

# SIMD 最適化向けソースコードレベルでのコード変形

蒲野 茂幸

佐々 政孝

東京工業大学 大学院情報理工学研究科 数理・計算科学専攻

## Abstract

*SIMD (Single Instruction Multiple Data stream)* 命令では、1 命令で同一演算、同一型の複数の命令を並列に実行でき、*SIMD* 命令をうまく使うことによってプログラムの実行時間を大きく改善することができる。*SIMD* 命令を生成するコンパイラとして、ソースコードレベルとアセンブリ言語レベルで解析ができるものを使用する。コンパイラによる *SIMD* 命令を生成するにはさまざまな変形方法を組み合わせることが必要であり、その手法には 2 つの言語レベルのどちらかに向いている方法があるからである。本研究では、これらの変形手法のうちソースコードレベルに向いている手法をえらび、ソースコードレベルでのコード変形器の設計を行った。また、*SIMD* 化の変形を行って、*SIMD* 化できなかった場合に実行時間が増加してしまうことがある。その対策として *SIMD* 化できるかどうかの判定を行い、できなかった場合には変形前のプログラムへもどした。本研究で設計したソースコードレベルでのコード変換器を *COINS* コンパイラインフラストラクチャ上で実装を行い、マルチメディア向けのベンチマークを使用して評価を行った。提案手法でコード変換を行い *SIMD* 化可能と判断されたループに対して、期待される命令レベルの変換が行われた場合、既存のコンパイラ以上の *SIMD* 命令が生成できることがわかった。また、*SIMD* 化ができなかった場合の実行時間が増加してしまうということがなくなった。

## 1 はじめに

*SIMD (Single Instruction Multiple Data stream)* 命令では、1 命令で同一演算、同一型の複数の命令を並列に実行できる。図 1 の例だと足し算のスカラ命令 4 つを、複数のスカラデータを保存できる *SIMD* レジスタを使用し

て *SIMD* 命令 1 命令で実行している。また、同一命令を並列実行する *SIMD* 命令は、大量の同一演算の命令を処理する必要があるマルチメディア処理に強い。よって、ゲームや物理現象のシミュレーションなどで高度な処理が求められている中、有効利用することが重要である。

プログラムを *SIMD* 化する方法として現在とられているのが、プログラムによるアセンブリ言語や *SIMD* 命令と 1 対 1 に対応する関数 (intrinsic ルーチン) を使って *SIMD* 並列化を行うことである。しかし、プログラムによる *SIMD* 命令の生成は特別な知識が必要になったり、プログラムがアーキテクチャ依存になってしまったりという問題がある。よって、通常のソースコードに対してコンパイラによって *SIMD* 命令を自動生成する技術を確立することが重要になってきている。

コンパイラで *SIMD* 命令を生成する方法としてパターンマッチングを行う方法がある。しかし、パターンマッチングだけだと記述したパターンと意味は同じでもわずかに書き方が違うプログラムに対して *SIMD* 命令を生成することができない。たとえば、配列のポインタ表現などがこれにあたる。そこで、1 つのパターンで多くの記述に対応できるようにするためにプログラム変形を行う必要がある。

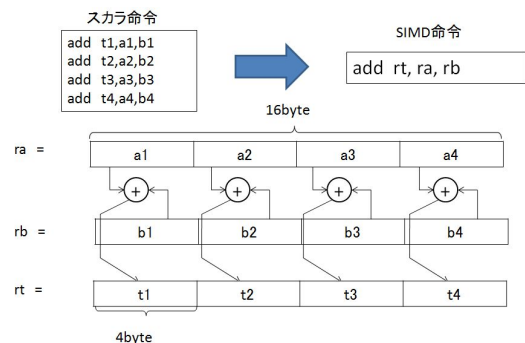


図 1. SIMD の例

本研究で使用するコンパイラとして、ソースコードレベルとアセンブリ言語レベルで解析を行うものを仮定する。ソースコードレベルではループや配列の解析などを容易にでき、アセンブリ言語レベルでは命令スケジューリングやレジスタへの操作などソースコードレベルより細かい操作ができる。本研究では、SIMD 命令を生成するには両方のレベルでの変形が必要だと考え、ソースコードレベルでの特徴を生かした変形方法を選択し、ソースコードレベルでのコード変形器の設計を行う。また、SIMD 化できなかった場合の実行時間の増加を抑えるために、ループ単位での SIMD 化判定を行うことを目的とする。これによって、通常のスカラ最適化を行った場合より SIMD 化を行った方が常に効率のよいコードを生成することができる。

## 2 SIMD 化向けコード変形

SIMD 化向けのコード変形には、ソースコードレベルとアセンブリ言語レベルの 2 つの段階が考えられる。本研究では、この 2 つの段階でのコード変形の操作を明確に分ける。

### 2.1 ソースコードレベルでの変形

ソースコードレベルでの変形では、ソースコードレベルで表現可能な変形を文単位まで行うことにする。

ソースコードレベルで行う変形の特徴を下記に示す。

- if 文や for ループなどの制御構造が容易に認識できる  
アセンブリ言語レベルだとこれらの制御構造は分岐命令を組み合わせることによって実現される
- 配列構造が容易に認識できる  
アセンブリ言語レベルだと配列構造がそのアドレス計算を含むいくつもの命令列に分解されてしまう

本研究では、以上のようなソースコードレベルの特徴を踏まえて、SIMD 命令生成のための変形手法の中からソースコードレベルに適したものを選び実装を行った。さらに、アセンブリ言語レベルでの変形とセットで行うことにより、SIMD 命令とのパターンマッチングを目指す。本研究の対象はこのレベルでの変形として、3 節で詳しく説明を行う。

### 2.2 アセンブリ言語レベルでの変形

アセンブリ言語レベルでの変形では、命令単位での変形を行う。ソースコードレベルとの違いとして、レジスタやアーキテクチャ特有の命令などを意識した変形が行える。また、演算レベルでのより細かい単位で命令スケジューリングを行うことができる。ここでは、このレベルで行う研究について示す。

SIMD 命令では、16byte などの SIMD 命令専用のレジスタを持っていて、メモリ上で連続している配列参照などのデータを SIMD のロード命令を使うことによって、1 命令で SIMD レジスタに代入することができる。しかし、このロード命令を使うにはデータの先頭アドレスが 16byte 境界でなければならないなどの制約がある。これに対し shift 命令や shuffle 命令などを使用してデータを整えるアルゴリズム [1] が開発されている。

また、SIMD レジスタ上のデータ位置を考慮して、SIMD 命令数を減らす研究 [2] もおこなわれている。このように、SIMD レジスタや SIMD 命令を意識した変形というのは命令レベルで行うべきである。

## 3 ソースコードレベルでのコード変形器の設計

3 節では、提案するソースコードレベルでのコード変換器の設計について述べる。図 2 には、コード変形器全体のフローチャートを示した。また、これらの手法はすべてループ単位で行うことにする。

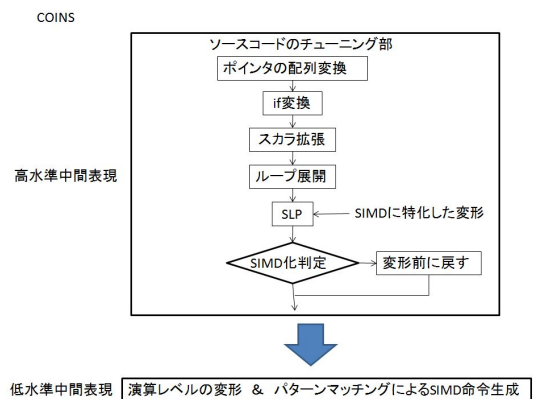


図 2. ソースコードレベルでのコード変形器

### 3.1 ポインタ配列の配列変換

C言語では、配列がポインタ表現として書かれていることが多い。しかし、ポインタ配列をそのまま使用するとコンパイラによる解析を難しくする。これは、SIMD化を行う場合でも問題となる。そこで、Symbolic Analysisの手法を使用して、ポインタ配列を標準形式に変形して、配列表現に逆変換する手法 [4] が提案されている。本研究では、この変形のアルゴリズムを図2のようにコード変形の最初に行うことによってそれ以降の変形のアルゴリズムでポインタ配列を考慮する必要がなくなるようにした。

次に、ポインタ表現の配列を配列に変換するアルゴリズムについて説明する。ポインタ演算の解析は、帰納変数を解析することと同等とみなすことができる。帰納変数とは、ループの中で定数刻みで値が変わる変数で下記のように定義することができる。

$$j = c1 * i + c2$$

ここで、 $i$  が帰納変数、 $c1, c2$  はループ不変式である。Engelen らはこの帰納変数を解析するために CR (Chains of Recurrences) 形式を導入した。CR 形式というのは Zima によって開発され、後に Bachmann, Zima, Wang によって改良された。CR 形式を説明するのに下記のようなプログラムのポインタ  $F(i)$  について考える。

```

F[0] :=  $\phi_0$ 
cr1 :=  $\phi_1$ 
      :
crk :=  $f_k$ 
for i = 1 to n do
    F[i] := F[i - 1]  $\odot_1$  cr1
    cr1 := cr1  $\odot_2$  cr2
      :
    crk-1 := crk-1  $\odot_k$  crk
end for

```

ここで、 $cr_j, j = 1, \dots, k$  は、帰納変数であり、 $\odot \in \{+, *\}$  である。ポインタ  $F(i)$  を解析しようとしたとき、 $F(i)$  は帰納変数である  $cr_1$  と依存関係にある。また、 $cr_1$  は  $cr_2$  と依存関係がある。ここで、CR 表現を使うことによって、ループ中のポインタ  $F(i)$  をそれと依存関係がある帰納変数と合わせて1つのCR表現で簡単に表すことができる。ポインタ  $F(i)$  に対するCR形式は下記のように書く。

$$F[i] = \{\phi_0, \odot_1, \{\phi_1, \odot_2, \dots, \{\phi_{k-1}, \odot_k, f_k\}_i\}_i\}_i$$

これは、カッコを省略して、

$$F[i] = \{\phi_0, \odot_1, \phi_1, \odot_2, \dots, \phi_{k-1}, \odot_k, f_k\}_i$$

のように書くことができる。

このCR表現を使って、ポインタ配列  $\Rightarrow$  CR形式  $\Rightarrow$  配列の手順で変形を行う。

次に、CR表現を使って配列を表し方について説明する。ポインタアクセスをCR形式で表すと、 $\{\phi_0, +, \dots, f_k\}_i$  となり、 $\phi_0$  はポインタ表現となり、 $\phi_j (j=1, \dots, k-1)$  と  $f_k$  は、整数型の表現となる。つまり、配列  $a[n]$  で言うと、これはポインタ表現の  $*(a+n)$  と同じ意味を表わし、 $\phi_0$  が  $a$  であり、 $\phi_j (j=1, \dots, k-1)$  と  $f_k$  が  $n$  にあたる。ここでは、プログラム

```

1: p = &a[0];
2: for(i = 0; i < N; i++) {
3:     ....
4:     ...*p...;
5:     p+=i;
6:     ...
7: }

```

を例にポインタ配列を配列に変換を行う。

#### i, 帰納変数のCR変形

ループ変数の初期値を  $a$ 、ループが1回まわるとの増加数  $s$  とすると、ループ変数  $i$  をCRであらわすと、

$$\{a, +, s\}_i$$

となる。このループでは  $a=0, s=1$  としているので、 $\{0, +, 1\}_i$

となる。ポインタ演算  $p+=i$  を見ると、 $i$  があるので先ほどのCRと置き換えることができ、

$$p = p + \{0, +, 1\}_i$$

となる。また、この文は  $V=V+C$  ( $C$  はループ変数  $i$  のループで定数、またはCR)の形をしているので、 $p$  は帰納変数であることがわかる。ここで、 $V+C$  は  $\{V_0, +, C\}_i$  ( $V_0$  はループ変数  $i$  の入り口での  $V$  の初期値)に変形できるので、

$$\{p_0, +, \{0, +, 1\}_i\}_i$$

になることがわかる。

#### ii, CRの置き換え

次に、 $p$  が参照されている部分を操作  $i$  で変形したCRと置き換える。 $*p$  を

$$*\{p_0, +, \{0, +, 1\}_i\}_i$$

に変形する．また， $p_0$  は  $a$  となるので，ここで置き換える．

### iii, 配列への変換

CR 形式  $\ast(\{a, +, \{0, +, 1\}_i\}_i)$  から配列に変換するには  $CR^{-1}$  という表 1 にあるルールを使用する． $\{0, +, 1\}_i$  に表 1 のルール 9 を適用すると， $\ast(\{a, +, i\}_i)$  となる．次に，ルール 1 を適用して， $\ast(a + \{0, +, i\}_i)$  となる．最後に， $\{0, +, i\}_i$  にルール 10 を適用すると， $\ast(a + (i^2 - i)/2)$  となり，配列  $a[(i^2 - i)/2]$  に変形できる．

表 1.  $CR^{-1}$  変換のルール

	変換前	変換後	条件
1	$\{\phi_0, +, f_1\}_i$	$\Rightarrow \phi_0 + \{0, +, f_1\}_i$	$\phi_0 \neq 0$
2	$\{\phi_0, *, f_1\}_i$	$\Rightarrow \phi_0 * \{1, *, f_1\}_i$	$\phi_0 \neq 0$
9	$\{0, +, f_1\}_i$	$\Rightarrow i * f_1$	$i$ は $f_1$ の中には存在しない
10	$\{0, +, i\}_i$	$\Rightarrow (i^2 - i)/2$	

このアルゴリズムには 3 つの制約がある．

- ポインタの別名がある場合は行わない
- break 文のあるループに対しては行わない
- 関数間の解析は実行されない

これらの制約に当てはまるループは配列への変換がされないので，SIMD 化は行われない．

### 3.2 if 変換

SIMD 並列化を行うにあたって if 文の扱い方が問題になってくる．ループ展開を行った同一の if 文に対しても，要素ごとの分岐方向は異なるため，1 つの方向で計算をしてしまうと間違った値が含まれていることがある．例えば，図 3 のような絶対値を求めるプログラムがある．

```

1: for (i = 0; i < SIZE; i++) {
2:     if (in[i] > 0) {
3:         out[i] = in[i];
4:     } else
5:         out[i] = -in[i];
6: }

```

図 3. 絶対値を求めるプログラム

これを SIMD 化するためにループ 4 回分を展開すると，

```

1: for (i = 0; i < SIZE; i=i+4) {
2:     if (in[i] > 0)
3:         out[i] = in[i];
4:     else
5:         out[i] = -in[i];
6:
7:     if (in[i+1] > 0)
8:         out[i+1] = in[i+1];
9:     else
10:        out[i+1] = -in[i+1];
11:
12:    if (in[i+2] > 0)
13:        out[i+2] = in[i+2];
14:    else
15:        out[i+2] = -in[i+2];
16:
17:    if (in[i+3] > 0)
18:        out[i+3] = in[i+3];
19:    else
20:        out[i+3] = -in[i+3];
21: }

```

のようになる．このプログラムに対して if 文の then パート，else パートをそれぞれ SIMD 命令で並列に計算すると，図 4 のようにそれぞれのレジスタで 2 個ずつの誤ったデータが含まれてしまうので，if 文を SIMD 化して正しいデータを得るためには工夫が必要である．

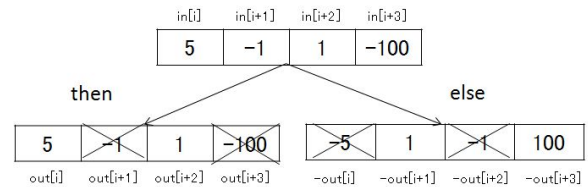


図 4. 条件分岐の例

そこで本研究では，図 5 のようなフローチャート図に従って if 変換を行う．最初に if 変換を行うかどうかの判定を行う．これは，if 文内部の文の数が多かった場合，if 変換を行って SIMD 化してもプログラムが高速化されない可能性があるからである．なぜなら，if 変換というのは条件が成り立っていてもいなくても if 文内部の計算を実行する変形であるからである．例えば，then パートに 100 個の文がある場合を考える．通常のプログラムでは条件が false の場合，100 文は実行しないが，if 変換を行うと常に 100 文を実行することになる．そこで，if 文内部の文の数がしきい値より小さい場合のみ if 変換を行うことにする．

if 変換には，パターンマッチングと select 文向きの変換という 2 つの手法をとっている．これは，ある特定の

形にマッチングすると select 文向きの変換よりプログラムが高速に実行できるためである。次に、この2つの手法について詳しく説明する。

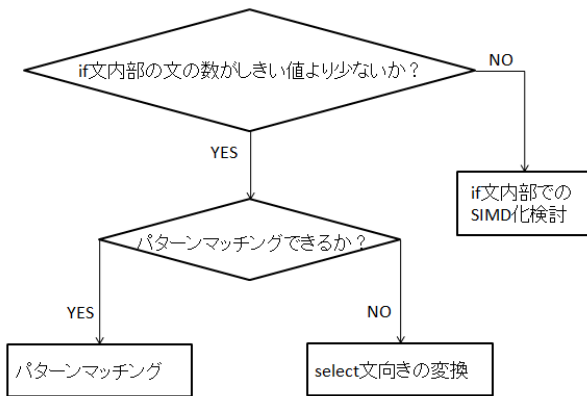


図 5. if 変換のフローチャート

### 3.2.1 パターンマッチング

パターンマッチングで行う方法として現在は2つの方法を実装している。どちらの手法も SIMD の比較命令の演算結果である真/偽が-1/0 となっていることに着目したものである。

1. SIMD の比較演算が生成するマスク値を-1 と 0 の数値として使用する場合



2. SIMD の比較演算が生成する-1 と 0 をマスク値として使用する場合



また、この変形が行えるのは、true が-1 になる時 (SIMD の比較演算) だけなので下記のようなマスクを使って明示的に与えることにする。

$$mask\_0 = (cond)? -1 : 0$$

上記のようにマスク値を与え、

$$a += -mask\_0$$

のようにマスク値を使って文をあらわすことにする。コード生成では、

$$mask\_0 = (cond)? -1 : 0$$

$$a += -mask\_0$$

をまとめて一つの SIMD 命令とおきかえることができる。したがって、マスク値を生成したことによる実行時間のオーバーヘッドはない。

### 3.2.2 select 文向きの変換

SIMD プロセッサには select 命令というデータ選択命令を備えているプロセッサがある。図 3 の絶対値を求めるプログラムに対して select 命令を使用して SIMD 化したものが図 6 のようになる。then パート、else パート、条件式をループ 4 回分計算したものをそれぞれ SIMD レジスタ va, vb, cond に保存する。then パート、else パートを計算したレジスタには、図 4 で示したように、誤ったデータが含まれているので、select 命令を使用して正しいデータを選択する必要がある。select 文を使用すると、cond の真偽によって、va, vb からデータを取り出して、if 文を実行した最終的な結果が得られる。

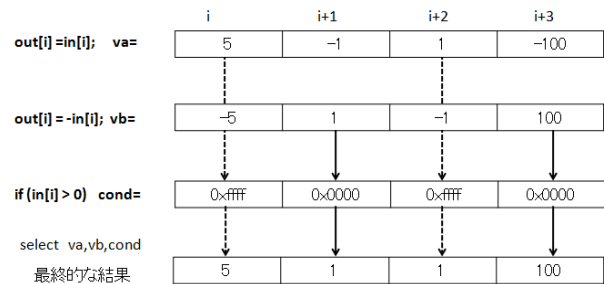


図 6. Select 文の例

この節で説明する変形方法は、この select 文をコンパイラによって自動的に生成するためのものである。また、select 文がないプロセッサに対しては select 文と同値の論理式を生成する。

IA64 の命令セットを対象にネストしている if 文にも適用可能なアルゴリズム [5] が提案されている。これは、SSA 変換のアルゴリズムを応用したものである。本研究では、この手法をソースコードレベルで SIMD 命令生成のために行うことにする。

ここでは、図 7 のようなプログラムを例にアルゴリズムの解説を行う。

```

1: if(a[i] < x){
2:     a[i] = x;
3:     if(a[i] < z)
4:         a[i] = z;
5: }
```

図 7. if 変換を行うプログラム例

if 変換を行うには制御フローグラフ上での解析が必要である。ここで、図 7 に対する制御フローグラフは図 8 のようになる。

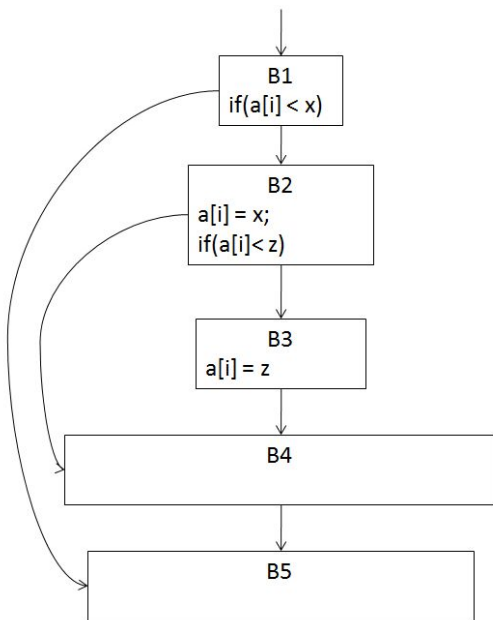


図 8. 図 7 の制御フローグラフ

図 8 のノードの  $\text{pred}()$  と  $\text{succ}()$  は以下ようになる。

```

pred(B2) = {B1}, succ(B2) = {B3, B4}
pred(B3) = {B2}, succ(B3) = {B4}
pred(B4) = {B1, B2, B3}, succ(B4) = {B5}
```

次に、if 変換のアルゴリズムについて説明する。

#### i, ループ中の if 文を探索する

1 回の if 変換では if 文が見つかった所から、if 文のある基本ブロックを直接後支配する基本ブロックま

でを範囲とする。これによって、上記の例では、if 文がネストしているが、最外部の if 文全体を 1 つの if 変換を行う単位として行う。また、if 文中に break 文がある場合は、if 変換は行わないが、continue 文があった場合は、if 変換の対象とする。その場合は、continue 文がある if 文の入口ブロックからそのブロックを直接後支配するループの最後のブロックまでを 1 つの if 変換の単位として処理する。

#### ii, 支配境界 (Dominance frontier) を求める

図 8 の制御フローグラフの各ノードに対する支配境界は以下ようになる。

$$DF(B2) = \{B5\}$$

$$DF(B3) = \{B4\}$$

$$DF(B4) = \{B5\}$$

#### iii, 見つけた if 文に対して支配境界に phi 命令を挿入する

ここでの phi 命令は、SSA 形式の  $\phi$  関数と似ているが、phi 命令は、 $\text{phi}(\text{cond}, v1, v2)$  という形をとっており、cond によって、true の場合  $v1$ 、false の場合  $v2$  を選ぶ。また、phi 命令を並列化できるものを複数個集めたものが SIMD の select 命令となる。

#### iv, if 文内部のみ SSA 形式に変形する

図 8 の制御フローグラフを SSA 形式にすると図 9 のようになる。なお、SSA 形式に変換するのは、if 変換を行う if 文内部だけなので、if 変換の出口ブロックである B5 では  $a[i]$  を一時変数で置き換えていない。ここで SSA 形式にする目的は、文と文の依存関係をなくすためである。これによって、if 変換を行って if 文を 1 つの基本ブロックにしたとき、それぞれの変数の定義使用関係が保たれる。

また、phi 文の右辺を設定するために使う phi-list を作成する。phi-list の求め方は、 $\phi$  関数の求め方と同じである。phi 文を定義したブロックから phi-list には  $\phi$  関数との違いとして、変数名と一緒にその変数が定義された基本ブロックの情報も保存する。基本ブロック情報は、phi 文の条件式を求めるために使用する。

図 9 の制御フローグラフを見ると、B4 の phi-list は B3 の支配境界となり追加されたものである。この

B4 の phi-list には, B2, B3 のように変数とペアでその変数が定義されている基本ブロックの情報も保存されている. また, phi-list の要素は基本ブロックの番号順になっている.

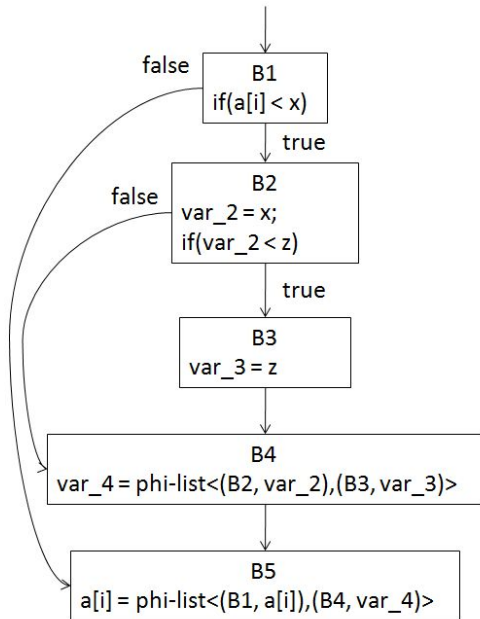


図 9. phi-list を作成

v, 制御依存 (control dependent) を求める

Y が制御依存するノード X から Y 方向へのエッジの集合を返す関数を CD(Y) と書く. 図 9 の制御フローグラフの各ノードに対する CD は以下のようになる.

$$\begin{aligned}
 CD(B1) &= \{\} \\
 CD(B2) &= \{\{B1 \ B2\}\} \\
 CD(B3) &= \{\{B2 \ B3\}\} \\
 CD(B4) &= \{\{B1 \ B2\}\} \\
 CD(B5) &= \{\}
 \end{aligned}$$

vi, それぞれの基本ブロックに対して GP (Guarding Predicate) を求め, select 文の右辺の値を決める

GP とは, それぞれの基本ブロックが実行されるとき true, されないとき false になるように定めた条件式である.

GP を求めるアルゴリズムを図 10 に示す. ループ中のすべての基本ブロックに対して, 制御依存の数で

場合分けをしているのがわかる. 0 の場合というのは, if 変換の入口ブロックと出口ブロックに当たる. これらのブロックは必ず通るので, true になる. 6 行目にある  $PE(edge\{X \ Z\})$  は, X の条件式で Z 方向を進むエッジを選んだとき, true となる条件式を返す. 図 9 の制御フローグラフに対する PE は以下ようになる.

$$\begin{aligned}
 PE(\{B1 \ B2\}) &= a[i] < b \\
 PE(\{B1 \ B5\}) &= !(a[i] < b) \\
 PE(\{B2 \ B3\}) &= var\_2 < z \\
 PE(\{B2 \ B4\}) &= !(var\_2 < z)
 \end{aligned}$$

1 の場合に, この PE を使用して GP を設定する. 5 行目にある Y が制御依存しているエッジとの関係は図 10 のようになる. 図 10 の 6 行目で, 関数 PE に 5 行目のエッジを引数にとって, 論理式 P を求める. 7 行目で, X が常に通るブロックであるときは, GP(Y) が P になる. 常に通るブロックでないときは, P と GP(X) を & した論理式が GP(Y) となる. たとえば, 図 12 の B3 をみると, p3 を  $PE(\{B2 \ B3\})$  & p2 により求めているのがわかる. また, 1 以上ある場合が書かれていないが, switch 文を対象にすると考慮する必要があるが本研究では if 文のみ考えることにする.

```

1: for each ループ do
2:   for each ブロック Y (トポロジカルソートされた順序) do
3:     if |CD(Y)| = 0 then GP(Y) を true にする
4:     if |CD(Y)| = 1 then
5:       E{X Z} CD(Y)
6:       P := PE(E)
7:       if GP(X) = true then
8:         GP(Y) := P
9:       else
10:        GP(Y) := P & GP(X)
11:      end if
12:    //if |CD(X)| > 1 は, switch 文のみ
13:  end for
14: end for
  
```

図 10. GP を求めるアルゴリズム

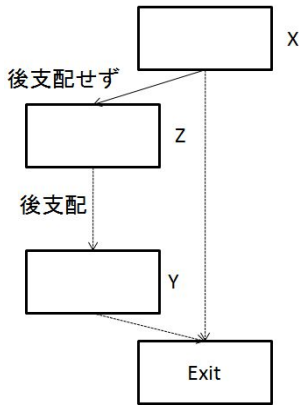


図 11. 図 10 に対する基本ブロックの関係

GP は phi 文の条件式として使用される。GP を用いて phi-list から phi 文を生成するアルゴリズムを図 12 に示す。通常の if 文は phi-list の要素が 2 個であるので、アルゴリズムの if 文では 9,13 行目が実行される。まず、2 行目から 5 行目で {block1, var1} に phi-list の 2 番目の要素を代入し、{block2, var2} に phi-list の 1 番目の要素が代入される。ブロック 1 の GP を求めて、phi-list のある代入文の右辺を phi (exp) var1, var2 で置き換える。これで、exp が true のとき var1 が選ばれ、false のとき var2 が選ばれる phi 文が作成される。また、phi-list の要素が 3 個以上ある場合 (continue 文がある場合) は、一時変数を使用する必要がある。たとえば、 $a[i] = \text{phi-list} < (B2, \text{var}_2), (B3, \text{var}_3), (B8, \text{var}_8) >$  のような phi-list があつたとき、  
 $\text{temp} = \text{phi}(GP(B8)) \text{var}_8, \text{var}_3$   
 $a[i] = \text{phi}(GP(B8)|GP(B3)) \text{temp}, \text{var}_2$   
 のように 2 つの phi 文に置き換えることができる。アルゴリズムでは、15 から 20 行目でこのような変形が実行される。

```

1: for phi-list が空ではない
2:   phi-list の最後の要素を{block1, var1}へ代入し,
3:   phi-list からその要素を削除する
4:   phi-list の最後の要素を{block2, var2}へ代入し,
5:   phi-list からその要素を削除する
6:   if block1 = null(17 行目で追加された要素) then
7:     exp := tempExp
8:   else
9:     exp := GP(block1)
10:  end if
11:
12:  if phi-list が空
13:    phi-list を phi (exp)var1, var2 で置き換える
14:  else
15:    新しい変数を作成する (temp)
16:    phi 文の前に
17:    temp := phi (exp)var1, var2;
18:    を追加する.
19:    phi-list の最後に{null, temp}を追加する
20:    tempExp := exp | GP(block2)
21:  end if
22: end for
  
```

図 12. phi-list から phi 文を生成するアルゴリズム

図 13 にはそれぞれの基本ブロックに対応する GP が書かれている。また、その GP を利用して図 9 の phi-list から phi 文への変換を行っている。

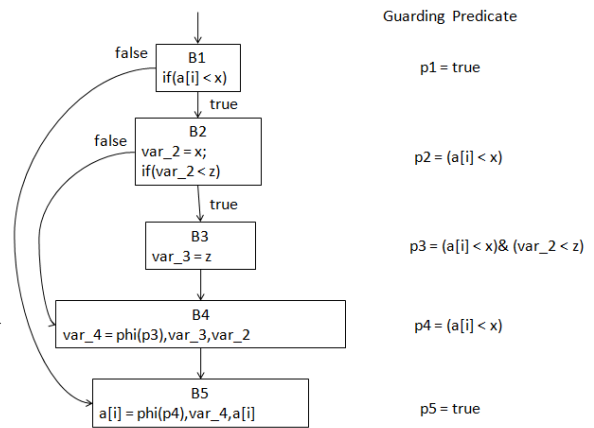


図 13. 制御フローグラフと対応する GP

vii, マスク文を追加して if 文を取り除く

select 文はソースコードレベルで直接表現できないので、代わりに論理式を使って表現し、命令レベルの処理でこの形を認識して、select 文を生成する必要がある。phi (cond), v1, v2 の代わりになる論理式は、

$$(v1 \& \text{cond}) | (v2 \& \sim \text{cond})$$

のように書く必要がある。しかし、前節のパターン

マッチングで示したように、この変形が行えるのは、true が-1 になる時 (SIMD の比較演算) だけなのでマスクを使って明示的に与る必要がある。マスクを使って書くと、

```
mask_0 = (cond)? -1 : 0;
(v1&mask_0)|(v2& ~ mask_0);
```

のようになり、命令生成では、マスク生成の if 文も含めて select 文とマッチングすることを期待する。また、select 文がない SIMD 命令セットでは上記のような論理式に対応する SIMD 命令を生成すればよい。最終的に出力されるプログラムは、図 14 のようになる。図 13 に対して、トポロジカルソートをした基本ブロックの順序 (ブロック番号順) で代入文を出力した結果である。

```
1: //B2
2: var_2 = x;
3: //B3
4: var_3 = z;
5: //B4
6: mask_0 = ((a[i] < x) & (var_2 < z))? -1:0
7: var_4 = (var_3 & mask_0) | (var_2 & (~mask_0));
8: //B5
9: mask_1 = (a[i] < x)? -1:0
10: a[i] = (var_4 & mask_1) | (a[i] & (~mask_1));
```

図 14. if 変換されたプログラム

### 3.3 ループ展開

ループ展開では、ループがネストしている場合、最内部のループのみを展開の対象とする。SIMD レジスタサイズとプログラムのデータサイズを考えて展開数を変える必要がある。SIMD のレジスタサイズを n byte、演算のデータサイズ m byte であった場合ループの展開数は n/m(と余り)となる。図 15 の例だと、16byte の SIMD レジスタを使用し、4byte のデータを扱っているので、展開数は 4 となる。また、ループ中に異なるデータサイズの計算がある場合の m の値は、それぞれの SIMD 化できる文の最小のサイズを指定する。

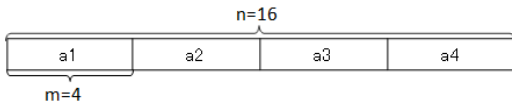
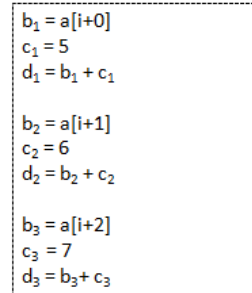


図 15. SIMD レジスタとデータサイズ

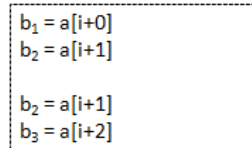
### 3.4 Super Level Parallelism

SLP 解析 [7] では、同一の演算を同一の型で行う文を見つけて SIMD 命令のスケジューリングを行う。SLP 解析の手順を以下のような例を使って説明する。



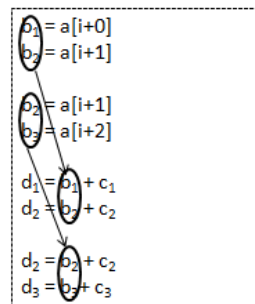
#### 1. 隣接メモリ参照を検知する

プログラム中で配列参照を行っているところを探して、メモリ上で隣接しているものを組にする。



#### 2. 組にした命令と def-use 関係にある組を探す

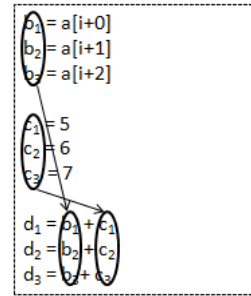
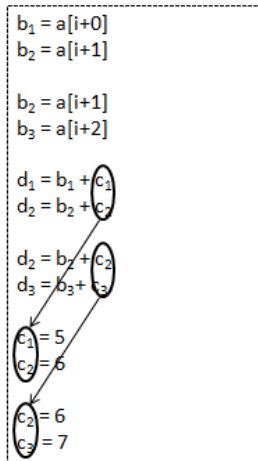
組にした命令で定義されている変数が、使用されている命令を新たに組みとして追加する。



#### 3. 組にした命令と use-def 関係にある組を探す

組にした命令で使用されている変数が、定義されて

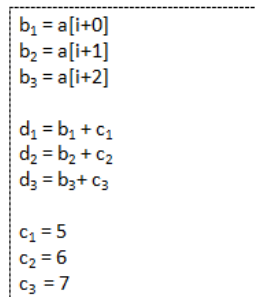
いる命令を新たに組みとして追加する .



本研究での SLP 解析は基本的には Larsen のアルゴリズム [7] を使用するが、アルゴリズムの対象が 3 番地コードとなっていて、コード生成までを行っている。しかし、本研究の SLP 解析の目的は、SIMD 化可能かどうかの判定を行うことにある。よって、文全体が同等かどうかの判定までを行うことにする。

#### 4. それぞれの組を結合して 1 つの SIMD 命令で実行できる命令列を作る

例では 1 つ目の組と 2 つ目の組で、 $b_2 = a[i+1]$  が重なっているので、2 つのブロックを結合する



#### 5. 元のプログラムと比較して、命令列間の依存関係を調べる

4 での作業時の命令順序では、 $(c_1, c_2, c_3)$  の def-use 関係が逆になっているので 2 番目の命令列と 3 番目の命令列を入れ替える。

### 3.5 SIMD 化判定

現在 SIMD 化が行えるコンパイラで SIMD 化が難しいプログラムをコンパイルすると、通常のスカラでの最適化よりも実行時間が遅くなってしまうことがある。図 16 に動画圧縮プログラムである quant5[8] に対して変形していないプログラムと SIMD 化向けの変形を行ったプログラムに対して gcc -O3 でコンパイルした結果を示す。このグラフは、原型を 1 とした実行時間の相対値である。グラフより SIMD 化向けの変形が SIMD 化できなかったとき、スカラでの最適化よりも遅くなってしまう場合があることがわかる。この実行時間の増加の主な理由は if 変換によるものである。quant5 にはネストしている if 文があり、その if 文を本研究の if 変換を行うことによるオーバーヘッドが表れている。if 変換の説明でも述べたが if 変換は条件が成立していてもいなくても文が実行されてしまう。たとえば、

```
if (a>0) {
    x[i]= exp1
    y[i]= exp2
    z[i]= exp3
}
```

のようなプログラムでは  $a \leq 0$  のとき文は実行されないが、if 変換を行うと、

```
var1 = exp1
var2 = exp2
var3 = exp3
x[i] = phi(a>0) var1, x[i]
y[i] = phi(a>0) var2, y[i]
z[i] = phi(a>0) var3, z[i]
```

となり常に実行されてしまう。また、図 16 ではこの if 変換を行ったプログラムをループ展開することによ

てさらに実行時間が増加してしまっている。しかし、if 文がないループに対してはループ展開により常に実行時間が増加するとは言えない。逆に、ループ展開を行ってループ制御命令の実行回数が減ることによって実行時間が改善されることもある。しかし、本研究ではループ展開によりループ中の命令数が増え、その分必要とするレジスタの数も増えてしまうという問題を考え、実行時間の改善する可能性があるとしても SIMD 化不可能だったら変換前のコードに戻すことにする。

本研究では、SLP 解析の命令スケジューリングを行った情報によってループ全体の SIMD 化判定を行う。本研究の SIMD 化判定で SIMD 化できないと判断される文は、

- ポインタ配列を配列変換できない
- break 文や return 文により if 変換ができない文
- メモリアドレスが隣接していない
- 関数呼び出し
- while 文
- switch 文

である。この判定を行うことによって、コンパイラが SIMD 化できないと判断したループに対しては変形前に戻し、スカラでの最適化を行う。現在の判定方法では、SLP 解析で SIMD 命令が 1 命令でも生成できると判断されたら、変形されたコードの出力を行っている。1 命令も SIMD 化できなかった場合、変換前のコードと置き換える。このような判定を行うことによって、プログラム中のループがすべて SIMD 化できなかった場合でも、スカラ最適化した時と同じ実行時間になり、実行時間が増加してしまうことがなくなる。

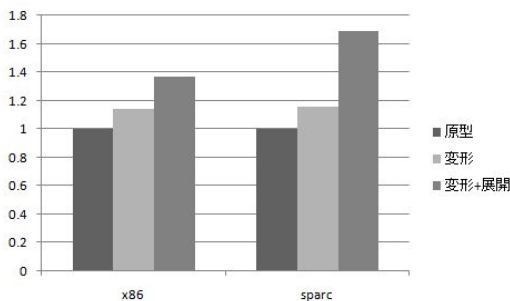


図 16. 変形前との実行時間の比較

## 4 実験と考察

### 4.1 実装

実装は、COINS 上の高水準中間表現 HIR 上におこなった。ループ展開はすでに COINS での実装が行われている。そこで、本実装では SIMD 化向けの展開数を指定する部分のみ実装を行った。また、COINS を使うことにより HIR の制御フローグラフや変換を行うメソッドが使用できたので容易に実装できた。本研究で実装したのはソースコードレベルの変形だけなので実際に SIMD 命令の生成を行うには、命令レベルでの変形やコード生成を行う機能が必要である。COINS 上には命令レベルでの SIMD 命令生成器が実装されているが、これはプロトタイプ実装ということもあり SIMD 命令への変換性能があまり高くない。よって、高水準中間表現上で提案手法の実装を行い、低水準中間表現は通さずに変形した中間表現を C 言語へ逆変換を行ったものを用いた。

### 4.2 実験

テストプログラムは、論文中にある条件に合う件数を数える例題 [9] と SIMD ベンチマーク [10] からそれぞれ count, quant5 を使用した。これらは、ループ中に if 文があり SIMD 化するには if 変換を要するものである。また、マルチメディア向けベンチマークプログラムである DSPstone[8] から matrix, convolution を使用した。これらは、ポインタ配列を用いているので配列変換の効果を検証できる。これらの 4 つのプログラムは、アセンブリ言語レベルでのアライメントを整える処理が必要でないものを選んだ。

実験方法は表 2 に示す。実験では、原型のプログラム、提案手法で変形したプログラムそれぞれに対して、icc (Var. 11.0) のスカラ最適化 (-O3), SIMD 化 (-axN -O3) を適用した。もう一つは、変形したプログラムに対して、本研究で SIMD 化可能と判断された部分に対して手で組み込み関数 [13] を使い SIMD 化を行ったものである。たとえば、DSPstone[8] の matrix に対して、ポインタ配列の配列変換を行ったプログラムの一部は、

```
C[k*Z+i] = C[k*Z+i] + A[i*X+f] * B[k*Z+f];
C[k*Z+i] = C[k*Z+i] + A[i*X+f+1] * B[k*Z+f+1];
C[k*Z+i] = C[k*Z+i] + A[i*X+f+2] * B[k*Z+f+2];
C[k*Z+i] = C[k*Z+i] + A[i*X+f+3] * B[k*Z+f+3];
```

のようになる。これに組み込み関数を適用すると、

```
Areg = *(__m128*)&A[i*X+f];
Breg = *(__m128*)&B[k*Z+f];
Creg = _mm_add_ps(Creg, _mm_mul_ps(Areg, Breg));
```

のようになる。\_mm\_add\_ps(SIMD の足し算), \_mm\_mul\_ps(SIMD の掛け算) が組み込み関数であり, Areg, Breg, Creg が SIMD レジスタ (16byte) である。また, それぞれの実験に対する目的は下記ようになる。

表 2. 実験方法

実験	ソースコードレベル	コード生成
1	原型	icc -O3
2	原型	icc -axN -O3 (SIMD)
3	提案手法で変形	icc -axN -O3 (SIMD)
4	提案手法で変形	組み込み関数

#### 実験 1 基準

スカラ最適化を行ったプログラムを基準にして, SIMD 化により基準よりも実行時間が改善されているプログラムに対して, SIMD 化の効果があったと判定できる。

#### 実験 2 icc の SIMD 命令生成能力

現在使用されている SIMD 命令を生成できるコンパイラである icc で, ポインタ配列や if 文へどれだけ対応できるかを考察する。

#### 実験 3 変形が icc の SIMD 化を助けるかどうか

実験 3, 4 は本研究でのコード変形器を使用した。本研究で作成したのはソースコードレベルでのコード変形器なので, 変形された C 言語のコードを SIMD 命令を生成できるコンパイラの入力として与えることができる。このような使い方を想定したとき, 実験 2 と比べてコード変形がどの程度実行時間に影響を与えるかを考察する。

#### 実験 4 ソースコードレベルでの期待通りにコード生成が行われた場合

本研究ではアセンブリ言語レベルでのコード変形器とコード生成器の実装は行っていない。そこで, テストプログラムとしてアセンブリ言語レベルでの変形が必要でないプログラムを選び, コード生成では組み込み関数を使用した。この実験では, 実験 1 と比べて SIMD 化を行うとどれだけ実行時間の改善されるかを考察する。

実験環境は, 表 3 に示す。

表 3. Let's note CF-W5 の仕様

機種	Let's note CF-W5
プロセッサ種別	Intel Core Duo 1.06GHz
プロセッサ数	2
1 次キャッシュ	2 × 32KB
2 次キャッシュ	2MB
メモリ容量	512MB
オペレーティング環境	Ubuntu-8.04

### 4.3 考察

提案手法で変形したときと提案手法を適用していない原型での実行時間の比較を図 17 に示す。このグラフは, 原形でスカラ最適化をした場合を 1 としている。これらの実験に対する考察を下記に示す。

#### 実験 2 icc の SIMD 命令生成能力

icc では簡単な if 文 (ネストしてない) に対して if 変換を行い, SIMD 命令を自動生成することができる。しかし, ネストしている if 文とポインタ配列があるプログラムに対しては, SIMD 命令を生成することができなかった。

#### 実験 3 変形が icc の SIMD 化を助けるかどうか

実験 2 と比べて, 実行時間は改善しなかった。quant5 ではスカラ最適化と比べて, 1.7 倍の実行時間となってしまった。これは, 本研究では SIMD 化できないと判定されたループがスカラ最適化されると期待している。また, SIMD 化できると判定されたループに対して SIMD 命令が生成されると期待している。しかし, icc がこれらの期待通りのコード生成ができていないことが原因だと考えられる。したがって, icc 向けにソースコードレベルでの変形器を使用する場合は, icc で SIMD 化できるプログラムの形を実験で調べることにより, コード変形器の生成するコードの形をチューニングする必要がある。

#### 実験 4 ソースコードレベルでの期待通りにコード生成が行われた場合

この実験は, ソースコードレベルでコード変形を行い, SIMD 化可能と判断されたループに対して, ソースコードレベルで期待されるコード生成を命令レベルで行ったと仮定した場合である。実験結果より, 50% 以上の実行時間の減少が見られた。これからわかるように, 提案手法を使って変形したものに

対して命令レベルで期待されるコード生成を行うと SIMD 化による効果が期待できる。

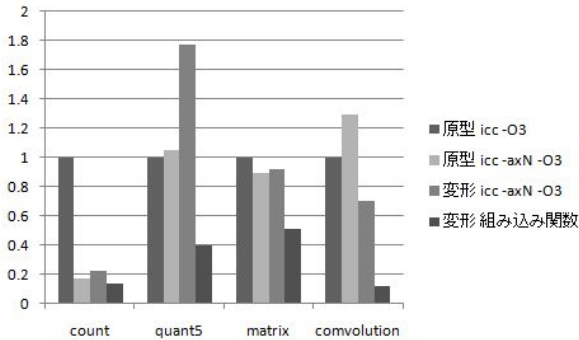


図 17. 実行時間の比較

また、DSPstone のすべてのプログラムの for ループに対して、SIMD 化ができるかどうかの判定を行い下記の 3 つに分類する静的な実験を行った。それぞれのパターンとプログラム中の出現率を下記に示す。

パターン 1 SIMD 化可能と判断され、命令レベルでの処理が必要ない 出現率 40%  
アライメントを整える必要がない

パターン 2 SIMD 化可能と判断されるが、命令レベルでアライメントを整える必要がある 出現率 13%  
SIMD 命令にはアドレスが 16byte 境界でないとロードできないなどの制約がある。下記の例だと、a[0], b[0], c[0] が 16byte 境界で配列の要素のサイズが 4byte だとすると、それぞれ 8,4,12byte ずれているのでアライメントを整える必要がある。

```
例: for(i=0; i<100; i++){
    a[i+2] = b[i+1] + c[i+3];
}
```

パターン 3 SIMD 化不可能と判断される 出現率 47%  
SLP のスケジューリングではメモリ上で隣接しているデータにのみ SIMD 化可能とした。SIMD 化不可能と判断されるのは、下記のようなプログラムである。配列の要素の 1 つ置きに同一の演算子の計算をしている。

```
例: for (i = 0; i < N*2; i+=2)
{
    d[i] = a[i]+b[i];
    d[i+1] = a[i+1]*b[i+1];
}
```

図 17 では上記のパターン 1 のループに対して、SIMD 化による実行時間の改善を見た。これから、それぞれのパターンのループに対してどれだけのオーバーヘッドがあり、SIMD 化による実行時間の改善が望めるかの考察を行う。

まず、SIMD 並列化を行うことによるオーバーヘッドとして、SIMD 命令を使用するには SIMD レジスタへロードしなければいけないことと、SIMD レジスタのデータは通常のレジスタへ置き換えなければスカラ演算ができないことがあげられる。1 はメモリ上に連続しているデータを 1 命令でロードできるパターンなので、このオーバーヘッドが少なく済み、実行時間の改善が期待できる。また、DSPstone ではプログラム中に 40%出現しているため、マルチメディア系のプログラムに対して十分実行時間の改善が見込めることがわかる。2 は 1 のオーバーヘッドに比べて、アライメントを整えるオーバーヘッドがある。本研究では、アライメントを整えるアルゴリズム [1] はアセンブリ言語レベルで行う手法として分類した。この手法を使うことにより、パターン 1 よりも実行時間は遅くなるが SIMD 化可能になる。1,2 のパターンが本研究で SIMD 化可能と判断されるパターンで DSPstone では全体の半分を占めた。3 は本研究では SIMD 化不可能と判断されているものである。これは、メモリ上で同一演算の命令が連続していない場合で、データの並べ替えを行ってから SIMD 化するアルゴリズム [3] により SIMD 化可能になる可能性がある。このように、パターン 3 に分類されたものにも SIMD 化可能なループも存在するが、SIMD 化効果を予測することが難しく新たな SIMD 化判定方法の開発が必要なため、本研究では SIMD 化不可能と判定することにした。

## 5 今後の課題

### 5.1 SIMD 化判定方法の検討

現在の SIMD 化判定は、1 命令でも SIMD 化が可能ならループ全体の SIMD 化を行っている。しかし、SIMD 化には考察で述べたようなさまざまなオーバーヘッドがある。このようなコストも考え判定方法を確立していく必要がある。

## 5.2 SIMD化できる文を増やす

本研究でSIMD化できないと判断される文の中には、switch文やwhile文などがあるがこれらもSIMD化できるように、アルゴリズムを拡張することが考えられる。これによって、SIMD化不可能と判断されていたループの一部がSIMD化可能となり実行時間の改善が期待できる。

## 5.3 命令レベルでのコード変形器の設計

2節で述べたように、本研究ではSIMD化変形手法をソースコードレベルと命令レベルに分類した。そこで、アセンブリ言語レベルでのコード変形器を作成し、ソースコードレベルでのコード変形器と共に使用することにより、大きな効果を得ることできる。

## 6 関連研究

現在自動でSIMD命令を生成するようなコンパイラがいくつか設計されている。本手法に関連の深いいくつかの手法と本手法の比較を述べる。

### 6.1 Bikらの研究

Intelでは、ループ展開からIntelのアーキテクチャに特化したSIMD命令の生成も行えるようになっている[11]。しかし、SIMD命令の生成に失敗した場合の対処については書かれていない。実際、本論文の実験でもiccでSIMD化オプションを付けてコンパイルを行った結果、実行時間が増加してしまった例が見られた。

### 6.2 Sreramanらの研究

ソースコードレベルでループ展開や変数の依存関係などの解析を行い、C+SIMD化されたインラインアセンブラで出力するコンパイラの研究もあった[12]。しかし、ソースコードレベルだけの処理で効率のよいコード生成するのは限界がある。なぜなら、2.2節で示したようなアセンブリ言語レベルの手法を行うことが困難になるからである。

## 6.3 鈴木らの研究

4章で述べたように、本手法を実装したCOINS上にもアセンブリ言語レベルでSIMD命令の生成を行っている[9]。SIMD命令と低水準中間表現でSIMD命令と対応する動作テンプレートとの照合を行い、SIMD命令を作成している。アセンブリ言語レベルでの変形なので、本研究で行ったループ展開や配列変換は行われない。よって、本研究で行ったようなソースコードレベルでの変形手法を行ったコードに対して、SIMD命令が生成されるはずである。しかし、現在の実装(COINS1.4.2.2)では、ソースコードレベルで変形を行った一部のプログラムしか変形ができていない。また、SIMD特有の命令であるshuffle命令などに対応していない。

## 7 まとめ

本研究では、SIMD命令を生成するためのコード変形方法の中から、ソースコードレベルに適した手法を選び、コード変形器の設計を行った。また、コード変形を行うことによる実行時間の増加を防ぐために、SIMD化可能かどうかの判定を行った。そして、COINS上で実装を行い、マルチメディア向けのベンチマークを使用して評価を行った。本研究でのコード変形を行い、期待されるコード生成が行われると既存のコンパイラ以上のSIMD命令が生成できることがわかり、SIMD命令が生成できなかった場合に実行時間が増加することがなくなった。

## 謝辞

本研究の一部は、日本学術振興会科学研究費補助金の援助を受けた。

## 参考文献

- [1] Peng Wu, Alexandre E. Eichenberger, Amy Wang, Efficient SIMD Code Generation for Runtime Alignment and Length Conversion, Proceedings of the international symposium on Code generation and optimization, pp.153-164, March 20-23, 2005.
- [2] 三好健文, 杉野暢彦: "レジスタスロットを考慮したSIMD向け細粒度自動並列化コンパイラ", 情報処理

- 学会論文誌コンピューティングシステム, Vol.1, No.2, pp.240-249, 2008年8月.
- [3] Gang Ren, Peng Wu, David Padua, Optimizing data permutations for SIMD devices, Proceedings of the Conference on Programming Language Design and Implementation, pp.118-138, June 2006.
- [4] R.A., van Engelen and K. A., Gallivan : An Efficient Algorithm for Pointer-to-Array Access Conversion for Compiling and Optimizing DSP Applications, Proc. of Innovative Architecture for Future Generation High-Performance Processors and Systems, pp.80-89, 2001.
- [5] Weihaw Chuang , Brad Calder , Jeanne Ferrante, Phi-Predication for light-weight if-conversion, Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization, pp.179-190, March 23-26, 2003, San Francisco, California.
- [6] 中田育男. コンパイラの構成と最適化. 朝倉書店, 1999.
- [7] S. Larsen and S. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In Conference on Programming Language Design and Implementation, pp. 145-156, June 2000.
- [8] V. Zivojnovic , J. Martinez , C. Schlager and H. Meyr. DSPstone: A DSP-Oriented Benchmarking Methodology. Proc. of ICSPAT'94 - Dallas, pp. 715-720, Oct. 1994.
- [9] 鈴木 貢, 藤波 順久: "COINS における SIMD 並列化 ", コンピュータソフトウェア, Vol.25, No.1, pp.65-81, 2008年1月.
- [10] 鈴木貢, 小川大介, 室田. 樹, 渡邊坦: "SIMD ベンチマークの設計と実装 ", 情報処理学会論文誌: コンピューティングシステム, Vol.46, No.SIG 16 (ACS 12), pp.95-107, 2005年12月.
- [11] Aart J. C. Bik , Milind Girkar , Paul M. Grey , Xinmin Tian, Automatic Intra-Register Vectorization for the Intel Architecture, International Journal of Parallel Programming, Vol.30 No.2, pp.65-98, April 2002.
- [12] N. Sreraman , R. Govindarajan, A Vectorizing Compiler for Multimedia Extensions, International Journal of Parallel Programming, Vol.28 No.4, pp.363-400, August 2000.
- [13] インテル株式会社:Linux\* 版インテル (R) C++ コンパイラ ユーザーズ・ガイド, 2004.