SIMD ベンチマークの設計と実装

鈴木 貢[†] 小川大介[††]
室田朋樹[†††] 渡邊坦[†]

コンパイラのメディア処理向け SIMD 拡張命令に対する最適化技術は，現在は研究途上であり，SIMD 命令が適切に用いられる場面は非常に限定的である．コンパイラのユーザは，どんなコーディングを行えば SIMD 命令の恩恵に与れるかを知っておく必要があるが，それはコンパイラの設計によりまちまちである．逆にコンパイラ開発者にとっても，設計目標を決めたり，コンパイラの検証を行ったりするためのテストプログラムは有用である．そこで，コンパイラによる SIMD 拡張命令の活用度の調査に重点を置いた，ベンチマークプログラムの設計法を提案し，実装と評価を行った．ベンチマークの例題は，平均等の基本的なものと，実際のアプリケーションプログラムから得た例題に基づくものとからなる．

Design and Implementation of SIMD Benchmark

Mitsugu Suzuki,[†] Daisuke Ogawa,[††] Tomoki Murota[†††] and Tan Watanabe[†]

Development of compiler optimization techniques for extended SIMD instruction sets for media processing is on going task, and efficient use for those instructions are currently limited. Though a compiler user must know appropriate coding that can extract the advantage of SIMD instructions, such coding are different according to designs of compilers. Conversely, test programs that can be used for deciding design items and verifying a compiler will help the designer. We designed, implemented, and evaluated a set of benchmark programs, where the set is concentrated on survey of utilization for extended SIMD instruction sets by compilers. Examples consisting of basic operation patterns such as averaging values, and processing patterns in real application programs are presented.

1. はじめに

SIMD（Single Instruction Multiple Datastream）命令の一種として，レジスタ分割式ショートベクタ命令がある．マルチメディア処理の高速化などのために，既存の命令セットの拡張という形で，この種の SIMD 命令セットを追加することは，多くの汎用プロセッサで行われており，アーキテクチャの 1 つの趨勢となっている．本論文では，この種の拡張命令セットを単に「SIMD 命令セット」と呼ぶことにする．

現行の多くのプロセッサに SIMD 命令セットが具備されているにもかかわらず，それをコンパイラで活用することはまだ研究途上であるため，SIMD 命令を利用するには，アセンブリ言語やイントリンシックルーチン（高級言語から呼び出せる機械語命令列ルーチン）を利用することを強いられる場合が多い．一部のコンパイラでは SIMD 命令列を生成できるが，まだ非常に限定的であり，自動的に活用できるとはとてもいえない段階にある．

このような現状では，プログラム作成者は，ソースプログラムにおいてどのようなコーディングを行えば SIMD 化されるかを知っておく必要がある．それは，アルゴリズムを素直にプログラム化した場合にくらべ，SIMD 命令を適用しやすいように変形したパターンとなる場合が多い．一般に，その変形には，もとのアルゴリズムに近いものから，SIMD 命令に近いものまで，いくつもの段階がありうる．SIMD 命令セットの適用できそうな問題を多く収集して，それらを体系的に整理し，多くの段階ごとにパターン化したプログラム集を作成すれば，プログラム作成者は，使用するコンパイラがどんなパターンならば高速処理できる SIMD 命令列を生成するか，つまり SIMD 命令を活用できる

† 電気通信大学
The University of Electro-Communications
†† セイコーエプソン株式会社
Seiko Epson Corporation
††† 東京大学大学院新領域創成科学研究科
Department Frontier Informatics, Graduate School of Frontier Sciences, The University of Tokyo

かを容易に知ることができる．

一般的な「ベンチマークプログラム」という語は，ハードウェアやシステムソフトウェアの性能評価を数値化するためのプログラム集という意味を持つが，本論文では，上記のようなプログラム集も，ベンチマークプログラム，あるいは単にベンチマークと呼ぶことにする．

このようにして構成された SIMD ベンチマークは，コンパイラ開発者にとっても，設計目標を定めやすくなり，テストプログラムとして利用することもできて有用である．

ベンチマークにはすでに多くの種類がある．しかし，5 章で述べるように，従来のベンチマークは，通常の命令セットやベクタ命令セットに対して，コンパイラによる最適化の効果や命令セットの実装方法による性能の変化を調べるのに適していても，以下の理由によりコンパイラが SIMD 命令セットを適切に活用したコードを生成できるかどうかの評価には適していない．

- それらのベンチマークは大きい実用プログラムをそのまま提示しているものであって，SIMD 命令セットを活用できる部位の特定が容易でなく，SIMD 命令セットの活用対象となりうるコーディングパターンを体系的に整理したものでもないので，上記の目的には合わない．
- 3.2 節の 2 つ目の例のように，アルゴリズム自体はデータ並列性などの点で SIMD 命令セット向きであるが，そのままではプログラム中の定数値やデータサイズなどの選択が不適切なため，現在の最適化技術では効果的な SIMD 命令の適用が不可能か困難であるものが多い．
- 3.4 節で詳述するアラインメントへの不整合やベクタの重なりがある場合に，誤った SIMD 命令を生成していないことが判定できない．

実際の（ソースが公開されている）プログラムにおいて，現在のコンパイラの SIMD 命令の自動的な活用による高速化は必ずしも多いとはいえない．しかし，たとえば動画圧縮プログラムにおいては，その実行時間の 3 割程度を占める画像間の比較演算において，SIMD 命令をうまく適用すると，その部位だけで 5 倍以上の高速化が可能であり，全体として 2 割以上の高速化が可能になる．最適化技術の高度化によって，その効果が飽和に近づいている現状では，通常命令関連の最適化だけでは，このような高速化の達成は非常に難しいため，SIMD 命令の活用による高速化の可能性を探ることは，十分に価値がある．また，プロファイリングではわずかな実行時間を占めるに過ぎない部位についても，コンパイラが SIMD 命令を自動的かつ適切に適用すれば，効果の積み重ねによる高速化を達成できる．

そして，SIMD 命令セットという比較的新しい機能について，それを有効活用し得る部位を実際のプログラムの中から見つけ出し，コンパイラのユーザや設計者にその部位の多様な表現を系統的に示すことは，最終的にプログラムの実行性能の向上におおいに寄与するものと考えられる．

本論文では，SIMD 命令セットに特化したベンチマークを作成することを提案し，その 1 つの実装例を示す．これには，SIMD 命令の処理機能を単体でテストする例題のほかに，実際のアプリケーションから抽出した処理パターンから得た例題も含める．現在の実装は整数演算の SIMD 命令セットに特化している．それは以下の設計要件を満たすことを目標とする．

- コンパイラの設計者に対して，SIMD 命令の適用対象となりうるパターンを体系的に提示することによって，最適化の設計目標を示す．
- SIMD 最適化コンパイラのユーザに対して，そのコンパイラ向けのコーディングパターンを示す．
- SIMD 命令の誤った適用の検出を可能にする．
- SIMD 命令セットの実装の違いによる性能比較を可能にする．

2. SIMD 命令セット

ここでは，多くの SIMD 拡張命令セットに共通の機能と，コード生成戦略で考慮すべき点について説明する．

2.1 レジスタ分割式ベクタ処理

同じ SIMD 型の計算機でも，スーパコンピュータに代表されるベクタプロセッサや CM-5 [1] のような超並列マシンでは，処理データ幅は，ベクタレジスタの幅や個別プロセッサの基本アーキテクチャが決めるレジスタ幅で決まり，いっぺんに処理できるデータの長さは，ベクタレジスタ長やシステムのハードウェア構成で決定される．

一方で，本論文が対象にしている SIMD 命令は，**図 1** や **図 2**，**図 3** に示すように，アーキテクチャ依存の 64 ビット，128 ビットといったサイズ N の汎用レジスタや浮動小数点レジスタあるいは専用レジスタをベクタレジスタのように扱い，8 ビット，16 ビット，32 ビットといった処理データサイズ M に分割し，同位置にあるフィールド間の演算を並列に行い，同位置にあるフィールドに値をセットするタイプのものである．したがって，演算の並列性は，ベクタレジスタのサイ

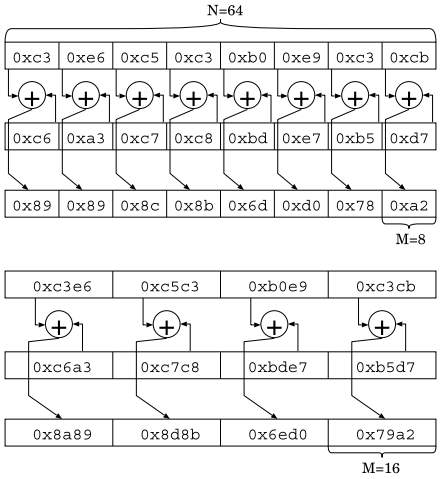

図 1 SIMD 命令の例
Fig. 1 Examples of SIMD instructions.

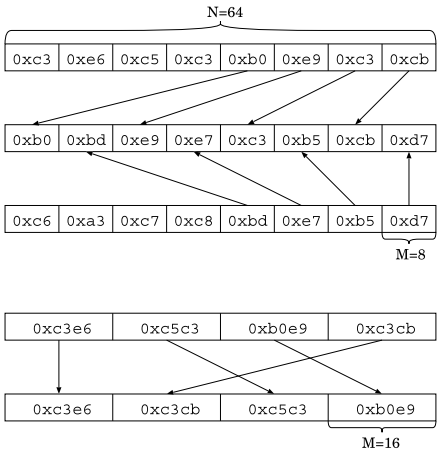

図 3 シャッフルやマージ命令の例
Fig. 3 Examples of shuffling and mergeing instructions.

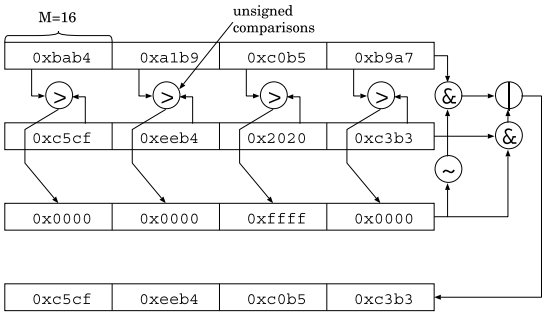

図 2 SIMD 命令セットにおける比較演算の例
Fig. 2 An example of comparison instructions in SIMD instruction sets.

ズを処理データサイズで割った値 N/M となる．

SIMD 命令セットを効率良く活用するには，処理データサイズ M をなるべく小さくして演算の並列性を高める必要があるが，高級言語には一般に汎整数拡張という規約が設定されいる．これは，8 ビットや 16 ビットといったサイズのデータをオペランドとして使う際には，汎整数型（integral type）と呼ばれるサイズの整数になるように符号拡張やゼロ拡張を施してから演算に用いるという規約である．文献 2)，3) で提案されている方法を用いて，汎整数拡張を行うのと同じ結果が得られるコードを，効果的に SIMD 命令を用いて生成することが望まれ，これはベンチマークの調査項目の 1 つとなる．

一方で，SIMD 命令セットにはビットごとの論理演算も用意されているが，この場合は処理データサイズの幅が N に固定である．

2.2 比較命令

Sparc の VIS[4] を除く現行の SIMD 命令セットのほとんどでは，比較演算の結果は図 2 に示すように，同じ幅のレジスタのフィールドの全ビットを 1 とする「真」と，全ビットを 0 にする「偽」として表現する．通常は，if 文や C 言語の条件式は，if 変換[5] を施されて選択演算に変換される．この真偽値は，図 2 の右側に C 言語の演算子を用いて表したような，積和の論理演算によってデータの選択をフィールド別に行う命令列のビットマスクとして使われる（この例では大きい方の値を求める演算になっている）．

この真偽値は，それぞれ −1 と 0 の数値として用いられることもある．これをうまく利用すると，積和を用いる場合に比べて，演算のステップ数を大きく減らすことによって高速化できる場合があり，そのことも調査項目の 1 つとなる．

2.3 SIMD 命令セット特有の特殊な演算

シャッフル・マージ命令

SIMD 命令セットの多くは，図 3 のようなシャッフル命令やマージ命令を有している．しかし，これらの有効な適用をユーザプログラムから抽出する技術は，まだ公表されていない．現状では，スカラベクタ変換（ベクタレジスタの各フィールドに同じ値をセットする演算）や，リダクション演算（ベクタレジスタの各フィールドの値の総和を求める演算等）等における定型コード生成に使われるにとどまっている．

乗算命令

乗算命令の結果を格納するには，通常はデータサイズの 2 倍のサイズのレジスタを必要とするが，

2.1 節で説明したフォーマットでは，積のすべての

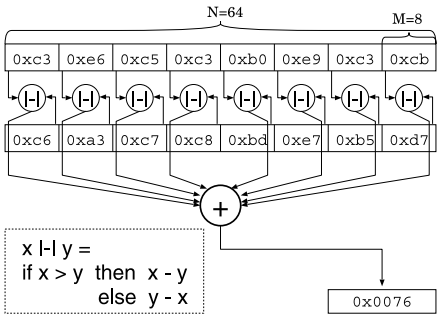

図 4 特殊なリダクション命令の例
Fig. 4 An example of particular reduction instruction.

結果を格納することはできない．乗算のデータの取り出し方や，結果の格納方式は拡張命令セットによってまちまちである．IA-32[6] の MMX や SSE2 では積の上半分を残す命令と下半分を残す命令があり，PowerPC の Altivec（Velocity Engine）では結果の上半分を残す命令のみであり，MIPS アーキテクチャ拡張の PlayStation2 の EmotionEngine では補助レジスタ HI と LO を使って積のすべてを残す．

飽和演算命令

演算の結果が決められたデータサイズと表示方式（符号あり/符号なし）で可能な表現範囲で溢れを起こした場合に，結果を表現可能な最大・最小値におさえ込む操作を「飽和」と呼ぶ．そして加算や減算の際に，同時に飽和処理を行う命令が用意されている機種もある．たとえば 8 ビットのデータサイズどうしの加減算に対して飽和処理を行う場合，飽和処理がない演算命令を使って飽和処理を行うと，加減算の結果を 9 ビット以上使って保持して，溢れの判定を行う必要があるので，並列度が半分になり，さらにデータサイズの変換や値の選択のオーバヘッドがかかる．しかし，飽和つきの加減算命令を適用すると，8 ビットの処理データサイズでだけで済む．コンパイラが飽和演算命令を活用するか否かは，調査項目の 1 つとなる．

特殊なリダクション命令

メディアデータの非可逆圧縮では，要素ごとに標本値と符号化後の値の差の絶対値を求め，それらの和を求めるという演算を多用しているため，たとえば SSE2 や VIS 等の拡張命令セットでは，**図 4** に示すような SIMD 命令としてフィールドごとの 8 ビットデータ間の差の絶対値を求める命令を有している．この命令は，本来の用途以外に，0 との絶対差を求めるような命令列を生成することによって，ベクタレジスタのフィールドの総和を求めるといった用途にも応用できる．

この種の命令の本来の適用範囲は比較的狭いが，上

```
#define AVE(x,y) (((x)>>1)+((y)>>1)+\\
                  (((x)|(y))&1))
struct {
  short r, g, b, a;
} *u1, *u2, *u3;

  short *v1, *v2, *v3;

  for (i = 0; i < M; i++)            // (A)
    *v1++ = AVE(*v2++, *v3++);
  for (i = 0; i < M; i++)            // (B)
    v1[i] = AVE(v2[i], v3[i]);
  for (i = 0; i < M; i += 4) {       // (C)
    v1[i] = AVE(v2[i], v3[i]);
    v1[i+1] = AVE(v2[i+1], v3[i+1]);
    v1[i+2] = AVE(v2[i+2], v3[i+2]);
    v1[i+3] = AVE(v2[i+3], v3[i+3]); }
  for (i = 0; i < M - 3; i += 4) {   // (D)
    v1[0] = AVE(v2[0], v3[0]);
    v1[1] = AVE(v2[1], v3[1]);
    v1[2] = AVE(v2[2], v3[2]);
    v1[3] = AVE(v2[3], v3[3]);
    v1 += 4; v2 += 4; v3 += 4; }
  for (i = 0; i < M; i++) {          // (E)
    u1[i].r = AVE(u2[i].r, u3[i].r);
    u1[i].g = AVE(u2[i].g, u3[i].g);
    u1[i].b = AVE(u2[i].b, u3[i].b);
    u1[i].a = AVE(u2[i].a, u3[i].a); }
```

図 5 SIMD 最適化可能なループの例
Fig. 5 Examples of SIMD optimizable loops.

のような使い方を 3.3 節で示すようなプログラミング上の工夫やコンパイラの最適化で行うことによって，適用範囲を広げることができる．また，シャッフルやデータサイズ変換を組み合わせてリダクションを行う場合に比べて，命令のステップ数が非常に少なくなり，高速化に寄与する可能性がある．この種の命令をうまく活用できるかどうかは，調査項目の 1 つとなる．

2.4 並列化戦略

SIMD 命令を生成するための並列化には，次の 2 つの方針がある．

(1) ベクトル化に基づく方式[7)~9)]
(2) 覗き穴最適化の拡張に基づく方式[10)]

図 5 の例の場合，(1) では (B) が SIMD 命令適用の対象となり，静的な解析や動的なポインタの値の解析を行えば (A) も対象となる．(2) の場合は，(A) や (B) はそのままでは SIMD 命令適用の対象とはならず，あらかじめソースレベルあるいは中間言語レベルで (C) や (D) のように展開しておく必要がある．しかし，コンパイラ内の中間言語の設計に依存することではあるが，たとえば COINS コンパイラインフラストラクチャ[11)] の低水準中間言語では，(D) と (E) は同じ表現になるので，(E) のようなプログラムに対しても SIMD 命令を適用できる．コンパイラが (A)〜

(D) のどの形のコードに対して適切な SIMD 命令を生成するか，あるいは (E) のような形のコードに対して SIMD 命令を生成できるかどうかは，調査項目となる．

ベンチマークの実装にあたっては，設計要件であげた「コンパイラのユーザに対して，対象コンパイラ向けのコーディングを示す」という事項を満たすために，処理系が (1) と (2) のいずれの方針を採っていても SIMD 命令の適用が促されるように，ループの形と展開済みの形の両方のコーディングを行う．

3. SIMD ベンチマークの設計

ベンチマークは，それ自体の処理系依存性を避けるために，ANSI C 言語の機能の範囲内で記述し，処理系依存性のあるベクタ処理等向けの特別な構文やプラグマは使用しない．3.1 節の方法で収集した処理パターンは，3.2 節に示すように SIMD 命令向きに変形して複数のバリエーションを構成し，それぞれを 3.3 節に示す方法で展開し，同じ処理パターンに対して複数のコーディングを作成する．各々のコーディングは 1 つの関数としてまとめ，それぞれについて実行時間を計測し表示する．実行時間の計測には，Unix 系 OS では関数 `getruseage()` を，Windows 系 OS では関数 `clock()` を使用する．

3.1 例題の収集

例題としては，加減算のような単体の SIMD 命令に対応する単純なもののほかに，平均や最大値のように数ステップの SIMD 命令で構成可能な基本処理パターンによるものと，C 言語のソースプログラムが公開されている実際のプログラムから抽出した処理パターンをもとに作成したものを収集した．この節では，実際のプログラムからの処理パターンの抽出について述べる．

実際のプログラムからの処理パターンの抽出は，以下のようにして行う．

(1) プログラムを gprof [12] か Intel VTune [13] を用いて，プロファイリングする．
(2) ホットスポット（実行時間に占める割合が多い箇所）を，ループや文レベルの実行部位単位で抽出する．
(3) それらに対して SIMD 命令の適用可能性を検討し，適用可能なものを残す．
(4) 適用可能なものに対してアセンブリ言語による SIMD 命令を活用したコーディングを行い，コンパイラが生成する通常命令（スカラ命令）のコードとの実行時間の比較を行い，高速化できたものを採択する．

```
typedef struct array_t{
  unsigned int cdt, col, pos, neg;
} array;
...
  int lsb = (-r) & r;
  a[h+1].cdt = ~(       ~r | lsb);
  a[h+1].col = ~(~a[h].col | lsb);
  a[h+1].pos =  ( a[h].pos | lsb) << 1;
  a[h+1].neg =  ( a[h].neg | lsb) >> 1;
  r = a[h+1].col & ~(a[h+1].pos | a[h+1].neg);
  h++;
```

図 6 N-queens 問題の繰返しによる解
Fig. 6 A solution with iteration for the N-queens problem.

これらの作業は，主に IA-32 と SSE2 の計算機の上で行ったが，一部 Power Mac G5 も用いた．gprof では抽出可能なホットスポットは関数単位なので，人手によるソースコードの解析で該当する実行部位を抽出した．

MediaBench [14] を構成するプログラム群も調査の対象としたが，多くのホットスポットを (3) により候補から外した．たとえば，暗号化や復号化プログラムの中には，一見したところ SIMD 命令の効果的な適用が可能であるように思われるものがあった．しかし，256 エントリの小規模な表引きの並列実行などのように，現状の SIMD 命令セットでは処理できない演算を含んでおり，やはり候補から外した．ファイル圧縮・伸長のプログラムも同様であった．

プロセッサの実装方法によって (4) で採択の可否が異なる場合があったが，その場合は採択とした．たとえば IA-32 の Northwood や Banias，Prescott と呼ばれる実装はいずれも SSE2 命令セットを実装している．Northwood では SIMD 命令の演算器には倍クロック演算器を用いていないが，後の 2 者はそれを用いており，SIMD 命令を適用したコードの実行性能がピークで 1.8 倍程度まで高まっている．この場合は，プロセッサを変えて通常命令のコードよりも高速化できたものがあれば，例題として採択した．

(4) に関連して，SIMD 命令を適切に活用しても通常命令のコードに比べて高速化できなかった例として，N-queens 問題の繰返しによる求解[15] を紹介する．そのループカーネルを SIMD 向きに変形したものを**図 6** に示す．このプログラムは，フィールドごとに独立にシフト量を決めることができる Altivec で効果的に処理できると思われ，Power Mac G5 で SIMD 命令を使ったコーディングを数通り試したが，最も実行効率が高いものでも，通常命令のコードより 5%程度遅くなってしまった．これは，構造体の 4 つのメンバへの

```
#define ABS(X) (((X)>0)?(X):-(X))
int func0(unsigned char *a, int sum, int sz){
  int i;  unsigned char v;
  for(i=0; i<SIZE; i++, a++){
    v = *a;
    /* 下記 (a)〜(e) のうちの 1 つ */ }
  return sum; }

(a) sum += (v < 128) ? v : 256 - v;
(b) sum += 128 - abs(128-v);
(c) sum += 128 - ABS(128-v);
(d) sum += (v < 128) ? v : (unsigned char)(-v);
(e) sum += (v < 128) ? v : (unsigned char)(~v + 1);
```

図 7 画像フォーマット変換プログラムからの例題（sad）
Fig. 7 An example code from an image-format transformer (sad).

代入には並列性があるので，通常命令でも命令レベル並列性が十分に発揮されているためだと思われる．

3.2 例題の変形

こうして選ばれた処理パターンに対して，同じ結果をもたらし，SIMD 命令の適用を可能とする方向への変形を行った．それを例によって説明する．

画像フォーマット変換プログラム bmp2png [16] から，**図 7** に示すような処理パターンを得た．図のコメントの箇所には (a)〜(e) のコードのいずれかが埋め込まれる．いずれも通常の C コンパイラでは同じ結果をもたらす．作業用変数 v に代入しているのは，多くの処理系ではポインタで参照された変数はレジスタへの割付けを行わないからである．オリジナルでは (a) のコーディングが使われていたが，このコードに対してそのまま SIMD 命令を適用すると，定数の 256 に対する汎整数拡張のために 9 ビット以上で演算を行うことになる．すると，文献 2), 3) のような解析を行っても，一般的な SIMD 命令の仕様では演算のデータサイズが 16 ビットになってしまい，並列度が低下したりサイズ変換のオーバヘッドがともなったりするので，SIMD 命令向きのコーディングではない．(b) から (e) のコーディングは 8 ビットの演算で済むように改良したものである．(b) では絶対値を求める関数 abs() を呼び出しているが，多くの処理系ではコンパイラが組み込み関数として認識し，関数呼び出しを行わないコードを生成する．ここでは，SIMD 最適化でもこれを期待し，命令セットに依存した最適なコード生成を期待する．(c) はコンパイラがそのような認識を行わない場合に，有効であると思われるコーディングである．(d) と (e) は 256-v の結果が 8 ビット目以上切り落としではラップアラウンドにより 0-v と同じ結果になることを利用したコーディングである．

図 8 の例は，MPEG4 動画圧縮プログラムから選

```
/* 割り算を掛け算とシフトで行うための表 */
const unsigned int multipliers[32] ={
  0,32768,16385,10923,8193,6554,5462,4682,
  4097,3641,3277,2979,2731,2521,2341,2185,
  2049,1928,1821,1725,1639,1561,1490,1425,
  1366,1311,1261,1214,1171,1130,1093,1058 };

unsigned int quant5  (
  int16_t *  coeff,
  const int16_t *  data,
  const unsigned int  quant )
{ const unsigned int mult = multipliers[quant];
  const unsigned short quant_m_2 = quant << 1;
  const unsigned short quant_d_2 = quant >> 1;
  int sum = 0;  unsigned int i;
  for (i = 0; i < M; i++) {
    int16_t acLevel = data[i];
    if (acLevel < 0) {
      acLevel = (-acLevel) - quant_d_2;
      if (acLevel < quant_m_2) {
        coeff[i] = 0;
        continue; }
      acLevel = (acLevel * mult) >> 16;
      sum += acLevel;    // sum += |acLevel|
      coeff[i] = -acLevel;
    } else {
      acLevel -= quant_d_2;
      if (acLevel < quant_m_2) {
        coeff[i] = 0;
        continue; }
      acLevel = (acLevel * mult) >> 16;
      sum += acLevel;
      coeff[i] = acLevel; } }
  return sum; }
```

(a) オリジナル

```
const unsigned short multipliers[32] ={
  0,32767, .... }; // (b)(c) に共通

int16_t acLevel, acLevel2;
acLevel = ((data[i] < 0) ? -data[i] : data[i])
          - quant_d_2;
acLevel2 = (acLevel * mult) >> SCALEBITS;
sum += ((acLevel < quant_m_2) ? 0 : acLevel2);
coeff[i] = ((acLevel < quant_m_2)
  ? 0
  : ((data[i] < 0) ? -acLevel2 : acLevel2));
```

(b) for 文の中の SIMD 命令向き変形 1

```
int16_t acMsk1, acMsk2, acLevel;
acMsk1 = (data[i] < 0) ? -1 : 0;
acLevel = ((data[i] & ~acMsk1)|((-data[i]) & acMsk1))
          - quant_d_2;
acMsk2 = (acLevel < quant_m_2) ? -1 : 0;
acLevel = (acLevel * mult) >> SCALEBITS;
sum += ~acMsk2 & acLevel;
coeff[i] = ~acMsk2 & (((-acLevel) & acMsk1)
           | (acLevel & (~acMsk1)));
```

(c) for 文の中の SIMD 命令向き変形 2

図 8 動画像圧縮プログラムからの例題（quant5）
Fig. 8 An example code from a motion picture compressor (quant5).

```
int func0(unsigned char *a, int sum, int sz){
  int i;  unsigned char v;
  int s;
  for(i=0, s=0; i<SIZE-15; i+=16, a+=16){
    s += abs(128-a[0]);  s += abs(128-a[1]);
    s += abs(128-a[2]);  s += abs(128-a[3]);
    ....
    s += abs(128-a[14]); s += abs(128-a[15]); }
  sum += 2048 - s;
  for(; i<SIZE; i++, a++){
    v = *a;
    sum += 128 - abs(128-*a); }
  return sum; }
```

図 9 図 7 の例題（sad）の特殊なリダクション命令向き変形
Fig. 9 A transformation of the example code in Fig. 7 (sad) aimed to particluar reduction instruction.

んだものの 1 つである．オリジナルでは (a) のように，`multipliers[]` が `int` として宣言されているため，多くの SIMD 命令セットの乗算命令では効率的に処理できない．また，for ループ内の処理が SIMD 最適化の対象になるが，2 重の入れ子になった if 文の平坦化をともなう if 変換，共通処理の括り出しと段階的に SIMD 命令の適用向きに変形し，最終的に図 8 の (b) や (c) のコーディングに帰着させる．

このようにして，抽出した例題に対して，複数の変形パターンを作成していく．

3.3 ループ展開

こうして SIMD 命令向きに何通りかの変形を作られた例題のそれぞれに対して，ループを展開しない形と展開した形を作成する．多くの場合はループの形で記述されているので，幾通りかのループ展開を行ったが，最初から展開された形で記述されていたものもあったので，この場合は逆にループの形に変形した．これによって，処理系が SIMD 並列化をベクトル化に基づいて行っているか，あるいは同型命令の認識に基づいて行っているか，あるいは両者を行っているかを判断できる．展開形では，ベクタレジスタ長は 128 ビットであるとし，最適な処理データサイズ[2] を選択しているとして展開数を決めた．

また，図 7 の例の `(b)` や `(c)` のコーディングと，2.3 節の特殊なリダクション命令に関連して，**図 9** のように展開したループ内では 128 と `v` の差の絶対値の総和を求めて，その後に 128 を展開数分足し合わせた値から総和を引くようにコーディングすると，展開したループ内にリダクション命令をそのまま適用可能になる．適用可能ならばこのような変形もループの展開と同時に行う．図 9 の変形の場合は，`s` に差の絶対値を積算していくのに，複数の代入文に分けた記述と，1 つの式にまとめる記述が可能である．ベンチマークでは，両者について試すようになっている．

3.4 誤ったコード生成の検出

SIMD 命令に関連して，コンパイラが誤ったコード生成を行う可能性のあるパターンは次のとおりである．

(1) ベクタレジスタのサイズのアラインメントに合っていないポインタを使ったベクタレジスタとメモリの間の誤った転送
(2) 書き込みのベクタと読み出しのベクタに重なりがある場合の SIMD 命令の誤った適用

(1) は IA-32 のような完全なバイトマシンでは問題にならないが，バイトマシンでない Altivec や EmotionEngine 等では問題となる．実際のアプリケーションプログラムを分析すると，ほとんどの場合で処理データサイズの倍数にポインタが設定されるように記述されているので，この項目の検査が必要である．そこで，SIMD 命令を適用されることが想定される関数の呼び出しで渡すポインタのアラインメントを，ポインタが指すオブジェクトのサイズの倍数でずらし，同じパラメータを渡された通常命令のコードの実行結果と比較することによって，この項目の検査とした．

(2) の例としては，図 5 で `v2` が `v1` の 1 エントリ後を追いかけていき，`v2` から読めるデータは `v1` を通して書かれた値である場合があげられる．この場合に SIMD 命令を適用すると，通常命令で 1 つずつ処理する場合と異なる結果になる．これに対するコンパイラによる対策としては，関数の先頭で，すべての書き込みベクタのポインタと読み出しベクタのポインタの組合せで，両者の距離がベクタレジスタのサイズよりも大きいことを検査し，接近しているものがあれば，通常命令のコードにスイッチするという仕組みが考えられる．コンパイラがこのような対策を施しているかどうかを検出するために，わざと接近したポインタを関数に渡し，比較参照用のコードの実行結果と比較することによって，この項目の検査とした．

4. 実装と実験

4.1 実　　装

現在のベンチマークの実装で取り上げている例題は，以下のとおりである．6 データタイプとは，符号つき/符号なしの `int` と `short` と `char` の組合せを意味する．3 データタイプとは，符号つきの `int` と `short` と `char` の組合せを意味する．コンパイラが 2.1 節で述べた最適な処理データサイズの選択を行うかどうかは，すべての例題に共通の課題である．

ave （6 データタイプ × 4 変形）2 つの配列の対応する要素の平均を求め，別の配列に書き出す問題．

通常の足してから2で割るアルゴリズム（符号拡張が必要）と，図5のような2で割ってから足すアルゴリズム（符号拡張は不要）を用意した．

add （3データタイプ × 2変形）2つの配列の対応する要素の和を求め，別の配列に書き出す問題．結果がラップアラウンドする加算と，符号つきと符号なしのそれぞれで飽和する加算の3つの場合を，要素のサイズを8，16，32ビットと変えてテストする．

max （6データタイプ × 2変形）2つの配列の対応する要素の大きい方の値を求め，別の配列に書き出す問題．要素のサイズ8，16，32ビットについてそれぞれ符号つき/符号なしと変えてテストする．

maxc （6データタイプ × 6変形）2つの配列の対応する要素を比較し，片方の配列について大きい値を持つ要素の数を調べる問題．比較命令の結果を数値として使う能力や，特殊なリダクション命令を適用する能力を試す．

sum （6データタイプ × 3変形）配列の総和を求める問題．要素のサイズ8，16，32ビットについてそれぞれ符号つき/符号なしと変えてテストする．

quant5 （6変形）図8の例題．オリジナルと図8の2つの変形をテストする．

sad （37変形）図7の例題．これには，特殊なリダクション命令の生成を促す変形が含まれている．

bsad （37変形）同じ出典に含まれていたsadの派生問題．**図10**に示すようにループの途中`sum`の値が定数値`BREAKPOINT`よりも大きくなったらループを脱出する．

実験に使用したのは，EPSON EDiCube S160（Pen-tiumM 1.3 GHz，機種1とする）と，ASUS P4P800とPentium4 HT 2.6 GHzの組合せの計算機（機種2とする）に，それぞれオペレーティングシステムとしてLinux（kernel ver. 2.4.27）を搭載したものである．コンパイラとSIMD命令生成や基本命令生成の指示に用いたオプションの組合せは**表1**のとおりである．

4.2 結　　果

実験で用いたgccのバージョンにはベクトル化機能が備わっているので，SIMD命令を活用したコードの生成を期待したが，本ベンチマークのような配列を関数のパラメータで渡す形のコーディングでは，この機能は有効にならなかった．さらにベンチマークを書き換えて，グローバル変数に直接アクセスするように変更してみたが，if変換が必要な箇所や簡単なシフト演

```
(1) ループのままで，一時変数を使う
  for (i=0;i<SIZE;i++,a++) {
    v = *a; sum += 128 - abs(128 - v) }
(2) ループのままで，ポインタで直接参照
  for (i = 0; i < SIZE; i++, a++)
    sum += 128 - abs(128 - *a)
(3) ループを展開し，一時変数を使う
  for(i=0,p=a;i<(SIZE-(SIZE&15));i+=16,p+=16){
    v = *p;       sum+= 128 - abs(128-v);
    v = *(p+1);   sum+= 128 - abs(128-v);
    ....
    v = *(p+15); sum+= 128 - abs(128-v);}
  (1) のループ /* 端数処理 */
(4) ループを展開し，ポインタで直接参照
  for(i=0,p=a;i<(SIZE-(SIZE&15));i+=16,p+=16){
    sum += 128 - abs(128-*p);
    sum += 128 - abs(128-*(p+1));
    ....
    sum += 128 - abs(128-*(p+15));
  (2) のループ /* 端数処理 以下では省略 */
(5) (4) をひとつの式にまとめる
  for(i=0,p=a;i<(SIZE-(SIZE&15));i+=16,p+=16){
    sum+= 128 - abs(128-*p) + 128 - abs(128-*(p+1))
    ....
(6) (4) の 128 の 16 回の加算をまとめる
  for(i=0,p=a;i<(SIZE-(SIZE&15));i+=16,p+=16){
    sum -= abs(128-*p);
    sum -= abs(128-*(p+1));
    ....
    sum += 2048;}
(7) (5) の 128 をまとめる
  for(i=0,p=a;i<(SIZE-(SIZE&15));i+=16,p+=16){
    sum -= abs(128-*p) + abs(128-*(p+1))
    ....
    sum += 2048;}
(8) (6) で別の集積変数を使う
  for(i=0,p=a;i<(SIZE-(SIZE&15));i+=16,p+=16){
    tmpsum += abs(128-*p);
    tmpsum += abs(128-*(p+1));
    ....
    sum += 2048-tmpsum; tmpsum = 0;}
(9) (8) の tmpsum の初期化位置を変更
  for(i=0,p=a;i<(SIZE-(SIZE&15));i+=16,p+=16,tmpsum=0){
    tmpsum += abs(128-*p);
    tmpsum += abs(128-*(p+1));
    ....
    sum += 2048-tmpsum;}
(10) (7) で別の集積変数を使う
  for(i=0,p=a;i<(SIZE-(SIZE&15));i+=16,p+=16){
    tmpsum = abs(128-*p) + abs(128-*(p+1))
    ....
    sum += 2048-tmpsum; tmpsum=0;}
(11) (10) の tmpsum の初期化位置を変更
  for(i=0,p=a;i<(SIZE-(SIZE&15));i+=16,p+=16,tmpsum=0){
    tmpsum = abs(128-*p) + abs(128-*(p+1))
    ....
    sum += 2048-tmpsum;}
```

図10 図7の例題（sad）の演算パターン (c) についてのループの変形

Fig. 10 Transformations for the operation pattern (c) of the example in Fig. 7 (sad).

算の翻訳でコンパイラが異常終了してしまった．

icc は多くの例題の SIMD 向け変形版に対して，SIMD 命令を生成していた．**表 2** に機種 1 での図 7 の例題（sad）について，図 10 に示すようなループ変形（(1)〜(11)）と演算の変形（(a)〜(e)）の組合せのそれぞれに対する機種 1 での実行時間を，SIMD 命令を生成した場合（斜線の左側）と基本命令のみの場合（斜線の右側）について示す．

また，**図 11** の例題（bsad）の実行時間を，**表 3** に示す．この例題のループ展開版では，脱出条件を満たした際の巻き戻し（ループを 1 つ前の状態に戻す）処理を含んでいる．

これらのすべてについて，SIMD 命令の生成を指示した場合には，「ベクトライズした」という趣旨のメッセージをコンパイラが表示していた．これら結果についての考察は，4.3 節で行う．

表 4 に図 8 の例題（動画像圧縮）の結果を示す．この問題では，変形 2 をループ展開した場合にのみ SIMD

表 1 実験に用いたコンパイラとオプション
Table 1 Compiler and options used in the experimentation.

コンパイラ	SIMD 命令生成時	基本命令生成時
Intel icc ver. 8.1	-axW -O3	-O3
gcc ver. 4.0.0	-O6 -ftree-vectorize -msse2	-O6

```
int func0(unsigned char *a, int sum, int sz){
  int i;  unsigned char v;
  for(i=0; i<SIZE; i++, a++){
    v = *a;
    sum += (v < 128) ? v : 256 - v;
    if (sum > BREAKPOINT) break; }
  return sum; }
```

図 11 図 7 の例題（sad）の派生版
Fig. 11 A derivative of the example in Fig.7 (sad).

表 2 図 7 の例題（sad）の実行時間（msec）
Table 2 Execution times for the example in Fig.7 (sad).

ループ	演算の変形				
の変形	(a)	(b)	(c)	(d)	(e)
(1)	8600/15230	4380/ 6280	4390/ 6280	20040/14790	10880/15440
(2)	8590/15410	4390/ 6350	4390/ 6350	19990/15260	10880/15140
(3)	5500/13290	4390/ 5710	4390/ 5700	13470/13490	5490/13480
(4)	5490/13260	4400/ 5720	4400/ 5720	13440/13460	5490/13290
(5)	6120/13750	5590/ 6010	5590/ 6010	13750/13750	6180/13900
(6)		5610/ 4880	5610/ 4890		
(7)		5610/ 5740	5610/ 5740		
(8)		5520/ 4990	5510/ 4990		
(9)		5530/ 5000	5530/ 5000		
(10)		5520/ 6100	5510/ 6100		
(11)		5520/ 6100	5520/ 6090		

（SIMD 命令の生成を指示時の実行時間/通常命令の生成を指示時の実行時間）
ループの変形は図 10 中の番号に対応．
空欄の箇所は，有効な該当する変形がないことを意味する．

表 3 図 11 の例題（bsad）の実行時間（msec）
Table 3 Execution times for the example in Fig. 11 (bsad).

ループ	演算の変形				
の変形	(a)	(b)	(c)	(d)	(e)
(1)	8420/14840	8300/ 8320	8290/ 8320	14280/14550	8420/15280
(2)	8090/14840	8270/ 8310	8270/ 8310	14640/14970	8080/15260
(3)	4450/10350	4600/ 4610	4600/ 4610	10450/10530	4470/10390
(4)	4460/10300	4100/ 4630	4100/ 4630	10340/10530	4440/10200
(5)	4950/10820	4540/ 4970	4540/ 4970	11020/10890	5120/11130
(6)		4580/ 3930	4590/ 3930		
(7)		4590/ 4910	4590/ 4910		
(8)		4600/ 4020	4600/ 4010		
(9)		1130/ 4010	1130/ 4020		
(10)		4590/ 4980	4590/ 4980		
(11)		1130/ 4980	1130/ 4980		

詳細は表 2 に同じ

表 4 図 8 の例題（quant5）の実行時間（msec）
Table 4 Execution times for the example in Fig. 8 (quant5).

	原型		変形 2 をループ展開		高速化比
	SIMD	基本 (a)	SIMD(b)	基本	(a/b)
機種 1（PentiumM）	820	840	410	900	2.05
機種 2（Pentium4）	420	400	460	629	0.87

命令が生成された．また，機種 1（PentiumM，実装は Banias）では SIMD 命令の適用により高速化しているのに対し，機種 2（Pentium4，実装は Northwood）では逆に低速になってしまった．これはプロセッサの実装の違いによるものと思われる．

3.4 節で示した誤ったコード生成に関するテスト項目について議論する．使用したコンパイラはここにあげたテスト項目をクリアするコード生成を行っていた．また，IA-32 はバイトマシンなので任意のデータサイズに対して任意のアドレス境界からの参照を許しており，メモリ参照時にアラインメントなしのメモリ参照（movdqu）命令を生成している限りは，(1) のテストを通過する．そこでアセンブリ出力を修正して，アラインメントつき（movdqa）に変更して実行すると，結果の間違いを報告した．また生成されたアセンブリ出力を検討したところ，icc は関数の先頭でベクタの重なりの動的な検査を行っており，(2) のテストも通過した．そこで，アセンブリ出力を修正して，検査のコードを実行しないようにすると，結果の間違いを報告した．よって本ベンチマークは，3.4 節にあげた検査を想定される範囲内で行う能力を有する．

4.3 本ベンチマークの利用法

コンパイラユーザの立場で，このベンチマークが示す結果の活用法を，表 2 や表 3 から得られる情報を例として説明する．

ユーザが記述しようとしている処理パターンがこれらの例題にあれば，原則として，表の中から SIMD 命令を適用して最も処理時間が短いものを採用すればよい．ただし，次に述べるように類似の処理パターンの結果も参考にすべきである．

表 3 で，実行時間が 1,000 msec 台のパターンについてアセンブリコードを検討したところ，psadbw（バイトデータ間の差の絶対値の，ベクタレジスタ内の総和を求める）命令というリダクション命令を生成していた．ただし，表の前後の対応するパターンを見比べると，この命令へのマッチングは非常に狭く，コンパイラ内では何かの特定処理向けの大きなパターンへのマッチングを行っていると推察される．図 7 の例題は，図 11 の例題の「脱出がない」という特殊な場合であると考えられるので，脱出の閾値 `BREAKPOINT` を十分大きい値に設定して，図 11 のような記述にする方が，psadbw 命令にマッチして，処理を大幅に高速化できることが分かる．このように類似の処理パターンの結果も見ていくと，より高い実行性能を得られるパターンを見出せる場合がある．

目的の処理パターンそのものがベンチマークにない場合は，ベンチマーク中から目的の処理パターンに近い例題を探し，その例題の結果を検討して目的の処理を記述していく．たとえば，1 つのベクタから（他への副作用なしで）スカラ値を得る例として，図 7 と図 11 の例題が選ばれたとする．

ループを展開すべきかどうかという検討課題に対しては，展開したコードの方が総じて実行時間が短くなっているので，展開する．

また，エイリアス解析の困難さからポインタで参照されるメモリのレジスタ割付けを行わないコンパイラでは，「一時変数の利用」は有効である．しかし，icc の場合はほとんど影響がないように思われるので，このような工夫は記述性を増すものでない限りは行わなくてよいことが分かる．

表 2 の (b) 欄や (c) 欄を見ると，途中でループを脱出しない場合には，`sum` の増分をまとめるのに定数部分をまとめあげてあとで足すような記述よりも，素直なものの方が高速であることが分かる．反対に，ループ脱出がある場合には，表 3 の (b) 欄や (c) 欄を見ると，そのような記述をしても遅くならないか，非常に高速化する場合があることが分かる．

また，SIMD 命令の中で負の値を求める演算については，表の (d) 欄と (e) 欄を検討すると，意外なことに「1 の補数に 1 を足す」式の記述の方が高速であることも分かる．

以上はあくまでもベンチマークが与える情報の一部に対する検討にすぎない．ベンチマークを通して，このような情報をあらかじめ参考にできれば，特定コンパイラとプロセッサに対する実行速度の向上のチューニングが容易かつスピーディになると思われる．

5. 関 連 研 究

ここではまず，既存のベンチマークと SIMD 最適化の関連について議論し，さらにベンチマーク例題に

関連する成果を紹介する．

3 章で述べたように，多くのアプリケーションやベンチマークのプログラムにおいて，そのままで SIMD 命令の活用による高速化可能な部位は比較的限定的で，SIMD 命令を適用可能な部位をプログラムごとに解析しなければならない．

SPEC ベンチマーク[17] は，コンパイラやハードウェアの性能評価の標準として広く用いられ，調査項目も整数演算や浮動小数点演算から Web サーバに至るまでの多岐にわたっている．しかし，本論文で話題にしている SIMD 命令セットに関連したベンチマークは特には用意されておらず，最も関連がある SPEC CPU でも，実働のアプリケーションをそのまま例題にしているため，設計要件にあげた情報をユーザに与えない．

MediaBench[14] は，例題の出典をメディア処理やファイル圧縮等のプログラムに絞ったベンチマークである．例題採取の方向性から SPEC に比べると SIMD 命令を活用できる可能性は高いが，SPEC と同様の構成方法であるため，そのままでは設計要件を満足する情報を与えない．

DSPStone[18] は，DSP による処理対象に的を絞ったベンチマークで，本論文であげている設計要件のいくつかを満たしているが，対象命令セットが SIMD 命令と少し異なる DSP 命令である点と，3.2 節や 3.3 節の事項に多様性がなく，3.4 節の事項のための考慮がない．

NpBench[19] や NetBench[20] 等のネットワーク処理を対象としたベンチマークは，命令レベル並列性やスレッドレベル並列性が要求される例題からなっており，SIMD 命令が対象とするデータ並列性を有する例題には主眼が置かれていない．

Dhrystone[21] は，コンパイラと計算機の性能の総合的な評価を目的としたベンチマークで，さまざまな言語で実現されている．その構成方法は，本論文のベンチマークと似ているが，対象が通常命令セットやオペレーティングシステムの入出力処理の性能であり，SIMD 命令向け最適化のテストにはならない．

Windows 系オペレーティングシステムについては，数多くのベンチマークが公表されているが，それらはソースレベルで提供されていないので，コンパイラの調整やユーザのコーディングパターンの選択には利用できない．

暗号やエラー訂正で用いられるガロア体の算術を SIMD 命令で効率良く扱う手法[22] が公表されている．この成果がプログラムとして公表されれば，3.1 節で述べた「候補から外された例題」を，ベンチマークに加えられるようになると考えられる．

6. 結　　論

コンパイラによるメディア処理向け SIMD 拡張命令セットの活用に主眼を置いたベンチマークプログラムの構成法を示し，それに基づく実装と評価を行い，提案する方式の有効性を確認した．

なお，本方式のベンチマークは，MediaBench などの既存のベンチマークを置き換えるものではなく，相補的関係にあるものである．

実装したベンチマークは，COINS プロジェクト[11] の配布物の一部として公開している．我々は，ユーザからの意見を採り入れ，利用形態の変化やハードウェア技術，コンパイラ技術の進化に合わせながら，ベンチマークを拡充していく予定である．

本論文では，例題の探索をアプリケーションプログラムのホットスポットに的を絞ったため，SIMD 命令の適用が可能な重要な例題でとりこぼしたものがあるかもしれない．N-queens 問題のようなボーダーライン上にあった問題は，より SIMD 向きの変形を探求していく．そして，本論文では浮動小数点の SIMD 命令を対象にはしなかったが，最近のメディア処理プログラムでは，IIR フィルタを用いるものをはじめとして，SSE2 や Altivec 等の SIMD 命令セットがサポートしている単精度浮動小数点演算命令によって高速化が可能なものが増えている．これらは今後の SIMD ベンチマーク拡充の方向を示している．

今回は，COINS コンパイラでは SIMD 最適化モジュールの調整不足で，比較検討できるコードを生成するまでには至らなかったが，今後 web 等でテスト結果を公表していきたい．

謝辞　本論文の執筆にあたり，有益なアドバイスをいただいた査読者の方々ならびに電気通信大学情報工学科の岩崎英哉教授に，心よりお礼を申し上げます．また，実験で協力をいただいた東京大学大学院新領域創成科学研究科の広松悠介氏に感謝します．なお，本研究は文部科学省科学技術振興調整費「並列化コンパイラ向け共通インフラストラクチャの研究」の補助を受けています．

参 考 文 献

1) Hwang, K.: *ADVANCED COMPUTER ARCHITECTURE*, McGraw Hill, Inc. (1993).
2) 鈴木　貢，藤波順久，福岡岳穂，渡邊　坦，中田育男：マルチメディア SIMD 命令活用のためのデータサイズ推論，情報処理学会論文誌：プロ

グラミング，Vol.45, No.SIG5(PRO21), pp.1–11 (2004).
3) Stephenson, M., Babb, J. and Amarasinghe, S.: Bitwidth Analysis with Application to Silicon Compilation, *Proc. SIGPLAN Conference on Program Language Design and Implementation* (*PLDI'00*), pp.108–120 (2000).
4) Sun microsystems: VIS Instruction Set. http://www.sun.com/processors/vis/
5) Allen, J., Kennedy, K., Porterfield, C. and Warren, J.: Conversion of Control Dependence to Data Dependence, *Proc. SIGPLAN Conference on Program Language Design and Implementation* (*PLDI*), pp.177–189 (1983).
6) Intel Corporation: IA-32 Intel(R) Architecture Software Developer's Manual. http://developer.intel.com/design/pentium4/manuals/index_new.htm
7) Bik, A.J.C., Girkar, M., Grey, P. and Tian, X.: Automatic Intra-Register Vectorization for the Intel(R) Architecture, *Int. J. Parallel Programming*, Vol.30, No.2, pp.65–98 (2002).
8) Sreraman, N. and Govindarajan, R.: A Vectorizing Compiler for Multimedia Extensions, *Int. J. Parallel Programming*, Vol.28, No.4, pp.363–400 (2000).
9) Larsen, S. and Amarasinghe, S.: Exploiting Superword Level Parallelism, *ACM SIGPLAN Notices*, Vol.35, No.5, pp.145–156 (2000).
10) 藤波順久，阿部正佳：SIMD 型拡張命令をもっと使った最適化への道のり，第 43 回プログラミング・シンポジウム報告集，pp.185–196 (2002).
11) COINS コンパイラ・インフラストラクチャ協会：並列化コンパイラ向け共通インフラストラクチャの研究. http://www.coins-project.org/
12) Free Software Foundation: GNU gprof. http://www.gnu.org/software/binutils/manual/gprof-2.9.1/gprof.html
13) Intel Corporation: Intel(R) VTune(tm) Performance Analyzer. http://www.intel.com/software/products/vtune/
14) Lee, C., Potkonjak, M. and Mangione-Smith, W. H.: MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communicatons Systems, *MICRO 30: Proc. 30th annual ACM/IEEE International Symposium on Microarchitecture*, pp.330–335, IEEE Computer Society (1997).
15) 吉瀬謙二，片桐孝洋，本多弘樹，弓場敏嗣：PC クラスタを用いた N-queens 問題の求解，電子情報通信学会論文誌レター，Vol.J87-D-I, No.12, pp.1145–1148 (2004).
16) Miyakawa, M.: bmp2png/png2bmp home page. http://hp.vector.co.jp/authors/VA010446/b2p-home/
17) Standard Performance Evaluation Corporation: SPEC benchmark. http://www.spec.org/
18) Zivojnovic, V., Martinez, J., Schlager, C. and Meyr, H.: DSPstone: A DSP-Oriented Benchmarking Methodology, *Proc. ICSPAT'94* (1994).
19) Lee, B.K. and John, L.K.: NpBench: A Benchmark Suite for Control plane and Data plane Applications for Network Processors, *Proc. International Conference on Computer Degisn* (*ICCD'03*) (2003).
20) Memik, G., Mangione-Smith, W.H. and Hu, W.: NetBench: A Benchmarking Suite for Network Processors, *Proc. 2001 IEEE/ACM International Conference on Computer-aided Design* (*ICCAD '01*), Piscataway, NJ, USA, pp.39–42, IEEE Press (2001).
21) Weicker, R.P.: Dhrystone: A Synthetic Systems Programming Benchmark, *Comm. ACM*, Vol.27, No.10, pp.1013–1030 (1984).
22) Bhaskar, R., Dubey, P.K., Kumar, V. and Rudra, A.: Efficient Galois Field Arithmetic on SIMD Architectures, *Proc. 15th annual ACM symposium on Parallel algorithms and architectures* (*SPAA '03*), pp.256–257 (2003).

(平成 17 年 4 月 28 日受付)
(平成 17 年 9 月 21 日採録)

鈴木　貢（正会員）

電気通信大学情報工学科助手．記憶管理アルゴリズム，並列/分散アルゴリズム，言語処理系等に興味を持つ．平成 14 年度本学会論文賞受賞．ACM，電子情報通信学会，日本ソフトウェア科学会各会員．

小川　大介

電気通信大学情報工学科卒業．現在（株）セイコーエプソン TFT 事業部．コンパイラ，SIMD 型命令セット等に興味を持つ．

室田 朋樹（学生会員）

電気通信大学情報工学科卒業．東京大学大学院新領域創成科学研究科修士課程在学中．計算機システムの性能評価，アドホックネットワーク，コンピュータネットワークの品質管理等に興味を持つ．電子情報通信学会，ACM，IEEE各会員．

渡邊 坦（正会員）

昭和 37 年京都大学理学部数学科卒業．日本 IBM（株），（株）日立製作所中央研究所，同社システム開発研究所，電気通信大学情報工学科教授を経て，現在電気通信大学名誉教授．工学博士．言語処理系，プログラミング言語に興味を持つ．ACM，日本ソフトウェア科学会各会員．