マルチメディア向けSIMD命令の生成手法

酒井淳嗣，枝廣正人，小長谷明彦
{sakai,eda,konagaya}@csl.cl.nec.co.jp

NEC C&C研究所
〒216 川崎市宮前区宮崎4-1-1

本稿では高級言語記述からSIMD命令を生成するコンパイル手法を提案する。マルチメディア処理の高速化をねらってSIMD命令を備えるマイクロプロセッサが登場してきたが、現状ではアセンブラでSIMD命令を記述しなければならないことが多い。本稿で述べる手法は、配列操作を行なうループを展開して並べ換えした後、SIMD命令に変換する。このSIMD化手法を中間コードレベルで実装してその有効性を確認したが、コンパイラによる自動SIMD化は困難な場合があることも分った。本稿ではSIMD処理記述に向いた言語拡張についても述べる。

A Compiler Technique with SIMD Instructions for Multi-media Processing

Junji SAKAI, Masato EDAHIRO, and Akihiko KONAGAYA

C&C Research Laboratories, NEC Corporation
4-1-1, Miyazaki, Miyamae-ku, Kawasaki, Kanagawa, 216 Japan

We propose a compiler technique to convert a program written in a high-level language such as C, into an object program that includes SIMD instructions. Some recent microprocessors have SIMD instruction sets that accelerate multi-media processing. In many cases, however, we have to write a program in assembly language to utilize such a SIMD instruction set. The method we propose in this paper enables high-level description of SIMD operations by means of unrolling the loop with array elements, rearranging the order of operations and replacing them by SIMD operations. Our prototype implementation proves its effectiveness for some multi-media programs. In this paper we also mention a language extension suitable for SIMD operations.

1　はじめに

近年のマイクロプロセッサの性能向上によって、従来専用チップを用いて行っていた画像処理や音声処理等が通常のマイクロプロセッサで行えるようになりつつある [1]。

マイクロプロセッサのデータ処理単位は 32bit から 64bit へと拡大されつつあるが、他方、画像処理 / 音声処理等のマルチメディア処理では必ずしも精度の高い演算が求められているわけではなく、むしろ短いビット幅の演算を多数並列に行う場合が多い。

最近では SunMicrosystems 社の UltraSPARC [2],[3] や Hewlett Packard 社の PA7100LC[4] など、短ビット長演算を複数同時処理する命令セットを有するマイクロプロセッサが登場しつつある。これらのプロセッサは、通常レジスタが等ビット幅の複数フィールドに分割されているとみなし、それぞれのフィールドに納められている数値に対して同一の演算を施すような命令を持っている。複数のデータに対して同一演算を施すことから、本稿ではこの種の命令を SIMD 命令と呼ぶことにする[1]。

配列処理などにおいて SIMD 命令が適用されると 2 〜 8 倍程度の高速化が可能だが、そのためには人手によるアセンブラプログラミングが必要になるのが現状である。そこで本稿では、高級言語で記述された配列要素処理から SIMD 命令を使用したオブジェクトコードへの変換 (SIMD 化と呼ぶ) を行う手法を提案する。ソースプログラム上に若干の指示を付加することで、ユーザは SIMD 命令で実行したい処理を高級言語で記述することができるようになる。

本稿の構成を以下に示す。まず第2章では本稿で用いる SIMD 命令の動作とその中間コード表記について説明する。続く第3,4,5章では、既存の C 言語の枠組を大幅に拡張しない範囲でソースプログラムを SIMD 化することを検討する。即ち第3章ではソースプログラム記述上の留意点、第4章ではコンパイラが SIMD 命令を生成する上での問題点とコンパイラによる SIMD 化の手順を説明し、第5章でその手法の実装と評価について述べる。第6章では命令セットやソース言語に対するコンパイラ側からの要望、特に SIMD 処理に向いた言語拡張について述べる。最後に第7章で、本稿のまとめと今後の課題について述べる。

2　SIMD 命令セット

この章では本稿で想定するプロセッサの SIMD 命令セットと、その中間コード表記について述べる。

[1]UltraSPARC ではこのような演算命令セットを VIS (Visual Instruction Set) と呼んでいる。

2.1　SIMD 命令

SIMD 命令セットは大きく分けて、基本 SIMD 命令、マスク付き SIMD 命令、マスク生成命令などから成る。これらを表 1 に示す。

命令	説明
基本 SIMD 命令	各フィールド毎に同一演算を施し、デスティネーションの対応フィールドに格納する。
マスク付き SIMD 命令	各フィールド毎に同一演算を施し、デスティネーションへ格納するか否かをマスクレジスタの値によってフィールド毎に選択する。
マスク生成命令	ソースレジスタの各フィールド毎に演算を行ない、その結果に応じてマスクレジスタの対応フィールドに値をセットする。
その他	各フィールドの総和演算命令、異種フォーマット間データ変換命令など。

表 1　SIMD 命令セット

演算オペランドは 64bit レジスタであり、それを 2, 4, 8 個に分割したものがフィールドである。4 つの 16bit フィールドに分割した場合の SIMD 演算命令のデータの流れを図 1 に示す。

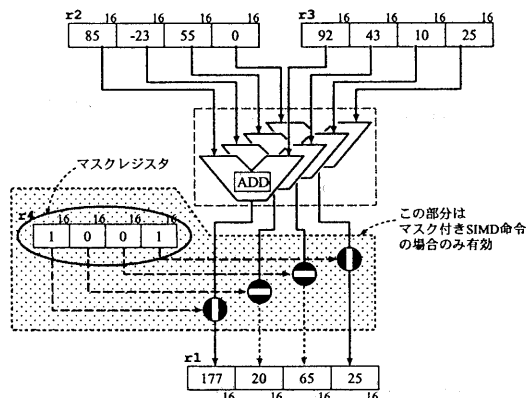

図 1　SIMD 命令におけるデータの流れ

64bit レジスタのフィールド分割としては他に 32bit × 2 や 8bit × 8 形式も考えられ、また固定小数点のみならず浮動小数点形式の SIMD 命令も考えられるが、本稿では 16bit 整数× 4 フィールドの形式のみを扱うことにする。

2.2　中間コード表記

中間コードでは個数に制限のないテンポラリレジスタ (中間項) を用いる。通常のスカラ型テンポラリレジスタは r_i、16bit × 4 などの SIMD 型データを保持するテンポラリレジスタは r_{Si} と表記する。制御構造以外の中間コードは単一の演算またはロードストアを表現する代入文であり、原則として通常命令や SIMD 命令と 1 対 1 に対応する。マスクつ

きSIMD演算を表す中間コードはマスクレジスタを条件式とするif文で、規則的なループ構造はfor文で表現する。

3 ソースプログラムの記述

SIMD命令を生成するコンパイラを構築する方法としては

1. 既存言語(C言語など)で記述されたプログラムをコンパイラが解析してSIMD化する方法。
2. 既存言語をSIMD化用に拡張し、それに従って記述されたプログラムをコンパイラがSIMD化する方法。

が考えられる。後者は第6章で述べることとし、この章では前者の方法を採用した場合の問題点について議論する。

SIMD命令は1レジスタが保持している複数のフィールドに対して同一の演算処理を施す。C言語では、この処理は基本的にはループ中にある配列要素アクセスとして表現される。また、マスク付きSIMD命令に対応する処理はif文を用いた選択的配列要素操作として表現される。しかし配列操作がコンパイラによってSIMD化されるためには、次のような点に留意してソースプログラムを記述しなければならない。

1. データサイズ
   SIMD命令が扱う個々のフィールドのサイズは8, 16, 32bitなどである。一方ユーザプログラムでは配列要素を単にint型として扱う場合も少なくない。int型で宣言されている配列要素の保持するデータが、実際には8/16/32bitで表現できる範囲に収まることをコンパイラが自動判別するのは困難であり、ユーザによる適切なサイズ型宣言[2]が必要である。
2. 配列の重なり
   SIMD化によって命令の実行順序、特に配列要素の定義参照順序が一部変化するので、2つ以上の配列の領域に重なりがあると、SIMD化によって誤った処理結果を生み出す可能性がある。コンパイラ側で高度なalias解析を行わない限り、配列間の重なりがないことをユーザに指示してもらわなくてはならない(#pragma safe)。
3. ポインタによる参照
   配列をポインタで参照する場合、上に述べた配列の重なり問題の他に、参照する要素のアラインメントが分らない、という問題がある。SIMD命令はメモリ上の任意のバイト位置の要素を参照できるわけではないので、後述するようなアラインメント調査を行わなければならない。アラインメント調査が困難な場合は何らかのユーザ指示を用いる必要がある。

[2]C言語ではshort, int, longなどに明確なビット数の取り決めがないため、言語処理系に依存した宣言とならざるを得ない。

これらの点に留意して記述したプログラム例を以下に示す。この例では#pragmaによって配列間の重なりがないことやアラインメントが正しいことを指示している。

```
short v,p1[],p2[];
#pragma safe p1[], p2[]
#pragma aligned p1[], p2[]
int   i,s;

for(i=0; i<16; ++i) {
  v=p1[i]-p2[i];
  if (v<0) v=-v;
  s=s+v;
}
```

図2 dist1() — MPEG2エンコーダより

4 コンパイラによる変換処理

SIMD化コンパイラは、前章で述べたようなスタイルで記述されたソースプログラムからSIMD命令を含むオブジェクトコードを生成するが、その際、コンパイラは以下のような点を考慮しなければならない。

1. 命令実行順序の変更
   SIMD化はベクトル化と同様、命令の実行順序を一部変更する。この命令実行順序変更を安全に行うために、データ依存関係解析を行う必要がある。
2. 配列要素のアラインメント
   SIMD命令はレジスタ上の複数フィールドに対して同時に演算処理を施すことができるが、複数フィールドをレジスタ－メモリ間で転送する場合には、その参照メモリアドレスがレジスタ長に対応するワード境界(例えば64bit境界)から開始していなければならないのが普通である(図3(a))。逆に、図3(b)のようにワード境界以外から開始する転送を行う場合は、特別な命令セットを用いたロード/ストア、あるいは通常ロード/ストアとシフト命令を組み合わせた命令を生成する必要がある。

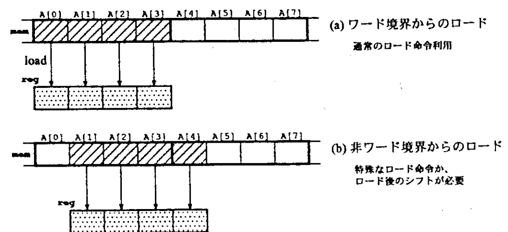

図3 配列要素のアラインメント

3. 配列要素の参照間隔(ストライド)
   ストライドが1の場合はSIMD命令をそのまま使用できる。ストライドが2の場合はマスク付きSIMD命令(図4(a))か、SIMD命令に通常の論理演算命令を組み合わせたオブジェクトコードを

生成する。これらの場合、マスクされたフィールドの不要な演算によって演算例外などを生じさせないよう配慮する。

一般にストライドがフィールド数の約数 (1, 2, 4,···) 以外の場合、この手法はそのままでは適用できない (図 4 (b))。またストライドが 4, 8 と大きくなると、SIMD 化による速度向上率が低下する (図 4 (c))。

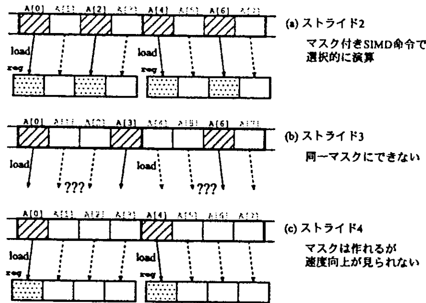

図 4　配列参照のストライド

4. プロローグ / エピローグ処理
   例えば 16bit × 4 の SIMD 命令を用いることを考える。ループ中でアクセスする配列領域の先頭が第 $4m$ 要素で末尾が第 $4n-1$ 要素でない場合は、ループ繰り返しの最初と最後は 4 要素同時処理できない (m, n: 整数)。そこで、最初と最後のアイテレーションの実行は条件つき SIMD 命令で行うようなコード生成を行う。

これらの点を考慮した SIMD 化の一手法について以下に述べる。この SIMD 化手法は中間コードレベルで逐次実行プログラムを SIMD 実行形態に変換するもので、前処理、命令並べ換え、後処理の 3 つに大きく分けられる。

4.1　前処理

SIMD 化のメイン処理が行いやすいように、以下のような処理を行う [6],[7]。

- ループの正規化 — 制御変数の値が 0 から 1 ずつ増加していくように変数を変換する。
- 帰納変数の処理 — ループアイテレーション毎に定数分だけ増加 / 減少する帰納変数を見つけ、それらを制御変数の線形式に置き換える。これは命令並べ換えを阻害する無用な依存関係を減らすためである。
- SIMD 化可否の判定 — 配列要素アクセスの添字式をもとに配列のアラインメントとアクセス時のストライドを解析し、そのアクセスパターンが SIMD 化に向いているかどうかを判定する。
- スカラ変数の配列化 — ループ本体の始めの部分で定義され、その後の本体中でのみ引用される単純変数は配列化し、アイテレーション間にまたがる無用な依存関係を除去する。

4.2　命令並べ換え

並べ換えを容易にするため、ループをいったん U 段 (ここで U は SIMD 命令のフィールド数) にアンローリングする。更に、図 5 の例に示すような 2 種類の番号 (SIMD 化順序数) を各中間コードに付与する。第 1 列目の番号 (第 1 の SIMD 化順序数) はアンローリング前の各中間コードのテキスト順を示しており、第 2 列目の番号 (第 2 の SIMD 化順序数) は何回目のアンローリングで生成された中間コードであるかを示している。

```
        for r1 :=0 to 15 step 4 begin
1 1         r2         := A[r1]
2 1         r3         := r2 + 1
3 1         B[r1]      := r3
1 2         r4         := A[r1+1]
2 2         r5         := r4 + 1
3 2         B[r1+1]    := r5
1 3         r6         := A[r1+2]
2 3         r7         := r6 + 1
3 3         B[r1+2]    := r7
1 4         r8         := A[r1+3]
2 4         r9         := r8 + 1
3 4         B[r1+3]    := r9
        end
```

図 5　SIMD 化順序数

アンローリング後、ループ本体内の各中間コード間の依存関係解析を行う。ここではループアイテレーション間にまたがった解析は不要である。そして第 1, 第 2 の SIMD 化順序数を第 1, 第 2 のソートキーとしてループ本体内の中間コード列をソートする。このとき依存関係が損なわれるような並べ換えは行わない[3]。

先の例を並べ換え後の中間コードを図 6 に示す。

```
        for r1 :=0 to 15 step 4 begin
1 1         r2         := A[r1]
1 2         r4         := A[r1+1]
1 3         r6         := A[r1+2]
1 4         r8         := A[r1+3]
2 1         r3         := r2 + 1
2 2         r5         := r4 + 1
2 3         r7         := r6 + 1
2 4         r9         := r8 + 1
3 1         B[r1]      := r3
3 2         B[r1+1]    := r5
3 3         B[r1+2]    := r7
3 4         B[r1+3]    := r9
        end
```

図 6　並べ換え後の中間コード

4.3　後処理

並べ換え後の中間コードは、SIMD 命令を用いて実行する場合の命令実行順序で並んでいる。そこで中間コード列を先頭から順に走査し、同一オペコードを持つ一連の中間コード列を 1 つの SIMD 中間コードに置き換えていく。

[3] そのような場合は本質的に SIMD 化できない。

その後以下に示すような最適化処理を行う。

- 共通部分式の削除 — 前処理で行った帰納変数の展開を元に戻すために必要である。
- ループ制御変数の整理 — ループ制御変数の値を直接参照することがない場合はそれを削除し、配列へのポインタをループ脱出判定に用いるようにする。
- 演算強度の軽減 — ループ制御変数の線形式を保持する中間項(一時レジスタ)に関して演算強度軽減処理を施す。

5 実装・評価

5.1 実装

前章で述べたSIMD化手法のうち、命令並べ換えに関する部分(ループアンローリング、SIMD化順序数付与、命令並べ換え)およびSIMD中間コードへの置換処理部を中心にSIMD化評価ツールとして実装を行った。実装したSIMD化処理部の入力および出力はいずれもテキスト形式の中間コードとした。

5.2 評価

評価には実アプリケーションプログラムの一部を用いることとし、MPEG2などのソースプログラムからSIMD化の対象となりうる部分を抜き出して手動で中間コード化し、それをSIMD化評価ツールへ入力として与えた。評価に用いたプログラムは

1. dist1 — MPEG2ソフトウェアエンコーダにおいて最も多くの実行時間を占める `dist1()` 関数(動き補償予測)中のforループの一つ(前出の図2)。if文と総和型演算を含む。
2. recon_comp — MPEG2ソフトウェアデコーダにおいて実行時間順位の上位を占める `recon_comp()` 関数(画像の再構成)中のforループの一つ(図7)。if文と非アラインメントロードを含む。
3. 行列積 — Mesaライブラリ[4]付属のデモプログラム群において、実行時間順位の上位にある `gl_transform_points()` 関数(Geometry変換)中のforループで記述されているもの。4×4 行列 M と各4次ベクトル p_i に対して $q_i \leftarrow Mp_i$ を計算する(図8)。

の3種である。

ただしこれらのプログラムは第3章で述べた点について手直ししたものである。特に行列積プログラムは、元々のコードではメモリアクセス順序がSIMD化に適さないため、二重ループの順序を入れ換えてある。

これら3つのプログラムを中間コードに変換し今回実装したSIMD化評価ツールに与えたところ、

[4] OpenGLライクな3Dグラフィックライブラリ

```
short v,s[],d[];
#pragma safe s[], d[]
#pragma aligned s[], d[]
int   i,w;

for(i=0; i<w; ++i) {
  v=d[i]+
    ((unsigned short)(s[i]+s[i+1]+1)>>1);
  if (v>=0) ++v;
  d[i]=v>>1;
}
```

図7 recon_comp() — MPEG2デコーダより

```
for(i=0; i<n; ++i) {
  for(j=0; j<4; ++j)
    q[i][j] = 0;
  for(k=0; k<4; ++k)
    for(j=0; j<4; ++j)
      q[i][j]+=m[k][j]*p[i][k];
}
```

図8 行列積 — Mesaライブラリより

いずれも配列操作部分がSIMD命令を用いる中間コードに変換された。図7のプログラムから得られたSIMD化中間コードの主要部を図9に示す。

r_{S19} := [4, 4, 4, 4]
r_{S20} := [0, 1, 2, 3]
r_{S21} := [1, 2, 3, 4]
r_{S10} := [1, 1, 1, 1]
r_{S12} := [2, 2, 2, 2]
r_{S14} := [0, 0, 0, 0]
for r_1 := 0 to 15 step 4 begin
  r_{S4} := D[r_{S20}]
  r_{S5} := S[r_{S20}]
  r_{S8} := S[r_{S21}]
  r_{S9} := $r_{S5} + r_{S8}$
  r_{S11} := $r_{S9} + r_{S10}$
  r_{S13} := r_{S11} / r_{S12}
  r_{S15} := $r_{S13} \geq r_{S14}$
  if r_{S15} r_{S16} := [1, 1, 1, 1]
  if r_{S15} r_{S13} := $r_{S13} + r_{S16}$
  r_{S17} := [2, 2, 2, 2]
  r_{S18} := r_{S13} / r_{S17}
  D[r_{S20}] := r_{S18}
  r_{S20} := $r_{S20} + r_{S19}$
  r_{S21} := $r_{S21} + r_{S19}$
end

図9 recon_compの中間コード

6 コンパイラ側からの提案

前章まででコンパイラによる自動SIMD化について述べたが、現状ではまだかなり制約があると言わざるを得ない。この章では、命令セットアーキテクチャやプログラミング言語に対するコンパイラ側からの要望などについて述べる。

6.1 命令セット

SIMD演算では8bit, 16bit, 32bitなどの短いビット長のデータを扱うが、最近のマイクロプロセッサの多くは短ビット長データの扱いが得意ではなく、プログラムの一部しかSIMD化できない場合にペナルティが課されている。各種ビット長のデータフォー

マット間のフォーマット変換命令の整備が望まれる。

また実行効率化のためにはアラインメントに沿ったメモリアクセスが求められるが、アラインメントに沿っていない場合にも少ないコストでアクセスできるメモリ操作命令があれば、コンパイラによる解析困難な場合にもSIMD化が適用できる可能性が高まると考えられる。

6.2 プログラミング言語

信号処理プロセッサ(DSP)向けには、DSP言語やデータフロー言語、関数型言語など、従来から様々な専用言語が提案されている。しかしDSP機能としてSIMD演算処理部を内蔵したマイクロプロセッサの場合、通常のCPU部分の動作をDSP向き専用言語で記述するのは困難である。

C言語はシステム記述からアプリケーション記述まで幅広く用いられており、詳細な動作記述が可能である。その反面、汎用的なC言語にはSIMD演算に対する枠組はなく、信号処理動作をCで記述するとまわりくどい記述になりがちである。

そこで、Cのような手続き型言語に対して、容易にSIMD演算を記述できるような拡張を施すことを考える。以下に拡張案を示す。

- マクロな操作
  各アイテレーションの実行順序を問わないループ構造(doall,foreachのような構造)を導入する。配列要素に対する順序を問わない処理はこのようなループで記述するのが自然である。また$\sum, \prod$演算などを容易に記述できる仕組みを組み込むと、行列演算などがより見やすく表記できる。
  これらの拡張によりコンパイラはユーザの記述したアルゴリズムをより容易に認識できるようになり、高いレベルでのプログラム変換を行なうことがより容易になると考えられる。
- ミクロな操作
  16bit × 4や32bit × 2といったデータの集合をひとまとめにしたSIMD型ともいうべきデータ型を導入する。SIMD型変数に対して通常の演算子を適用すればSIMD演算を記述したことになり、異なるSIMD型変数の間、あるいはSIMD型変数と整数型変数の間で相互に代入するなどして型変換させることで、SIMDデータ間のフォーマット変換を記述する。
  これらの拡張により、ユーザはきめ細かい演算処理をスマートに記述することができ、コンパイラはそれに基づいた効率的なコード生成を行なえる。
- その他
  部分配列やポインタの指している要素が正しいアラインメントに沿っているか否かを明示できる仕組みを用意する。例えば関数宣言部で

```
int func(aligned short *ary)
```

のようにアラインメントに沿った配列が渡されることを宣言できるようにする。
また、信号処理でしばしば用いられる飽和演算を記述するための枠組みも求められる。実現容易な方法の一つとしては、飽和させるための組み込み関数を用意することが考えられる。

7 おわりに

高級言語で記述されたソースプログラムをSIMD命令を含むオブジェクトコードへ変換する方法について述べた。プログラム中の配列操作部分をSIMD化する部分については簡単な実装を行ってその有効性を確認することができた。中間言語レベルでのSIMD化処理は、現在、アンローリング ⇒ 並べ換え ⇒SIMD命令へ置換 というステップを踏んでいるが、アンローリング処理を経ず直接SIMD命令へ変換する方法も検討すべきである。

コンパイラによるSIMD化には現段階では若干のユーザ指示の助けが必要である。本稿では自動SIMD化のために望まれる命令セットや言語仕様についても述べた。言語に関しては、従来の言語の枠組にはなかったSIMD演算などを表記できるミクロな拡張とともに、繰り返し処理などをコンパクトに記述できるマクロな拡張も必要であろう。

今後はSIMD化手法や言語拡張について検討を進めるとともに、SIMD化処理部の実装および評価についてもより詳細に行なっていく予定である。

謝辞 本研究の遂行にあたって有益な助言をいただいたNEC C&C研究所, マイクロエレクトロニクス研究所の方々に感謝します。

参考文献

[1] P. Lapsley, "NSP Shows Promise on Pentium, PowerPC," Microprocessor Report, May 8, 1995

[2] L. Kohn, et.al., "Out-of-order実行機能を省いて動作周波数を上げたUltraSPARC", 日経エレクトロニクス, Jan 30, 1995

[3] L. Gwennap, "UltraSparc Adds Multimedia Instructions," Microprocessor Report, Dec 5, 1994

[4] R. B. Lee, "Accelerating Multimedia with Enhanced Microprocessors," *IEEE Micro*, pp22-32, Apr 1995

[5] Microprocessor Forum 資料, Oct 1995

[6] A.V.Aho, R.Sethi, J.D.Ullman 原著, 原田賢一訳: "コンパイラ I,II," サイエンス社, 1990

[7] H.Zima, B.Chapman 原著, 村岡洋一訳: "スーパーコンパイラ" ("Supercompilers for Parallel and Vector Computers"), オーム社, 1995