複雑な制御構造を持つプログラムのSIMD命令セットによる最適化

廣松悠介† 黒田久泰†† 金田康正††

近年の汎用プロセッサの多くは，複数のパックされたデータを1命令で演算可能なSIMD（Single Instruction Multiple Data）命令セットを搭載している．この命令セットはデータの並列性を利用して，大量のデータを通常の命令よりも高速に処理することが可能である．そのため，マルチメディア処理や数値計算処理の高速化に利用されている．これまで，自動解析によってSIMD命令セットを使ったプログラムの並列化を実現するための研究が多くなされており，コンパイラによるSIMD並列化も行われるようになりつつある．ところで，SIMD命令はパックされたデータ1つ1つに対して，異なる演算を実行するということができない．そのため，条件分岐やループのような複雑な制御構造は，あまり最適化対象として扱われなかった．しかし，そのような制御構造がSIMD並列化の適用範囲となれば，より多くのプログラムが最適化可能となることが期待できる．そこで本論文では，複雑な制御構造を持つプログラムをSIMD並列化するための手法を提案する．本論文の提案手法をCOINSコンパイラインフラストラクチャに実装し，テストプログラムをPowerPCのSIMD命令セット向けにSIMD並列化して速度を比較したところ，本来のプログラムの1.19倍から12.3倍の速度で動作した．

An Optimizing Method with SIMD Instruction Set for Program with Complex Control Structure

Yusuke Hiromatsu,† Hisayasu Kuroda†† and Yasumasa Kanada††

Modern general purpose processors have SIMD (Single Instruction Multiple Data) instruction set which computes packed data in parallel. Using data parallelism, this instruction set processes mass data faster than the scalar. Therefore it is used to optimize multimedia or mathmatic processing. There are researches to analyze programs to vectoize with SIMD instruction set, that make compilers to enable to generate SIMD codes. By the way, SIMD instruction set cannot select instructions for every packed data. Accordingly the complex control flow which includes conditional branches or loops are not treated for optimization with the instruction set. However if they became applicable to parallelize with that instruction set, more programs are expected to be optimized. In this paper, the method vectorizing the programs containing complex control structure with SIMD instruction set is proposed. It was implemented with COINS compiler infrastructure and converted some programs from scalar to vector. They achieved from 1.19 to 12.3 times speedup on PowerPC's SIMD instruction set.

1. はじめに

x86[1]，PowerPC[2]，ARM[3]といった汎用プロセッサの多くは，信号処理や数値計算処理などの大量のデータを扱うプログラムの高速化を目的として，SIMD（Single Instruction Multiple Data）命令セットを搭載している．ここで，SIMD命令セットは複数のデータを1命令で演算可能な命令の集まりである．SIMD命令セットを使ってプログラムを高速化するためには，プログラマがアセンブリ言語や組み込み関数を使ってプログラミングする必要がある．しかし，それにはSIMD命令セットやSIMD命令特有の問題を解決するための知識を要するうえに，プログラミングやデバッグに時間がかかる．さらに，SIMD命令で記述したプログラムは別プラットフォームで動作しないため，移植性が低くなる．

近年は解析手法に基づいてSIMD命令を生成するための研究[4)~6)]がなされているが，コンパイラがSIMD

† 東京大学大学院新領域創成科学研究科
Graduate School of Frontier Sciences, The University of Tokyo
†† 東京大学情報基盤センター
Information Technology Center, The University of Tokyo

命令による最適化を実現するための制約は多く，まだ発展途上である．特に，条件分岐や，ループ実行前に繰返し回数が定まらない WHILE 型ループなどがあり，制御構造が複雑なプログラムの最適化は，SIMD 命令の実行条件が制約となってほとんどなされていない．SIMD 命令の適用範囲を拡大し，SIMD 命令による最適化が自動化可能なプログラムを増やすためにも，制御構造を SIMD 並列化する的確な手法が要求される．本論文では，条件分岐や WHILE 型ループなどの複雑な制御構造を持ったプログラムブロックの SIMD 並列化処理を自動化するための手法を提案する．

本論文の流れは以下のようになっている．2 章では，SIMD 命令と，SIMD 並列化や提案手法に関わる研究動向について紹介する．3 章では，制御依存関係を明確にし，SIMD 並列化を容易にするために利用する制御フローグラフという解析手法について説明する．4 章では，本論文の提案手法である，制御構造の SIMD 並列化を行うためのアルゴリズムを解説する．5 章では，本論文で提案する手法を実装したプログラムによって，いくつかのテストプログラムを SIMD 並列化することで提案手法の性能評価を行い，6 章で全体のまとめを述べる．

2. 関 連 研 究

SIMD 命令セットには 64 ビットや 128 ビットなどの大きさを持つ専用レジスタが定義されており，そこには 1，2，4 バイト整数や単精度浮動小数点数など，レジスタサイズ未満の型のデータが複数個まとめて格納されている．演算の際には**図 1** のように，要素別に並列に演算する．この演算方法によって，SIMD 命令はデータ間の並列性を利用した高速処理を実現する．

近年は，プログラムを自動解析して，SIMD 命令を利用した最適化を行うことを可能にするための研究がなされている．たとえば，SLP（Superword Level Parallelism）解析[4] は同一演算を行う箇所を発見し，SIMD 並列化することを目的としている．SIMD メモリアクセス命令はメモリ境界が厳密でなければならない場合が多いため，それを克服するための研究[5),6)] もある．また，演算に必要なデータを正しい順序に並べるための整列演算のステップ数の最小化の研究[7)] もなされている．そして，これまでの SIMD 最適化に関する研究成果から，自動 SIMD 並列化機能を搭載したコンパイラも現れ始めている[8),9)]．

しかし，現在のコンパイラの SIMD 並列化機能は，複雑な制御構造を持つプログラムの SIMD 並列化については，ごく簡単なパターンマッチングでしか扱わない．つまり，ある SIMD 命令セット固有の命令にマッチするパターンの分岐のみを変換する，といった程度のきわめて限定的な利用であり[9)]，WHILE 型ループに関してはまったく扱われていない．

複雑な制御構造の SIMD 並列化がなぜ容易でないのかを，例をあげて説明する．まずは**図 2** (a) のように並列実行可能なループ中にある分岐を SIMD 命令で処理することを考える．a の値が **図 3** のようになっていたとすると，要素ごとに分岐方向が異なるため，一方の分岐先の処理を行うだけでは，ある要素へ間違った演算結果が渡される可能性がある．そのため，分岐を SIMD 命令によって実現しようとした場合，演算ステップを変える必要がある．また，図 2 (b) のように，ループ中に存在する WHILE 型ループを SIMD 命令で複数の要素に対して並列に行う形に変換する場合，**図 4** のように各要素の値が異なると，要素ごとに終

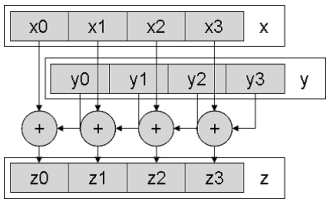


図 1 SIMD 命令の例
Fig. 1 Example of a SIMD instruction.

(a)

```
for i = 0 ··· n do
  if a[i] > 0 then
    a[i] ← a[i] + 1
  else
    a[i] ← −a[i]
  end
end
```

(b)

```
for i = 0 ··· n do
  while b[i] > 0 do
    b[i] ← b[i] − 1
  end
end
```

図 2 制御の流れが複雑な例
Fig. 2 Examples of non-trivial control flow.

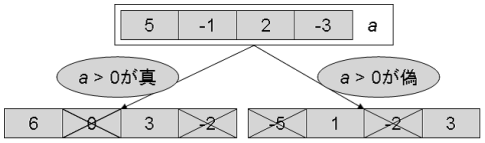


図 3 各要素によって分岐先の処理が異なる条件分岐
Fig. 3 Branch with multiple branch destinations.

了タイミングがずれてしまう．すべての要素がループ終了時に正しい値を指すようにするためには，ループ構造の大幅な書き換えが 必要となる．

制御構造の並列化に関する問題は，過去にベクトルプロセッサにおいて取り組まれている．ベクトルプロセッサとは，256 や 512 個といった大量のレジスタを 1 つのグループとして扱い，同じ演算を並列に実行する機能を持つプロセッサで，HPC 分野において利用されている[10]．

条件分岐を並列化するための代表的な方法として IF 変換[11] がある．これは分岐先に含まれる命令に，命令実行のための論理条件式を貼りつけることで条件分岐を取り除く手法である．ベクトルプロセッサには条件分岐の並列化を容易に実現するための機能として，各要素の条件比較結果を真理値で保持するマスクベクトルが用意されている[12] ので，ベクトルプロセッサ向け並列化コンパイラでは，マスクベクトルと IF 変換を利用することで分岐の並列化を実現している．SIMD 命令にはマスクベクトルのような機能はないが，この手法の考え方は SIMD 命令においても適用可能である．

また，WHILE 型ループを並列化するための手法もいくつか提案されており[13]，以下のようなものがある．

- ループの一部をスカラ処理して，ループ回数を数えてからベクトル処理する．ループ分割を利用した手法である．スカラ演算する箇所ができるため，ベクトル化効率が悪い．
- 数回ループ処理をまとめて行った後に全要素が終了条件を満たしたかどうかを調べ，満たしていたら，条件を満たした段階の値を取り出す．ストリップマイニングを利用した手法である．無駄な繰返し処理が発生するため，ベクトル長が短かったり，ループ回数が少なかったりする場合は実行効率が悪くなる．
- FOR 型ループという，繰返し回数が既知のループ内にネストしている WHILE 型ループがある場合，FOR 型ループと WHILE 型ループの順序を入れ替えることでベクトル化する．ループ交換を利用した手法で，WHILE 型ループが多重になっている場合にも適用できるが，制約が厳しい．

これらは並列処理のためのオーバヘッドが大きいため，並列度が小さい SIMD 命令に適用したとしても，並列化に見合う性能が出るとは考えられない．また，IF 変換と WHILE 型ループの並列化手法は別々のものとして扱われており，条件分岐とループが互いに入れ子になるなどして，制御構造が複雑になっている場合には適用が難しい．SIMD 命令でも効果が見込めるように，低オーバヘッドで，より一般化された手法が必要とされる．

3. 制御フローグラフ

条件分岐やループなどの複雑な制御構造を演算ステップから直接解析するのは困難である．そこで，プログラム中の制御構造を制御フローグラフ[14] という依存関係を表すツリーにすることで，制御構造を単純化して考える．制御フローグラフはデータの依存関係を表すデータフローグラフとともに，コンパイラの解析手法として頻繁に使われている．制御フローグラフはプログラムの制御依存関係をツリー型のグラフで表現したものであり，ツリーのノードの単位となるブロックとして以下の 3 つを定義する．

定義 1 代入文などの基本的な演算の集まりを基本ブロックと定義する（**図 5** 参照）．

定義 2 比較を行い，比較結果によって制御の方向を決定する処理を行うブロックを分岐ブロックと定義する（図 5 参照）．

定義 3 繰り返して同じ処理を行う領域を示すブロッ

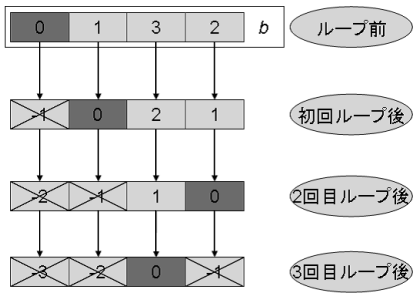


図 4 終了条件が各要素によって異なる WHILE 型ループ
Fig. 4 While loop with multiple exit timings.

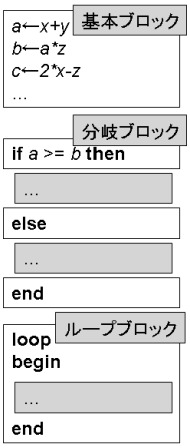


図 5 制御フローブロック
Fig. 5 Control flow blocks.

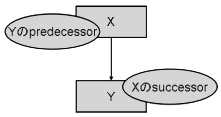


図 6 successor と predecessor
Fig. 6 Successor and predecessor.

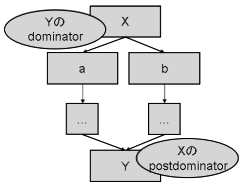


図 7 dominator と postdominator
Fig. 7 Dominator and postdominator.

クをループブロックと定義する．ループブロックは内部の繰返し処理として，他のブロックを内包しうる．ループ脱出の条件分岐ブロックも内含している（図 5 参照）．

これらのブロックは，互いに制御の依存関係でつながっている．また，依存関係を指す単語として，以下のようなものがある．

predecessor

ブロック X からブロック Y へ制御依存関係が伸びている場合の，X の Y に対する関係をさす（**図 6** 参照）．

successor

ブロック Y に対してブロック X から制御依存関係が伸びている場合の，Y の X に対する関係をさす（図 6 参照）．

dominator

ブロック Y のどの predecessor から制御依存関係を遡っていっても，必ずブロック X を通る場合の，X の Y に対する関係をさす（**図 7** 参照）．

postdominator

ブロック X のどの successor から制御依存関係をたどっていっても，必ずブロック Y を通る場合の，Y の X に対する関係をさす（図 7 参照）．

制御フローグラフ中に現れる各ブロックを，依存関係を利用して SIMD 命令で実行可能な形に変換することで，制御構造の SIMD 並列化を実現する．

4. 制御フローの SIMD 並列化

2 章で述べたように，スカラ実行を前提としたプログラムの条件分岐や WHILE 型ループを SIMD 並列化する場合，条件の比較結果が各要素ごとに異なるために，各演算を SIMD 命令に置き換えるだけでは正しいプログラムにならない．SIMD 命令でスカラの条件分岐や WHILE 型ループと同等の処理を行うためには，あらゆる条件分岐や WHILE 型ループを条件の成立不成立を問わず依存関係順に並べ，分岐集合地点で条件を利用して，各変数が正しい値をさすようにデータ調整処理を追加する必要がある．制御の依存関係は 3 章で示した制御フローグラフを作成することで解析が容易になる．各ブロック間の依存関係が条件分岐やループの依存関係なので，制御フローグラフを見ればブロックの処理順序も容易に解析できる．

この章では，制御フローグラフを利用して制御構造を SIMD 並列化するための手法について述べる．まず，4.1 節で今回の提案手法における制約と，手法の実現に必要とされる SIMD 演算命令について述べる．そして，4.2 節でアルゴリズムについて説明し，4.3 節ではアルゴリズムの実際の動きを，サンプルプログラムを例にして解説する．

4.1 SIMD 並列化のための制約と命令

制御構造の SIMD 並列化を考えるにあたって，問題を単純化するためにいくつかの制約を設ける．また，制御の SIMD 並列化を行う際に必要となるいくつかの SIMD 命令を定義する．

SIMD 並列化における制約は以下のようになる．ループに関する制約が複数あるが，プログラムに現れるようなたいていのループ文はこの制約を満たすので，ほとんどのループに適用できる．

- メモリアクセスやメモリアドレスなどの SIMD 命令で扱えないデータや演算は，SIMD 並列化対象となるプログラムブロックに含まれない．
- 手法を用いてプログラム構造を書き換えることで，例外を起こしうる演算（0 除算など）が発生することはない．
- ループの入口となるブロックは 1 つであるとする．つまり，ループの中へ入ったときに最初に実行されるブロックが条件に依存せず，確実に決まっている．
- ループの出口となるブロックは 1 つであるとする．つまり，ループブロックの successor は 1 つである．
- 入れ子になったループを一度に脱出するようなパスは存在しない．

次に，制御の SIMD 並列化を行う際に必要となる SIMD 命令を定義する．

vselect(v_1, v_2, v_{cond})

ベクトル v_1，v_2 の各要素から部分を取り出して新たなベクトルを生成する演算であると定義する．ベクトル v_{cond} の内容が偽の要素では v_1 の要素を取り出

Exp ::= Var | Var Op Exp | Op Var ; 演算を表す.
Stmt ::= Exp | v_{dest} '=' Exp ; 式を表す.
StmtList ::= Stmt | Stmt StmtList ; 式のリストを表す.
Block ::= *pred succ dom pdom* StmtList ; 制御フローグラフ中のブロックを表す.
BlockList ::= Block | Block BlockList ; ブロックのリストを表す.

v_{dest} ::= Var ; 代入文において代入先を表す.
pred ::= Block | ϕ ; ブロックの predecessor.
succ ::= Block | ϕ ; ブロックの successor.
dom ::= BlockList | ϕ ; ブロックの dominator.
pdom ::= BlockList | ϕ ; ブロックの postdominator.
Var 変数.
Op 演算子．単項演算子も含む.

図 8 データ構造
Fig. 8 Data structures.

し，真の要素では v_2 の要素を取り出す.

vcond_any(v_{cond})

ベクトル v_{cond} の各要素の内容のいずれかが真である場合に真を返す演算であると定義する.

SIMD 命令セットの中にはこれらに 1 対 1 で対応する命令を持たないものも存在するが，そのようなSIMD 命令セットであっても，基本的な論理命令，シフト命令，算術命令が揃っていれば，組合せによって実現可能である.

たとえば vselect 演算は，マスク値が真か偽かで取り出すデータを変える演算である．これは，

$$(v_1 \wedge \neg v_{cond}) \vee (v_2 \wedge v_{cond})$$

のような論理式で等価の処理を実現できる.

4.2 アルゴリズム

制御構造を SIMD 並列化するために，まず，制御フロー解析を行う．この際に生成されるデータ構造は**図 8** のようになっている．以下では，それらのデータ構造を用いて SIMD 並列化を進めていく.

まず，制御フロー全体を処理する大まかなアルゴリズムは**図 9** のようになる．SIMD 並列化対象とする制御ツリーグラフに対し，依存関係が解消されたブロックから SIMD 並列化していき，全ブロックが SIMD 並列化された時点で変換処理を終える.

条件分岐の SIMD 並列化

具体的な制御構造の変換ステップのうち，まずは条件分岐の SIMD 並列化について述べる．**図 10**，**図 11** が，条件分岐の SIMD 並列化に関わる部分である.

SIMD 命令はすべての要素に対し同じ演算を行うことしかできないので，条件成立の有無にかかわらず，成立時と不成立時の処理両方の結果を出して並列化を実現する．具体的には，双方の処理をつねに通過するように処理ブロックを直列に並べる．これによって，成立，不成立両方のパターンを処理した結果が得られる.

```
vectorize_block_list: BlockList L → BlockList
  BlockList L' ← ∅
  while L ≠ ∅ do
    Block B ← find_next_block(L, L')
    Block B' ← gen_new_block_from(B)
    if num_of_predecessor(B) > 1 then
      B' ← merge_branch(B')
    B' ← vectorize_block(B', B, L')
    L' ← L' ∪ {B'}
    L ← L − {B}
  return L'

vectorize_block: Block B', Block B,
          BlockList L → Block
  if is_loop_block(B) then
    B' ← vectorize_loop(B', B)
  elseif is_branch_block(B) then
    B' ← vectorize_branch_block(B, B')
  else
    foreach Stmt s = ⟨s_0, ··· , s_n⟩ ∈ B.stmt do
      Stmt s' = vectorize_stmt(s, B')
      B'.stmt ← B'.stmt ∪ {s'}
  if is_exit_of_loop(B) then
    Block LB ← get_parent_loop(B')
    StmtList term ←gen_loop_terminal(LB, B')
    B'.stmt ← B'.stmt ∪ term
  return B'
```

gen_new_block_from: Block B → Block
ブロック B の successor, predecessor, dominator, postdominator の依存関係をコピーしたブロックを返す.

図 9 全体の流れ
Fig. 9 General flow.

ところで，処理ブロックが直列に並べられているため，ある変数が先に並べられた分岐先ブロックの処理で内容が変更され，後に並べられた分岐先処理ブロックで参照されてしまうということが起こりうる．これでは本来，両方の分岐先にデータの依存関係ができてしまい，正しい結果が得られない．そこで，分岐中の処理ブロック内に存在する代入文は，図 11 中の vectorize_stmt，find_replacement にあるように代用の変数を用意し，分岐が集合するまでの間はそれを本来の変数の代わりとして利用する．そして，分岐が終わって制御の流れが集合した地点で，図 10 の merge_branch のコードで，分岐条件を使って分岐内で起こった演算の結果を合成し，分岐終了後に続くブロックが必要とする正しい値を得られるようにする.

以上で，条件分岐の SIMD 並列化は実現される.

ループの SIMD 並列化

次にループの SIMD 並列化について述べる．ループの SIMD 並列化ステップは**図 12** のようになっており，メインルーチンは vectorize_loop である.

```
vectorize_branch_block: Block B,Block B' →Block
  Var cond ← gen_new_variable()
  Exp c ← get_cond_exp(B)
  Stmt s ← vectorize_stmt({c})
  s ← gen_assign_stmt(cond, s)
  B'.v_cond ← cond
  B'.stmt ← B'.stmt ∪ {s}
  return B'

get_cond_path: Block from, Block to,
        Block term → Exp
  Exp e ← ∅
  if to ≠ ∅ ∧ to ≠ term then
    foreach Block p =
          ⟨pred_0, ··· , pred_n⟩ ∈ to.pred do
      Exp c ← get_cond_path(to, p, term)
      if e ≠ ∅ then
        c ← gen_or(e, c)
      e ← c
  if is_branch_block(to) then
    Exp c ← to.v_cond
    if to.else_part = from then
      c ← gen_not(c)
    if e ≠ ∅ then
      c ← gen_and(e, c)
    e ← c
  return e

merge_branch: Block B → Block
  while num_of_predecessor(B) > 1 then
    Block p_0 ← pred_0 ∈ B.pred
    Block p_1 ← pred_1 ∈ B.pred
    Block p ← find_nearest_branch(p_0, p_1)
    Exp cond ← get_cond_path(B, p_1, p)
    VarList L = find_replaced_varlist(p_0, p)
        ∪ find_replaced_varlist(p_1, p)
    Block pred' ← gen_new_block(p_0, p_1, B)
    foreach Var v = ⟨v_0, ··· , v_n⟩ ∈ L do
      Var v_0 ← find_replacement(v, p_0)
      Var v_1 ← find_replacement(v, p_1)
      Var v' ← gen_new_variable()
      B.var ← B.var ∪ {⟨v, v'⟩}
      Exp op ← gen_vselect(v_0, v_1, cond)
      Stmt s ← gen_assign_stmt(v', op)
      B.stmt ← B.stmt ∪ {s}
    B.pred ← B.pred − {p_0, p_1} ∪ {pred'}
  return B
```

gen_new_variable: → Var
SIMD 型の変数を生成して返す.
get_cond_stmt: Block B → Stmt
条件分岐ブロック B で利用されている分岐条件式を返す.
find_nearest_branch: Block B_0, Block B_1 → Block
ブロック B_0 とブロック B_1 が合流する分岐ブロックを検索して返す.

図 10 分岐の SIMD 並列化
Fig. 10 Vectorization of branch.

ループはベクトル中のすべての要素が終了条件を満たすまで行わなければ，すべての要素に対応する正しい結果が得られない．また，終了条件を満たした要素がそれ以上ループを繰り返すと，誤った結果をさしてしまう．それを回避するために，ループ条件変数を用意し，各要素のループ継続状態を保持させる．ループ

```
vectorize_stmt: Stmt s, Block B → Stmt
  Stmt s' ← ∅
  foreach Exp e = ⟨e_0, ··· .e_n⟩ ∈ s.exp do
    Var e'
    if is_variable(e) then
      e' ← find_replacement(e, B)
    else
      e' ← operator_to_simd_instruction(e)
    s'.exp ← s'.exp ∪ {⟨e, e'⟩}
  if is_assign_stmt(s) then
    Var v ← s.v_dest
    Var v' ← get_replacement(B, l)
    if is_in_branch(B) ∧v' = ∅ then
      v' ← gen_new_variable()
      B.var ← B.var ∪ {⟨v, v'⟩}
    s'.v_dest ← ⟨v, v'⟩
  return s'

find_replacement: Var v, Block B → Var
  Var v' ← get_replacement(B, v)
  if v' = ∅ then
    if num_of_predecessor(B) = 0 then
      Block BL ← get_parent_loop(B)
      if BL ≠ ∅ then
        v' ← get_replacement(BL)
      else
        v' ← v
    else
      Block pred ← pred_0 ∈ B.pred
      v' ← find_replacement(v, pred)
  return v'
```

operator_to_simd_instruction: Op op → Exp
演算子 op に対応する SIMD 命令を返す.

図 11 複雑な制御フローに対応したステートメントの SIMD 並列化処理
Fig. 11 Statement vectorization for complex control flow.

条件変数の初期値はそのループに到達するための論理式である．そのため，たとえば分岐中のループであれば分岐条件が，ネストしたループであれば親ループのループ条件変数が初期値に反映される．ループ中の分岐中にあるループであれば，条件と親のループ条件変数両方が初期値に反映される．

ループ中の処理は終了条件を満たした要素に対して反映させてはならないので，代入処理は条件分岐ブロック同様，すべて代用変数に対して行う．そして，ループを抜ける分岐を発見した時点でループ条件変数の内容を更新し，代用変数に格納された処理結果を反映と，ループ全体の終了条件の判定を行うためのコードを生成する．その処理を生成するのが図 12 の gen_loop_terminal である．また，ループの先頭に戻る際には，各データがループへ入ったときと同じ状態になっていなければならないので，gen_loop_terminal で代用変数の内容を反映するためのコードを生成する．

以上で，ループの SIMD 並列化が実現され，制御構造の SIMD 並列化が達成される．

```
find_next_block: BlockList L, BlockList L' → Block
  Block next ← ∅
  foreach Block B = ⟨B_0, ··· , B_n⟩ ∈ L do
    Block LB ← get_parent_loop(B)
    if (∀pred ∈ B.pred) ⊂
          (L' ∩ LB.blocks_in_loop) then
      next ← B
      break
  return next

vectorize_loop: Block B, Block B' → Block
  Block root ← get_root_block(B)
  B'.v_cond ← gen_variable()
  Exp p ← get_cond_path(B, pred_0 ∈ B.pred, root)
  Block LB ← get_parent_loop(B)
  if LB ≠ ∅ then
    p ← gen_and(p, LB.v_cond)
  Stmt s ← gen_assign_stmt(B'.v_cond, p)
  B' ← set_stmt_before_loop(B', {s})
  BlockList L ← B.blocks_in_loop
  VarList VL ← find_varlist_changed_in_loop(B)
            ∩ find_varlist_used_after_loop(B)
  StmtList S ← ∅
  foreach Var v = v_0, ··· , v_n ∈ VL do
    Var v' ← find_replacement(v, B')
    Var v'' ← gen_variable()
    Stmt a ← gen_assign_stmt(v'', v')
    B'.var ← B'.var ∪ {⟨v, v''⟩}
    S ← S ∪ {a}
  BlockList L' ← vectorize_block_list(L)
  StmtList term ← gen_loop_terminal(B', B')
  L' ← {S} ∪ L' ∪ {term}
  B'.blocks_in_loop ← L'
  return B'

gen_loop_terminal: Block LB, Block B → StmtList
  StmtList L ← ∅
  VarList VL ← find_varlist_changed_in_loop(LB)
  foreach Var v = ⟨v_0, ··· , v_n⟩ ∈ VL do
    Var v' ← find_replacement(LB)
    Var v'' ← find_replacement(B)
    Exp c ← get_cond_path(B, pred_0 ∈ B.pred, LB)
    c ← gen_and(c, LB.v_cond)
    Exp op ← gen_vselect(v', v'', c)
    Stmt s ← gen_assign_stmt(v', op)
    L ← L ∪ {s}
  if B ≠ get_lastblock_of_loop(LB) then
    Stmt s ← gen_branch_vcond_any(
                  LB.v_cond, succ ∈ LB.succ)
    L ← L ∪ {s}
  return L
```

get_root_block: Block B → Block
B がループに含まれている場合，ループの先頭ブロックを返し，ループの外にある場合，全体で最も最初に位置するブロックを返す．
set_stmt_before_loop: Block B, StmtList S → Block
ブロック B の前処理として処理リスト S を置く．
find_varlist_changed_in_loop: Block B → VarList
ループブロック B に含まれるブロックで行われる代入処理の代入先となっている変数をすべて返す．
find_varlist_used_after_loop: Block B → VarList
ブロック B の predecessor につながる全ブロックが使用する変数をすべて返す．

図 12 ループの SIMD 並列化
Fig. 12 Vectorization of loop.

```
while x < y do
  z ← z * x
  x ← x + 1
end
if a < x then
  a ← x
elseif a < z then
  a ← z
end
b ← a + c
```

図 13 サンプルプログラム
Fig. 13 Sample program.

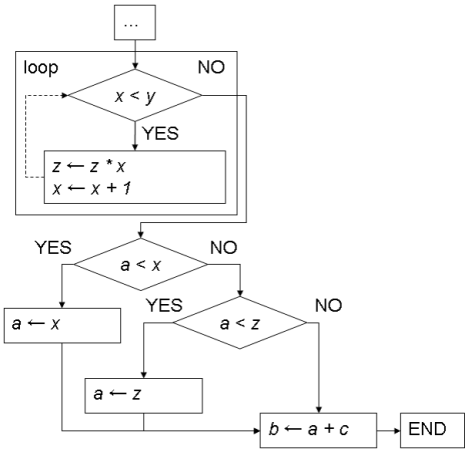


図 14 サンプルコードの制御フローグラフ
Fig. 14 Control flow of sample code.

4.3 サンプルプログラムによるアルゴリズムの解説

提案手法の実際の流れを，**図 13** の簡単なコードを例にあげて解説する．このコードの前半は WHILE 型ループ，後半は 2 つの条件分岐で構成されており，制御フローグラフにすると**図 14** の形になる．

まず，先頭のループブロックの前準備としてループ条件変数 $condL$ を用意し，初期化しなければならない．このループブロックは必ず通るブロックなので，初期条件は全要素真である．繰返し部分の最初の処理が，ループ脱出条件による条件分岐である．脱出分岐を発見したら，ループ条件変数の更新と脱出分岐前までに作られた代用変数を本来の変数に反映する処理を生成する．この段階で代用変数は存在しないので，ループ条件変数を更新し，繰返し処理を終えるための分岐を生成するのみである．次の x，z の演算は演算結果を直接反映してはならないので，代用変数 $x0$，$z0$ を用意して代入する．ここでループブロックの終端に到達するので，代用変数 $x0$，$z0$ の内容を反映する処理を生成する．この変換したコードの制御フローは**図 15** のようになる．図の濃い色の部分が，SIMD 並列化に

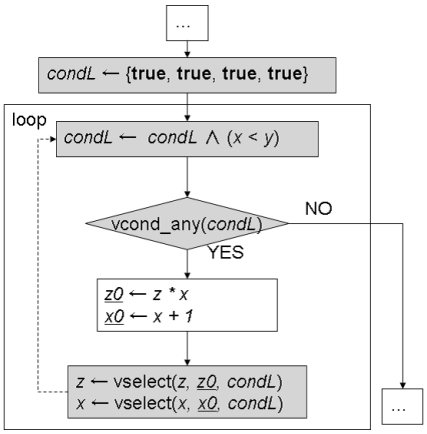


図 15 ループブロックの SIMD 並列化
Fig. 15 Vectorization of loop block.

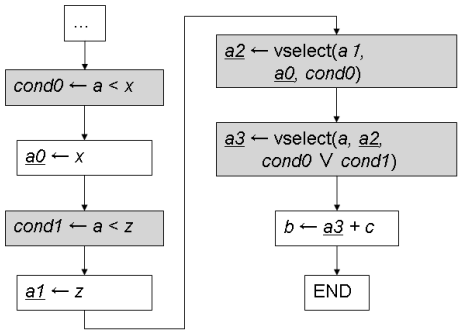


図 16 条件分岐ブロックの SIMD 並列化
Fig. 16 Vectorization of condition branch block.

よって追加した部分である．

ループブロックを抜けると条件分岐ブロックにやってくる．まずは，最初の条件分岐 $a < x$ の比較演算結果を分岐条件変数 $cond0$ に代入する．この条件分岐ブロックの successor は 2 つあるが，どちらから処理してもかまわないので，条件が真の場合である $a \leftarrow x$ を先に処理することにする．このブロックは条件分岐先にあるため，代入は代用変数に対して行わなければならない．この場合は，変数 a に対して代用変数 $a0$ を用意する．このブロックの successor，すなわち $b \leftarrow a + c$ を擁する基本ブロックは predecessor の一部が未処理なので，次は先ほどの条件分岐ブロックのもう 1 つの successor を処理する．これも条件分岐ブロックであるので，$a < z$ の比較演算結果を条件分岐変数 $cond1$ に代入する．次に，条件が真であった場合のブロックを処理する．やはり変数 a への代入は，代用変数 $a1$ への代入へと置き換える．

これで分岐処理は終了し，複数のパスが集合する地点に到達する．ここでは 3 つのパスが集合しているが，パスの合成は 2 つずつ行う．パスの合成順序は問われないが，ここでは最初の条件分岐が真であった場合と，次の条件分岐が真であった場合とを先に合成する．両方のパスで操作された変数は a のみなので，a に対して合成を行う．$a0$ を vselect 演算の条件が真の場合に取り出す内容とすると，$a1$ が偽の場合の内容，$cond0$ が条件となる．これらを利用して生成した vselect 演算は，新しい代用変数 $a2$ に対して行う．次は，今合成した $a2$ と，2 つの条件分岐が失敗した場合の元の値 a との vselect 演算を生成する．$a2$ を真の場合の内容とすると，a が偽の場合の内容となり，$cond1 \vee cond0$ が条件となる．この vselect 演算の結果を新たな代用変数 $a3$ を用意して代入する．これで変数 a の内容が $a3$ に格納されている状態となる．以後のブロックでの a の扱いはこのサンプルコードでは存在しないが，a への参照をすべて $a3$ で置き換えるか，$a3$ の内容を a へコピーすることで正しい振舞いをすることができる．この変換したコードのフローは**図 16** のようになる．ループと同様，図の濃い色の部分が追加部分である．

以上で全ブロックを処理したので，変換処理を終える．変換後のコードは**図 17** である．

```
condL ← {true, true, true, true}
loop
  condL ← condL ∧ (x < y)
  if vcond_any(condL) = false
    break
  end
  z0 ← vector_mul(z, x)
  x0 ← vector_add(x, {1, 1, 1, 1})
  z ← vselect(z, z0, condL)
  x ← vselect(x, x0, condL)
end
cond0 ← a < x
a0 ← x
cond1 ← a < z
a1 ← z
a2 ← vselect(a1, a0, cond0)
a3 ← vselect(a, a2, cond0 ∨ cond1)
b ← vector_add(a3, c)
```

図 17 SIMD 並列化したサンプルコード
Fig. 17 SIMD-vectorized code.

5. 手法の評価

4 章において提案した手法を COINS コンパイラインフラストラクチャ[15]（以下 COINS）に実装した．そして，手法の適用の有無によるプログラムの速度向上比率を比較することで提案手法の評価を行った．COINS は研究用のコンパイラ基盤で，ソースコード

を言語に近い高水準中間表記，アセンブリ言語に近い低水準中間表記，各マシン用アセンブリ言語の順番に変換し，その過程で最適化を行う．今回の提案手法は高水準中間表記に対して実装し，SIMD 命令に対応する組み込み関数を使って並列化したプログラムを C 言語のコードとして出力することで機能を実現した．

性能評価は提案手法を組み込んだ COINS でテストプログラムを SIMD 並列化し，並列化前後で実行速度を比較して行った．本論文の提案手法による速度向上が期待できるプログラムは，プログラムの一部が SIMD 命令で並列処理でき，かつ，その部分に分岐やループ文といった複雑な制御を持つものである．

lifegame ライフゲームを行うプログラムである．各セルが次の世代で生き残る，または発生するかどうかの判定計算を変換して，16 並列で実行するようにした．変換対象は入れ子になっている複雑な条件分岐があった．ライフゲームの規模を 256×256 としてランダムに初期配置をして 3,200 世代進める，という処理を 10 セット行うのにかかった時間を比較した．

threshold 画像処理の一種で，しきい値を利用して与えられた画像の配色を二極化するプログラムである．各画素に対する計算を変換し，4 並列で実行可能にした．変換対象には条件分岐 1 つが含まれる．この画像処理を一般的なピクセル数 256×256 のサンプル画像 12 枚を 1 枚につき 2,000 回実行し，すべての画像に対する処理が終わるまでにかかった時間を比較した．

edging 画像処理の一種で，ある画像のエッジを検出するプログラムである．各画素に対する計算を変換し，4 並列で実行可能にした．変換対象には条件分岐 1 つが含まれる．対象画像や比較方法は **threshold** と同じである．

add 画像処理の一種で，2 つの画像の各画素を加算するプログラムである．各画素に対する計算を変換し，4 並列で実行可能にした．変換対象には条件分岐が複数含まれる．対象画像や比較方法は **threshold** と同じである．

beta ベータ分布の計算に用いられる関数 $\beta(x,y) = \frac{\Gamma(x)\Gamma(y)}{\Gamma(x+y)}$ の計算を並列化した．具体的には，$\beta(x,y) = \exp(\ln(\Gamma(x))+\ln(\Gamma(y))-\ln(\Gamma(x+y)))$ という形にして，3 つの $\ln(\Gamma(x))$ を並列で実行可能にした．性能比較は，$\beta(x,y)$ 関数に毎回ランダムな値を渡して，合計 10,000,000 回実行するのにかかった時間を計測することで行った．変換対象の $\ln(\Gamma(x))$ はループ 1 つと複数の条件分岐が含まれ，今回のテストプログラムの中で最も演算のステップ数が多かった．

mandel Mandelbrot 集合を計算し，描画するプログラムである．各座標の色を計算するブロックを変換し，4 並列で実行可能にした．変換対象にはループが 1 つあり，内部に複数のループ脱出分岐が含まれる．1,024 × 1,024 の領域の全体像の各画素を 128 段階調に分ける処理を 800 回行うのにかかった時間を比較した．

表 1 テストプログラムの実行環境
Table 1 Environment of test programs.

プロセッサ	PowerPC 970 1.8 GHz
SIMD 命令セット	AltiVec(VMX)
2L キャッシュ	512 KB
メモリ	1 GB
コンパイラ	gcc 4.0.0
コンパイラオプション	-O2 -faltivec

表 2 速度向上率表
Table 2 Table of speedup ratio.

プログラム名	スカラ (sec)	SIMD(sec)	速度比率
lifegame	20.78	1.70	12.3
threshold	15.3	11.56	1.32
edging	44.19	31.01	1.43
add	24.4	13.47	1.81
beta	6.89	5.78	1.19
mandel	32.63	10.95	2.98

これらのプログラムを，SIMD 並列化しなかったものと，SIMD 並列化したものとで時間計測し，処理速度を比較した．テスト環境は**表 1** のとおりである．SIMD 並列化した場合もしなかった場合も，コンパイラおよびコンパイラオプションは同じものを使っている．

テスト結果は**表 2** のようになった．実行結果は 40 回テストした結果の計測時間の平均を，プログラムごとに載せている．平均時間の速度比率をグラフ化したのが **図 18** である．

lifegame は各セルの計算を，SIMD 命令を使って 16 並列実行する形にしたものであり，それに見合った速度向上率となっている．ところで，256 × 256 の規模においては，すべてのデータが 2 次キャッシュに載っていた．そこで，規模を拡大し，2 次キャッシュにちょうど載るサイズと，それを上回ってしまうサイズでの向上率比較も行い，**表 3** の各 **lifegame** 項目に表した．括弧の中の数字はそれぞれ問題サイズを表しており，上から 10,000 回，8,000 回，4,000 回実行するのにかかった時間を 40 回計測した．400 × 400 はちょうどキャッシュに載るサイズで，それ以上のサイズはキャッシュから溢れている．400 × 400 では表 2 とほ

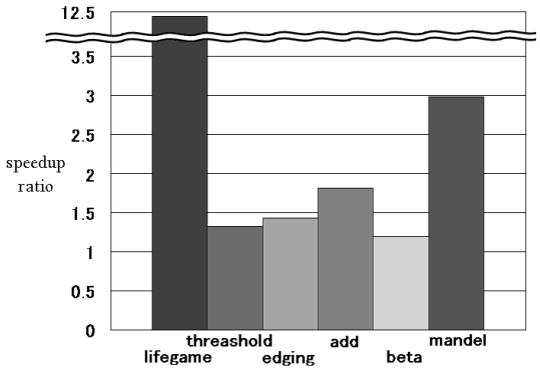


図 18 平均速度向上率グラフ
Fig. 18 Graph of average speedup ratio.

表 3 追加テストプログラムの実行結果
Table 3 Result of extra test programs.

プログラム名	スカラ (sec)	SIMD(sec)	速度比率
lifegame(400)	17.50	1.44	12.16
lifegame(512)	22.57	2.26	9.97
lifegame(1024)	47.74	5.71	8.36
mem-graphic	2.42	1.88	1.29
loggamma	4.27	1.81	2.36

ぼ同じ速度向上率が見られたが，それ以上の規模になるに従って，向上率が低下している．つまり，SIMD 命令で大規模データを処理しようと考えた場合，メモリアクセス速度は重要な要素となる．

また，**threshold**，**edging**，**add** といった画像処理系プログラムは，4 並列実行になったにもかかわらず速度向上率は 2 倍にも達していない．そこで，画像処理系プログラムで利用した画像と同規模である 256 KB のデータ転送処理を 10,000 回行うのにかかった時間を 40 回計測した．これが表 3 中の **mem-graphic** である．これと画像処理系の速度向上率を見てみると，**lifegame** の問題規模を拡大したときと同様に，メモリアクセス速度に速度向上率が制約される形となっているのが分かる．各画像処理プログラムの向上率は，おおむね **mem-graphic** の向上率に沿った値になっているので，SIMD 並列化の効果は十分にあったといえる．

beta は SIMD 命令で実行されるステップが多いプログラムであり，メモリアクセスも定数の取り出し以外ほとんどないプログラムであったが，1.19 倍程度の速度向上にとどまった．$\beta(x, y)$ 関数は $\Gamma(x)$ 関数後の計算は並列実行不能であった．SIMD からスカラ実行へ移るためのコストと，スカラ実行のコストとが速度向上の妨げになった可能性がおおいにある．そこで，$\Gamma(x)$ 関数部分の計算のみの実行速度を比較してみた．それが表 3 の **loggamma** である．これは 15,000,000 回分の $\Gamma(x)$ 関数部分の計算をスカラで実行したときと，SIMD 命令で 3 並列実行したときとの時間の平均を比較している．2.36 倍程度の速度向上が見込めていることから，$\beta(x, y)$ 関数においては，$\Gamma(x)$ 関数部分の SIMD 並列化による速度向上効果はあったが，その後の計算が SIMD 並列化不可能であったため，結果的に速度向上率が低くなったと考えられる．

mandel は演算にメモリ上のデータを必要としないため，負荷がかかるのは SIMD 命令による演算のみである．さらに，ループで実行される処理も比較的単純で，SIMD 命令が得意な形であった．そのため，SIMD 命令による 4 並列演算に見合った速度が出た．ループを SIMD 命令で並列実行することで，十分に速度向上が得られるパターンであるといえる．

6. ま と め

SIMD 命令という複数のデータを並列処理するための命令セットで条件分岐や WHILE 型ループといった複雑な制御構造を並列処理するための手法はあまり提案されておらず，自動 SIMD 並列化の対象からはずされてきた．そこで本論文では，複雑な制御構造を SIMD 命令で並列実行可能にするための手法を提案した．提案した手法を COINS コンパイラインフラストラクチャに実装し，テストプログラムを SIMD 並列化して SIMD 並列化前後で速度比較したところ，平均 1.19 倍から 12.3 倍の速度向上が見られた．

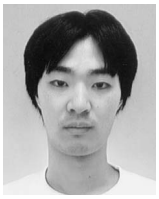
今後の課題としては以下のような項目があげられる．まず，今回扱った AltiVec という SIMD 命令セットには vselect，vcond_any に相当する命令があったが，対応する命令が欠けている SIMD 命令セット（たとえば，Intel x86 系の SSE など）において，どれほどの速度向上が得られるものかは検証する必要がある．また，SIMD 命令による最適化のための手法として提案されている SLP 解析は演算レベルで SIMD 並列化可能な部分を発見するための手法なのに対して，今回提案した手法は制御レベルの SIMD 並列化を行う手法である．SLP と提案手法を併用できるような SLP 解析の拡張が必要とされる．

参 考 文 献

1) インテル株式会社：IA-32 インテルアーキテクチャ・ソフトウェア・ディベロッパ・マニュアル中巻：命令セットリファレンス (2003).
2) Diefendorff, K., Dubey, P.K., Hochsprung, R. and Scales, H.: AltiVec Extension to PowerPC

Accelerates Media Processing, *IEEE Micro*, Vol.20, No.2, pp.85–95 (2000).
3) ARM. http://www.arm.com/products/CPU/arch-simd.html
4) Larsen, S. and Amarasinghe, S.: Exploiting Superword Level Parallelism with Multimedia Instruction Sets, *Programming Language Design and Implementation*, Vancouver, Canada, *Proc. ACM SIGPLAN Conf.*, pp.145–156 (2000).
5) Eichenberger, A.E., Wu, P. and O'Brien, K.: Vectorization for SIMD Architectures with Alignment Constraints, *PLDI'04*, Vol.9-11, pp.82–93 (2004).
6) Peng Wu, A.E.E. and Wang, A.: Efficient SIMD Code Generation for Runtime Alignment and Length Conversion, *Proc. International Symposium on Code Generation and Optimization*, San Jose, *CGO'05*, pp.153–164 (2005).
7) Kudriavtsev, A. and Kogge, P.: Generation of Permutation for SIMD Processors, *2005 ACM SIGLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems, LCTES'05*, pp.147–156, ACM Press (2005).
8) Autovectorization in GCC, *Proc. 2004 GCC Developer's Summit* (2004). http://www.gccsummit.org/2004/
9) XLSOFT: Intel C++ Compiler リファレンスマニュアル. http://www.xlsoft.com/jp/products/intel/compilers/index.html
10) 地球シミュレータセンター. http://www.es.jamstec.go.jp/
11) Allen, J.R., Kennedy, K., Porterfield, C. and Warren, J.: Conversion of Control Dependence to Data Dependence, *Annual Symposium on Principles of Programming Languages*, *Proc. 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (1983).
12) 長島重夫，田中義一：スーパコンピュータ，オーム社 (1992).
13) 村井，末廣，岡部，國枝，津田：ループ交換による while 型ループの自動ベクトル化，日本ソフトウェア科学会第 11 回大会論文集，pp.65–68 (1994).
14) Ferrante, J., Ottenstein, K.J. and Warren, J.D.: The Program Dependence Graph and Its Use in Optimization, *ACM Trans. Prog. Lang. Syst.* (*TOPLAS*), Vol.9, Issue 3, pp.319–349 (1987).
15) COINS コンパイラ・インフラストラクチャ協会. http://www.coins-project.org/

(平成 18 年 9 月 12 日受付)
(平成 18 年 12 月 11 日採録)



廣松 悠介

1983 年生．2005 年電気通信大学情報工学科卒業．同年東京大学新領域創成科学研究科基盤情報学専攻修士課程へ入学．2007 年 3 月同修士課程修了見込．

黒田 久泰（正会員）

1970 年生．1993 年名古屋大学理学部物理学科卒業．1995 年京都大学大学院工学研究科応用システム科学専攻修士課程修了．2000 年東京大学大学院理学系研究科情報科学専攻博士課程満期退学．同年より東京大学情報基盤センター助手．博士（理学）．ACM，IEEE，SIAM，人工知能学会，日本応用数理学会各会員．

金田 康正（正会員） 1949 年生．1973 年東北大学理学部物理第二学科卒業．1975 年東京大学理学系研究科物理学専攻修士課程修了．1978 年東京大学理学系研究科物理学専攻博士課程修了．理学博士．同年名古屋大学プラズマ研究所附属電子計算機センター助手．1981 年東京大学大型計算機センター助教授．1984 年ケンブリッジ大学計算機研究所客員研究員．1997 年東京大学大型計算機センター教授．1999 年東京大学情報基盤センタースーパーコンピューティング研究部門教授．1983 年情報処理学会論文賞（邦文）．1994 年情報処理学会 Best Author 賞．1998 年情報処理学会論文賞（和文）．2003 年兵庫県揖保川町「金もくせい」賞．2005 年第 37 回市村産業賞貢献賞．ACM，SIAM，日本応用数理学会，プラズマ核融合学会，電子情報通信学会各会員．