等式の整数解は一意に定めることができず、また解が必ずあるとは限らない。

そこで次のようなアルゴリズムを用いて $\vec{x}$ を求める。

＜アルゴリズム＞

(1) 依存ベクトルの正規化

m個ある依存ベクトルのうち、一次従属なものを取り除き、昇順にソートする。

(2) 暫定解の導出の準備

n次であれば、正規化された依存ベクトルの中からn個を取り出し、$\vec{x}$ と組み合わせたn個の連立一次方程式を作る。この際、定数項部分は全て -1 とする。これは、図 6 で示されている条件を満たすように、定数項が0以下もしくはブロック数以上となればよく、依存距離の絶対値が小さい方が効率が良いためである。定数項が負であるほうが、ブロック数によらずベクトル実行可能な形であるため、-1 としている。0とすると、$\vec{x} = \vec{0}$ が解となってしまい、変換行列が得られない。

(3) 掃き出し方による導出

(2) によって作成されたn元連立一次方程式を掃き出し方を用いて解く。しかし、掃き出し方では除算を頻繁に行うが、除算を行うと得られる解が整数となる保証がなくなるため、本アルゴリズムではこの点を改良し、除算は一切行わない。最小公倍数を用いることで除算なしに掃き出しを行っていく。

掃き出しの最後に $px_n = q$ という形になったところで、$x_n = lcm(p,q)/p$ として仮の解 x_n を定める。もともと定数項部分を-1とおいているだけなので、定数項である q の値を $q' = lcm(p,q)$ と置き換えても、不等式を満足することが保証されている。

(4) 後退代入

仮の解が求まったら、掃き出したのと逆の手順で残りの $\vec{x}$ の成分を求め、暫定解とする。このときも x_n を求めたときと同じように、最小公倍数を用いて整数解を得る。

(5) 暫定解の検証

得られた暫定解とすべての依存ベクトルが図 6 のn元一次連立不等式を満たしているか検証する。満たしていれば解が求まったことになり、計算終了となる。満たしていない場合は2に戻り、今選んでいたn個とは異なる組み合わせのn個の正規化された依存ベクトルを取り出して暫定解を求める。

(6) 反復処理

2から5を解が求まるまで繰り返す。正規化された依存ベクトルの中から、n個取り出す組み合わせ全てを取り出して計算しても、図 6 のn元一次連立不等式を満たさなかった場合は、解なしとなり、ベクトル実行させるための変換行列が存在しないため、ベクトルか不可能なループであることが分かる。

以上のようにして求められた変換行列を用いて依存ベクトルを変換することで、変換行列が求まるならば、最内周ループをベクトル実行可能な形に変形することができる。

5. 評　　価

本章では、4章で述べたアルゴリズムをベンチマークプログラムに適用し、最内周ループがどれだけベクトル実行可能な形に変形することができたかを示す。

評価をとるに当たっては、最内周ループ内の命令をステップと呼ぶ粒度に分解し、各ステップを本手法を用いなくてもベクトル実行可能なステップ、本手法を用いることによってベクトル実行可能なステップ、本手法を用いなくてもベクトル実行不可能なステップの3つに分類し、各ベンチマークプログラムの総ステップに占める割合を示す。

ステップとは、定数・配列ではない変数・配列の1要素を項とし、項同士の四則演算および代入ならびに項の累乗を1ステップとた。配列の添字式については、ステップの対象外とした。

対象としたベンチマークは、固有値問題を解決するeispack、線形問題を解決するlinpack、24のマイクロカーネルから構成されるlivermore、Perfect Benchmarkからpueblo,qcdと、5種類である。

評価結果を表す表2において、分類とは以下に示す通りである。また、ベンチマークプログラム名の後の数字は、そのプログラムに含まれる最内周ループのステップ数である。

表1 分類表

分類	意味
A	本手法も用いなくてもベクトル実行可能なステップ
B	本手法を用いることでベクトル実行可能なループ
C	本手法を用いてもベクトル実行不可能なループ

表2 本手法によるループの変換結果

プログラム名 (ステップ数)	結果		
	A	B	C
eispack(1433)	63.1%	10.6%	26.3%
linpack(378)	27.8%	19.6%	52.6%
livermore(369)	62.6%	6.2%	31.2%
pueblo(2714)	80.1%	9.5%	10.4%
qcd(2997)	86.0%	13.1%	0.9%

eispackでは、$S = S + A[i,j]$というような、自分自身を被演算子とする演算結果を自分自身に代入する命令が多く含まれていた。このような命令は、依存距離1の真依存でサイクルが形成されており、ショートベクトルプロセッサでのベクトル実行に向いていないが、本手法によって全体の1割がベクトル化できた。

linpackでは、依存距離1の真依存を持つ1次元配列同士の演算命令が多く含まれていた。1次元配列では、ユニモジュラ変換によるスキューイングを用いて依存の性質を変えることが難しく、ベクトル実行可能な形に変形しにくい。その中で、20%近くがベクトル化できたことは、本手法が効果的に働いたものと思われる。

livermoreは、自分自身を被演算子とする演算結果を自分自身に代入する命令と、依存距離の短い真依存を持つ1次元配列同士の演算命令が多く、評価対象とした5つのプログラムの中で最もベクトル化に向いていなかった。その中でも本手法によってベクトル実行可能な部分を増やすことができた。

puebloでは、配列の添字式にループ変数ではない変数が含まれていることが多く、本アルゴリズムの対象とはならない命令が多かったが、全体の1割ほどベクトル化することができた。この点はアルゴリズムの改良やループバージョニング技法[5]を取り入れることによって、さらに多くの命令をベクトル化できるものと思われる。

qcdでは、短い依存距離を持つ命令が少なく、本手法によって全体の99%がベクトル実行できる形となった。

本手法によってベクトル実行できるようになった命令が、今回評価対象となった5つのベンチマークプログラムにおいて、それぞれ全体の1割から2割を占めることが分かった。このことから、本手法はショートベクトルプロセッサのループ並列化技法の一つとして、激的な効果を生むほどではないが、有効な手法であることが示されたと思われる。

6. おわりに

本研究では、ループをベクトル実行可能な形に変形し、ベクトル実行することを、近頃登場したショートベクトルプロセッサと、スーパーコンピュータなどに搭載されているベクトルプロセッサで比較した場合、アーキテクチャ上の違いから、ショートベクトルプロセッサの方が変形する際の制限が緩いことに着目した。ショートベクトルプロセッサは登場してからあまり時間が経っていないので、専用のベクトル化アルゴリズムというものが少ないので、ショートベクトルプロセッサに特化したベクトル化アルゴリズムを考案した。

評価にあたっては、ベクトル化の難しい依存距離の短いものを多く含むプログラムを用いても、全体的に1割から2割のベクトル化の向上が認められた。このことから、本手法はショートベクトルプロセッサのループ並列化技法の1つとして有効であることが認められた。

現段階では依存ベクトルの成分に変数が含まれていないものと仮定している。今後は変数があった場合に、配列の添字式に含まれる変数の性質による分類[4]やループバージョニング手法[5]と組み合わせて、ベクトル実行可能であるための変数の条件を抽出することで、より多くのループをベクトル実行できる形に変換できるアルゴリズムを改良していく。

参考文献

1) R. Allen and K. Kennedy: 'Vector Register Allocation', IEEE Transactions on Computers, VOL.41, NO.10, pp.1290-1317, OCT. 1992

2) Michael E. Wolf, and Monica S. Lam: 'A Loop Transformation Theory and an Algorithm to Maximize Parallelism', IEEE Transactions on Parallel and Distributed Systems, VOL.2, NO.4, pp.452-471, OCT. 1991

3) H. Zima and B. Chapman, 村岡 洋一訳: 'スーパーコンパイラ', オーム社(1995)

4) G.Gina, K.Kennedy and C.Tseng: 'Practical Dependence Testing', PLDI, Vol.26, No.6, pp.15-29, 1991

5) M.Byler, J.R.B.Davies, C.Huson, B.Leasure and M.Wolfe: 'Multiple Version Loops', In Proceedings of the 1987 Interntional Conference on Parallel Processing, NewYork, pp312-318, 1987