卒業研究論文

解像度非依存型動画像処理ライブラリRaVioliのSIMD命令による高速化

指導教員　　松尾 啓志 教授
津邑 公暁 准教授

名古屋工業大学 工学部 情報工学科
平成 17 年度入学　17115074 番

白木 夢斗

解像度非依存型動画像処理ライブラリ RaVioli の SIMD 命令による高速化

白木 夢斗

内容梗概

近年，ビデオカメラ等の入力機器と計算機間でのデータ転送速度が向上し，リアルタイムで計算機に画像を取り込むことができる環境が整ってきている．また，汎用 PC の高性能化によって，従来専用ハードウェアで行っていた高度な動画像処理を汎用 PC 上で行うことも可能となった．そのため汎用 PC 上でリアルタイム動画像処理を行うことが増加すると考えられる．しかし，汎用 PC では他プロセスも実行されているため使用可能な CPU リソースは限られており，動画像処理のリアルタイム性を保証するのは困難である．

そこで，擬似的なリアルタイム性を保証する動画像処理ライブラリ RaVioli が提案されている．RaVioli では使用可能な CPU リソースの変動に応じて，時間解像度と空間解像度を動的に変化させ，画像処理の処理量を調整する．これにより擬似的なリアルタイム処理を実現する．また，抽象的な画像処理の記述が出来る環境も提供している．しかし，RaVioli を用いたプログラミングを行うことによって，実行時間が低下するという問題があり，時間解像度と空間解像度が低下する原因となるため，高速化が必要である．

そこで，本研究では CPU プロセッサに搭載されている SIMD 命令を用いた高速化を行うこととした．しかし，RaVioli ならではの特性上，RaVioli ライブラリ内部に直接 SIMD 命令を用いることが出来ない．そのため，ユーザ記述の画素単位の処理関数と RaVioli ライブラリの高階メソッドから，SIMD 命令を用いた特定のアルゴリズム専用処理関数に変換するトランスレータを実装した．

また，このトランスレータをサンプルプログラムを用いて評価した．プログラムの変換前と変換後とを比較し，速度向上が確認できた．

解像度非依存型動画像処理ライブラリ RaVioli の SIMD 命令による高速化

目次

1 はじめに 1

2 解像度非依存型動画像処理ライブラリ RaVioli 2

2.1 RaVioli の概要 2

2.1.1 擬似的なリアルタイム性 4

2.1.2 抽象的な画像処理の記述 4

2.2 高階メソッド 4

2.3 RaVioli の問題点 7

3 SIMD 命令を用いた RaVioli の高速化手法 7

3.1 SIMD 命令 7

3.2 RaVioli に SIMD 命令を用いる際の問題点 8

3.3 SIMD 命令を用いた具体的な記述方法 9

4 SIMD 命令を用いたプログラムへの自動変換 13

4.1 変換手順の概略 13

4.2 変換対象部分 14

4.3 関数のインライン展開 16

4.4 SIMD 命令への変換 19

5 評価 21

6 おわりに 23

参考文献 24

付録

A.1 変換前

A.2 変換後

1 はじめに

近年，ビデオカメラ等の入力機器と計算機間でのデータ転送速度が向上し，リアルタイムで計算機に画像を取り込むことができる環境が整ってきている．また，計算機の性能向上に伴って，従来専用ハードウェアを用いて行っていた高度な動画像処理が汎用 PC 上でもある程度実行可能となってきた．汎用 PC は専用ハードウェアに比べて安価であるため，コスト削減のために今後リアルタイム性が重要である侵入者検知システムのような動画像処理を汎用 PC 上で行うことが増加すると考えられる．

しかし，Linux 等の汎用 OS と汎用 PC を組み合わせた環境では，1/30 もしくは 1/60 秒毎に必ず処理を開始する必要があるリアルタイムストリーミング処理は未だに困難である．その主な原因として，処理する画像に依存して処理量が増加したり，他プロセスとの競合により使用可能な CPU リソースが変動することによって，処理遅延が発生することが挙げられる．そのため，汎用 OS と汎用 PC を組み合わせた環境上でリアルタイム性が重要である動画像処理を行いたい場合は，処理量を低下させる等によって擬似的なリアルタイム処理を行うことで対応する必要がある．しかし，静的に処理量の削減を行ったのでは使用可能な CPU リソースが余っている時にも低い精度の処理結果しか得られない．そこで，出来るだけ高い精度の動画像処理結果を得たい場合，使用可能な CPU リソースが少なくなっているときのみ動的に処理量を低下させる必要がある。しかし，ユーザ自身の手で動的な処理量変化を行うプログラムを記述するのは非常に困難であり，動画像処理アルゴリズムに関する記述以外の部分でユーザに負担がかかる．そのため，ライブラリ側で動的な処理量変化に対応することで，ユーザが画像処理アルゴリズムを動画像処理にそのまま反映できる記述が出来ればよいと考えられる．

現在提案されている画像処理ライブラリとしては，VIGRA[1] や OpenCV[2] といったものがある．これらのライブラリではエッジ処理や二値化等の様々な画像処理アルゴリズムの処理関数が提供されている．これらの処理関数は，対象画像と処理を選択するパラメータ等を与えるだけで画像処理が行えるような仕様となっている．そのため，ユーザはあらかじめ用意された画像処理アルゴリズムを用いて画像処理プログラムを記述する必要があり，記述できる画像処理プログラムの自由度は低い．また，VIGRA は動画像処理に対応しておらず，ユーザ自身の手で動画像に対応させる必要がある．一方，OpenCV は動画像処理に対応しており，読み込んだフレームを逐次処理することが出来る．しかし，処理結果のリアルタイム性は保証されていないため，ユーザ自身

の手でこれを満たすプログラムを記述する必要がある．そのため，これらの画像処理ライブラリを用いて擬似的なリアルタイム動画像処理を行うことは非常に困難である．

そこで，擬似的なリアルタイム性を保証する動画像処理ライブラリとしてRaVioli[3]が提案されている．RaVioliでは処理対象画像の解像度及びフレームレートをパラメータ化し，使用可能なCPUリソース等の負荷の変動に応じて自動的にパラメータを変化させ，画像処理の処理量を調整する．これにより擬似的なリアルタイム動画像処理を実現する．また，CPU リソースが足りない時に画像解像度とフレームレートのどちらをどれだけ優先するかを比率によって指定することができる．例えば，画像解像度を若干優先的に維持したい場合，画像解像度7，フレームレート3といった具合に指定すればよい．更に，RaVioliでは画像処理アルゴリズムをユーザ自身の手で記述出来る．そのため，複雑な画像処理を行う場合は，先に述べたVIGRAやOpenCV等の画像処理ライブラリでは用意された処理関数を工夫して組み合わせる必要があるが，RaVioliではそのまま画像処理アルゴリズムを記述すればよい．このように，記述できる画像処理プログラムの自由度が高いというのも特徴である．

しかし，RaVioliの問題点として，処理を抽象化することでユーザは容易にプログラムを記述できる半面，抽象化によるオーバーヘッドによって実行速度も低下するという問題点がある．そこで，本研究では，解像度非依存型動画像処理ライブラリRaVioliにSIMD命令を用いることで，処理の高速化を目指す．そのためにユーザが記述した画像処理関数から，その画像処理アルゴリズムの専用関数を生成し，更に画像処理アルゴリズムをSIMD命令に自動変換する．

本論文では，2章で解像度非依存型動画像処理ライブラリ RaVioliの概要を説明する．3章ではSIMD命令を用いたRaVioliの高速化手法，4章ではSIMD命令を用いたプログラムへの自動変換するトランスレータについて説明する．5章で生成したプログラムの評価と考察を行い，6章で結論を述べる．

2　解像度非依存型動画像処理ライブラリRaVioli

本章では解像度非依存型動画像処理ライブラリ RaVioliの特徴と問題点について説明する．

2.1　RaVioliの概要

解像度非依存型動画像処理ライブラリRaVioliは，処理対象画像の解像度及びフレームレートをパラメータとして管理している．これによって，

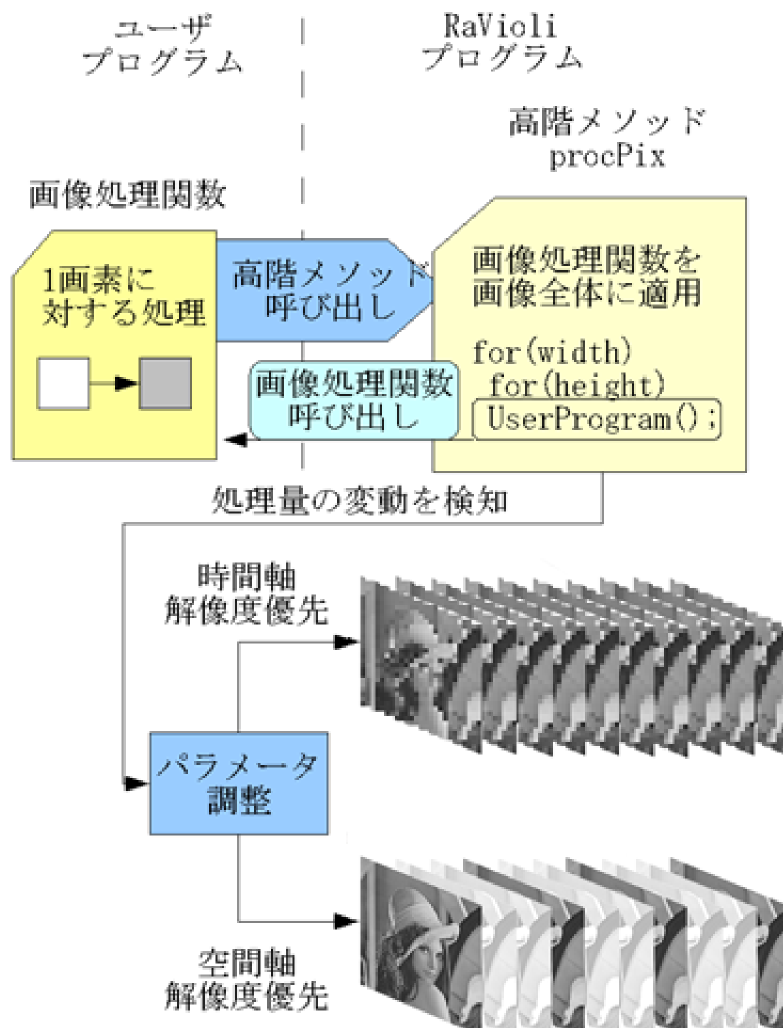

図 1: RaVioli の概念図

- 動画像処理の擬似的なリアルタイム性
- 抽象的な画像処理の記述

の 2 点を実現している．

RaVioli では，画素単位の処理を画像全体に適用出来る高階メソッド procPix をユーザに提供する．そのためユーザは画素単位の処理関数を記述し，高階メソッドにその処理関数の関数ポインタを引数として渡すだけで，画像全体に画素単位の処理を適用することが出来る．一方高階メソッドでは，空間解像度である画像解像度と時間解像度であるフレームレートをパラメータとして管理し，処理量の変動を検知するとこれらのパラメータを変化させる．このとき，ユーザはどちらのパラメータを優先的に維持したいかを，比率で指定する事が出来る．これらの概念図を図 1 に示す．

2.1.1 擬似的なリアルタイム性

一般に動画像処理プログラムでは，出力フレームレートと１フレームあたりに割ける時間はトレードオフの関係にある．そのため，汎用 PC 等の使用可能な CPU リソースが限られる環境においては，時間解像度であるフレームレート，もしくは空間解像度である画像解像度のいずれかを下げる必要がある．RaVioli では，これら二つの解像度を使用可能な CPU リソースに応じて動的に変化させることで擬似的なリアルタイム動画像処理を実現している．しかし，リアルタイム性を重要とする動画像処理でも目的の違いによっては，時間解像度と空間解像度のうち，より重要である解像度が異なる場合がある．例えば侵入者検知システムでは，画像の解像度よりも常にサンプリング頻度の方が重要であるため，時間解像度を重視すべきである．一方，工場での不良品検出システムでは，サンプリング頻度よりも精度の方が重要であるため，空間解像度を重視すべきである．このような場合にも対処出来るように，RaVioli ではユーザが優先度を指定することによって，重要である方の解像度を出来るだけ維持するようにすることも可能である．

2.1.2 抽象的な画像処理の記述

デジタル画像は一つ一つが色情報を持っている画素の集合によって構成されている．そのため，画像処理を行う場合に，ユーザは画像の幅と高さの画素数を意識してプログラムを記述する必要がある．そこで，RaVioli ではユーザに画像全体を構成する画素数や画像の幅と高さの画素数を意識させないために，RV_Pixel と呼ばれる画素クラスと RV_Image という画像クラスを定義してカプセル化を行っている．RV_Pixel クラスでは一つの画素の色情報を集約してカプセル化しており，ユーザが色を扱いやすくしている．色を変更する場合には，RV_Pixel クラスが持つメソッドを用いて，RGB または HSV のそれぞれの値を 0〜100%で変化させることで，全ての色を表現する．RV_Image クラスでは画像中の全ての画素の情報と，画像の幅と高さの画素数の情報を持っており，これらをカプセル化によって外部からは見えないようにしている．また画像への抽象的なアクセスを実現させる様々な高階メソッドも提供している．

2.2 高階メソッド

通常，デジタル画像に対する処理を行う場合，基本的には一つ一つの画素に対する処理を繰り返して実現する．例えば，あるデジタル画像を二値化したい場合には，一つ一つの画素に対して輝度が閾値以上なら 1，そうでないなら 0 にするといった処理を繰り返す．この画像処理のプログラム例を図 2 に示す．

```
for(int j=0;j<height;j++){
  for(int i=0;i<width;i++){
    new_Image[i][j]=Binalization(Image[i][j]);
  }
}
```

図 2: 通常の二値化プログラムの例

Binalization() は一つの画素を輝度が閾値以上なら 1 に，そうでないなら 0 にするという処理を記述した関数であるとする．x 軸方向と y 軸方向の座標に対応する変数 i と j をループ変数としてループを構成し，画像を構成する画素数分だけ Binalization() の処理を繰り返している．なお，width と height は画像の幅の画素数と高さの画素数を表している．図 2 のようなプログラムを記述するためには，ループの条件に画像の幅の画素数と高さの画素数を必要とする．そのため，ユーザはこれらを意識してプログラムを記述する必要がある．

前節で述べたように，RaVioli では画像を構成する画素数を意識させない環境を提供するために，RV_Image クラスで複数のメソッドを高階関数の形で定義している．これを**高階メソッド**という．図 2 に示した二値化の画像処理プログラムを RaVioli を用いて記述した例を図 3 に示す．また，図 3 で用いた高階メソッドのプログラムを図 4 に示す．

ユーザ記述プログラムでは，main 関数内で image を RV_Image クラスのインスタンスとして定義し，RV_Image クラスで宣言されている高階メソッドを，画素単位の処理関数を引数として呼び出す．そして，画素単位の処理関数では一画素に対する処理内容を記述する．一方，RaVioli ライブラリでは高階メソッドが定義されており，全画素に対して引数で指定された画素単位の処理関数を適用するループ処理を行っている．また，インクリメント幅に用いられている grain という変数は処理画像の画像解像度を調整するためのものであり，前節で述べた空間解像度を管理するためのパラメータである．

このように，通常の二値化では，ユーザは一つ一つの画素に対して処理を施さなければならなかったが，RaVioli の高階メソッドを使用した二値化では，繰り返し回数を意識する必要はなく，高階メソッドを用いることで間接的に画像処理を行うことが出来る．高階メソッド内で，画像の幅の画素数と高さの画素数を意識する必要がある繰り返し処理を行っているからである．そのため，ユーザが一つの画素に対する処理を

```
ユーザ記述プログラム
void Binalization(Pixel* p){
 /*p の輝度が閾値以上なら 1，そうでないなら 0 にする*/
}

void main(argc,argv[]){
 RV_Image image;
 /*中略*/
 new_Image=Image.proc(Binalization);
 /*中略*/
}
```

図 3: RaVioli を用いた二値化プログラムの例 (ユーザ記述)

```
RaVioil ライブラリ
RV_Image* RV_Image::proc(RV_Pixel (* UserProgram)(RV_Pixel)){
  RV_Image* tmpImage;

  for(int ny=0;ny<height;ny+=grain){
    for(int nx=0;nx<width;nx+=grain){
      tmpImage->pixel[ny*width+nx] = UserProgram(*_getPixel(nx,ny));
    }
  }
  return(tmpImage);
}
```

図 4: RaVioli を用いた二値化プログラムの例 (RaVioli ライブラリ)

記述することは，画像全体に対する処理を記述したことと等しいと考えられる．よって，RaVioli を使用することにより，ユーザは繰り返し回数を意識する必要がなくなり，直感的にプログラムを記述することが出来るようになる．また，図 3 に示したような例では，特定の範囲にのみ処理を施す必要はなかったのだが，特定の範囲にのみ処理がしたい場合も存在する．そのような場合には，高階メソッドの引数に処理対象範囲の座標の記述を加えることで，指定された範囲にのみ画像処理関数を適用することが出来る．また，処理対象範囲の座標の記述には，(0,0) を基準として，画像の横の画素数と縦の画素数をそれぞれ 100.0 %とした値を用いる．このため，ユーザは処理対象範囲の記述にも画像の画素の概念を意識する必要がない．

	テンプレートマッチング	グレーススケール化
RaVioli 不使用時 (s)	2.520	0.040
RaVioli 使用時 (s)	11.032	0.156
不使用時/使用時 (倍)	4.38	3.9

表 1: RaVioli 使用時と不使用時の実行時間

2.3 RaVioli の問題点

RaVioli の高階メソッドを用いることによって，ユーザが抽象的な画像処理記述が行えるようになったということは 2.2 節で既に述べた．ところが，RaVioli を使用した場合は使用しない場合に比べて数倍の実行時間を必要とする．例えば，表 1 に示すように，テンプレートマッチングでは約 4.4 倍，グレースケール化では約 3.9 倍の実行時間低下が見られた．この主な原因として，カプセル化を行ったことによるプログラム記述の容易化と関数呼び出し等のオーバーヘッドの増加がトレードオフの関係にあることが考えられる．そしてプログラムの実行に時間がかかるということは，フレームレートや画像解像度が大きく低下するということにつながるため，高速化を施す必要がある．

そこで本研究では，RaVioli を高速化することによって，フレームレートや画像解像度を少しでも高くすることを目的とする．プログラムの高速化手法としては，ブロック分割やパイプライン化手法を用いたスレッド並列化，SIMD 命令を用いたベクトル並列化といった方法が存在するが，本研究では SIMD 命令を用いた高速化を行うこととした．

3 SIMD 命令を用いた RaVioli の高速化手法

3.1 SIMD 命令

SIMD とは，Single Instruction Multiple Data の略であり，一命令で複数データを同時に処理することでベクトル並列化を行う概念のことである．これを実現する命令セットが SIMD 命令セットである．図 5 に概念図を示す．昔はスーパーコンピュータのベクトル命令や DSP(デジタル信号プロセッサ) の積和演算専用レジスタを用いた積和演算命令等で実装されていたものであるが．Intel の Pentium Ⅲ 等のパーソナルコンピュータ用の汎用 CPU でも実装されて以来，現在一般的に使われている殆どの汎用 PC の CPU に実装されている．

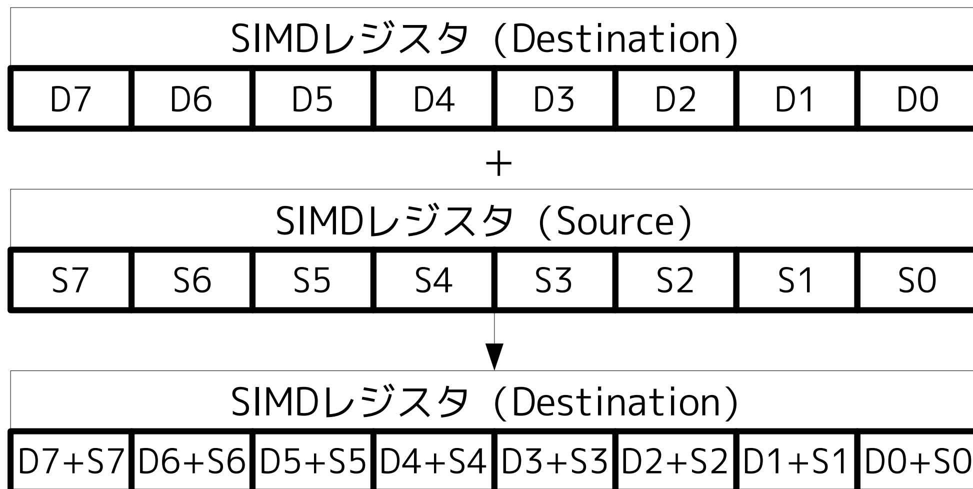

図 5: SIMD の概念図

SIMD を搭載しているプロセッサは PowerPC 系列や Pentium 系列や Cell の SPE 等の CPU，また Geforce 系列や RADEON 系列等の GPU プロセッサ等，多数のプロセッサに搭載されている．本研究では汎用 PC 上で実行する際の高速化を対象とするため，汎用 PC を構成する上で必ず必要とされる CPU プロセッサで SIMD 命令を用いることとした．現在，汎用 PC に用いられている主流の CPU プロセッサには Intel 社の Pentium 系列と AMD 社の Athlon 系列が挙げられる．しかし，両 CPU プロセッサとも SIMD 命令は実装されているが，双方の完全な互換性が無いといった欠点がある．例えば，SIMD 拡張命令セットとして，Pentium 系列には SSE が，Athlon 系列には 3DNow!が実装されており，命令が若干異なる．そのため，アセンブリレベルでの記述が必要となる SIMD 命令を用いるには，CPU プロセッサを考慮する必要がある．本研究では Intel 社の Pentium 系列の CPU を対象とし，SIMD 拡張命令セット SSE を用いた．なお，Intel 社の CPU アーキテクチャに関しては参考文献 [4] が，SIMD 命令については参考文献 [5] が詳しい．

3.2 RaVioli に SIMD 命令を用いる際の問題点

RaVioli を用いた画像処理に SIMD 命令を適用するために，RaVioli ライブラリ内部で直接 SIMD 命令を使うことによって，ユーザがベクトル化を意識したプログラミングを行わなくて済むのが最も望ましい．しかし，SIMD 命令とは，先に述べたとおり，

一命令で複数データを同時に処理するというベクトル並列化を実現する命令のことである．従って，SIMD 命令を用いる箇所は複数データに対して同じ演算を繰り返す等といった箇所に限られる．このことから，画像処理ライブラリにおいて，SIMD 命令による高速化の効果を得るには，複数の画素データを並列に処理する必要がある．ところが，RaVioli では，2 章で述べたように，高階メソッドによって，ユーザは画素単位の処理関数を記述し，処理範囲を指定するだけで対象範囲への画像処理を可能にしている．一方，RaVioli ライブラリの高階メソッドは，ユーザによって指定された画素単位の処理関数を，指定範囲にループを用いて適用する処理をしている．つまり SIMD 命令を適用すべき具体的な処理はユーザプログラム内に記述され，RaVioli ライブラリ側で提供する高階メソッドが担うのは，その処理内容によらない汎用的な部分に限られる．よって高階メソッド自体を，具体的な SIMD 命令を使用する形に書き換えることは不可能である．

それでは，ユーザプログラム中に SIMD 命令を記述する場合はどうだろうか．RaVioli の高階メソッドを用いたプログラミングでは，ユーザは画素単位の処理しか記述することが出来ない．そのため，複数画素を並列処理する SIMD 命令をユーザプログラム中に記述することは出来ない．そこで，高階メソッドのソース内の画素を走査する二重ループのインクリメント幅を，用いた SIMD 命令が並列処理できる画素数分に変更出来るような仕様に変更する方法がある．しかし，この方法ではユーザが SIMD 命令を記述しなければならない上，画像の画素数を意識したプログラミングを行う必要があり，2.1 節で述べた RaVioli の特徴の一つである抽象的な画像処理の記述が出来なくなるという欠点がある．

このように，RaVioli を用いた画像処理を行うプログラムに対して，RaVioli 側からもユーザ側からも SIMD 命令を実装することは困難である．

3.3 SIMD 命令を用いた具体的な記述方法

RaVioli に SIMD 命令を用いた高速化を施すために，まず RaVioli を用いない画像処理に SIMD 命令を適用した場合にどのようなプログラムとなるのかを考えてみることとする．テンプレートマッチングを行う画像処理関数を SIMD 命令を用いて記述したプログラム例を図 6 に示す．

図 6 のプログラムは入力画像とテンプレート画像との RGB 値を比較するものである．具体的には，入力画像 `input_image` からテンプレート画像 `input_tp` の大きさ分の画像を切り出して，その画像とテンプレート画像との RGB 値の差の絶対値の和 `allsum` を

```
int min=2147483646;
int allsum=0;
for(j=0;j<input_image.height-input_tp.height;j++){
  for(i=0;i<input_image.width-input_tp.width;i++){
    for(jj=0;jj<input_tp.height;jj++){
      for(ii=0;ii<input_tp.width;ii+=16){
        //SIMD 命令で記述された画像処理アルゴリズム
        asm volatile (
           "movdqu (%1),%%xmm0\n\t"
           "movdqu (%2),%%xmm1\n\t"
           "movdqu (%3),%%xmm2\n\t"
           "movdqu (%4),%%xmm3\n\t"
           "movdqu (%5),%%xmm4\n\t"
           "movdqu (%6),%%xmm5\n\t"
           "psadbw %%xmm1,%%xmm0\n\t"
           "psadbw %%xmm3,%%xmm2\n\t"
           "psadbw %%xmm5,%%xmm4\n\t"
           "paddd %%xmm4,%%xmm2\n\t"
           "paddd %%xmm2,%%xmm0\n\t"
           "movdqu   %%xmm0,%0\n\t"
           "emms" : "=g" (sum) :
           "r" (&input_image.R[(j+jj)*input_image.width+i]),
           "r" (&input_tp.R[(jj)*input_tp.width]),
           "r" (&input_image.G[(j+jj)*input_image.width+i]),
           "r" (&input_tp.G[(jj)*input_tp.width]),
           "r" (&input_image.B[(j+jj)*input_image.width+i]),
           "r" (&input_tp.B[(jj)*input_tp.width]));
         //SIMD 命令ここまで
         allsum+=sum[0]+sum[2];
       }
     }
     if(min > allsum) {
       min=allsum;
       mini=i;
       minj=j;
     }
     allsum=0;
   }
}
```

図 6: SIMD 命令を用いたテンプレートマッチング

計算する．そして，入力画像全体に同様の処理を繰り返し、RGB 値の差の絶対値の和の最小値とその時の入力画像の左上の座標を求める．SIMD 命令で記述されている部分はコメントで示した部分である．記述にはインラインアセンブラを用いている．インラインアセンブラについては後述する．

しかし，ユーザが一画素に対する画像処理アルゴリズムを記述し，高階メソッドが処理対象範囲へのループ処理を行う RaVioli で図 6 のようなプログラムを直接記述するのは困難であることは前節で示した．SIMD 命令を記述するためには，ユーザが記述した画像処理関数を RaVioli の高階メソッドの中にインライン展開することが必要である．そうすることで、複数画素のデータを並列に処理するようにループを書き換えることが可能となり，図 6 のようなプログラムに書き換えることが出来る．しかし，このような変換を多種多様な画像処理関数とそれを呼び出している高階メソッドとを組み合わせて手作業で行う場合，多大な労力が必要となる．

そこで，本研究ではいくつかのプログラムを手作業で変換することで，変換のパターンを発見し，ユーザ記述の画素単位の処理関数と使用した高階メソッドのソースから，二つのソースを統合し，SIMD 命令を適用したプログラムへの変換を自動で行うことが出来るトランスレータを実装する．変換例については付録 A.1,A.2 に掲載する．具体的な変換手法については 4 章で説明することとし，本節では SIMD 命令をどのように記述するかについて述べる．C++プログラムに SIMD 命令を適用するには，次のような方法がある．

- コンパイラによる自動 SIMD 化
- イントリンシック命令を用いる
- インラインアセンブラを用いる
- 外部アセンブラでアセンブルしたオブジェクトファイルをリンクする

コンパイラによる自動 SIMD 化とは，コンパイル対象のプログラムに対して，コンパイラが SIMD 命令の使用が有効であるかどうかを検討し，有効だと判断した場合に SIMD 命令を用いるというものである．コンパイラによる自動 SIMD 化を行うには，コンパイルオプションを変更するだけでよく，非常に容易である．しかし，ベクトル化が可能な箇所を見つけられる場合は非常に限られているため，高速化効果が高いとは言えない．

イントリンシック命令とは，`mmintrin.h` 等の，使用する命令セットに応じたヘッダファイルをインクルードすることで，組み込み関数を用いることで SIMD 命令を使用する事が出来るというものである．組み込み関数なので直感的に記述しやすい上，コ

```
gcc コンパイラ (GNU アセンブラ)
add %xmm0,%xmm1

Intel コンパイラ (Microsoft Macro Assembler)
add xmm1,xmm0
```

図 7: アセンブリ言語記述方式

ンパイラが使用する SIMD レジスタを自動で決定してくれるのでレジスタ管理が必要ないという利点もある．しかし，必ず SIMD レジスタが使用される保証が無く，必要となるヘッダファイルが無い環境では動作しないという欠点がある．

インラインアセンブラとは，C++プログラム内でアセンブリ言語での記述を可能としたものである．インラインアセンブラは，C++で標準サポートされており，asm 文で記述を行う．コンパイラによっては__asm 文の場合もある．asm 文では対象 CPU のアーキテクチャに合ったアセンブリ文法を用いて記述する必要がある．利点として，高速化したい部分のみアセンブリで記述することが可能であるという点が挙げられる．また，インラインアセンブラを更に発展させた拡張インラインアセンブラというものもある．この拡張インラインアセンブラを用いることで，通常のインラインアセンブラでは C++の変数にアクセスする際に，スタックポインタやフレームポインタを用いた記述を必要とするが，それらの記述なしに C++の変数にアクセスすることが出来る．

最後に，外部アセンブラを用いる方法であるが，アセンブリ言語で書いたソースファイルを用意し，それを外部アセンブラでアセンブルした結果生成されたオブジェクトファイルをリンクして利用する．これはインラインアセンブラではコンパイラで行うことを外部アセンブラを用いて行う以外はあまり違いはなく，ソースコード管理等の点で外部アセンブラを用いた方がよいことがある．ただし，外部アセンブラを用いるほうが手軽さの点でインラインアセンブラに劣る．

このように，SIMD 命令を記述する方法は色々あるが，本研究ではインラインアセンブラを用いた記述を行うこととする．しかし，アセンブリによる記述を行うことの欠点として，環境による影響が非常に大きいということが挙げられる．依存する環境の一つとして，使用するコンパイラがある．その理由は，コンパイラが使用しているアセンブラが異なるため，アセンブリ言語の記述形式が異なるからである．記述方式の例を図 7 に示す．

図 7 で示した以外にもアセンブリ言語記述方式は存在するが，ここでは割愛する．図

第一引数	高階メソッド名
第二引数	ユーザプログラム名
第三引数	高階メソッド呼び出しを行っている関数
第四引数	返り値に設定する変数名

表 2: 引数対応関係

7 で示した例のように記述すると，両方とも xmm0 レジスタと xmm1 レジスタの値を加算するという意味になる．このように，アセンブリ言語を用いたプログラミングでは環境も考慮する必要がある．本研究では gcc コンパイラを使用した場合のアセンブリ言語記述形式を採用することとする．なお，x86 アーキテクチャCPU を対象とした gcc を用いたインラインアセンブラに関しては参考文献 [6] が詳しい．

4 SIMD 命令を用いたプログラムへの自動変換

本章では，ユーザの記述した画像処理アルゴリズムから，専用高階メソッドを生成することで，画像処理アルゴリズムを SIMD 命令化するといった変換を自動で行うトランスレータについて説明する．

4.1 変換手順の概略

ユーザが記述した画像処理プログラムを SIMD 化されたプログラムへと変換する作業を自動で行うトランスレータを実装した．変換手順は次の様になる．

1. ユーザプログラムと rv_image.cpp の読み込み
2. 変換対象部分を変数に格納
3. 画素単位の処理関数を高階メソッド内にインライン展開
4. 画像処理アルゴリズムを SIMD 命令に変換
5. 専用高階メソッドとして出力する

このうち，2〜4 について次節から詳しく説明していく．また，トランスレータに必要な引数として，高階メソッド名，ユーザプログラム名，高階メソッド呼び出しを行っている関数名，返り値に設定する変数名を設定した．今後，それぞれ順に第一引数，第二引数，第三引数，第四引数と呼ぶことにする．それぞれの引数の対応関係を表 2 にまとめる．

高階メソッドを呼び出している関数名で検出

```
void countTP(RV_DoppelImage* image,RV_Coord Cstart,RV_Coord Cend){
  image->procImgComp(SAD,input_tp);
  if(min > sum) {
   min=sum;
    tmps=Cstart;
    tmpe=Cend;
  }
  sum=0;
}
```

高階メソッド名が記述されている
行から様々な情報が解析できる

```
void SAD(RV_Pixel* p1,RV_Pixel* p2){
  int r1,g1,b1,r2,g2,b2;
  p1->getRGB(r1,g1,b1);
  p2->getRGB(r2,g2,b2);
  sum+=abs(r1-r2)+abs(g1-g2)+abs(b1-b2);
}
```

図 8: 変換対象部分の検出 (ユーザ定義の一画素に対する処理が記述されている関数)

4.2　変換対象部分

まず，トランスレータは第二引数で指定された名称のユーザプログラムと高階メソッドが記述されている RaVioli の rv_image.cpp プログラムを読み込む．そして変換を行うために，これらのプログラム中から，変換対象となるソース部分を特定する必要がある．そこで，第一引数と第三引数で指定された情報を用いる．これらの引数から変換対象部分を検出する様子をユーザ定義の一画素に対する処理が記述されている関数は図 8 に，高階メソッドは図 9 に示す． 図 8 に示したように，まず，ユーザプログラム中で第三引数で指定された関数が記述されている部分の検索を行う．第三引数で指定された関数の内部ではトランスレータに第一引数として渡された高階メソッドが呼び出されているはずなので，呼び出しを行っている行を検索する．そしてその行に対して，パターンマッチングによる字句解析を用いた情報抽出を行う．この解析をする理由は，この行にはユーザ定義の一画素に対する処理が記述されている関数名だけで

```
RV_Image* RV_Image::procImgComp(RV_Pixel (* UserProgram)
(RV_Pixel*,RV_Pixel*),RV_Image* cmpImg){
        /*省略*/
}
```

同名高階メソッドだが返り値の型が異なる

```
void RV_Image::procImgComp(void (* UserProgram)(RV_Pixel*,
RV_Pixel*),RV_Image* cmpImg){
        /*省略*/
}
```

図 9: 変換対象部分の検出 (高階メソッド)

なく，変換対象の高階メソッドの返り値型とユーザ記述の画素単位の処理関数名以外の引数の型を示す情報が記述されているからである．図 8 のプログラムの例では，次に示す記述が対象の行である．

```
image->procImagComp(SAD,input_tp);
```

この記述を例に字句解析の内容を説明する．まず，記述された式の中に‘‘=’’が存在するかどうか調べる．‘‘=’’が存在する場合は，左辺の変数の型が高階メソッドの返却値型である．この例では‘‘=’’が存在しない．従って，ここで呼び出している高階メソッドには返り値が存在しないので，変換対象の高階メソッド procImgComp の返り値型は void 型であることがわかる．

次に，高階メソッドの第一引数は必ずユーザ記述の画素単位の処理関数をとる．従って，この例では SAD がユーザ記述の画素単位の処理関数名であることがわかる．

最後に，高階メソッドの第一引数以外の引数の型を調べる．この例では第一引数以外には第二引数が存在し，変数 input_tp が渡されている．従って，変数 input_tp の型である RV_image*が高階メソッドの第二引数に指定される必要のある型だとわかる．

ユーザ記述の画素単位の処理関数名と，変換対象の高階メソッドに関する情報が必要な理由について説明する．ユーザ記述の画素単位の処理関数名は，変換対象の画素単位の処理関数をユーザプログラム中から検出するために必要である．また，変換対象の高階メソッドに関する情報は，変換対象の高階メソッドを特定するために必要である．高階メソッド名だけで変換対象の高階メソッドを特定できないのは，rv_image.cpp では同名の高階メソッドが記述されている場合があるためである．

図 10: インライン展開 (展開前)

この解析によって，変換対象のユーザ定義の一画素に対する処理が記述されている関数名がわかったため，ユーザプログラム中から対象のユーザ定義の処理関数を検索し，関数全体をトランスレータ内で定義している変数に格納する．

図 9 に示したように，次に RaVioli プログラム中から，第一引数で指定された高階メソッドを検索する．そして，当該メソッドの解析を行い，先程ユーザプログラムで行った解析による情報と照らし合わせ，変換対象の高階メソッドを特定する．最後に，特定した変換対象の高階メソッドの関数全体をトランスレータ内で定義している変数に格納する．

4.3 関数のインライン展開

4.2 節で変換対象となるユーザ定義の画素単位の処理が記述されている関数と高階メソッドをそれぞれ変数に格納した．次に，変換対象となる画素単位の処理が記述されている関数のソースを，対象高階メソッドのソース内にインライン展開する処理を行う．

具体的にインライン展開する手順を説明する．図 10 に示したように，高階メソッド

```
void RV_Image::SIMD_SAD(void (* UserProgram)(RV_Pixel*,
RV_Pixel*),RV_Image* cmpImg){
 int nx,ny;
 int tmpflag;
 int cmpgrain;
 int tmpgrain=0;

 _InputCheck();
 cmpgrain=cmpImg->getgrain();
 for(ny=0;ny<Bheight;ny+=grain){
  for(nx=0;nx<Bwidth;nx+=grain){
   RV_Pixel* p1=_getPixel(nx,ny);
   RV_Pixel* p2=cmpImg->_getPixel(nx,ny);
   int r1,g1,b1,r2,g2,b2;
   p1->getRGB(r1,g1,b1);
   p2->getRGB(r2,g2,b2);
   sum+=abs(r1-r2)+abs(g1-g2)+abs(b1-b2);
  }
 }
}
```

図 11: インライン展開 (展開後)

のソース内には，画素を走査する二重ループの中に関数 UserProgram がある．UserProgram は高階メソッドの引数として受け取る関数ポインタであり，ユーザが定義した画素単位の処理が記述されている．UserProgram が定義されている行をユーザ定義の画素単位の処理が記述されている関数のプログラムに書き換え，引数の対応付けを行う．このような変換を行うと，図 11 のようなプログラムに変換することが出来る．以後，図 11 のようなインライン展開によって得られた関数を**専用高階メソッド**と呼ぶことにする．また，画素単位の処理関数によっては，図 12 のようにユーザプログラム中で大域変数を用いた記述を行っている場合がある．インライン展開を行うことによりユーザプログラムの大域変数を用いることが出来なくなるため，その場合は専用高階メソッドに返り値を設定する必要がある．そのように返り値の設定が必要な場合

```
int sum=0;
/* 他の変数宣言 */

void countTP(RV_DoppelImage* image,RV_Coord Cstart,RV_Coord Cend){
  image->procImgComp(SAD,input_tp);
  if(min > sum) {
   min=sum;
    tmps=Cstart;
    tmpe=Cend;
  }
  sum=0;
}

void SAD(RV_Pixel* p1,RV_Pixel* p2){
  int r1,g1,b1,r2,g2,b2;
  p1->getRGB(r1,g1,b1);
  p2->getRGB(r2,g2,b2);
  sum+=abs(r1-r2)+abs(g1-g2)+abs(b1-b2);
}
```

図 12: 大域変数を用いた画素単位の処理関数

のために，第四引数に返り値に設定する変数名を用意する．第四引数の指定に応じて，専用高階メソッドの関数の頭部に記述されている返り値型の値を変更する．引数の指定がある場合は，指定された変数の型に変更し，無い場合は変更を行わない．図 11 のプログラムで，第四引数に sum が与えられた場合を例として説明する．第四引数で与えられた変数 sum は int 型で宣言されているものとする．この変数を返り値に設定するにあたって，専用高階メソッドの関数の頭部に記述されている返り値型 void を int に変換する必要がある．更に，引数の指定がある場合には，専用高階メソッドの最後で返り値としてその値を返す必要があるため，return(第四引数); の記述を加える．第四引数によるこれらの変換の結果，専用高階メソッドは次に示すようなプログラムとなる．

```
int r,g,b,y;                              変数宣言部
p.getRGB(r,g,b);                          画素値取得部
y=(int)(0.299*r + 0.587*g + 0.114*b);     画像処理部
p.setRGB(y,y,y);                          演算結果出力部
```

図 13: 画像処理プログラム部分 (抜粋)

```
int RV_Image::procImgComp(void (* UserProgram)
      (RV_Pixel*, RV_Pixel*),RV_Image* cmpImg){
      /* インライン展開を施した画像処理アルゴリズム */
      return(sum);
}
```

4.4 SIMD 命令への変換

4.3 節でインライン展開を行った関数の画像処理プログラムの部分は，ループイテレーション変数のインクリメント幅を変更することによって，SIMD 命令に書き換えることが可能である．ここで，ユーザが RaVioli を用いて記述した画像処理プログラムを SIMD 命令に変換する手順について説明する．RaVioli を用いて記述した一般的な画像処理プログラムは，図 13 に示したように大まかに変数宣言部，画素値取得部，画像処理部，演算結果出力部の四つにブロック分割することができる．プログラムによっては演算結果出力部が存在しないこともある．変換手順としては，画像処理部を SIMD 命令化した後に変数宣言部と画素値取得部と演算結果出力部を SIMD 命令に対応させるといった流れとなる．

まずは，画像処理部を SIMD 命令化する．RGB 色空間での画像処理の計算式は，例えば

$$out = 0.299 * r + 0.587 * g + 0.114 * b; \tag{1}$$

のように記述される．これを SIMD 命令に変換するには，コンパイラで行われているような字句解析をトランスレータで行い，解析結果に基づいてアセンブリ命令を出力すればよい．

この解析の際に，変数の型が問題となってくる．SIMD 命令で効率よく計算するためには，1 個の要素を計算するのに最低限必要とするバイト数でパックされたデータを

```
for(ny=0;ny<Bheight;ny+=grain){
 for(nx=0;nx<Bwidth;nx+=grain){
  byte r1,g1,b1,r2,g2,b2;
  p1 = _getPixel(nx,ny);
  p2 = cmpImg->_getPixel(nx,ny);
  p1->getRGB(r1,g1,b1);
  p2->getRGB(r2,g2,b2);
  asm volatile (
      /*テンプレートマッチングのSIMD命令による計算*/
     : "=g" (sum) :
    "r" (&r1),
    "r" (&r2),
    "r" (&g1),
    "r" (&g2),
    "r" (&b1),
    "r" (&b2));
  }
 }
```

図 14: 変換前

用いらなければならない．例えば，(1) で示した式において r,g,b の値はそれぞれ 8bit の byte 型で格納されているものとする．$0.299 * r$ を行うのに，0.299 は小数点を含む値なので浮動小数点を格納できる float 型 (32bit)，r は整数を格納する byte 型 (8bit) となる．これらを計算するためには，演算の型を浮動小数点が格納できる型に統一せねばならず，float 型 (32bit) で計算する必要があり，結果も float 型となる．(1) の式では，$0.587 * g$ および $0.114 * b$ の処理結果も同様に float 型となるので，32bit の単精度浮動小数点でパックされた値を計算する SIMD 命令を用いなければならない．このように，字句解析の際に変数の型も調べる必要がある．

次に，SIMD 命令に合わせて専用高階メソッドを書き換える．書き換える前のプログラム例を図 14 に示す.

まず，変数宣言部を画像処理部の SIMD 命令に合わせた宣言に書き換える．図 14 に示すようなプログラムならば，SIMD 命令で 8bit の byte 型でパックされた値を用いているので，128bit の SIMD レジスタならば 16 個同時に演算する事が出来る．そこで，SIMD 命令で用いている変数を配列に変換する．この例ならば 16 個同時に演算できるので，演算に用いる変数の配列を 16 個確保するといった記述に変換する．

更に，画素値取得部を画像処理部の SIMD 命令に対応させた記述に変更する．例え

```
for(ny=0;ny<Bheight;ny+=4*grain){
 for(nx=0;nx<Bwidth;nx+=4*grain){
  byte r1[16],g1[16],b1[16],r2[16],g2[16],b2[16];
  for(int i=0;i<16;i++){
   p1 = _getPixel(nx+(i%4),ny+(i/4));
   p2 = cmpImg->_getPixel((nx+(i%4)),((ny+i/4)));
   p1->getRGB(r1[i],g1[i],b1[i]);
   p2->getRGB(r2[i],g2[i],b2[i]);
  }
  asm volatile (
      /*テンプレートマッチングの SIMD 命令による計算*/
     : "=g" (sum) :
    "r" (&r1),
    "r" (&r2),
    "r" (&g1),
    "r" (&g2),
    "r" (&b1),
    "r" (&b2));
  }
 }
```

図 15: 変換後

ば図 14 のプログラム例ならば，画像処理部の SIMD 命令で一度に 16 個のデータを必要とする．そのため，画素値取得も 16 個分行う必要があるので，その分画素値取得部をループさせなければならない．演算結果出力部が存在する場合は，同様の変換を行う．

最後に，画像処理アルゴリズムをループ実行する回数を変更しなければならない．その理由は，１回の演算で複数画素分の演算を行うためである．図 14 のプログラム例では 16 画素分のデータを同時処理するので，縦横それぞれ 4 画素分進むようにインクリメント幅を変更する．図 14 のプログラムに対してこれらの変換を行った結果，図 15 に示すようなプログラムを得ることが出来る．

5　評価

SIMD 命令を使用せず，ユーザ定義の一画素に対する画像処理が記述された関数と RaVioli の高階メソッドを用いた場合と，トランスレータによって生成された SIMD 命令を使用した専用高階メソッドを用いた場合との実行時間の比較を行った。

評価には次の表 3 に示すような環境を用いたところ，図 16 に示すような結果が得られた。値はそれぞれの評価プログラムに対して，変換前の実行時間を 1 に正規化した

CPU	Opteron 2.0GHz
メモリ	2GB
コンパイラ	GNU C++コンパイラ version 4.1.2

表 3: 評価環境

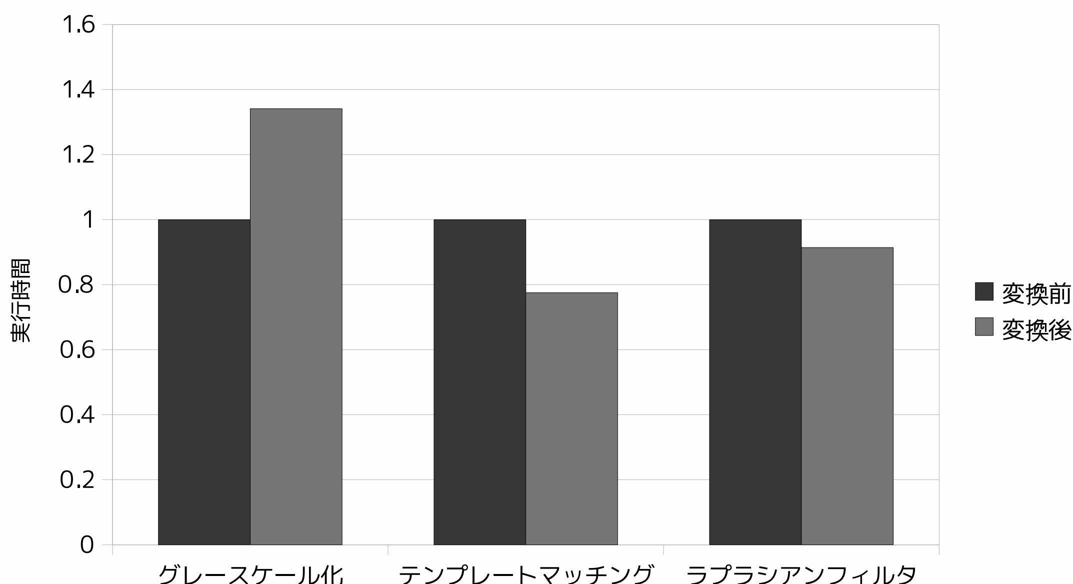

図 16: 評価結果

値を用いている．

図 16 から，変換前と変換後の実行時間を比較してみると，テンプレートマッチングでは約 22%，ラプラシアンフィルタでは約 9%の実行時間の減少を確認する事が出来る．このことから，画像処理アルゴリズムを SIMD 命令を用いたプログラムに変換したことによる高速化が実現できていると考えられる．

しかし，テンプレートマッチングでは 16 画素を，ラプラシアンフィルタでは 8 画素を並列に処理しているので，単純に考えるとそれぞれ約 16 倍，約 8 倍の速度向上が見込めてもよいはずである．ところが，速度向上比率は 2 倍にも満たなかった．また，グレースケール化に至っては約 34%実行時間が増加してしまっている．

これらの原因として，計算に必要な画像の画素値をパックする時間が SIMD 命令によるベクトル演算での時間短縮分を埋めてしまったためだと考えられる．例えば 8 画素分をベクトル演算しようとした際には，ループを用いて 8 回 `getRGB` や `getR` 等の RV_Pixel クラスで定義されている画素値取得インスタンスメソッドを呼び出す必要が

	グレーススケール化	テンプレートマッチング	ラプラシアンフィルタ
画像処理演算時間 (s)	0.028	7.378	0.024
関数呼び出し時間 (s)	0.083	3.737	0.128
関数全体実行時間 (s)	0.111	11.115	0.152

表 4: 画素単位の処理関数実行時間

ある．その際に，通常の演算では必要なかった，取得した値を配列に格納しておく処理を行わなければならない．この処理では，配列にアクセスする必要があるのでメモリアクセスを伴う可能性があり，速度低下を招く．また，インラインアセンブラでパックされた値を SIMD レジスタに読み込み，SIMD 命令によるベクトル演算を行うが，その演算結果を C++で用いるためには，SIMD レジスタから配列に格納する必要がある．この操作も通常の演算では必要ないため，速度低下の一因である．

また，グレースケール化で遅くなった理由を考える．表 4 に示すように，グレーススケール化では関数呼び出し時間に比べて画像処理演算にあまり時間がかからない．そのため，先に述べた SIMD 命令によるベクトル演算を行うための準備によるオーバーヘッドが，ベクトル演算による速度向上分よりも大きかったのではないかと考えられる．

6　おわりに

擬似的なリアルタイム処理を実現する，解像度非依存型動画像処理ライブラリ RaVioli を SIMD 命令を用いて高速化した．しかし，RaVioli ならではの特性上，そのまま SIMD 命令を実装することは出来なかった．そこで，ユーザが記述した一画素に対する画像処理関数プログラムと RaVioli 高階メソッドプログラムから，その画像処理専用関数を生成し，その関数を SIMD 命令に変換するトランスレータを実装することとした．実装したトランスレータが生成した専用処理関数を用いた場合と，RaVioli 高階メソッドを用いて画素単位の処理関数を実行した場合と比べて，プログラムによっては高速化出来たことを確認することが出来た．

今後の課題としては，本トランスレータを複雑な画像処理関数にも対応することや，生成した専用関数に自動最適化を施して更なる高速化を行うといったこと，使用コンパイラに対応して出力アセンブリを変更することが挙げられる．複雑な画像処理関数とは，画像処理部分に if 文を含んでいるようなアルゴリズム等である．これらのアルゴリズムは，SIMD 命令に変換するパターンがまだ分かっていないため，現在のトランスレータでは対応することが出来ない．また，現在トランスレータが生成すること

が出来る専用処理関数は，更なる最適化を施す余地がある場合があるので，SIMD レジスタを効率よく利用できるよう最適化も必要である．更に，ユーザプログラム中で高階メソッドを呼び出している関数で，同じ高階メソッドが 2 回以上呼び出されている場合の動作が未定義であるといった問題もある．

謝辞

本研究のために，多大な御尽力を頂き，御指導を賜わった名古屋工業大学の松尾啓志教授，津邑公暁准教授，齋藤彰一准教授，松井俊浩助教に深く感謝致します．また，本研究の際に多くの助言，協力をして頂いた松尾・津邑研究室の方々に深く感謝致します.

参考文献

[1] Köthe, U.: *VIGRA - Vision with Generic Algorithms*, 1.6.0 edition (2008).

[2] Bradski, G. and Kaehler, A.: *Learning OpenCV: Computer Vision With the Opencv Library*, Oreilly & Associates Inc (2008).

[3] 岡田慎太郎, 桜井寛子, 津邑公暁, 松尾啓志: 解像度非依存型動画像処理ライブラリ RaVioli の提案と実装, 情報処理学会論文誌：コンピュータビジョンとイメージメディア（CVIM）, Vol. 1, No. 4 (2009).

[4] Corp., I.: IA-32 インテルアーキテクチャ・ソフトウェア・デベロパーズ・マニュアル, `http://www.intel.co.jp/jp/download/index.htm`.

[5] : IA-32 SIMD の扉, `http://www.icnet.ne.jp/~nsystem/simd_tobira/index.html`.

[6] SAITOH, A.: GCC でインラインアセンブリを使用する方法と留意点等 for x86, `http://www.mars.sannet.ne.jp/sci10/on_gcc_asm.html`.

付録

A.1 変換前

ユーザ記述プログラム

```
void SAD(RV_Pixel* p1,RV_Pixel* p2){
  byte r1,g1,b1,r2,g2,b2;
  p1->getRGB(r1,g1,b1);
  p2->getRGB(r2,g2,b2);
  sum+=abs(r1-r2)+abs(g1-g2)+abs(b1-b2);
}
```

RaVioli プログラム

```
void RV_Image::procImgComp(void (* UserProgram)
      (RV_Pixel*, RV_Pixel*),RV_Image* cmpImg){
  int nx,ny;

  _InputCheck();
  cmpgrain=cmpImg->getgrain();
  for(ny=0;ny<Bheight;ny+=grain){
    for(nx=0;nx<Bwidth;nx+=grain){
      UserProgram(_getPixel(nx,ny), cmpImg->_getPixel(nx,ny));
    }
  }
}
```

A.2 変換後

```
int RV_Image::SIMD_procImgComp(RV_Image* cmpImg){
  int nx,ny;
  int sum[4];
  int allsum;
  int i;
```

```
byte r1[16],g1[16],b1[16],r2[16],g2[16],b2[16];

RV_Pixel* p1;
RV_Pixel* p2;

_InputCheck();

asm volatile ("pslldq \$255,%xmm3");//0に初期化
for(ny=0;ny<Bheight;ny+=4*grain){
  for(nx=0;nx<Bwidth;nx+=4*grain){
    for(i=0;i<16;i++){
      p1 = _getPixel(nx+(i%4),ny+(i/4));
      p2 = cmpImg->_getPixel((nx+(i%4)),((ny+i/4)));
      p1->getRGB(r1[i],g1[i],b1[i]);
      p2->getRGB(r2[i],g2[i],b2[i]);
    }
    asm volatile (
      "movdqu (%1),%%xmm0\n\t"
      "movdqu (%2),%%xmm1\n\t"
      "psadbw %%xmm1,%%xmm0\n\t"
      "movdqu (%3),%%xmm1\n\t"
      "movdqu (%4),%%xmm2\n\t"
      "psadbw %%xmm2,%%xmm1\n\t"
      "paddw  %%xmm1,%%xmm0\n\t"
      "movdqu (%5),%%xmm1\n\t"
      "movdqu (%6),%%xmm2\n\t"
      "psadbw %%xmm2,%%xmm1\n\t"
      "paddw  %%xmm1,%%xmm0\n\t"
      "paddd  %%xmm0,%%xmm3"
      : "=g" (sum) :
      "r" (&r1),
      "r" (&r2),
```

```
        "r" (&g1),
        "r" (&g2),
        "r" (&b1),
        "r" (&b2));
    }
  }
  asm volatile (
    "movdqu %%xmm3,%0\n\t"
    "emms" : "=g" (sum));
  allsum=sum[0]+sum[2];
  return(allsum);
}
```