社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2008－HPC－116 (33)
2008／8／7

配列処理言語におけるSIMD化向けプログラム変換

城田祐介† 瀬川淳一† 金井達徳†

マルチコアプロセッサのアーキテクチャが多様化するに従い、マルチコアプログラムの生産性が大きな課題となってきている。この課題に対して、我々は、マルチコアプロセッサ上での並列処理に向いている配列処理に特化したアルゴリズム記述言語を提案している。本配列処理言語は、プログラムの記述の抽象度をアルゴリズムレベルに上げることで、特定のプロセッサアーキテクチャに依存しないマルチコアプログラムの開発を実現する。本配列処理言語の処理系は、本言語で記述されたプログラムを各ターゲットアーキテクチャに最適化したC++プログラムに変換するが、この際に処理系が最適化用のアルゴリズムレベルの情報を抽出しやすいように本言語は設計されている。本稿では、最適化の一つとして、変換後のC++プログラムでターゲットアーキテクチャのSIMD命令を利用することを検討する。SIMD命令の生成に、C++コンパイラの自動SIMD化機能を利用することを想定しているが、C++コンパイラが自動SIMD化できる処理は限定的であり、効率的なSIMD化ができない。そこで本稿では、アルゴリズムレベル情報を利用することで、C++コンパイラが自動SIMD化しやすいプログラムへ変換する方式を提案する。

Program Transformation for Vectorization in Array Processing Language

Yusuke Shirota †, Jun'ichi Segawa † and Tatsunori Kanai †

Productivity of multicore programs for evolving multicore architecture is increasingly becoming an important issue. To address this issue, we designed an architecture-independent programming system, with our new high-level language dedicated to array processing. Array processing programs are translated into C++ programs optimized for each architecture by our translator. To exploit SIMD instructions effectively with minimal effort of supporting new architectures, it is preferable to use auto-vectorization of C++ compiler. However, auto-vectorization is restricted to inner most loops in which all memory access are consecutive. Our language design allows the translator to easily extract high-level information needed for optimization. To enhance compilation with auto-vectorization, we propose a program transformation method using these information. We confirmed that C++ programs with inner most loops amendable to auto-vectorization can be generated.

1. はじめに

近年、プロセッサはマルチコア化により飛躍的に高速化すると同時に、そのアーキテクチャも多様化・複雑化している[1)2)]。そのため、効率よく動作するマルチコアプログラムの開発には、ターゲットアーキテクチャに関する深い知識が必要となっている。また、特定のアーキテクチャを考慮して書かれたプログラムは、再利用性が低くなってしまうという課題もある。このような背景のもと、プログラムの生産性を向上させるためにターゲットアーキテクチャを抽象化したマルチコア言語[3)4)]も登場している。

これらの課題に対して、我々は、プログラムの記述レベルをアルゴリズムレベルまで上げるマルチコアプログラム開発方式を提案している[6)7)8)]。プログラムをアルゴリズムレベルで記述することで、特定のアーキテクチャに依存しないプログラムにすることができる。我々はアルゴリズム記述に用いる言語として、配列処理に特化した関数型言語を提案している。本言語ではマルチコア上での並列処理を前提としているため、対象を並列処理と相性のよい配列処理に限定している。

本配列処理言語で記述されたプログラムは、本配列処理言語の処理系によって、各ターゲットアーキテクチャで効率よく動作するC++プログラムに変換される。また、プログラム変換時に処理系が最適化用のアルゴリズムレベル情報が抽出しやすいように言語が設計されている。このため、データ依存解析などの複雑な解析は不要になり、処理系が行う解析を軽くできることから、処理系の開発コストや実行時のオーバヘッドを削減できる。

プログラム変換時に考慮する最適化の一つとして、各アーキテクチャに実装されているSIMD命令を利用した高速化が効果的である。SIMD命令を生成する方法として、C++コンパイラの自動SIMD化機能[5)]を利用する方法がある。自動SIMD化を利用すれば、処理系でアーキテクチャ毎にSIMD命令を直接生成する必要がなくなるため、処理系の実装を容易にすることができる。しかし、C++コンパイラの自動SIMD化は、SIMD化対象のループに十分な並列性があることを前提としているため、小さい配列同士の配列演算では十分な並列性が抽出できないなど万能ではない。

また、一般にSIMD化は、メモリアクセスの最適化をした上で行わないと、実行時間がメモリアクセスに律速してしまい、その効果が隠れてしまうという課題もある。

そこで本稿では、メモリアクセスを各ターゲットアーキテクチャにあわせて最適化した上で、配列演算の対象となる配列間の間隔などのアルゴリズムレベルでは容易にわか

† (株) 東芝 研究開発センター
Toshiba Corporation, Corporate Research & Development Center

$$f(x) \leftarrow \left| \sum \left(x \times \begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix} \right) \right|$$

$$Laplacian(M_{\langle 480,720 \rangle \sharp uint8})_{\langle 480,720 \rangle \sharp uint8} \leftarrow \forall f \left(\bigoplus_{\langle -1,-1 \rangle \sharp \langle 1,1 \rangle}^{\rho_M \sharp \langle 3,3 \rangle} M \right)$$

図 1 ラプラシアンフィルタプログラム

る情報を利用して、配列演算間での並列性を抽出して C++ コンパイラに解析しやすい形で提示するプログラム変換方式を提案する。

以下、2 章で本配列処理言語について述べ、3 章で本開発方式の主なターゲットである画像処理の近傍処理における SIMD 化の課題を説明する。4 章ではアルゴリズムレベル情報である近傍情報を用いた SIMD 化向けプログラム変換方式を提案し、5 章で性能評価を行う。6 章で関連研究について述べ、7 章でまとめる。

2. マルチコアプログラム開発方式

2.1 配列処理に特化したアルゴリズム記述言語

本配列処理言語では、図 1 に示すラプラシアンフィルタプログラムのように、配列データの処理を配列演算を組み合わせた抽象度の高い記述でプログラミングできる。

本配列処理言語の最大の特長は、C++で多重ループで記述していた処理が、2 つの配列切り出し演算子 $\bigoplus$(繰返し切り出し演算子) と $\odot$ (単一切り出し演算子)、および、引数の配列の要素ごとに関数を適用するマップ関数呼び出し $\forall$ を使うことでループレスで記述できる点である。

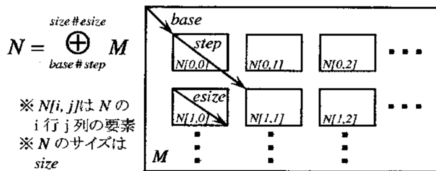

図 2 繰返し切り出し演算子

演算子 $\bigoplus$ は、図 2 に示すように、引数の配列 M から部分配列を繰返し切り出し、切り出した部分配列を要素とする配列 N を作成する演算子である。パラメータ $base$ に最初の部分配列の切り出し開始位置、$step$ に切り出し位置の行方向および列方向のずらし幅、$size$ に切り出す配列の行方向および列方向の個数、$esize$ に切り出す個々の配列のサイズを指定する。これらのパラメータは、配列のインデックスやサイズを表す行と列を示す整数値のペアを ⟨ と ⟩ で囲んで表記する。

演算子 $\bigoplus$ で作成した配列を引数にして関数を呼び出す場合、関数名の前に $\forall$ を付けると配列の要素毎に関数が適用される。このように組み合わせて利用することで、C++ の多重ループに相当する処理が記述できる。このようにして、性能に影響を及ぼしやすいループを陽に記述させず、ターゲットアーキテクチャに適したループを処理系で生成するのが本配列処理言語の大きな特長である。

演算子 $\odot$ は、既存の配列から部分配列を 1 つ切り出す演算子である。プログラム中では、$\underset{base}{\overset{size}{\odot}} M$ の形で記述し、切り出し開始位置をパラメータ $base$ に、部分配列のサイズを $size$ に指定する。

このように、アルゴリズムレベルの情報である配列のサイズや切り出し間隔を、配列演算子のパラメータとしてプログラマに明示させることで処理系が最適化に必要な情報が取得しやすくなっている。なお、言語仕様の詳細に関しては、文献[8] を参照されたい。

2.2 近傍処理プログラムの記述例

本配列処理言語の主なターゲットは、画像処理アルゴリズムにおける近傍処理である。近傍処理は、データ局所性が高いため、マルチコア上での並列処理に向いている。近傍処理では、近傍画素を使った同じ処理を画像の全画素に対して繰返し適用する。このため、2.1 節で説明した配列処理演算子を使ってシンプルに記述できる。

図 1 に、代表的な近傍処理であるラプラシアンフィルタのプログラムを示す。ラプラシアンフィルタは、8-近傍を用いた 2 次微分値を求めるアルゴリズムである。プログラムでは、演算子 $\bigoplus$ を使って入力画像 M の各画素の周辺の 9 画素からなる 3 行 3 列の配列からなる配列を作り、その各配列に対して関数 f をマップ関数呼び出しで適用している。f は、切り出した 3 行 3 列の配列 x に 3 行 3 列のフィルタ係数の配列を掛けて、9 つの要素の総和 $\sum$ を求める。ここで、ρ_M は配列 M のサイズを返す組込み関数である。

2.3 配列処理言語の処理系

本配列処理言語の処理系の構成を説明する。本処理系の入力は、本配列処理言語で記述されたプログラムである。まず、フロントエンドにより、入力プログラムを中間表現に変換する。次に、プログラム変換部により、中間表現に対して最適化を行う。そして最後にバックエンドにより、中間表現から、それぞれのターゲットアーキテクチャ毎に用意するランタイムライブラリを呼び出しながら実行する C++プログラムに変換する。生成された C++プログラムは、自動 SIMD 化機能を有する C++コンパイラによって、SIMD 命令を利用する実行プログラムにコンパイルされる。

プログラム変換部では、C++プログラムが自動 SIMD 化しやすくなるように 4 章で述べるプログラム変換を施す。

3. SIMD 化の課題

3.1 SIMD 化による高速化の課題

一般に、SIMD 化の効果が出るようにするためには、メモリアクセスもあわせて最適化する必要がある。プロセッサがキャッシュを持つ場合には、配列データがキャッシュサ

```
for(i=1; i<479; i++){
  for(j=1; j<719; j++){
    tmp = 0;
    T1_p = T1b + 720*(i-1) + (j-1);
    T2_p = T2b;
    for(x=0; x<=2; x++){
      for(y=0; y<=2; y++){ /* trip count too small */
        tmp += *T1_p++ * *T2_p++;
      }
      T1_p += 717;
    }
    tmp = abs(tmp);
    *T0_p++ = (uint8)((tmp > 255)? 255: tmp);
  }
  T0_p += 2;
}
```

図 3 ラプラシアンフィルタの C++プログラムの概要

イズに収まらないとキャッシュミスによりメモリアクセスのレイテンシが大きくなってしまう。その場合、仮に SIMD 化できても、その効果はメモリアクセスのレイテンシに隠れて無駄になってしまう。

3.2 自動 SIMD 化の課題

C++コンパイラの自動SIMD化を利用することで、アーキテクチャ毎の SIMD 命令の生成をコンパイラに任せることができる。しかし、自動SIMD化できる処理は限定的である。

一般に、自動SIMD化の対象となるのは、多重ループの最内ループの処理で、かつ、メモリアクセスが連続である処理が基本となる。さらに SIMD レジスタサイズに対してループ回転数が小さすぎると十分な並列性がないと判定され自動SIMD化できない。このため、メモリアクセスが連続で、かつ、ループ回転数が大きい最内ループを生成してコンパイラに提示する必要がある。

しかし、このようなループを生成しても、ループのデータの依存関係の有無やループ回転数をコンパイラが解析できない場合があるため、コンパイラが自動 SIMD 化できるとは限らない。そのため、関連するアルゴリズム情報をプラグマの挿入という形でコンパイラにヒントすることが必要となる。

3.3 近傍処理の SIMD 化の課題

画像処理における近傍処理では、3x3 画素や 5x5 画素などの範囲にある近傍画素を使った処理を全画素に対して繰返し適用する。この繰返しの単位で SIMD 化しようとすると、演算量やデータ転送サイズの観点からも並列性が十分でない。このため、仮に制限の多い自動SIMD化に頼らずに、SIMD 命令を直接記述しても効率がよい SIMD 化ができないという課題がある。よって、配列処理用ライブラリを SIMD 化して用意しておくことも解にならない。

繰返し単位の配列処理に対して、処理系の基本的なループ生成方式では、対応する 2 重ループを生成する。そのため、近傍処理では、生成される 2 重ループの最内ループのループ回転数が 3 や 5 などとなる。ラプラシアンフィルタの例では、図 3のような C++プログラムを生成する。このような 2 重ループに対して SIMD 命令の生成をコンパイラの自動SIMD化で行おうとすると、最内ループのループ回転数が小さすぎるため自動SIMD化できないという課題がある。また、2 重ループのループ回転数が小さいため、

表 1 評価環境

項目	スペック
Intel Core 2 Quad	2.66GHz
L1 データキャッシュ	Four 32[KB], 8-way set associative
L2 キャッシュ	Two 4[MB]
Linux	2.6.22
C++コンパイラ	Intel C++ コンパイラ 10.0

ループオーバヘッドが顕著化するという課題もあった。

4. 近傍情報を利用した SIMD 化向けプログラム変換方式

本章では、配列処理プログラムに明示されている近傍情報を利用した簡単な解析を行うことで、効率のよい自動SIMD化が可能なC++プログラムを生成するプログラム変換方式を提案する。

4.1 提案方式の概要

提案方式の概要を説明する。近傍情報を利用した SIMD 化向けプログラム変換は 3 段階で行う。

最初に、SIMD 化の効果がメモリアクセスのレイテンシで隠れないように、メモリアクセスを最適化する。プロセッサが、本稿の評価で利用する表 1 の Intel Core 2 Quad のようにキャッシュを持つ場合☆、キャッシュの容量を考慮して処理順序を変更するループブロッキングを行うことでキャッシュミスを減らすことができる。本開発方式では、ターゲットアーキテクチャにとって最適な処理順序の近似解を求める、レンジスケジューリング[6] と呼ぶメモリアクセスの順序の制御を行う。これによりメモリアクセスレイテンシを抑制し、SIMD 化が効きやすい状態を整える。

次に、コンパイラが自動 SIMD 化しやすいループをつくるための変換を行う。近傍処理の繰返しの単位で SIMD 化しようとしても十分な並列性を抽出できない。そこで、本稿では、近傍処理間の並列性を利用できるようにプログラム変換を行う。これにより、自動 SIMD 化しやすいループ、すなわち、メモリアクセスが連続した、ループ回転数が大きい最内ループをつくる。

最後に、生成された最内ループに対して、プラグマを挿入する。コンパイラにはポインタが指す配列間の依存関係などが解析できない場合がある。プラグマを挿入することで、アルゴリズムレベルの知識をコンパイラにヒントとして与える。

以降、4.2 節でメモリアクセスの最適化について、4.3 節で近傍処理間の並列性の抽出について述べ、4.4 節でコンパイラヒントの挿入について説明する。

4.2 メモリアクセスの最適化

メモリアクセスの最適化をレンジスケジューリングを利用して行う。レンジスケジューリングでは、プログラムのトップレベルの式の結果配列全体をレンジとして、そのレンジを小さい矩形領域のレンジに階層的に区切っていき、区切ったレンジ間の実行順序を決めていく。

表 1 の Intel Core 2 Quad のようなキャッシュを持つマルチコアの場合について説明する。まずは、レンジを複数

☆ レンジスケジューリングでは、プロセッサがローカルメモリを持つ場合も統一的に扱うが、本稿ではキャッシュを持つ場合に絞って説明する。

コアで分割する。最も単純にはレンジをコア数で上下方向に分割してそれぞれのコアが処理する。

このレンジをブロッキングでキャッシュが有効に働く幅のレンジにさらに分割する。レンジの幅の計算には、レンジの処理に必要なデータの集合であるワーキングセットをまず計算する。そして、ワーキングセットの各部分配列が、キャッシュサイズをウェイ数で分割したサブブロックのいずれかに収まる、できるだけ幅の大きなレンジのサイズを選択する。

レンジのワーキングセットは、近傍情報を用いて簡単に解析することができる。図1のラプラシアンフィルタの例を使って、サイズ $\langle 1,c\rangle$ のレンジのワーキングセットを求めてみる。入力変数 M のワーキングセットは、$esize{+}step{\times}\langle 1{-}1,c{-}1\rangle$ と計算できるので、$\langle 3,3{+}(c{-}1)\rangle$ となる。出力変数のワーキングセットは $\langle 1,c\rangle$ と計算できる。これを格納するのに必要なメモリ容量は、行成分と列成分の積に、さらに $sizeof(uint8)$ を掛けた値になるので、それぞれ、$3c+6$、c になる。また、後述する、計算過程で必要となる型変換解決用の 32 ビット整数型の配列については、ワーキングセットは $\langle 1,c\rangle$ と計算できる。必要なメモリ容量は、これに $sizeof(int32)$ を掛けた値である $4c$ になる。以上により、この中で一番大きい、型変換解決用の配列に依ってレンジの幅が決定する。表1よりL1データキャッシュの1ウェイが 4[KB] なので、求めるレンジの幅は $4c \leqq 4[KB]$ を満たす最大の c になるので、c は画像の横幅の 720 と求めることができる。

次節以降で、最内のレンジの中の処理の SIMD 化について述べる。

4.3 近傍処理間の並列性の抽出

近傍処理間の並列性の抽出は、本配列処理言語の最も特徴的なイディオムである、演算子 $\bigoplus$ とマップ関数呼び出しの組み合わせで以下のように記述される近傍処理に対して適用する。

$$\forall f\left(\bigoplus_{base\sharp\langle 1,1\rangle}^{size\sharp\langle I,J\rangle} M\right)$$
$$\textbf{where} \quad f(x) \leftarrow \sum g(x)$$

具体的には、$\bigoplus$ で $\langle 1,1\rangle$ 間隔で繰返し切り出すサイズ $\langle I,J\rangle$ の部分配列 x に対して、エレメントワイズな配列演算 g を適用した後、アグリゲーション演算 $\sum$ を適用し、スカラー型のデータを返す処理である。これは C++では多重ループに相当する処理で、アプリケーションのホットスポットになる可能性が高く、SIMD 化の効果が大きい。

この処理の SIMD 化は、次のように近傍処理間の並列性を利用して行う。まず、$\bigoplus$ で繰返し切り出すサイズ $\langle I,J\rangle$ の $size$ 個の部分配列について、そのインデックス $\langle i,j\rangle$ の要素のみを集めた配列データを i 行 j 列の要素とするサイズ $\langle I,J\rangle$ の配列を V とする。これらを用いると、近傍処理全体の処理は、$\sum g(V)$ と表すことができる。g を V に合わせて拡張すると、近傍処理全体の結果である配列データが返る配列演算に変換できる。

本配列処理言語では、$\bigoplus$ に明示された近傍情報である $step = \langle 1,1\rangle$ から近傍処理間の配列の間隔がわかる。この

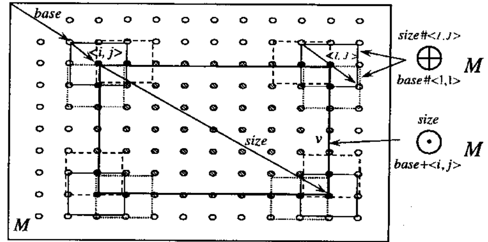

図4 $\bigoplus$ 演算子で切り出す各部分配列のインデックス $\langle i,j\rangle$ の要素の集合を $\bigodot$ 演算子で切り出した配列 v

```
for (i=1; i<479; i++){
  for (j=1; j<719; j++){
    tmp = abs( (*T1_p) + (*T2_p)      + (*T3_p)
             + (*T4_p) + (*T5_p * -8) + (*T6_p)
             + (*T7_p) + (*T8_p)      + (*T9_p));
    tmp = (tmp > 255)? 255: tmp;
    *T0_p = (uint8)tmp;
    T1_p++;
    T2_p++;
    ....
    T9_p++;
    T0_p++;
  }
  T1_p += 2;
  T2_p += 2;
  ....
  T9_p += 2;
  T0_p += 2;
}
```

図6 ラプラシアンフィルタのプログラム変換後の C++プログラムの概要

ことから、$\bigoplus$ で切り出す部分配列の各インデックスの要素の集合は、行方向と列方向に連続している部分配列であることがわかる。部分配列のサイズは $size$、切り出し位置のずらし幅が $\langle 1,1\rangle$ なので、V は $\bigoplus_{base\sharp\langle 1,1\rangle}^{\langle I,J\rangle\sharp size} M$ で切り出せることがわかる。これにより、処理系でこのイディオムを中間表現上で認識すると、次のような変換を行う。

$$\sum g\left(\bigoplus_{base\sharp\langle 1,1\rangle}^{\langle I,J\rangle\sharp size} M\right)$$

V の i 行 j 列の要素配列は、図4に示すように $\bigodot$ を使って $\bigodot_{base+\langle i,j\rangle}^{size} M$ で表現できるのでこれを使って $\sum$ を展開する。

図1のラプラシアンフィルタプログラムの例では、中間表現上で、図5のプログラム相当の中間表現に変換する。

この変換後は、処理系では図6のような C++プログラムを出力する。最内ループに注目すると、部分配列の各要素を表すポインタ T1_p〜T9_p、および、結果を格納する要素を表すポインタ T0_p のメモリアクセスが連続していることがわかる。また、ループ回転数も 718 と十分に大きくなっている。以上のことから、コンパイラの自動 SIMD

$$\left| \bigodot_{\langle -1,-1\rangle}^{\rho_M} M + \bigodot_{\langle -1,0\rangle}^{\rho_M} M + \bigodot_{\langle -1,1\rangle}^{\rho_M} M + \bigodot_{\langle 0,-1\rangle}^{\rho_M} M - 8M + \bigodot_{\langle 0,1\rangle}^{\rho_M} M + \bigodot_{\langle 1,-1\rangle}^{\rho_M} M + \bigodot_{\langle 1,0\rangle}^{\rho_M} M + \bigodot_{\langle 1,1\rangle}^{\rho_M} M \right|$$

図 5 プログラム変換後のラプラシアンフィルタプログラム

化が期待できるループが生成できている。

4.4 コンパイラヒントの挿入

4.3 節までにおいて、SIMD 化向けの最内ループが生成できた。次は、この最内ループの直前にプラグマを挿入することで、コンパイラにこのループに関するアルゴリズム情報を提示する。ここでは、本稿の評価で利用する Intel C++コンパイラを想定する。

まず#pragma ivdep を挿入する。これにより、最内ループにデータ依存がないことを指示する。これでコンパイラは SIMD 化できることが静的にわかる。この挿入ができるのは、本配列処理言語が関数型であるためマップ関数呼び出しの処理間でデータ依存がないことを保証しているからである。

さらに、#pragma loop count を挿入する。これにより、ループ回転数の目安を指示する。SIMD 化向けプログラム変換を施したので、ループ回転数は十分に大きくなっている。しかし、コンパイラにはこれが解析できず SIMD 化されない場合があるために必要となる。ここでは、#pragma loop count (16) などと SIMD レジスタサイズより大きくすればよい。

最後に、型変換が自動 SIMD 化できない制約を解決するための処理を行う。画像処理では、入力画像の符合なし 8 ビット整数で表現される画素を入力とし、符合付き 16 ビット整数などで計算を行い、計算結果を再び出力画像の符合なし 8 ビット整数の画素に戻す型変換が必要となる。この後半の型変換が、Intel C++コンパイラでは自動 SIMD 化できないという制約がある。そこで、計算結果を一時保存する 16 ビット整数型の型変換解決用の配列を用意し、そこへ代入するようにする。これにより、その代入の処理までは自動 SIMD 化できない型変換が入らなくなるため、自動 SIMD 化が可能となる。型変換解決用の配列から出力画像にデータをコピーする後続の処理のみが自動 SIMD 化されないようになる。

5. 性能評価

SIMD 化向けプログラム変換方式の効果を確認するために、性能評価を行った。評価には、近傍処理である、ラプラシアンフィルタ (*Laplacian*)、平滑化フィルタ (*Blur*)、Prewitt フィルタ (*Prewitt*) を用いた。

平滑化フィルタは、画像を平滑化するために、各画素データに対してその近傍 8 画素との平均値を求める。本配列処理言語では、図 7 のように記述できる。

$$\forall f\left(\bigoplus_{\langle -1,-1\rangle \sharp \langle 1,1\rangle}^{\rho_M \sharp \langle 3,3\rangle} M \right)$$
$$\textbf{where} \quad f(x) \leftarrow \frac{\sum x}{9}$$

図 7 平滑化フィルタプログラム

Prewitt フィルタは、エッジ検出を行うために、各画素データとその近傍 8 画素に対して、2 つのフィルタ v、h を適用する。本配列処理言語では、図 8 のように記述できる。

$$\forall f\left(\bigoplus_{\langle -1,-1\rangle \sharp \langle 1,1\rangle}^{\rho_M \sharp \langle 3,3\rangle} M \right)$$
$$\textbf{where} \quad h \leftarrow \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}; \quad v \leftarrow \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$
$$f(x) \leftarrow \frac{\left|\sum (x \times h)\right|}{2} + \frac{\left|\sum (x \times v)\right|}{2}$$

図 8 Prewitt フィルタプログラム

評価環境は、表 1 の Core 2 Quad を用いた。また、C++コンパイラには、Intel C++ コンパイラ 10.0 を用いた。

まず、メモリアクセス最適化の効果を確認する。図 9 に、レンジの幅を変化させた際の実行時間の変化を示す。レンジの幅は、1 から画像の横幅まで変化させている。レンジスケジューリングによるレンジの幅の計算値は、ラプラシアンフィルタと平滑化フィルタがそれぞれ 720、Prewitt フィルタは 360 となる。図 9 より、よい実行時間となる幅が計算できていることがわかる。今回評価に用いたアプリケーションは、実際にはレンジの幅を画像の横幅としても、ワーキングセットが L1 データキャッシュに収まるほど小さい。よって、実行時間はレンジの幅を画像の横幅とした時に一番よい結果となっている。

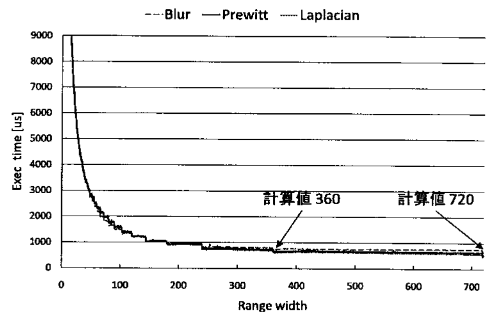

図 9 メモリアクセス最適化の評価

次に、SIMD 化および並列化の効果を図 10 で確認する。速度向上は、4.3 節で説明した $\bigodot$ への変換を行わない場合の実行時間 (図中の $Oplus(AutoVec)$) を基準としている。$\bigodot$ への変換までを行った場合を $Odot(AutoVec)$、同変換後にコンパイラヒントを挿入した場合を $Odot + CH(AutoVec)$、さらに 4 コアで並列化した場合を *4core* で示す。また、比較のために、Intel C++コンパイラのイントリンシックを使って SIMD 化をした場合を

$Odot(Intrinsics)$ と示す。

⊙への変換を行わない場合、自動SIMD化はできなかった。⊙への変換までを行った場合でも自動SIMD化はできなかったが、ループオーバヘッドが減ったことやフィルタ定数との必要のない演算が減ったことで、それぞれ3.3倍、5.8倍、8.9倍の速度向上を得ることができた。

同変換後にコンパイラヒントを挿入して自動SIMD化することで、それぞれさらに1.4倍、2.4倍、2.6倍の速度向上が得られることが確認できた。自動SIMD化しない場合は、1画素単位で処理が行われる。これに対して、自動SIMD化した場合では、4画素単位で処理されるようになる。ラプラシアンフィルタの例では、4[B]のロードが9回実行され、それぞれがパックドダブルワードに変換された後、パックドダブルワード同士の演算が行われる、4並列のSIMD化がなされる。

一方で、イントリンシックを使った場合との比較では、それぞれさらに1.6倍、1.7倍、2.1倍速度向上できる余地を残していることがわかる。イントリンシックを使ったものでは、パックドワードでの8並列のSIMD化を行っている。自動SIMD化の場合に4並列になっているのは、現状の処理系で、プログラムの処理過程で必要な中間値の型をC++のデフォルト型変換ルールに従って素直に決定しているためである。そのため、本来は16ビット整数でビット長が足りるところを32ビット整数を利用している。また、データ転送は、16[B]で転送を行った方が効率がよい。このため、イントリンシックを使ったものでは、16画素分に相当する16[B]のロードを9回行い、レジスタ上で上位ビットと下位ビットに分けて計算し、結果を合成してから再び16[B]でストアしている。このような細やかなSIMD化が、自動SIMD化では難しいことも性能差に影響している。

最後に、4コアでの実行では、それぞれ、3.4倍、3.0倍、2.5倍の速度向上が得られることが確認できた。

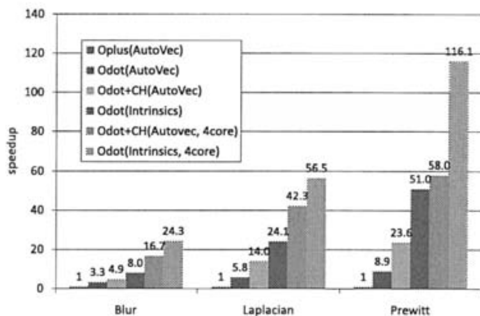

図10 SIMD化および並列化による速度向上

6. 関連研究

X10[3]やSequoia[4]などのマルチコア言語では、ターゲットアーキテクチャを抽象化してプログラマにみせ、その上で問題の分割方法を明示させている。これに対して、本開発方式では、記述レベルをあげることでターゲットアーキテクチャに依存する処理の記述をプログラマから隠蔽し、ターゲットアーキテクチャへのマッピングの一切を処理系で自動的に行う。

中西ら[9]は、SIMD記述の可搬性を保つために、演算レベルでSIMD向けの共通記述方式を提案している。本処理系でも、より抽象度の高いレベルで共通記述を定義し、これを利用したC++プログラムを生成することも考えられる。

7. おわりに

本配列処理言語のアルゴリズムレベル情報を利用した簡単なプログラム変換と、C++コンパイラの自動SIMD化を組み合わせることで、近傍処理アプリケーションがSIMD命令を直接記述することなくSIMD化できた。これは、本配列処理言語がアルゴリズムレベル情報が取得しやすくなっているために、複雑なデータ依存解析などが不要になり、処理系で容易に最適化ができるという本開発方式の有効性の一端を示すものであると考えられる。

処理系の実装コストという観点でコンパイラの自動SIMD化を利用するアプローチは有効ある一方、不連続アクセスに対応していないなど未だに制約が多い。今後は、さらにSIMD化できる範囲を拡大することも必要であるため、コンパイラの自動SIMD化の進歩にも期待しつつ、処理系で一部SIMD命令を生成する方法などを併用することなどを検討する。

また、SIMD化の並列度を上げるためにビット長推論を行いさらに効率のよいSIMD化をすることも課題である。

参考文献

1) Intel Core 2 Quad: http://www.intel.com/products/processor/core2quad/index.htm.
2) D. Pham, et al. "The Design and Implementation of a First Generation CELL Processor", *International Solid State Circuits Conference*, 2005.
3) X10: http://x10.sourceforge.net/
4) K. Fatahalian et al. "Sequoia: programming the memory hierarchy," *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, 2006.
5) Aart J. C. Bik. The Software Vectorization Handbook, Intel Press, 2004.
6) 金井他. 組み込みプロセッサのメモリアーキテクチャに依存しない画像処理プログラムの記述と実行方式, 情報処理学会論文誌:コンピューティングシステム, Vol. 48, No. SIG13(ACS19), pp. 287–301, 2007.
7) J. Segawa et al. "The Array Processing Language and the Parallel Execution Method for Multicore Platforms", *The First International Symposium on Information and Computer Elements*, pp. 98–103, 2007.
8) 瀬川他. 配列処理向けドメイン特化言語によるマルチコアプログラミング, SWoPP2008(掲載予定), 2008.
9) 中西他. コードの性能可搬性を提供するSIMD向け共通記述方式, 情報処理学会論文誌:コンピューティングシステム, Vol. 48, No. SIG13(ACS19), pp. 95–105, 2007.